Projet Couches Logicielles Basses : Passer en multicœur

Table des matières

Contexte

Après avoir réalisé un kernel très léger pour RasPi3, comprenant un ordonnanceur, une isolation mémoire ainsi que des appels systèmes, nous souhaitons maintenant adpater ce kernel à un système multicœur. Pour faire cela nous avons premièrement placé un biglock sur le kernel pour permettre un usage multicœur avant de le remplacer, dans un second temps, par une multitudes de spinlock sur les structures de données critiques.

1 Multicoeur via un biglock sur le kernel

Chaque cœur configure son propre timer générique et ses registres système avant de se mettre en attente grâce a WFI, Ils regardent tous les X temps si une tache leur a été donné en passant dans le scheduler.

On a ajouté un tableau de taches à faire contenant les fonctions liées. Le tableau des taches à faire du scheduler est lock pas une mutex pour éviter que 2 cœurs fasses la même tâche. quand un cœur fini sa tâche, il repasse dans le scheduler. Si il n'y a pas de tâche dans le tableau, le cœur repasse en attente active.

méthode pour le big lock : load byte exclusive big lock = mutex un seul cœur exécute une section critique(SC). pour ça faut qu'on puisse entrer en SC seulement si personne en SC pour ça variable commune qui dit si SC → doit être en mémoire (accès mémoire prend plusieurs temps) on utilise un load exclusif pour ne pas avoir d'accès mémoire concurrent

« Nous allons utiliser l'outil « Bottom » afin de voir l'état des cœurs donc pour savoir si actif use de UART avec fonction num cœur qui tourne pour savoir quel cœur fait quelle tâche nous allons faire un tableau de résultat des timer et des tâches. »

Nous avons aussi codé « user.c » qui fait qu'un cœur aléatoire crée les process, les autres cœurs en prennent un et créent eux mêmes un autre process qui lui va boucler sur une chaîne de caractères. Cela montre que n'importe quel cœur est capable de créer un processus sans conflits et que les données communes ne sont pas corrompues (sortie uart, liste des tâches etc..)

sources: doc ARM

1.1 Pas-à-pas

Modification Boot.S

```
.globl start_cores
start_cores:
  adrp x2, _start
  add x2, x2, #:lo12:_start
  ldr x3, =VA_START
  sub x2, x2, x3 // Since the PC will be virtual, we need to
give the other cores a physical address by subtracting the
virtual offset
  mov x1, #0xd8
  str x2, [x1,#8]!
  str x2, [x1,#8]!
  str x2, [x1,#8]
  ret
```

Les cœurs secondaires restent initialement en attente en lisant en boucle un registre de démarrage, et commencent leur exécution dès qu'une adresse valide est écrite dans ce registre.

Une fonction dédiée au réveil des différents cœurs a été ajoutée. Son rôle est de transmettre l'adresse du du démarrage du boot.s (start) aux cœurs secondaires. Comme cette fonction est appelée après l'activation de la MMU sur le premier cœur, l'adresse de start, initialement une adresse virtuelle, est convertie en adresse physique avant d'être écrite dans le registre de démarrage des cœurs secondaires.

Les différents cœurs peuvent alors quitter leur boucle d'attente et exécuter leur propre routine Master.

Dans la fonction Master, plusieurs registres systèmes doivent être initialisés pour l'ensemble des cœurs, et chacun d'eux doit disposer de sa propre pile

d'exécution. Cependant, certaines parties de la fonction sont communes à tous les cœurs et ne doivent être exécutées qu'une seule fois, comme la gestion du segment BSS ou la création des tables de pages.

Pour cela, le registre x19 est utilisé afin de stocker l'identifiant du cœur actif. Cette valeur permet de conditionner l'exécution de certaines sections du code, garantissant que seul le cœur 0 réalise les initialisations partagées.

```
  cbnz x19, skip_create_tables // Only let core 0 perform
the page table creation
  bl __create_page_tables
```

Afin de permettre l'initialisation correcte des timers génériques propre à chaque cœurs, une entrée PMD supplémentaire a été ajoutée aux tables de pages. Cette entrée permet de mapper une section mémoire dédiée contenant les registres de configuration des timers génériques utilisés par les différents cœurs.

Le mapping est effectué en adresse virtuelle, ce qui garantit que les timers puissent être correctement initialisés et utilisés après l'activation de la MMU.

```
// PGD --+PAGE_SIZE--> PUD --+PAGE_SIZE--> PMD1 --
+PAGE_SIZE--> PMD2
// Fetch PUD address
adrp    x10, pg_dir
add     x10, x10, #PAGE_SIZE

// Fetch PMD address
adrp    x11, pg_dir
add     x11, x11, #(3 * PAGE_SIZE)

// Map second PMD address
orr     x12, x11, #MM_TYPE_PAGE_TABLE
str     x12, [x10, #8]
```

```
// Map timer addresses
mov     x0, x11
ldr     x1, =GENERIC_TIMER_START                // Physical
address of timer region
ldr     x2, =(VA_START + GENERIC_TIMER_START)    // Virtual
address start
ldr     x3, =(VA_START + GENERIC_TIMER_START)    // Virtual
address end : same address as start to enforce the mapping
of a single section
```

Modification Kernel.c

```
void kernel_main() {
    unsigned char core_id = get_core_id();
    // Core 0 initializes the uart and IRQ controller before
waiting for the other cores to start
    lock();
    if (core_id == 0){
      uart_init();
      init_printf(NULL, putc);
      printf("kernel boots ...\n\r");
      enable_interrupt_controller();
      int res = copy_process(PF_KTHREAD, (unsigned
long)&kernel_process, 0);
      if (res < 0) {
          printf("error while starting kernel process");
          return;
      }
      start_cores();
      printf("Starting other cores...\n\r");
    }
    irq_vector_init();
    generic_timer_init();
    enable_irq();

    printf("Core %d : Successfully started\n", core_id);
    unlock();
    while(1){
      lock();
```

```
            schedule(core_id);
            unlock();
            asm("wfi");
        }
    }
```

La fonction kernel_main a été modifiée afin d'être exécutée correctement par les différents cœurs du système.

Chaque cœur est identifié grâce à la fonction get_core_id(), ce qui permet de différencier leur rôle lors de l'initialisation.

Le cœur 0 est responsable de l'initialisation globale du système, notamment de l'UART et du contrôleur d'interruptions via enable_interrupt_controller(), afin de permettre l'utilisation du timer générique sur l'ensemble des cœurs. Les adresses liées au timer (« TIMER_INT_CTRL_x ») ont été ajustées pour prendre en compte l'offset de la MMU.

Le démarrage des autres cœurs est ensuite déclenché par l'appel à start_cores().

Tous les cœurs procèdent ensuite à l'initialisation des timers, des interruptions et de la table des vecteurs d'IRQ. Ils entrent enfin dans une boucle principale où l'accès au scheduler est synchronisé à l'aide de verrous, avant de se placer en attente lorsque nécessaire.

Modification sched.c

```
static struct task_struct init_task_0 = INIT_TASK;
static struct task_struct init_task_1 = INIT_TASK;
static struct task_struct init_task_2 = INIT_TASK;
static struct task_struct init_task_3 = INIT_TASK;
struct task_struct *currents[NB_CPU] = { &(init_task_0),
&(init_task_1), &(init_task_2), &(init_task_3)};
```

```
struct task_struct *task[NR_TASKS] = {
    &(init_task_0),
    &(init_task_1),
    &(init_task_2),
    &(init_task_3),
};

void schedule(unsigned char core_id) {
    currents[core_id]->counter = 0;
    _schedule(core_id);
}
void preempt_disable(unsigned char core_id) {
    currents[core_id]->preempt_count++;
}
```

Dans un premier temps, nous avons redéfini current en un tableau currents[NB_CPU] afin de gérer une tâche courante distincte pour chaque cœur.On peut ainsi identifier le cœur avec Core_Id et après modifier les paramètres de la tache propre a ce cœur.

Nous avons également introduit une tâche initiale par cœur, utilisée comme tâche d'attente lorsque le cœur n'a aucune autre tâche à exécuter

```
void _schedule(unsigned char core_id) {
    printf("Core %d : SCHEDULE -- Disabling preemt of task :
%d with counter : %d\n",core_id, currents[core_id],
currents[core_id]->counter);
    preempt_disable(core_id);
    printf("Core %d : Arrived in _schedule\n",core_id);
    int next, c;
    struct task_struct *p;
    c = 0;
    next = core_id;
    for (int i = NB_CPU; i < NR_TASKS; i++) {
        p = task[i];
        if (p && p->state == TASK_RUNNING && p->counter > c
&& !p->taken) {
```

```
            c = p->counter;
            next = i;
        }
    }
    if (c == 0) {
      for (int i = 0; i < NR_TASKS; i++) {
          p = task[i];
          if (p && !p->taken) {
              p->counter = (p->counter >> 1) + p->priority;
          }
      }
    }
    else {
      task[next]->taken = 1;
    }
    switch_to(task[next]);
    core_id = get_core_id();
    printf("Core %d : SCHEDULE -- Enabling preemt of task :
%d with counter : %d\n",core_id, currents[core_id],
currents[core_id]->counter);
    preempt_enable(core_id);
}
```

La fonction de scheduling a également été modifiée. La boucle while a été retirée afin d'éviter qu'un cœur monopolise le scheduler. Le core_id est utilisé pour modifier la préemption de la tâche courante correspondant au bon cœur.

Lors de la sélection des tâches, les tâches initiales propres à chaque cœur sont ignorées, et une condition supplémentaire a été rajouté afin de s'assurer qu'une tâche n'est pas déjà prise par un autre cœur.

Dans le cas où une tâche est sélectionnée, elle est marquée comme prise pour les autres cœurs grâce au champ taken = 1, qui est relâché dans switch_to au moment du changement de tâche.

Enfin, après le changement de tâche, le core_id est recalculé afin de ne pas modifier la préemption de la mauvaise tâche courante.

spinlock.S

```
// spinlock.S
.globl lock
lock:
    mrs x9, mpidr_el1
    and x9, x9, #0xFF           // x9 = core_id
    adrp x10, mutex
    add x10, x10, #:lo12:mutex

    // Check si déjà propriétaire
    ldrb w11, [x10]
    cmp w11, w9
    b.eq 1f

    // Spinloop
spinloop:
    ldxrb w11, [x10]
    cmp w11, #0xFF
    b.ne spinloop // Check si mutex libre
    stxrb w11, w9, [x10]
    cbnz w11, spinloop // Check si exclusivité valide
    dmb sy

1:  ret

.globl unlock
unlock:
    mrs x9, mpidr_el1
    and x9, x9, #0xFF
    adrp x10, mutex
    add x10, x10, #:lo12:mutex

    ldrb w11, [x10]
    cmp w11, w9
```

```
    b.ne 1f

    dmb sy
    mov w11, #0xFF
    strb w11, [x10]

1:  ret

.section ".data"
.globl mutex
mutex :
  .byte 0xFF
```

La fonction *lock* permet de load la *mutex* puis store l'identifiant du cœur à l'intérieur.

Pour faire une *mutex*, on utilise le load exclusif (`ldxrb`) et le store exclusif (`stxrb`).

Le load exclusif enregistre l'adresse physique en tant qu'accès exclusif au cœur qui a load.

Le store exclusif accède à une adresse et écrit dessus **uniquement** si l'adresse est enregistrée en tant qu'accès exclusif du cœur qui a store.

Grâce à l'exclusivité, si deux cœurs entrent en section critique, le premier échouera si il n'a pas fini de store avant que le second ne load. Et le second attendra si le premier à store avant la fin du load.

Contrairement à un CAS, un load/store exclusif peut échouer à la place d'attendre que l'adresse soit accessible. Il faut donc boucler jusqu'à ce que le load/store exclusif passe.

La première boucle est là pour s'assurer que la *mutex* est libre.

La seconde boucle est là pour vérifier si le store a été interrompu par un load exclusif d'un autre cœur.

Le code permet également de définir *mutex* en tant que variable globale. Cela permet que chaque cœurs accèdent à la même *mutex*. Ce système de lock/unlock va être utilisé au moment d'entrer dans le kernel (*kernel_entry*) afin de s'assurer qu'un seul cœur exécute du code kernel.

La valeur de la *mutex* dépend du cœur qui la possède pour ne pas avoir de problème si on appelle lock alors que l'on possède déjà la *mutex*.

```
struct task_struct *currents[NB_CPU] = { &(init_task_0),
&(init_task_1), &(init_task_2), &(init_task_3)};

void schedule(unsigned char core_id) {
    currents[core_id]->counter = 0;
    _schedule(core_id);
}
void preempt_disable(unsigned char core_id) {
    currents[core_id]->preempt_count++;
}
```

Nous avons transformé la variable current en un tableau *currents* qui contient le tick actuel de chacun des cœurs.

Nous avons transformé la variable current en un tableau *currents* qui permet de suivre la tache en cours pour chaque cœur .

Nous avons ainsi adapté le reste du code pour qu'il fonctionne avec ce tableau.

1.2 Test de la V1

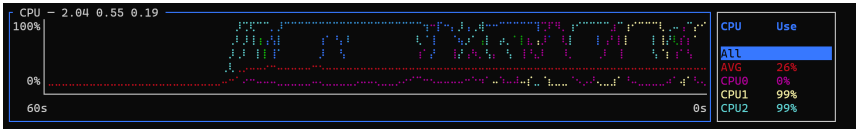
```
void create_and_loop(char* str, char* arg)
{
  char buf[25] = {""};
```

```
  for (int i = 0; i < 25; i++){
    buf[i] = str[i];
  }
  call_sys_write(buf);
  int pid = call_sys_fork();
  if (pid < 0) {
    call_sys_write("Error during fork\n\r");
    call_sys_exit();
    return;
  }
  if (pid == 0)
    loop(arg);

  call_sys_write("Process over\n\r");
  call_sys_exit();
}

void user_process()
{
  call_sys_write("User process\n\r");
  // First child
  call_sys_write("Creating process 1\n\r");
  int pid = call_sys_fork();
  if (pid < 0) {
    call_sys_write("Error during fork\n\r");
    call_sys_exit();
    return;
  }
  if (pid == 0)
    create_and_loop("Child 1 creating
child\n\r","azazazaz");

  // Second child ..
  // Third child..
  // Four child ..
  }
```



Pour valider le bon fonctionnement de l'implémentation multi-cœur, nous avons utilisé deux méthodes de test .

La première repose sur l'outil Bottom, qui permet d'observer l'activité de chaque cœur du processeur. Cet outil nous a permis de vérifier que les cœurs passent correctement en attente via l'instruction wfi (100% → 0% d'utilisation) lorsqu'ils n'ont aucune tâche à exécuter, et qu'ils se réveillent correctement lorsqu'une nouvelle tâche est planifiée.

La seconde méthode consiste en une modification du programme utilisateur (user.c). Dans cette version, le cœur 0 crée plusieurs processus, tandis que les autres cœurs prennent en charge ces processus et en créent eux-mêmes de nouveaux. Chaque processus exécute ensuite une boucle affichant une chaîne de caractères différente.

Cette approche permet de démontrer que n'importe quel cœur est capable de créer un processus sans provoquer de conflits, et que les données partagées du noyau ne sont pas corrompues. Cela inclut notamment la sortie UART ainsi que la liste globale des tâches.

2 Multicoeur via plusieurs spinlocks sur les structures du kernel

Désormais, nous allons retirer le biglock sur le kernel pour le remplacer par des locks plus précis sur les ressources critiques. Les ressources en question sont les fonctions :

- le scheduling dans schedule
- la création de process dans fork
- les écritures via printf
- les allocation mémoires via get_free_page()

Pour que l'accès à ces ressources critiques ne pose pas de problème, nous ajoutons un spinlock devant chacune d'entre elles, sauf pour le fork qu'on peut réaliser avec une opération atomique.

2.1 Pas à pas

2.1.1 Scheduling

On lock avant de schedule dans **sched.c**.

```
lock_sched();
    for (int i = NB_CPU; i < NR_TASKS; i++) {
        p = task[i];
        if (p && p->state == TASK_RUNNING && p->counter > c
&& !p->taken) {
            c = p->counter;
            next = i;
        }
    }
```

On unlock une fois que le changement de contexte à eu lieu.

```
cpu_switch_to(prev, next);
unlock_sched();
core_id = get_core_id();
```

2.1.2 Fork

Seul le nombre de tâches est une ressource critique car chaque cœur modifiera la liste à des indices différent. Nous pouvons donc gérer cette ressource critique via une simple addition atomique.

```
int pid = atomic_add(&nr_tasks, 1);
```

2.1.3 Write

On lock du début à la fin du printf pour s'assurer qu'il n'y ai pas d'interférence sur l'UART.

```
void tfp_printf(char *fmt, ...)
    {
    lock_write();
    va_list va;
    va_start(va,fmt);
    tfp_format(stdout_putp,stdout_putf,fmt,va);
    va_end(va);
    unlock_write();
    }
```

2.1.4 Memory

On lock dans la fonction get_free_page() pour éviter que deux cœurs allouent une page au même moment.

```
unsigned long get_free_page() {
  lock_mem();
  for (int i = 0; i < PAGING_PAGES; i++) {
    if (mem_map[i] == 0) {
      mem_map[i] = 1;
      unsigned long page = LOW_MEMORY + i * PAGE_SIZE;
      memzero(page + VA_START, PAGE_SIZE);
      unlock_mem();
      return page;
    }
  }
  unlock_mem();
  return 0;
}
```

Nous pouvons maintenant créer plusieurs tâches en même temps car deux cœurs peuvent entrer dans le kernel simultanément.

2.2 Optimisation avec des runqueues

En se documentant sur ce qui existait en OS multicœur, en particulier sur la manière dont Linux fait du scheduling, nous avons appris qu'il était possible de scheduler sans section critique en utilisant des runqueues. Chaque cœur a maintenant sa propre liste de tâches ce qui permet de scheduler sans section critique.

Cela peut poser un problème d'équilibrage de charge : une runqueue peut contenir 4 millions de tâches alors que les 3 autres sont vides. Pour palier à cela, il est possible de faire du rééquilibrage, où un cœur compte régulièrement le nombre de tâches de chaque cœur et les déplace si besoin. Une solution alternative ou supplémentaire est de permettre la migration de tâches, où un cœur qui n'a plus de tâches à exécuter va piocher dans les runqueues des autres. Ces deux solutions au rééquilibrage réintroduisent des section critiques mais qui s'exécutent beaucoup plus rarement qu'à l'origine.

Pour aller plus loin, nous avons essayé d'ajouter ce mécanisme de runqueues et potentiellement d'équilibrage. Cela n'a pas pu être entièrement débogué à temps et nous n'allons donc pas fournir de guide d'implémentation pas à pas dysfonctionnel, mais cependant présenter les modifications réalisées.

2.2.1 Création de plusieurs runqueues (une par CPU).

Linux utilise des listes chaînées pour leur runqueues (listes de tâches), qui prennent donc la structure suivante :

```
struct runqueue {
    struct task_struct *current;
    struct task_struct *idle;
    struct task_struct *task_list; // chained list of
runnables tasks
};
```

Pourquoi une liste chaînée et pas un tableau global comme avant ?

- plusieurs CPUs liraient/écriraient dedans
- contention énorme
- migration difficile
- impossible de savoir “à qui appartient” une tâche

Nous devons aussi changer la structure des tâches :

```
struct task_struct {
    struct cpu_context cpu_context;
    long state;
    long counter;
    long priority;
    long preempt_count;
    unsigned long flags;
    struct mm_struct mm;

    unsigned char cpu;
    struct task_struct *next;
};
```

Nous n’avons plus besoin du champ *taken*, le champ *cpu* indique à quel cœur appartient cette tâche, et le champ *next* permet de faire une liste chaînée.

Maintenant que chaque cœur à sa liste de tâche, nous allons toujours scheduler sur soi-même, on peut donc se

passer de passer id_core en paramètre de *schedule()*, *preempt_disable()*, *preempt_enable()*.

On change INIT_TASK pour correspondre à la nouvelle structure : on met *cpu* à 0 et *next* à NULL.

Nous changeons *fork.c* et *mm.c* pour fonctionner avec la nouvelle structure des tâches.

Les tests sur ce premier niveau d'implémentation qui devrait nous permettre de scheduler sans spinlock montre que les tâches utilisateur ne semble pas être créées ce qui ne permet donc pas de vérifier si plusieurs tâches peuvent être schedulées en même temps.

Dû à des limites de temps, nous ne pouvons pas continuer à déboguer ces fonctionnalités mais cette implémentation se trouve dans la branche develop du dépôt git.

3 Les différentes alternatives

Au lieu de se servir du WFI nous aurions pu utiliser un Wait For Event (WFE) mais celui ci ne marche pas dans QEMU.

Au lieu de se servir d'un load exclusif et d'un store exclusif pour l'accès en section critique, il est possible d'utiliser l'instruction CAS *Compare And Swap* (opération atomique). Cependant, cela n'était pas possible dans notre cas car nous sommes avec un processeur ARMv8.0 et que l'instruction CAS n'existe que depuis ARMv8.1.

4 Sources

https://www.minix3.org/doc/

https://pdos.csail.mit.edu/6.S081/2023/xv6/book-riscv-rev3.pdf

https://docs.kernel.org/

https://deepdives.medium.com/kernel-locking-deep-dive-into-spinlocks-part-1-bcdc46ee8df6

https://github.com/bztsrc/raspi3-tutorial/tree/master/02_multicorec

https://lwn.net/Articles/615809/

https://www.youtube.com/watch?v=sofh0gXzwlg

https://drive.uqu.edu.sa/_/mskhayat/files/MySubjects/2017SS%20Operating%20Systems/Abraham%20Silberschatz-Operating%20System%20Concepts%20(9th,2012_12).pdf

https://www.rpi4os.com/part10-multicore/#:~:text=You%E2%80%99ll%20notice%20that%20we%E2%80%99ve%20set,ld

https://s-matyukevich.github.io/raspberry-pi-os/docs/lesson03/linux/interrupt_controllers.html

https://linux-kernel-labs.github.io/refs/heads/master/lectures/smp.html