Télécom Paris

Rapport du stage ingénieur
mai–octobre 2020

Implémentation de l'aspect multi-cœurs de la norme ARINC653

Romain Guilloteau

encadré par
Samuel Tardieu
Laurent Pautet

Enseignant référent
Tarik Graba

24 octobre 2020

Résumé

Dans le cadre de mes études d'ingénieur, j'ai réalisé mon stage dans le laboratoire Systèmes Embarqués Critiques Autonomes (ACES) de Télécom Paris. J'ai travaillé sur le développement de POK, un système d'exploitation temps réel pour l'avionique implémentant la norme ARINC653. L'objectif de ce stage était de découvrir comment fonctionne ce système d'exploitation et plus généralement un système temps réel. Mais il s'agissait également de découvrir la norme ARINC653 et les contraintes que cela ajoute au développement du système. De manière plus spécifique, je me suis d'abord attelé à la mise en place de tests et la détection de bugs dans le système. Puis je me suis chargé de l'extension de POK à des architectures x86 multiprocesseurs. Enfin, j'ai travaillé plus en profondeur sur les exemples liés à la norme ARINC653. Je me suis occupé de les débuguer et de les faire fonctionner. Mais, cela m'a aussi permis de m'approprier cette norme puis de vérifier et corriger la conformité de POK et des exemples avec cette dernière.

Table des matières

Chapitre 1

Introduction

1.1 Cadre du stage

1.1.1 Présentation du laboratoire

J'ai réalisé mon stage dans l'équipe ACES (*Autonomous Critical Embedded Systems*) du département INFRES de Télécom Paris. Télécom Paris est une école d'ingénieurs dont le domaine principal est le numérique où j'ai passé les 3 années de mon cursus ingénieur. Elle est organisée en plusieurs départements de recherche dont le département INFRES (Informatique et Réseau). Ce département couvre tous les domaines de l'informatique en étudiant aussi bien les systèmes, les réseaux et les infrastructures mais également les modélisations mathématiques qui y sont liées, comme la cryptographie ou la théorie de l'information. C'est au sein de ce département que se trouve l'équipe de recherche ACES [1]. Les thèmes de recherche de cette équipe sont axés autour des systèmes concurrents pour lesquels des propriétés non fonctionnelles telles que la performance ou la sécurité doivent être garanties. De plus, leurs thèmes sont répartis entre les systèmes faiblement couplés ou fortement couplés. Dans les systèmes faiblement couplés, on retrouve la sécurité de l'internet des objets ou encore les services distribués. Dans les services fortement couplés, on retrouve les problématiques de sécurité et de sûreté mais aussi les systèmes temps réel et la conception de systèmes critiques dont POK fait partie.

1.1.2 Contexte et objectifs

POK est un système d'exploitation temps réel basé sur un micro-noyau pour des systèmes embarqués temps réel critiques et en particulier pour l'avionique grâce à son implémentation de la norme ARINC653 spécialement conçue pour cette problématique. Ce système d'exploitation a été développé à l'origine par Julien Delange dans le cadre de sa thèse [2]. Ce stage s'inscrit dans une volonté de la part d'ACES de reprendre le développement de POK et cela sur plusieurs points. Tout d'abord, POK n'était tout simplement plus fonctionnel. En effet, le projet original s'appuyait beaucoup sur AADL et en particulier sur Ocarina pour générer le code utilisateur. Or, cette génération de code ne fonctionne plus correctement avec la dernière version d'Ocarina. Par ailleurs, il y a une volonté de rendre POK compatible avec RAMSES [3] pour la génération de code. De plus, il n'y avait pas de tests unitaires pour vérifier les fonctionnalités de base du noyau. Enfin, POK n'est compatible qu'avec des systèmes de moins en moins utilisés comme les processeurs monocœurs sur architecture x86. Aujourd'hui, la majorité des processeurs de cette famille sont multicœurs et en 64-bits.

Ainsi l'un des premiers objectifs a été de pouvoir créer des applications pour POK indépendamment d'Ocarina et notamment des tests. Ces tests nous ont permis de trouver de nombreux bugs sur différentes parties du programme. Puis, j'ai laissé de côté la recherche de bugs pour me concentrer sur l'implémentation du multiprocessing pour architecture x86. Enfin, je me suis concentré sur les fonctionnalités liées à ARINC653 ainsi que sur le respect de la norme.

1.1.3 Organisation du stage

Ce stage s'est déroulé dans un contexte particulier lié à la pandémie de Covid-19. J'ai réalisé une grande partie de mon stage en télétravail chez mes parents dans le Calvados. J'étais donc loin de Télécom Paris et il m'était impossible de voir mes encadrants pour discuter en face-à-face. Cependant, cela ne nous a pas empêchés de travailler et communiquer efficacement durant le stage. Tout d'abord, nous avions une conversation Télégram avec Samuel Tardieu qui a été le moyen privilégié pour moi de poser des questions mais aussi de discuter avec lui des différents bugs rencontrés et de la manière de les résoudre. Nous avons aussi beaucoup utilisé les systèmes d'*issues* et de *merge requests* offerts par GitLab. Nous ouvrions une *issue* pour chaque problème ou chaque fonctionnalité à ajouter. Chaque ajout passait par une *merge request* sur laquelle nous discutions beaucoup du code écrit et notamment des améliorations possibles dans mes ajouts. Enfin, nous organisions une réunion à distance chaque semaine avec Laurent Pautet pendant laquelle nous discutions des avancées que nous avions réalisées et des problèmes encore à résoudre.

Début septembre, je suis revenu sur Paris pour travailler directement à Télécom Paris. Je commençais à perdre ma concentration et ma motivation suite à la difficulté croissante à bien séparer le travail et les loisirs puisque mon bureau situé dans ma chambre était à la fois le lieu où je travaillais mais aussi le lieu où je me divertissais. De plus, ma famille qui était en vacances avait un rythme parfois différent du mien et avait du mal à se dire que même si j'étais dans ma chambre, j'étais techniquement au travail et qu'il fallait éviter de me solliciter durant ces horaires.

1.2 Spécificités techniques de POK

1.2.1 Un système d'exploitation temps réel

POK est un système d'exploitation basé sur un noyau comme pourrait l'être une distribution Linux. Cependant, il y a un point sur lequel ils diffèrent : l'aspect temps réel. En effet, ce paradigme se distingue par le respect de contraintes temporelles, à la différence du *best-effort* que l'on retrouve traditionnellement dans nos ordinateurs personnels. La spécificité de ce paradigme est l'ajout d'une contrainte liée au temps dans l'exécution des différentes applications. Chaque tâche a une période d'exécution, une échéance qu'il faut éviter de dépasser et un temps d'exécution, qui sera toujours calculé comme étant le pire temps d'exécution (*Worst Execution Case Time* ou WCET).

Les contraintes du temps réel nécessitent de nombreuses fonctionnalités complémentaires qu'on ne retrouve pas dans un noyau grand public. Il faut notamment ajouter des systèmes de contrôle qui vérifient qu'une tâche n'a pas dépassé son échéance et si c'est le cas, réparer cette erreur (cela peut être tout simplement le fait de relancer la tâche par exemple).

1.2.2 Les spécificités de la norme ARINC653

Dans le cadre de l'avionique, les différents systèmes sont régis par différentes contraintes en fonction de leur niveau de criticité. Ces niveaux sont présentés dans la norme DO-178 [4]. Et parmi les contraintes imposées, il y a un nombre maximal d'erreurs par heure comme le montre le tableau 1.1.

Niveau de criticité	Pourcentage des fonctions dans cette catégorie	Conséquence	Occurrence maximum des erreurs
E	5%	Aucune	N/A
D	10%	Mineure	$10^{-3}/h$
C	20%	Majeure	$10^{-5}/h$
B	30%	Dangereuse	$10^{-7}/h$
A	35%	Catastrophique	$10^{-9}/h$

TABLE 1.1 – Catégories de criticité de la norme DO-178 [5]

Dans une première approche, on pourrait penser à ne mettre qu'une seule fonction par calculateur afin de ne pas mélanger différents niveaux de criticité sur une même machine. C'est ce qu'on appelle l'architecture fédérée [5]. Cependant, cette solution est peu portable, énergivore et prend beaucoup de place. Ainsi, il existe une seconde approche qui est celle de l'architecture intégrée : plusieurs fonctions sur le même calculateur. Cette solution permet de prendre moins de place, a une consommation plus faible et est facilement portable [5]. Pour autant, si plusieurs fonctions avec des niveaux de criticité différents s'exécutent sur la même machine, il faut s'assurer que les fonctions moins critiques n'impactent pas les fonctions plus critiques en cas d'erreur. C'est justement là qu'intervient la norme ARINC653.

L'objectif de cette norme est d'assurer la sûreté d'une architecture intégrée. Les différentes applications sont situées au niveau au-dessus du noyau. Ce noyau de son côté doit fournir des fonctionnalités d'isolements spatial et temporel afin d'empêcher les erreurs de se propager d'une application à une autre [5]. Les différentes applications ont toutes leur propre partition qui a ses propres zone mémoire et plage temporelle d'exécution. L'autre avantage d'un noyau ARINC653 est de pouvoir s'affranchir du matériel pour les développeurs des applications et ainsi de pouvoir augmenter leur portabilité. En plus de ces isolements spatial et temporel des partitions, la norme ARINC653, à travers son API nommée APEX, permet à ces dernières de créer des processus légers et d'utiliser des objets de synchronisation comme les sémaphores, les verrous et les événements. Enfin, elle permet aussi de réaliser des communications intra-partition avec les *blackboards* et les *buffers* mais aussi des communications inter-partitions avec notamment le *port queueing* et le *port sampling*. Cependant, une partition utilise cette API dans un cadre prédéfini. En effet, de nombreux paramètres sont fixés lors de la compilation à travers un fichier de configuration, pour POK ce sont les fichiers `config.yaml` des différentes applications qui remplissent ce rôle. Ce fichier de configuration contient entre autres le nombre de partitions, leur taille, leur fenêtre d'exécution, les fonctionnalités qu'elles peuvent utiliser ou encore le nombre de processus et d'objets de synchronisation et de communication par partition.

Chapitre 2

Découverte de POK et premières tâches

2.1 Prise en main de POK

Au début du stage, j'ai commencé par prendre POK en main en lisant la documentation générée mais aussi le code. Cependant ce dernier était assez complexe à assimiler puisque participer au développement d'un noyau était quelque chose de nouveau pour moi. De plus, le côté succinct des commentaires a rendu la prise en main difficile. D'ailleurs, avec les problèmes de génération de code liés à Ocarina, nous ne pouvions pas générer de code qui compilait correctement et pouvait être exécuté. Samuel Tardieu a donc proposé de tout remettre à plat pour le processus de développement. Nous avons redéfini les règles de style et mis en place le formatage du code afin d'harmoniser le style sur tout le projet, mais nous avons surtout décidé de reprendre à zéro la création et le développement d'applications et surtout des tests, ce qui était l'un des objectifs du stage.

2.2 Création des tests et applications

Tout d'abord j'ai dû comprendre comment doit être réalisé l'exécutable d'une application. Chaque exécutable est en fait conçu en plusieurs étapes. Chaque partition est compilée en un `.elf` et le noyau, de son côté, est compilé mais l'édition de liens n'est pas faite. Ensuite les fichiers binaires de chaque partition sont regroupés dans un fichier `partition.bin` puis copiés dans la section `.archive2` de l'exécutable lors de l'édition de lien du noyau, comme le montre l'image 2.1. Cela donne un exécutable avec le format décrit par la figure 2.2.

2.2.1 Alternative à Ocarina

Comme expliqué dans l'introduction, nous avions un problème de génération de code avec Ocarina. Ce dernier est censé se baser sur un modèle AADL du programme souhaité afin de générer les fichiers nécessaires à la création et la compilation de l'application. Ce modèle AADL contient les informations sur les partitions comme leur taille ou le nombre de processus et pour chaque processus des informations comme le nom de la fonction à appeler lors de son exécution.

Or lorsque nous avons testé les différents exemples, dans le meilleur des cas, le code était généré mais ne compilait pas correctement et dans le pire, il n'était même pas généré. Pour la création des tests, nous avons d'abord dû trouver une alternative à Ocarina qui génère différents fichiers :

— le code source de chaque partition ;

— les fichiers de déploiement de chaque partition et du noyau : les fichiers `deployment.h`. Ces fichiers contiennent toutes les macros nécessaires au code source qui dépendent de la configuration ;

— les `makefiles` de chaque partition, du noyau et de l'exécutable global.

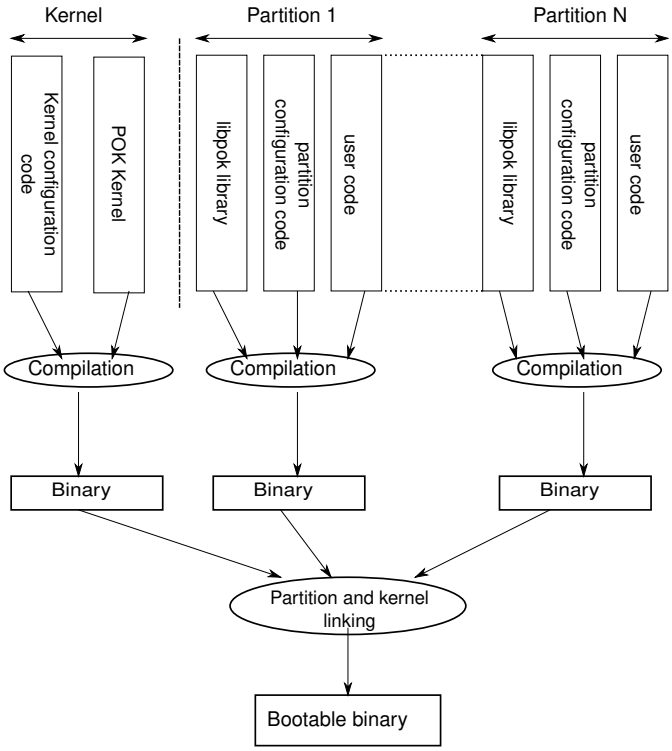

FIGURE 2.1 – Étapes de création de l'exécutable

FIGURE 2.2 – Format de l'exécutable final

Pour le code source, nous avons décidé de l'écrire à la main car il ne demandait pas une grande complexité. De plus, pour certains exemples, Ocarina permettait de générer le code même si ce dernier ne compilait pas. Cela donnait donc une base d'inspiration pour les différents tests. Pour les `makefiles` ainsi que les `deployment.h`, un script Python a été développé dans le but de les générer à partir d'un fichier de configuration au format YAML.

2.2.2 Écriture des tests

Une fois que nous pouvions créer différentes applications, nous avons commencé à réfléchir à comment écrire les tests. Nous avons décidé de créer des tests unitaires pour chaque fonctionnalité de la `libpok` (sémaphores, création de threads, etc). Pour cela nous avons utilisé le *framework* `avocado` qui permet de réaliser plusieurs tests à partir d'un script python et d'un fichier YAML. Pour chaque test, `avocado` va compiler le test, l'exécuter sur QEMU (un émulateur) puis comparer ce qui est renvoyé sur la sortie standard avec un fichier de référence contenant la sortie attendue, c'est le fichier `expected.txt`. Pour rendre les tests les plus pertinents possibles, nous avons ajouté de nombreux messages de debug, notamment lors de l'ordonnancement.

Cependant cette technique a ses défauts notamment lors de l'ordonnancement. En effet, en fonction des machines exécutant QEMU, différents timings peuvent être légèrement modifiés. Par exemple, si deux processus `P1` et `P2` affichent tous les deux un message puis s'endorment pendant un certain laps de temps, il est possible que pour certaines instances de QEMU, `P1` termine sa phase de sommeil avant que `P2` ne

commence la sienne et que pour d'autres instances, `P2` commence sa phase de sommeil avant que `P1` ne termine la sienne. Les deux scénarios sont corrects, cependant lorsque `P2` commence sa phase de sommeil avant que `P1` ne termine la sienne, l'ordonnanceur ne passera pas directement de `P2` à `P1` mais passera d'abord par le processus inactif , appelé *idle thread* en attendant que `P1` termine sa phase de sommeil, le processus inactif étant un processus qui exécute l'instruction `hlt` en boucle. Ce passage par le processus inactif étant affiché sur la sortie standard, les deux scénarios, bien que tous deux corrects, donnent des sorties différentes, ce qui lève une erreur avec `avocado`. De plus, cela est aussi très compliqué pour les tests en multiprocessing car les différents appels à `printf` ne sont pas systématiquement dans le même ordre bien que l'exécution reste correcte.

Cependant, on ne peut pas ignorer ces différences puisque parfois ce n'est pas dû à un problème de timing mais bien à un bug ne se manifestant que sur une seule machine.

2.3 Premiers tests

Malgré les problèmes avec Ocarina mentionnés plus haut, il était possible de générer le code de certains exemples puis de le modifier pour qu'il compile. C'est donc ce que j'ai fait au début et cela a permis de relever de nombreux bugs et implémentations incomplètes. Dans ce rapport, je vais parler de certaines erreurs et implémentations marquantes qui sont fortement liées au développement d'un noyau sur processeur x86.

2.3.1 *Opcode* non valide

J'ai commencé par l'exemple avec les sémaphores qui nous a permis de découvrir de nombreux bugs. Le tout premier était une erreur fatale levée par le processeur : un *Invalid Opcode*, une instruction assembleur inconnue. Cette erreur venait de l'instruction `movq 0x4(%esp),%xmm0` qui sert à déplacer un mot de 64 bits depuis le registre `xmm0`. Cette dernière n'est présente que sur les architectures possédant les extensions SSE (*Streaming SIMD Extensions*). Ces extensions, comme leur nom l'indique, permettent de réaliser du *Single Instruction Multiple Data*, soit d'exécuter une même instruction sur plusieurs données en simultané. Ici, cela revient à réaliser l'instruction `mov` sur 2 mots de 32 bits.

Nous avons donc essayé de comprendre pourquoi cette instruction n'était pas reconnue. La première hypothèse retenue était que les processeurs simulés par QEMU n'étaient pas compatibles. Cependant l'exécution du test à l'aide d'un processeur x64, qui est compatible SSE, donnait la même erreur. Finalement, il s'est avéré que ces extensions sont bien présentes sur les deux processeurs émulés par QEMU, cependant elles sont désactivées par défaut sur les processeurs Intel, il faut donc vérifier que le processeur est compatible et dans ce cas, activer le SSE [6]. Nous avons donc ajouté la portion de code nécessaire dans les routines des démarrages :

```
1       mov %cr0, %eax
2       andb $0xFB, %al
3       mov %eax, %cr0
4       mov %cr4, %eax
5       orw $(3 << 9), %ax
6       mov %eax, %cr4
```

Ces problématiques de rétro-compatibilité de l'architecture x86 ont souvent été rencontrées et notamment dans la partie liée au multiprocessing.

2.3.2 Un `printf` incomplet

Suite au réglage du bug précédent, l'exemple des sémaphores s'exécutait normalement. En voulant vérifier les timings de l'exemple, j'ai remarqué que les données affichées par la fonction `printf` étaient

erronées. Afin de déterminer si ce problème était lié à une mauvaise gestion des temps ou à la fonction `printf`, j'ai réalisé la fonction ci-dessous.

```
void* pinger_job () {
   pok_time_t new = 0x4f591e979f61321b;
   printf("Begin of task\n");
   while (1) {
      printf("%d %d %d %d %d\n",2, 2, new, 4, 2);
      pok_thread_sleep (5000);
   }
}
```

L'exécution de cette fonction donne le résultat suivant :

```
Begin of task
2 2 9f61321b 4f591e97 4
2 2 9f61321b 4f591e97 4
```

On remarque que la variable `new` est affichée en 2 parties avec les 32 bits de poids faible en premier puis les 32 bits de poids fort, ce qui provoque également un décalage des entiers suivants dans l'affichage. En regardant le code de `printf`, je note que seules les données codées en au plus 32 bits étaient prises en compte. En effet, lors d'un appel à `printf`, les différentes variables à insérer dans la chaîne de caractères globale sont stockées dans une `va_list` puisque ni le nombre d'arguments ni leur type ne sont fixés. Ainsi, c'est en fonction des indicateurs de conversion (`%d`, `%s`, *etc.*) présents dans la chaîne de caractères que l'application détermine le type de l'argument à afficher à cet endroit. Dans la fonction `printf` de POK, le caractère de conversion `%f` était lié au type `double` et tous les autres étaient par défaut liés à un entier de 32 bits :

```
switch (*(format + 1)) {
    case 'f':
      arg.vdouble = va_arg (args, double);
      break;

    default:
      arg.value = va_arg (args, uint32_t);
      break;
}
```

L'appel à `va_arg` va enlever le premier élément de la `va_list` (ici appelée `args`) en fonction du type passé en argument. Ainsi pour un entier de 64 bits, les 32 bits de poids faible seront extraits puis les 32 de poids fort. C'est pour cela qu'un entier de 64 bits est affiché en 2 fois et décale l'affichage des paramètres suivants.

Suite à cela, j'ai modifié la fonction `printf` afin de permettre un affichage des nombres en 64 bits. La gestion des formats a également été mise à jour afin de refuser les indicateurs de conversion non pris en charge mais également de prendre en compte les modificateurs de longueur comme `l` pour les variables de type `long int` et `h` pour celles de type `short`. Cette nouvelle implémentation n'utilise que des variables de type `long long int` pour stocker les données à afficher afin de conserver une version fonctionnelle sur les architectures x64 qui contiennent des entiers de type `long long int` sur 128 bits. Cette solution bien que simple n'est pas optimale d'un point de vue de la mémoire car même un caractère encodé sur 8 bits sera stocké dans au moins 64 bits. Cependant, les caractères étant stockés sur la pile temporairement, cela ne pose pas de problème de gestion de la mémoire.

2.3.3 Le débordement de pile

C'est lors de cette modification de `printf` que je suis tombé sur le bug le plus difficile à trouver et à comprendre. Le programme de test des sémaphores contient 3 partitions et la troisième contient un seul thread qui doit juste éteindre QEMU. Cependant, suite à la modification de `printf`, la partition 3 n'arrêtait plus le programme et semblait être bloquée dans une boucle infinie. Or cette partition se contente normalement d'afficher seulement un message avant d'arrêter QEMU. Le message `[P3] Will shutdown` s'affichait correctement puis le processeur bouclait à l'infini dans la fonction `__udivdi3`. Cette fonction correspond à l'implémentation de la division pour les entiers codés sur 64 bits. Cependant, il n'y a aucune raison que le processeur exécute une division à ce moment-là du programme. Il est donc très étrange qu'il soit situé dans cette zone. En regardant le code de la fonction `__udivdi3` lors de l'exécution, j'ai remarqué que cette zone a été réécrite et pour être précis, la fonction précédente aussi. En effet, j'obtiens cette portion d'assembleur en regardant la zone autour de `__udivdi3` :

```
1 0x10349f <memset+30>    add    %dl,(%rdx)
2 0x1034a1 <memset+32>    add    %al,(%rax)
3 0x1034a3 <memset+34>    add    %cl,(%rdx)
4 0x1034a5 <__udivdi3>    add    %al,(%rax)
5 0x1034a7 <__udivdi3+2>  add    %dl,-0x73(%rbx)
6 0x1034aa <__udivdi3+5>  movsb  %ds:(%rsi),%es:(%rdi)
```

Et cela au lieu de :

```
1 0x10349f <memset+30>    jne    103497
2 0x1034a1 <memset+32>    mov    %ebx,%eax
3 0x1034a3 <memset+34>    pop    %ebx
4 0x1034a4 <memset+35>    ret
5 0x1034a5 <__udivdi3>    push   %ebp
6 0x1034a6 <__udivdi3+1>  push   %edi
7 0x1034a7 <__udivdi3+2>  push   %esi
8 0x1034a8 <__udivdi3+3>  push   %ebx
9 0x1034a9 <__udivdi3+4>  lea    -0x102c(%esp),%esp
```

On voit donc que l'instruction `ret` a notamment disparu. Ainsi la présence du processeur dans la fonction `__udivdi3` est due à un appel de la fonction `memset` qui est positionnée juste avant dans l'espace mémoire.

QEMU possède une console graphique qui affiche aussi la sortie standard. Or lors de l'appel à `printf` par la partition 3, la chaîne de caractères contient le caractère `\n`, ce qui va conduire à la création d'une nouvelle ligne sur l'interface graphique à l'aide d'un appel à `memset`. Et c'est cet appel à `memset` qui va conduire à la boucle infinie dans la fonction `__udivdi3`.

Lors de l'exécution du programme, j'ai remarqué que l'instruction `ret` à la fin de `memset` était réécrite lors de la gestion d'une interruption *timer* pendant l'exécution de la partition 3. La nouvelle valeur écrite étant `00dead00h`. Cette valeur correspond à l'un des deux arguments passés à un thread lors de sa première exécution et sont poussés sur la pile par la fonction `pok_dispatch_space`. On note également que lors de la première exécution du thread de la partition 3, la pile est déjà au mauvais endroit, le problème est donc antérieur à l'exécution de ce thread.

En regardant plus en détail les zones de code réécrites, nous avons remarqué que lors du dernier appel système de la partition 2, la pile est bien placée mais cela n'est plus le cas lors du premier appel système de la partition 3. Le problème se situe donc entre les deux et nous avons donc commencé à penser que le problème venait du changement de contexte. Entre ces deux événements, il y a 2 changements de contexte qui se produisent : le premier lors du dernier appel système de la partition 2 et le deuxième lors du début de la fenêtre de la partition 3.

Le premier changement de contexte est dû à ce dernier appel système de la partition 2 et qui va

endormir le dernier thread de cette partition s'exécutant encore. Cela va provoquer un changement de contexte vers l'*idle thread*, ce qui est attendu. Le deuxième changement de contexte se produit lors d'une interruption *timer*. Le gestionnaire de l'interruption va détecter une fin de *quantum*. Cette fin de *quantum* arrive en même temps que le début de la fenêtre d'exécution de la partition 3. Ainsi il y a un changement de contexte vers le thread principal de la partition 3 qui s'exécute pour la première fois.

Son exécution va commencer par un appel à `pok_dispatch_space`. Et je remarque, comme pressenti au-dessus, que cette fonction écrase bien le code avec la valeur `00dead00h`, valeur qui correspond aussi à un des paramètres de la fonction. Or c'est dans cette fonction que la pile est placée grâce à un autre paramètre : `kernel_sp`. En regardant sa valeur, on remarque qu'elle pointe vers une zone de code et non de pile. Or cette valeur est initialisée lors de la création du thread et à ce moment-là, elle est correcte. La variable `kernel_sp` fait partie d'une structure de données qui est placée sur la pile (jusque là encore vide) du nouveau thread. En surveillant cette structure de données, nous nous sommes rendu compte qu'elle était réécrite lors d'un appel à `printf` par l'*idle thread*.

Lors du démarrage, POK va créer les différentes partitions ainsi que leur thread principal puis, il va créer l'*idle thread*. Ainsi l'*idle thread* est créé juste après le thread principal de la partition 3. Les piles étant allouées sur le tas, leurs piles respectives se retrouvent adjacentes. Cependant, la pile de l'idle thread ne mesure que 128 octets, or avec des variables de 64 bits poussées sur la pile par la nouvelle version de printf, il est très facile pour celle de l'idle thread de déborder et donc de réécrire la structure d'initialisation du thread principal de la partition 3. Ainsi lors de sa première exécution, le thread principal de la partition 3 va placer son pointeur de pile vers une zone qui ne correspond pas à sa pile réelle. Pour régler ce problème, nous avons donc passé toutes les piles à la taille par défaut qui est de 8192 octets.

2.4 Découverte de l'ordonnanceur et du partitionnement pour x86

Au cours du réglage des bugs précédents mais aussi pour les points à venir, j'ai dû comprendre comment fonctionnent l'ordonnanceur et le partitionnement qui sont également des points très importants dans POK et dans la norme ARINC653.

2.4.1 Le partitionnement

Comme expliqué dans la sous-section 1.2.2, il y a un partitionnement spatial et temporel. Pour le partitionnement temporel, chaque partition a une fenêtre d'exécution et en fonction du temps actuel, c'est l'ordonnanceur qui va passer d'une partition à une autre. Pour le partitionnement spatial, POK utilise une *Global Descriptor Table* pour l'architecture x86 car cela permet de segmenter la mémoire.

Global Descriptor Table (GDT)

La GDT est utilisée par les processeurs x86 pour définir différents segments mémoires. On peut définir plusieurs segments de plusieurs types notamment de codes ou de données mais également des segments systèmes. Chaque segment contient plusieurs paramètres comme le montre le tableau 2.1.

Dans le cadre de POK, les segments de code seront toujours activés en lecture, ceux de données en écriture. Leur direction/conformité sera de 0 pour tous.

Ainsi l'intérêt pour POK est de pouvoir créer plusieurs segments afin de séparer les partitions mais aussi le noyau. Ce qui donne le format suivant pour la GDT de POK :

Entrée 0 **:** Pas utilisée.

Entrée 1 **:** Segment « code du noyau ».

Entrée 2 **:** Segment « données du noyau ».

Entrée 3 **:** Le *Task State Segment*.

Entrée $4 + 2n$ **:** Segment « code de la partition n ».

Entrée $5 + 2n$ **:** Segment « données de la partition n ».

Nom	Bits	Description
Limite 0 :15	0 :15	16 bits de poids faible de la taille du segment.
Base 0 :23	16 :39	24 bits de poids faible de l'adresse la base du segment.
Bit d'accès	40	Mis à 1 par le processeur lorsqu'il y accède.
Bit lecture/écriture	41	Active la lecture pour les segments de code. Active l'écriture pour les segments de données.
Bit de direction Bit de conformité	42	0 : Le segment croit vers le haut, ne peut être exécuté que depuis le niveau de privilège privl. 1 : Le segment croit vers le bas, ne peut être exécuté que depuis un niveau de privilège de numéro supérieur à privl.
Bit d'exécution	43	Vaut 1 si le segment peut être exécuté (segment de code), 0 sinon.
Type	44	Mis à 1 pour les segments de codes et de données et à 0 pour les segments systèmes.
Privilège (privl)	45 :46	Le niveau de privilège de 0, le plus élevé, à 3 le plus faible.
Bit présent	47	Mis à 1 si le segment est disponible, 0 sinon.
Limite 16 :19	48 :51	Les bits 16 à 19 de la taille du segment.
Bit de taille	54	0 pour le mode 16-bits, 1 pour le 32-bits.
Bit de granularité	55	0 : la limite est en octets, 1 : la limite est en pages de 4kio.
Base 24 :31	56 :63	Bits 24 à 31 de l'adresse de la base du segment.

TABLE 2.1 – Paramètres d'une entrée de la Global Descriptor Table [7]

Une fois cette table créée, il faut spécifier son adresse au processeur grâce à l'instruction `lgdt`. Ensuite, le processeur utilise ses registres de segment pour déterminer quelle entrée de la table regarder. Pour cela, on stocke dans ces registres un sélecteur qui permet de connaître le numéro de l'entrée (noté `entry`), si c'est une entrée de la GDT ou l'*Interruption Description Table* (IDT) (noté `table` et il vaut 0 pour la GDT et 1 pour l'IDT) et le niveau de privilège souhaité (noté `rpl`). La valeur du sélecteur vaut donc `(entry<<3)|(table<<2) | rpl` [8]. Ainsi pour accéder au segment de code de la partition 1 avec le niveau de privilège 3, il faut enregistrer la valeur `(6<<3)|(0<<2)|3=0b110011=33h` dans le registre *Code Segment*.

Les segments de données et de code du noyau couvrent l'intégralité de la mémoire. Cela permet notamment au noyau de pouvoir accéder aux données des partitions. De plus, ces derniers ont un niveau de privilège de 0, soit le plus élevé. En revanche, les segments des partitions ne couvrent que leur zone mémoire respective et ont un niveau de privilège de 3 (le plus bas), ce qui est normal puisque les partitions exécutent du code utilisateur.

De plus, nous avons aussi le *Task State Segment* (TSS) qui permet de sauvegarder l'adresse de la pile à utiliser lors d'un appel système ou d'une interruption. En effet, lorsque l'un de ces 2 événements se produit, le processeur va passer du niveau de privilège 3 vers le niveau privilège 0 afin d'exécuter du code du noyau. Pour cela, il doit, au préalable, changer de pile en positionnant son pointeur à l'adresse contenue dans le champ `ESP0` du TSS, `ESP0` étant la pile utilisée pour tout passage au niveau de privilège 0. Ainsi chaque thread doit mettre à jour la valeur de `ESP0` afin que sa propre pile noyau soit utilisée lors du prochain appel système ou lors de la prochaine interruption. Dans POK, ce changement est réalisé par la fonction `update_tss`.

Pour que le processeur sache où se trouve le TSS il faut lui spécifier la valeur du sélecteur associé à l'entrée du segment dans la GDT grâce à l'instruction `ltr`.

2.5 L'ordonnancement

Lors de l'appel à l'ordonnanceur, POK se situe au niveau noyau 0 car cela fait suite à un appel système ou à une interruption du *timer*. Ce passage provoque aussi la création d'un cadre d'interruption permettant de sauvegarder plusieurs informations et notamment les valeurs du pointeur d'instruction et des registres de segment.

Lors de cet appel, le noyau va réaliser 3 tâches :

1. Calculer quelle est la fenêtre d'exécution courante et à quelle partition elle correspond.
2. Sélectionner le thread à exécuter parmi ceux de cette partition.
3. Réaliser un changement de contexte afin d'exécuter le nouveau thread.

L'étape 1 est assez simple et consiste à vérifier si la fenêtre de la partition courante est terminée. Si c'est le cas, l'ordonnanceur choisit la partition suivante.

L'étape 2 est déjà un peu plus complexe. Tout d'abord, le résultat change en fonction de l'état de la partition. En effet, si la partition est dans un des deux états d'initialisation, le seul thread pouvant s'exécuter est son thread principal (celui d'indice le plus bas pour la partition). Au contraire, si elle se situe dans le mode normal, tous les autres threads sauf le principal peuvent s'exécuter. Le thread principal ne sert qu'à l'initialisation, comme spécifié par la norme ARINC653. Pour choisir parmi tous les autres threads possibles, en comptant l'*idle thread*, on exécute la politique d'ordonnancement correspondant à celle choisie lors de la configuration de la partition.

Enfin, l'étape 3 qui est la plus complexe et qui se divise en 2 sous-étapes :

1. Si la nouvelle partition est différente de la précédente, l'ordonnanceur va désactiver les segments de données et de code de la précédente puis activer ceux de la nouvelle. Pour cela il passe à 0 le bit de présence pour les anciens segments et le passe à 1 pour les nouveaux. C'est cette possibilité d'activer et de désactiver les segments qui permet de sécuriser le partitionnement.
2. Ensuite il va réaliser un changement de contexte entre l'ancien et le nouveau thread.

Pour l'architecture x86, le changement de contexte se fait grâce à la fonction suivante :

```
1  void pok_context_switch(uint32_t *old_sp, uint32_t new_sp);
2  asm(".global pok_context_switch \n"
3      "pok_context_switch:        \n"
4      "pushf                      \n"
5      "pushl %cs                  \n"
6      "pushl $1f                  \n"
7      "pusha                      \n"
8      "movl 48(%esp), %ebx        \n"
9      "movl %esp, (%ebx)          \n"
10     "movl 52(%esp), %esp        \n"
11     "popa                       \n"
12     "iret                       \n"
13     "1:                         \n"
14     "ret");
```

Cette fonction va mettre sur la pile les *flags* du processeur, son *Code Segment* (CS), l'adresse de l'instruction `ret` puis les registres d'usage général avec l'instruction `pusha`. Ensuite le pointeur de pile est changé et pointe sur la pile du nouveau thread. Les valeurs sauvegardées des registres d'usage général du nouveau thread sont récupérées puis l'instruction `iret` est appelée. Cette instruction va réaliser un *far jump*, un saut avec un CS spécifique. Elle récupère sur la pile l'instruction sur laquelle sauter, puis le CS et enfin les *flags* afin de les restaurer [9].

La prochaine fois que le thread précédent sera ordonnancé, les instructions `popa` et `iret` permettront de rétablir son état grâce aux données présentes sur sa pile. Ensuite le retour au gestionnaire d'interruption

permet, grâce au cadre d'interruption, de restituer les valeurs du pointeur d'instruction mais aussi des registres de segment afin de reprendre l'exécution, au niveau utilisateur, là où elle avait été laissée.

Cependant cela n'est possible que si ces valeurs et notamment le cadre d'interruption existent. Pour les threads exécutés pour la première fois, une structure de données a été placée sur leur pile lors de leur création. Elle contient les éléments pour que l'appel à `iret` renvoie vers une fonction du noyau qui va créer un cadre d'interruption nécessaire. Suite à cela, le nouveau thread remonte au niveau utilisateur et commence à s'exécuter. Dans POK, l'initialisation du thread ainsi que son passage au niveau utilisateur est effectué par la fonction `pok_dispatch_space`. De plus, avant de retourner au niveau utilisateur, le thread met à jour le champ `ESP0` du TSS à l'aide de la fonction `update_tss`.

Chapitre 3

Prise en charge du multiprocessing

Après deux mois de prise en main de POK, de création des tests et de réglage des bugs existants, j'ai commencé à travailler sur la partie principale de mon stage, le passage de POK en multiprocesseur. Pour être précis, je ne l'ai fait que pour l'architecture x86, POK étant également compatible PPC et SPARC.

3.1 Architecture

Pour la suite du rapport, nous appellerons processeur un processeur logique.

L'architecture x86 est conçue pour être rétro-compatible, ainsi n'importe quel CPU peut exécuter du code ayant été compilé pour un autre CPU plus vieux. Or les premiers CPUs étaient monoprocesseurs et en 16 bits. Ainsi par souci de compatibilité, tous les CPUs x86 démarrent dans cette configuration.

De plus, sur les anciens CPUs, les interruptions matérielles étaient toutes gérées par une *Programmable Interrupt Controller* (PIC). Cette puce recevait la source de l'interruption et envoyait le numéro d'interruption correspondant au processeur. Dans les architectures multiprocesseurs en revanche, le système contient plusieurs puces appelées des *Advanced PICs* (APIC). Il y a un APIC pour chaque processeur, appelé *Local APIC* (LAPIC) et au moins un pour les interruptions extérieures, appelé I/O APIC. Ainsi, il y a aussi une rétro-compatibilité matérielle pour le contrôle des interruptions.

3.1.1 État au lancement de POK

Lorsque POK commence à son exécution, le BIOS vient de terminer la sienne. Ce dernier réalise plusieurs tâches avant d'appeler le système d'exploitation et passe notamment le processeur principal de 16 bits à 32 bits. Cela signifie que lorsque POK démarre, le système est configuré en mode monoprocesseur et en 32 bits. Ainsi pour pouvoir passer le système en mode multiprocessing, il va falloir passer le mode d'interruption en mode multiprocesseur, puis réveiller les autres processeurs et les configurer afin qu'ils puissent exécuter du code applicatif.

3.2 Connaître le système

On se situe juste après le démarrage de POK, le noyau a seulement configuré sa GDT et ses interruptions. Par convention, le processeur principal, le seul allumé, est appelé le *Bootstrap Processor* (BSP) et les autres sont appelés les *Application Processors* (APs). Avant de pouvoir démarrer les APs et passer le système en mode multiprocesseur, il faut savoir quel est le nombre total de processeurs, le nombre d'I/O APICS, les bus, *etc.* Il existe deux techniques pour cela, une qui est plus lente mais universelle et qui utilise la *MP Configuration Table* (MPCT) et une autre plus récente mais moins universelle consistant à utiliser la *Multiple APIC Description Table* [10]. Je détaillerai la première approche puisque c'est celle que j'ai décidé de mettre en place en raison de son aspect universel.

L'approche utilisée consiste donc à trouver la MPCT et à l'analyser, cette dernière étant créée par le BIOS. En effet, ce dernier ne se contente pas seulement de passer le BSP en 32 bits, il va aussi analyser le système afin de détecter si c'est un système multiprocesseur et enregistrer, dans la MPCT, différentes informations sur les processeurs, les APICs, les bus, *etc.* Enfin il va passer les APs en mode *halted*, l'état arrêté, afin que leur comportement ne soit pas indéterminé [11]. Pour savoir où se trouve la MPCT, il faut chercher la *MP Floating Pointer Structure*.

3.2.1 La MP Floating Pointer Structure

Cette structure peut se situer dans différentes zones :

1. Le premier kilo-octet de l'*Extended BIOS Data Area* (EBDA).
2. Le dernier kilo-octet de la mémoire de base du système si l'EBDA n'existe pas.
3. Au début de la mémoire physique.
4. Dans la section en lecture seule du BIOS comprise entre `0E0000h` et `0FFFFFh`.

Elle peut être trouvée en cherchant dans ces zones la signature de cette structure. Cette signature est composée des 4 caractères `_MP_`. Une fois que l'on a trouvé cette structure, cela signifie que le système est bien compatible avec le multiprocessing. Mais il y a aussi d'autres données importantes :

— l'adresse physique de la MPCT. Ce champ vaut 0 si ce tableau n'existe pas ;

— le premier octet du champ *MP Feature Information* qui permet de savoir si le système actuel correspond à l'une des configurations par défaut. C'est notamment dans ce cas que le champ précédent est nul ;

— le second octet du champ *MP Feature Information*, appelé IMCRP, qui permet de connaître le mode de compatibilité utilisé pour les interruptions en monoprocesseur.

Dans le cas de POK, nous simulons différentes architectures avec QEMU et ces architectures sont toujours explicitement décrites dans la MPCT, je n'ai donc pas traité le cas des configurations par défaut.

3.2.2 *MP Configuration Table* (MPCT)

Une fois que l'adresse de la MPCT est connue, il est possible d'y accéder afin de connaître la configuration du système. Tout d'abord le noyau doit vérifier que l'adresse pointe bien vers la bonne structure grâce à la signature du tableau : `PCMP`. Le tableau est constitué de plusieurs blocs : il y a un entête contenant plusieurs informations importantes puis plusieurs sections pour chaque type d'entrée.

L'entête de la MPCT

Cette entête contient plusieurs données importantes comme l'adresse des LAPICs, souvent `FEE00000h`, et le nombre d'entrées de la table. Il n'y a qu'une seule adresse pour les LAPICs car un processeur ne peut accéder qu'à son propre LAPIC et il lui suffit de faire un accès mémoire à cette adresse pour communiquer avec. Juste après cette entête, il y a les différentes entrées dans un ordre prédéfini que l'on identifie grâce à un code de type :

— 0 pour les processeurs ;

— 1 pour les Bus ;

— 2 pour les I/O APICs ;

— 3 pour les interruptions reliées aux I/O APICs ;

— 4 pour les interruptions reliées aux LAPICs.

Les processeurs

Pour chaque processeur, il y a une entrée de 20 octets dans le tableau permettant de savoir si le processeur est utilisable et quel est le numéro d'identification du LAPIC. Ce dernier paramètre sera utile plus tard pour numéroter les processeurs. Ainsi pour chaque entrée, on vérifie si le processeur est utilisable et dans ce cas, on incrémente un compteur afin de connaître le nombre de processeurs que POK pourra utiliser.

Les Bus

Chaque bus a une entrée de 8 octets. Cette dernière contient un ID qui identifie le bus et une chaîne de caractères permettant de connaître le type de bus.

Les I/O APICs

Pour chaque I/O APIC, il y a une entrée de 8 octets qui contient notamment un ID unique pour chaque I/O APIC, un bit pour savoir s'il est activé et l'adresse de base de l'I/O APIC.

Les interruptions reliées aux I/O APICs

Chaque entrée correspond à une interruption et pour chaque entrée, le bus source ainsi que l'identifiant de l'I/O APIC et la ligne d'interruption correspondante sont donnés.

Les interruptions reliées aux LAPICs

Chaque entrée correspond à une interruption et pour chaque entrée, le bus source ainsi que l'identifiant du LAPIC cible et la ligne d'interruption correspondante sont donnés.

La gestion des interruptions extérieures n'a pas encore été prise en compte. Ainsi je n'utilise que les informations données par les entrées correspondant aux processeurs pour l'instant.

3.3 Configuration du système et réveil des processeurs

Maintenant que le noyau possède toutes les informations du système et notamment des processeurs, il est possible de configurer le système pour réaliser le passage en multiprocesseur, en commençant par les interruptions matérielles.

3.3.1 La gestion des interruptions matérielles

Comme expliqué précédemment, les interruptions matérielles sont gérées par des APICs sur les systèmes multiprocesseurs. Or ce n'est pas le cas sur les systèmes monoprocesseurs qui eux, utilisent seulement une puce PIC. Ainsi il existe, avec les APICs, plusieurs modes permettant de la gestion multiprocessing ou du monoprocessing des interruptions. De plus, il existe sur les systèmes multiprocesseurs, une puce jouant l'équivalent d'un PIC et qui est reliée au BSP de différentes manières en fonction du mode d'interruption.

Le mode PIC

Dans ce mode, l'architecture matérielle est conçue de telle sorte que le LAPIC du BSP soit contourné et que seule l'information issue du PIC soit prise en compte. Cela se fait grâce à l'*Interruption Mode Configuration Register* (IMCR) puisque ce registre pilote un multiplexeur permettant de choisir entre le PIC et le LAPIC comme le montre le schéma 3.1. Lorsque ce registre est à 0, le mode d'interruption est le mode PIC car l'entrée PIC est celle sélectionnée. Si le registre vaut 1, le mode choisi est le *Symmetric I/O Multiprocessing* puisque l'entrée du LAPIC est sélectionnée.

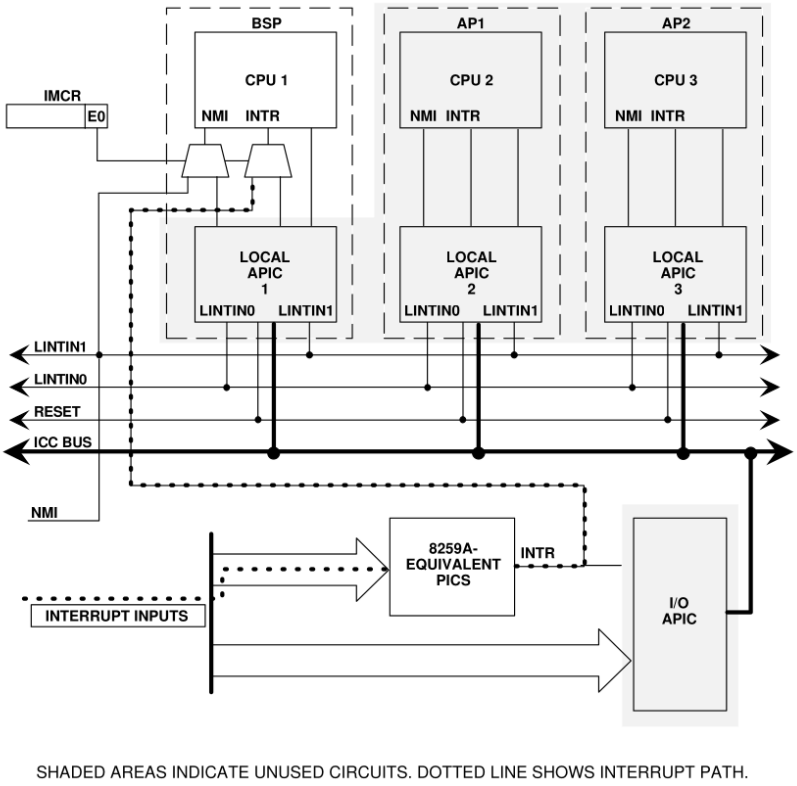

FIGURE 3.1 – Mode PIC [11]

Le mode *Virtual Wire*

Le mode PIC n'est pas le seul permettant à un système multiprocesseur d'être compatible avec le monoprocessing. Il y a aussi le mode *Virtual Wire*, dans ce mode, le PIC est perçu comme un contrôleur d'interruptions extérieures et délivre au LAPIC du BSP une interruption précise. Il existe ensuite deux configurations possibles :

— soit le LAPIC du BSP est en *Virtual Wire*, cela signifie qu'il va directement transmettre les interruptions en entrée au BSP comme sur le schéma 3.2. Dans cette configuration, les I/O APICs sont désactivés ;

— soit le LAPIC est utilisé dans son mode normal et un des I/O APICs est utilisé comme *Virtual Wire* comme sur le schéma 3.3. Pour le LAPIC, cela revient à recevoir une interruption de l'I/O APIC.

Côté noyau, il est facile de savoir quel mode est utilisé entre le PIC ou le *Virtual Wire* grâce au registre IMCRP que l'on retrouve dans la *MP Floating Pointer Structure*. Lorsque ce registre vaut 1, le système est en mode PIC, sinon il est en mode *Virtual Wire*. Par exemple, sur QEMU, le mode utilisé est le *Virtual Wire*, ainsi c'est le seul pris en compte pour l'instant.

Le mode *Symmetric I/O*

C'est le mode pour le multiprocessing. Dans ce mode le chemin d'interruption ne passe plus par le PIC, qui est donc désactivé, et passe par l'I/O APIC puis par un LAPIC avant d'atteindre le BSP ou un des APs. Pour passer dans ce mode, il y a plusieurs étapes à réaliser. Tout d'abord, le noyau doit passer le registre IMCR à 1 si et seulement si ce dernier existe. C'est à dire si et seulement le système est en mode PIC . Ensuite il doit activer le LAPIC du BSP et enfin configurer et activer les I/O APICs. Dans le cadre de mon stage, je n'ai ni configuré ni activé les I/O APICs puisque je ne me suis pas occupé des interruptions extérieures.

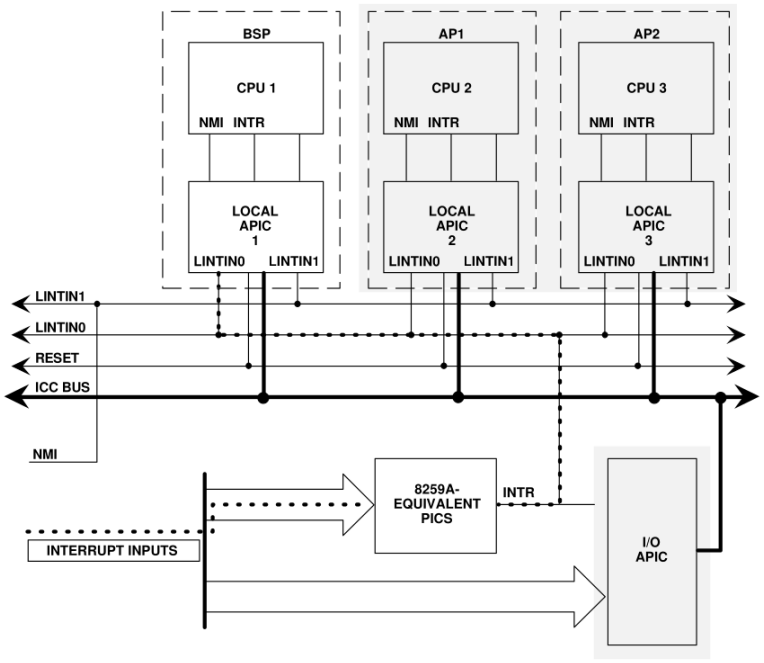

FIGURE 3.2 – Mode *Virtual Wire* via LAPIC [11]

3.3.2 Démarrage des processeurs

Maintenant que le système est en mode multiprocessing au niveau des interruptions, il va être possible de réveiller les autres processeurs. En effet, le passage au mode *Symmetric I/O* permet d'utiliser les interruptions inter-processeurs (IPIs) qui sont, comme leur nom l'indique, des interruptions envoyées depuis un processeur vers d'autres. Ces interruptions vont notamment être utilisées par le BSP afin de réveiller les autres processeurs.

Intel fournit un algorithme universel [11] qui permet de réveiller les autres processeurs grâce aux IPIs. Cependant, le guide d'OSDEV sur le multiprocessing [12] conseille de modifier les temps d'attente entre les différentes interruptions car ils les jugent trop courts. Ainsi l'algorithme 1 correspond à la version conseillée par OSDEV.

Algorithm 1 Algorithme théorique de démarrage des APs

```
BSP sends an INIT IPI to APs
BSP delays (10mSec)
if APIC_VERSION ≠ 82489DX then
    BSP sends a STARTUP IPI to APs
    BSP delays(1mSec)
    BSP sends a STARTUP IPI to APs
    BSP delays(1Sec)
end if
```

Pour simplifier, j'ai supposé que les LAPICs étaient plus récents que le modèle 82489DX, qui est le tout premier APIC et c'est notamment le cas sur QEMU. En effet, la séquence de démarrage pour cette version des LAPICs nécessite beaucoup plus de connaissances dans le fonctionnement du BIOS et la documentation d'Intel pour ce mode est beaucoup plus complexe.

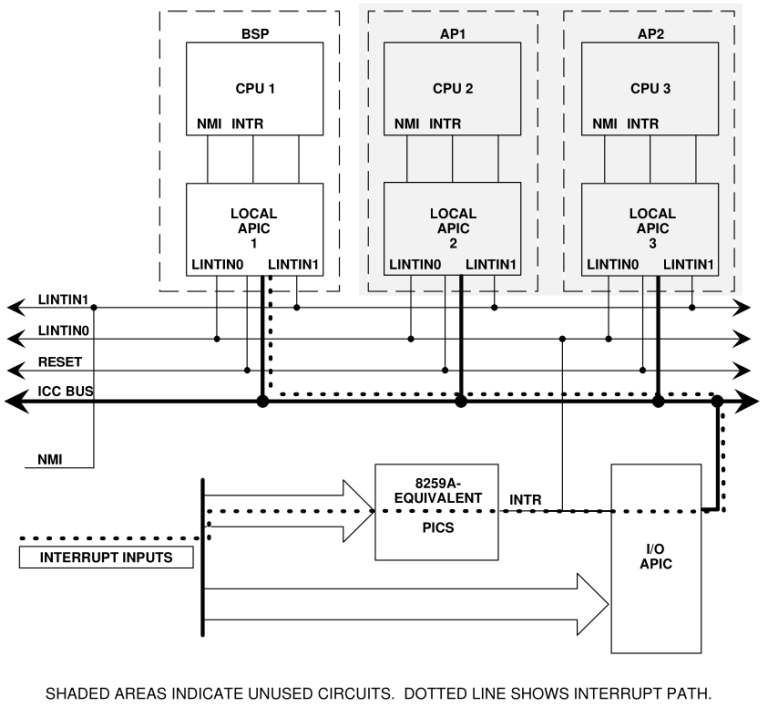

FIGURE 3.3 – Mode *Virtual Wire* via I/O APIC [11]

Fonctionnement des IPIs

Le LAPIC de chaque processeur permet d'envoyer des interruptions à un ensemble de processeurs désigné. Pour envoyer ces interruptions, il suffit d'écrire dans 2 registres du LAPIC, appelés *Interrupt Command Register* (ICR) comme le montre le schéma 3.4 qui détaille les différents champs de l'ICR. Le deuxième registre n'est utile que lorsque la valeur du *Destination Shorthand* est 0, dans les autres cas, le *Destination Shorthand* seul permet de déterminer les processeurs de destination. Dans le cas du démarrage des APs, je n'ai utilisé que les interruptions aux *Delivery Modes* égaux à INIT et STARTUP.

INIT IPI

Pour envoyer cette interruption, il faut en réalité écrire à la suite dans l'ICR les deux messages contenus dans le tableau 3.1, sachant que la valeur du vecteur est ici arbitraire. Le *Destination Shorthand* permet ici au BSP d'envoyer l'interruption à tous les APs. Le fait d'envoyer cette interruption aux autres processeurs va les faire passer dans l'état *Wait-SIPI* pour *Wait STARTUP IPI*.

Destination Shorthand	*Trigger Mode*	*Level*	*Destination Mode*	*Delivery Mode*	*Vector*	Résultat
11	1	1	0	101	11111111	11001100010111111111
11	1	0	0	101	11111111	11001000010111111111

TABLE 3.1 – Messages à envoyer pour générer une INIT IPI

STARTUP IPI

Une fois que les APs sont dans l'état *Wait-SIPI*. Ils doivent recevoir une STARTUP IPI afin de démarrer. D'ailleurs, ils doivent obligatoirement être dans l'état *Wait-SIPI* pour prendre en compte la STARTUP IPI. Normalement, une seule interruption de ce type est nécessaire pour les faire démarrer,

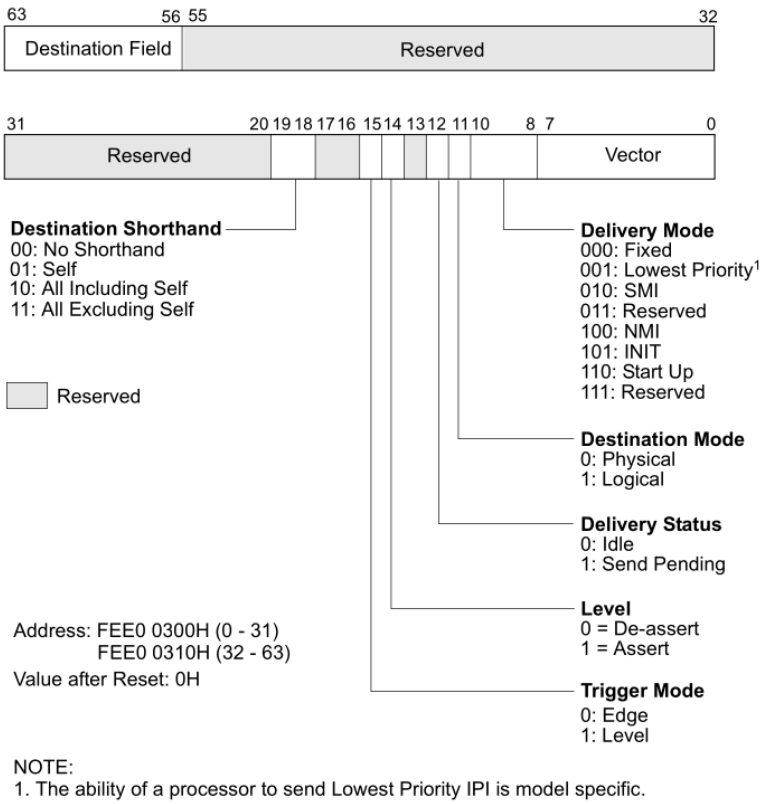

FIGURE 3.4 – *Interrupt Command Register* [13]

mais l'algorithme 1 en contient 2 par sûreté au cas où le premier envoi n'est pas reçu. Le BSP envoie donc 2 fois le même message. Les seuls champs utiles sont *Destination Shorthand* (qui ne change pas par rapport à avant), le *Delivery Mode* (`0b110` pour Start Up) et le vecteur. Ici, le vecteur est très important car lorsque que le processeur va être réveillé, il va copier la valeur du vecteur sur son registre *Code Segment* (CS). Ainsi, si on note cette dernière `VV`, le processeur se réveillera et ira à l'adresse `000VV000h`. Cette valeur a été fixée à `01h` pour des raisons qui seront expliquées en sous-section 3.4.1. Ainsi le BSP écrira deux fois `0b11000100011000000001` dans l'ICR. Ainsi l'implémentation en C de l'algorithme 1 donne le programme suivant :

```
1  void pok_start_ap() {
2    // Send INIT IPI
3    *(uint32_t *)(lapic_address + 0x300) = 0b11001100010111111111;
4    *(uint32_t *)(lapic_address + 0x300) = 0b11001000010111111111;
5    // Wait 10 milliseconds (OSDEV delay)
6    pok_x86_wait_mp(0x2e9b);
7    // Send first STARTUP IPI
8    *(uint32_t *)(lapic_address + 0x300) = 0x000C4600 + (0x01);
9    // Wait 1 millisecond (OSDEV delay)
10   pok_x86_wait_mp(0x4AE);
11   // Send second STARTUP IPI
12   *(uint32_t *)(lapic_address + 0x300) = 0x000C4600 + (0x01);
13   // Wait 1 second (OSDEV delay)
14   for (char i = 0; i < 19; i++)
15     pok_x86_wait_mp(0xffff);
16 }
```

Suite à l'exécution de cette fonction, tous les APs sont sortis de l'état *halted* et commencent à exécuter le code situé en `00001000h` qui va donc les configurer.

3.4 Configuration des processeurs applicatifs (APs)

3.4.1 Situation juste après le réveil

Juste après leur réveil chaque processeur se trouve à l'instruction située à l'adresse `00001000h` en mode 16-bits, appelé *Real Mode*. En effet, ils ne démarrent pas en 32-bits puisqu'ils n'ont pas été configurés par le BIOS. De plus, l'adresse de la première instruction n'a pas été choisie par hasard. Tout d'abord, il fallait une adresse comprise entre `00000000h` et `000FF000h` puisque c'est le vecteur de la STARTUP IPI qui fixe cette adresse, comme vu précédemment. Cependant, on ne peut pas mettre n'importe quelle adresse puisque que dans cette portion de la mémoire, il y a aussi des zones utilisées par le BIOS qu'il ne faut pas réécrire. J'ai décidé, par sécurité, de ne pas aller plus loin que l'adresse `00010000h` puisque l'utilisation de la mémoire au-delà n'est pas standardisée. Ensuite dans la zone précédente, il existe une section libre comprise entre `00007E00h` et `0007FFFFh` [14]. C'est donc pour cela que j'ai décidé de placer la première instruction à `00001000h` afin de garder la plus grande marge possible dans la zone de mémoire libre.

3.4.2 Préparation du code de configuration du 32-bits

Cependant, il y a un problème lié à cette adresse. En effet, l'usage du BIOS veut que, par convention, le noyau à charger se situe en mémoire à partir de l'adresse `01000000h`, ce qui ne comprend ni la section à l'adresse `0001000h` ni une zone mémoire atteignable en 16 bits. J'ai donc décidé de laisser le code avec une adresse physique de stockage (LMA) dans la partie de la mémoire après `01000000h` mais de lui donner une adresse virtuelle (VMA) égale à `0001000h`. Cela se traduit donc par un changement du *ldscript* avec l'ajout du segment `.text.realmode`. Ainsi j'ai également dû écrire un script de copie de la LMA vers la VMA qui est exécuté par le BSP juste avant le réveil des processeurs.

3.4.3 Le code de configuration du 32-bits

Les APs arrivent donc en *Real Mode* sur le code de configuration après leur réveil. Ce dernier correspond au mode 16-bits. La première étape va être de les passer en *Protected Mode*, le mode 32-bits. Pour cela il y a 2 étapes, il faut activer le bit *Protection Enable* du *Control Register 0* et configurer la GDT.

En *Real Mode*, les registres de segment sont ajoutés à un *offset* afin d'obtenir la vraie adresse. Cela permet notamment de pouvoir aller au-delà des 65 536 octets de mémoire que permettent les 16 bits. Par exemple, pour l'adresse d'une instruction, on ajoute le *Code Segment* (CS) à l'*Instruction Pointer* (IP) de la manière suivante : `(CS << 3) | IP`. C'est pour cela que la première instruction du code de configuration se situe en `00001000h` car le vecteur d'interruption du STARTUP IPI était à `01h`, ce qui fixe la valeur du CS à `01h`. La formule est équivalente pour les autres registres de segment. En *Protected Mode*, le processeur utilise la Global Descriptor Table comme expliqué dans la sous-section 2.4.1. Ainsi, j'ai ajouté une GDT par défaut juste pour le passage en 32 bits. Je ne pouvais pas utiliser celle déjà en place pour le BSP, car son adresse est inaccessible en mode 16-bits.

Après ces deux étapes, le processeur réalise un *far jump* vers une nouvelle section de code qui est en 32-bits. Une fois dans cette section, le code pousse le processeur à changer les valeurs de ses registres de segment de données, de pile, *etc.* à `10h` afin de correspondre à l'entrée des données de la GDT. Puis le code active l'extension SSE, comme vu dans la partie 2.3.1. Enfin, l'ID du LAPIC est récupéré afin d'attribuer une pile au processeur à l'adresse `pok_stack`+`LAPIC_ID`×`STACK_SIZE`. Pour cette allocation, j'ai remarqué que la convention du BIOS qui consite à numéroter de façon incrémentale les LAPICs était respectée. Enfin, le processeur réalise un appel à la fonction `main_ap` et quitte ainsi le code de configuration du 32-bits.

3.4.4 Configuration plus précise des processeurs applicatifs

Maintenant que les APs sont sortis du code de passage en 32-bits et peuvent ainsi accéder à toute la mémoire, de la même manière que le processeur principal, il faut les configurer plus en détail. Tout d'abord il faut les numéroter. Pour cela, j'ai créé un tableau qui permet à un processeur de récupérer son numéro en mettant en indice le numéro de son LAPIC. Les processeurs sont numérotés à partir de 0 de manière incrémentale, comme le demande ARINC653, 0 étant toujours le processeur principal. Ensuite, il faut configurer une GDT correcte, pour cela deux choix étaient possibles : une GDT commune ou une GDT unique pour chaque processeur. Au moment de cette implémentation, le postulat était qu'une seule partition allait s'exécuter à la fois, ainsi les segments de données et de codes allaient être identiques en tout temps pour tous les processeurs et la seule chose différente d'un processeur à l'autre serait le TSS, puisque que sa valeur ne dépend que du thread courant sur chacun. J'ai donc décidé de n'utiliser qu'une seule GDT mais avec une entrée TSS pour chaque processeur. On obtient donc une GDT différente de celle en section 2.4.1 et qui est maintenant ordonnée de la façon suivante :

Entrée 0 **:** Pas utilisée.

Entrée 1 **:** Segment "code du noyau".

Entrée 2 **:** Segment "données du noyau".

Entrée 3 **:** Le Task State Segment du processeur 0.

Entrée $3 + \texttt{nb\_proc} - 1$ **:** Le Task State Segment du processeur $\texttt{nb\_proc} - 1$.

Entrée $4 + 2n + \texttt{nb\_proc}$ **:** Segment code de la partition n.

Entrée $5 + 2n + \texttt{nb\_proc}$ **:** Segment données de la partition n.

Comme expliqué précédemment, cela fonctionne dans le cas où une seule partition est exécutée à la fois. Cependant, depuis la dernière mise à jour de la norme ARINC653, plusieurs partitions doivent pouvoir s'exécuter de façon concurrente sur un système multiprocesseur. Ainsi, plusieurs partitions peuvent voir leurs segments activés et un processeur peut donc par erreur passer d'une partition activée à une autre, ce qui limite la sécurité du partitionnement.

Pour les interruptions, j'ai décidé de n'avoir qu'une seule table d'interruptions commune à tous les processeurs, puisque pour l'instant je ne voyais pas de raison pour laquelle certains processeurs devraient avoir un comportement différent des autres sur une même interruption.

Enfin, leur configuration se termine par l'activation de leur LAPIC respectif. Suite à cette activation, les processeurs applicatifs sont prêts à être utilisés.

Chapitre 4

Ordonnanceur multiprocesseur

Pour l'instant, POK est un noyau monoprocesseur. L'objectif est d'adapter les fonctions d'ordonnancement existantes pour le multiprocessing et d'expliquer son implémentation sur l'architecture x86. Il y a plusieurs défis à relever, comme la synchronisation entre les processeurs par exemple, mais aussi l'identification des différents scénarios d'ordonnancement.

4.1 Les sources d'ordonnancement

Dans POK, il est possible de dénombrer 15 sources d'ordonnancement réparties dans 4 catégories :

1. Les sources d'un ordonnancement global : ces sources conduisent à un ordonnancement des partitions et un ordonnancement des threads de chaque processeur.
2. Les sources d'un ordonnancement de tous les processeurs : ces sources conduisent à un ordonnancement des threads de chaque processeur mais pas des partitions.
3. Les sources locales d'un ordonnancement local : ces sources conduisent à un ordonnancement des threads sur le même processeur.
4. Les sources externes d'un ordonnancement local : ces sources conduisent à un ordonnancement des threads sur processeur spécifique.

4.1.1 Les sources d'un ordonnancement global

Cette catégorie contient 2 sources : Le *timer* et la fonction `pok_partition_set_mode`. En effet, la seule raison pour laquelle le noyau doit réaliser un changement de partition est lorsque le *timer* atteint la fin de la fenêtre d'exécution de la partition ou si la partition doit arrêter son exécution (état IDLE). Comme l'ordonnancement des partitions donnera le même résultat pour chaque processeur et pour des raisons de synchronisation, j'ai décidé que seul le processeur qui provoque l'ordonnancement, appelé processeur source, ordonnera les partitions. Il s'agira toujours du processeur 0 pour le *timer*.

4.1.2 Les sources d'un ordonnancement de tous les processeurs seulement

Cette catégorie ne contient qu'une seule source : la fonction `pok_lockobj_eventbroadcast`. Elle diffuse un événement qui peut ainsi débloquer d'autres threads sur plusieurs processeurs. C'est pourquoi cette source conduit à un ordonnancement des threads sur chaque processeur. Cependant, il n'y a pas besoin d'un ordonnancement de partition.

4.1.3 Les sources locales d'un ordonnancement local

Il y a 6 sources dans cette catégorie :

`pok_lockobj_lock :` bloque le thread courant sur un sémaphore ou un verrou ;

`pok_lockobj_eventwait :` bloque le thread courant en attendant qu'un événement soit signalé ;

`pok_sched_stop_self :` arrête le thread courant ;

`pok_sched_end_period :` arrête le thread courant jusqu'au début de sa prochaine période ;

`pok_thread_sleep :` arrête le thread courant pendant une période passée en argument ;

`pok_thread_sleep_until :` arrête le thread courant jusqu'à une date donnée.

Chaque source arrête le thread en cours et le processeur doit donc savoir quel est le prochain thread à exécuter. Cependant, cela n'ayant aucune conséquence sur les autres threads en cours d'exécution, il n'y a pas de raison d'ordonnancer les autres processeurs.

4.1.4 Les sources externes d'un ordonnancement local

Il y a 6 sources également dans cette catégorie :

`pok_thread_resume :` permet au thread passé en argument de reprendre son exécution ;

`pok_thread_set_priority :` change la priorité d'un thread ;

`pok_partition_stop_thread :` arrête le thread passé en argument ;

`pok_partition_restart_thread :` redémarre le thread passé en argument ;

`pok_lockobj_unlock :` libère une ressource et donc le thread en tête de la liste d'attente ;

`pok_lockobj_eventsignal :` libère le thread en tête de liste d'attente sur un événement.

Chaque source d'ordonnancement a une action sur un thread spécifique passé en argument. La seule différence avec la catégorie précédente est qu'il est possible de modifier un autre thread et potentiellement un thread sur un autre processeur. Ainsi, le processeur source doit provoquer un ordonnancement local des threads sur le processeur exécutant le thread ciblé.

4.2 Le nouvel ordonnanceur

L'approche que j'ai choisie est de diviser l'ordonnanceur en 4 fonctions, une pour chaque type de source :

1. POK Global Schedule.
2. POK Global Thread Schedule.
3. POK Local Schedule.
4. POK Send Thread Schedule.

J'ai fourni, plus bas, l'implémentation en pseudo-code pour chaque algorithme. Dans les différents algorithmes, les macros suivantes seront utilisées :

`CURRENT_THREAD :` permet d'obtenir le numéro du thread courant sur le processeur appelant cet macro ;

`PREV_THREAD :` permet d'obtenir le numéro du thread précédent sur le processeur appelant cet macro ;

`CURRENT_PARTITION :` permet d'obtenir le numéro de la partition courante.

Il y a également différentes fonctions de POK qui seront appelées :

`pok_elect_partition :` permet de savoir quelle partition doit s'exécuter ;

`pok_elect_thread :` permet de savoir quel thread doit s'exécuter sur le processeur appelant la fonction ;

`pok_space_switch :` change les segments actifs dans la GDT pour ceux de la nouvelle partition ;

`pok_context_switch :` effectue le passage d'un thread à un autre.

4.2.1 POK Global Schedule

L'algorithme 2 est l'implémentation de l'algorithme d'ordonnancement global et il est divisé en 3 fonctions ayant chacune un rôle spécifique. La première fonction est la fonction « entrée », cette fonction est appelée pour lancer la procédure d'ordonnancement global. Cette fonction ne sera exécutée que par le processeur source et elle calculera la prochaine partition à exécuter. Elle conduit ensuite à l'exécution de la deuxième fonction sur chaque processeur. La deuxième fonction sert à calculer le prochain thread à exécuter sur chaque processeur. Enfin, chaque processeur exécute la troisième fonction dont l'objectif est de réaliser un changement de contexte entre deux threads et deux partitions.

Le changement de partition n'est réalisé que par le processeur source car il affecte la GDT qui est commune à chaque processeur. Le rendez-vous est là pour s'assurer que chaque processeur est dans l'étape d'ordonnancement avant de changer de partition sinon, cela pourrait entraîner des problèmes.

Algorithm 2 POK Global Schedule

```
1: function POK_GLOBAL_SCHEDULE
2:     global previous_partition ← CURRENT_PARTITION
3:     global elected_partition ← pok_elect_partition()
4:     CURRENT_PARTITION ← elected_partition
5:     start_rendezvous()                                  ▷ Start the rendezvous process
6:     send_global_thread_schedule() ▷ Send a message to the other processors to tell them to execute
   global_thread_schedule()
7:     global_thread_schedule()
8: end function
9:
10: function GLOBAL_THREAD_SCHEDULE
11:     elected_thread ← pok_elect_thread()
12:     if CURRENT_THREAD ≠ elected_thread then
13:         if CURRENT_THREAD ≠ IDLE_THREAD then
14:             PREV_THREAD ← CURRENT_THREAD
15:         end if
16:     end if
17:     pok_sched_context_switch(elected_thread)
18: end function
19:
20: function POK_SCHED_CONTEXT_SWITCH(elected_thread)
21:     if processor = source_processor then
22:         spin_wait_for_rendezvous()                      ▷ Waits other processors
23:         pok_space_switch(previous_partition, CURRENT_PARTITION)
24:         unblock_rendezvous()                            ▷ Unblock the other processors
25:         wait_for_leave()                    ▷ Wait other threads leave the rendezvous
26:     else
27:         join_rendezvous()
28:     end if
29:     CURRENT_THREAD ← elected_thread
30:     pok_context_switch(elected_thread)
31: end function
```

4.2.2 POK Global Thread Schedule et POK Thread Schedule

L'algorithme 3 est plus simple que le précédent. Dans ce cas, les différents processeurs sont indépendants dans leur ordonnancement, il n'y a donc pas de rendez-vous cette fois-ci. De plus, la partition reste la même, il n'y a donc pas besoin de la programmer ni de modifier la GDT.

Algorithm 3 POK Global Thread Schedule

```
1: function POK_GLOBAL_THREADS_SCHEDULE
2:     send_thread_schedule()          ▷ Send a message to the other processors to tell them to execute thread_schedule()
3:     thread_schedule()
4: end function
5:
6: function THREAD_SCHEDULE
7:     elected_thread ← pok_elect_thread()
8:     if CURRENT_THREAD ≠ elected_thread then
9:         if CURRENT_THREAD ≠ IDLE_THREAD then
10:            PREV_THREAD ← CURRENT_THREAD
11:        end if
12:    end if
13:    pok_sched_thread_switch(elected_thread)
14: end function
15:
16: function POK_SCHED_THREAD_SWITCH(elected_thread)
17:    CURRENT_THREAD ← elected_thread
18:    pok_context_switch(elected_thread)
19: end function
```

Pour exécuter l'algorithme POK Thread Schedule, il suffit au processeur d'exécuter directement la deuxième fonction de l'algorithme 3

4.2.3 POK Send Thread Schedule

L'algorithme 4 est utilisé lorsqu'une action sur un thread spécifique est exécutée. Ce thread a un paramètre permettant de savoir sur quel processeur il est exécuté, il est donc facile de spécifier quel processeur doit être ordonnancé.

Algorithm 4 POK Send Thread Schedule

```
1: function POK_SEND_LOCAL_SCHEDULE(target_processor)
2:     send_thread_schedule_to(target_processor) ▷ Send a message to the the target processor to tell it to execute thread_schedule()
3: end function
```

4.3 Implémentation pour l'architecture x86

4.3.1 Envoyer des ordres avec les Interruptions Inter-Processeurs

Pour plusieurs algorithmes, un processeur doit envoyer l'ordre d'exécuter une fonction spécifique à d'autres processeurs. Sur l'architecture x86, la voie privilégiée est l'utilisation d'interruptions inter processeurs (IPI). Il est possible d'envoyer une IPI à un groupe, tous sauf soi-même pour POK Global Schedule

par exemple, ou à un processeur spécifique comme pour POK Send Local Schedule. L'IPI contient également le numéro du vecteur d'interruption du gestionnaire d'interruption. J'ai donc implémenté des gestionnaires d'interruptions pour l'ordonnancement des threads. Juste avant de terminer l'ordonnancement, le processeur doit spécifier qu'il a terminé l'exécution de l'IPI avec la fonction `pok_end_ipi`.

Cependant, ce dernier point semble poser quelques problèmes. Lorsqu'un autre thread est exécuté, l'instruction `iret` dans l'implémentation x86 de `pok_context_switch` renvoie directement au niveau du code utilisateur. Dans ce cas, cet `iret` signifie la fin de l'interruption. Nous devons donc exécuter `pok_end_ipi()` avant le changement de contexte. Cependant, cette fonction ne doit pas être exécutée par le processeur qui lance l'ordonnancement mais seulement par les processeurs qui reçoivent une IPI. De plus, le LAPIC ne peut stocker que deux interruptions vers le même vecteur. Les autres sont ignorées.

Mais tout cela n'est finalement pas un problème. En effet, il y a une interruption pour chaque type de fonction d'ordonnancement. Ainsi, si une IPI pour une fonction d'ordonnancement spécifique est reçue avec deux interruptions déjà stockées, ce n'est pas un problème car cela signifie que la fonction d'ordonnancement est en cours d'exécution et sera à nouveau exécutée juste après. Ainsi, les changements induits par la fonction qui a envoyé le troisième appel identique à l'ordonnanceur seront pris en compte lors du traitement de l'interruption en attente.

4.3.2 Le rendez-vous

Pour le rendez-vous, j'ai utilisé un nombre entier `r` pour le créer. Le processeur source (le processeur qui exécute la source de l'ordonnancement) initialise le rendez-vous en mettant `r` à 1 seulement si sa valeur précédente est 0, c'est la fonction `start_rendezvous`. Ensuite, le processeur principal attend les autres processeurs sur la fonction `spin_wait_for_rendezvous`. Cette fonction vérifie que `r` est initialisé, c'est-à-dire sa valeur est strictement positive et attend pendant que $r \neq$ `compteur` avec `compteur` = `multiprocessing_system` et `multiprocessing_system` égal au nombre de processeurs. Les autres processeurs rejoignent le rendez-vous avec la fonction `join_rendezvous`. Cette fonction vérifie que `r` est initialisé, puis incrémente atomiquement `r` et attend ensuite que `r` repasse à 0. Enfin, lorsque le processeur principal a terminé, il peut débloquer les autres processeurs avec la fonction `join_rendezvous` en réinitialisant la valeur de r à 0.

En réalité, j'ai dû utiliser deux rendez-vous car avec un seul, le processeur principal peut recevoir une nouvelle interruption de planification et redémarrer la procédure de rendez-vous alors qu'un autre processeur n'a pas eu le temps de sortir du rendez-vous précédent. Ce problème peut se produire avec des processeurs de vitesse différente et il se produit également avec la QEMU. C'est pour cela que la fonction `wait_for_leave` est présente.

Chapitre 5

ARINC653 et problèmes à résoudre

Suite à la fin du développement de l'aspect multiprocessing, j'ai commencé à regarder tous les exemples que propose POK et à les corriger pour les faire fonctionner. Cela m'a permis de m'attaquer à la bibliothèque d'ARINC653 mais aussi de trouver d'autres problèmes qu'il faudrait résoudre.

5.1 ARINC653

Lors de la mise en place du multiprocessing mais aussi après, le travail que j'effectuais était influencé par la norme ARINC653.

5.1.1 Les spécifications pour le multiprocessing

La norme ARINC653 impose des spécifications supplémentaires aux systèmes multiprocesseurs aussi bien à l'échelle des partitions que des processus.

Spécifications des partitions pour le multiprocessing

Tout d'abord, il faut savoir qu'une partition ne peut pas utiliser les processeurs comme elle en a envie. En effet, l'utilisateur doit spécifier lors de la configuration à quels processeurs une partition donnée peut avoir accès. J'ai donc rajouté cette information dans les fichiers de configuration des applications de POK avec « tous les processeurs disponibles » comme valeur par défaut. Cependant cet ajout pousse les noyaux respectant la norme ARINC653 à devoir respecter un nouveau mode d'exécution, un mode avec plusieurs partitions s'exécutant de manière concurrente. En effet, si par exemple la partition 1 ne peut s'exécuter sur les processeurs 1 et 2 et la partition 2 sur les processeurs 3 et 4, rien n'empêche les deux partitions de pouvoir s'exécuter en parallèle.

Cependant ce cas pose de nombreuses questions, comme la séparation de la mémoire, que je n'ai pas eu le temps de traiter lors de mon stage.

Spécifications des processus pour le multiprocessing

Une fois qu'une partition a la liste des processeurs sur lesquels elle peut s'exécuter, l'utilisateur doit spécifier sur quel processeur un processus doit s'exécuter : c'est le *Processor Affinity*. Cependant, il y a une couche d'abstraction. En effet, du point de vue de la partition, la numérotation des processeurs est continue, ainsi dans le cas de POK, un processeur a 2 identifiants :

— un identifiant noyau qui est unique et correspond au numéro attribué au réveil pour un processeur auxiliaire et à 0 pour le principal ;

— un identifiant partition qui est unique parmi les autres processeurs de la même partition. L'attribution se fait par ordre croissant des numéros noyau. Par exemple si une partition peut utiliser les

processeurs 1, 2 et 4 alors le numéro partition du processeur 1 sera 0, celui de 2 sera 1 et celui de 4 sera 2.

Ces spécifications du multiprocessing d'ARINC653 m'ont poussé à modifier les ordonnanceurs pour que chaque processeur ordonnance les processus qui lui sont liés et seulement ceux-là. Suite à l'implémentation de la norme ARINC653 pour le multiprocessing, je me suis occupé de tester et de vérifier les implémentations des autres points de la norme.

5.1.2 Les objets de synchronisation

Il existe 3 types d'objets de synchronisation avec ARINC653 : les *mutexes*, les sémaphores et les *events*. Les sémaphores et les *mutexes* ne sont pas différents de ceux présents dans POSIX. Cependant dans ARINC653, il n'y a pas de variables conditionnelles mais des *events* qui ont un objectif proche mais un comportement différent.

Les events

Le fonctionnement des *events* est très similaire à celui des variables conditionnelles. Tout d'abord, il est possible d'attendre sur un *event* mais aussi de réaliser un *broadcast* pour libérer les processus bloqués comme pour une variable conditionnelle. Cependant, il existe des différences notables entre les deux comme le fait que l'attente n'est pas nécessairement bloquante. En effet, un *event* a un paramètre d'état qui vaut soit UP soit DOWN et un processus n'est bloqué que si l'état est à DOWN. Cet état peut d'ailleurs être modifié par 2 fonctions :

— la fonction `RESET` qui va passer l'état à DOWN ;

— la fonction `BROADCAST` qui va passer l'état à UP et va ainsi libérer tous les processeurs bloqués. Il n'existe donc pas la possibilité de libérer un seul processus comme le fait la fonction `signal` des variables conditionnelles.

Au moment où j'écris ce rapport, seuls les *events* et les sémaphores fonctionnent. Les mutexes n'ont pas été implémentés pour ARINC653, cependant une version des mutexes de POSIX est présente.

5.1.3 Les objets de communication intra-partition

Avec ARINC653, il existe 2 objets permettant de réaliser une communication intra-partition : les *buffers* et les *blackboards*.

Les *blackboards*

Les *blackboards* sont des objets dont le fonctionnement est assez proche d'un vrai tableau, d'où leur nom. Ces objets ne permettent de stocker qu'un seul message à la fois de taille maximale prédéfinie. Lorsqu'un processus écrit un message dessus, cela écrase le message précédent. En revanche, lorsqu'un processus lit un message, cela ne l'efface pas, il faut pour cela utiliser la fonction `CLEAR_BLACKBOARD`. Chaque accès se fait à l'aide d'un mutex pour contrôler les accès concurrents.

Les *buffers*

Pour les *buffers*, le fonctionnement est différent. Le *buffer* possède une FIFO d'une taille limitée pouvant comporter des messages d'une taille limitée. Chaque écriture dans le *buffer* ajoute un élément à la fin de la FIFO et chaque lecture retire l'élément en tête, ce qui rend pertinent l'utilisation du modèle de synchronisation producteur/consommateur à l'aide de 2 sémaphores.

5.1.4 Les objets de communication inter-partitions

En plus des objets de communication intra-partition, il existe leurs analogues pour la communication inter-partitions. Ce sont les ports avec le *port queueing* et le *port sampling*.

Le *port sampling* et le *port queueing*

Le *port sampling* est l'analogue inter-partitions du *blackboard*. C'est à dire que comme pour le *blackboard*, il n'y a qu'un seul message que l'on écrase et qu'on peut lire autant de fois que souhaité. De son côté le *port queueing* est l'analogue du *buffer* puisqu'il possède lui aussi une FIFO. Cependant malgré leurs points communs avec les objets de communication intra-partition, leurs implémentations sont assez différentes.

Implémentation des ports

Tout d'abord, contrairement aux objets de communication intra-partition, l'implémentation des ports ne peut se faire au niveau de la partition, elle doit se faire obligatoirement au niveau du noyau. En effet, les ports correspondent à un échange de données entre partitions. Or ARINC653 stipule que les partitions sont séparées en mémoire et qu'aucune partition ne peut accéder aux données de l'autre. C'est pour cela que tout doit se faire à travers le noyau qui lui peut interagir avec les zones mémoires des deux partitions. Un port ne peut pas fonctionner tout seul. En effet, il est soit en lecture soit en écriture. Il faut donc toujours un port en écriture et un ou plusieurs ports en lecture qui seront liés au niveau noyau. Ainsi une partition ne peut que lire ou écrire une même donnée, sauf si elle crée un port en lecture un autre port en écriture sur la même donnée. Dans le cas des ports tout est fixé à la configuration, le nom, la direction, la taille etc, mais également les liens entre les ports.

Lorsqu'une partition écrit sur son port, cela va réaliser un appel système qui va copier le message sur une zone mémoire correspondant au port, soit une FIFO pour le *port queueing* soit en écrasant la donnée précédente pour un *port sampling*. De même, lorsqu'une partition va lire depuis son port, le noyau va lui renvoyer la donnée correspondant à la zone mémoire du port, soit la tête de la FIFO pour le port queueing soit l'unique donnée pour un *port sampling*. Mais les zones mémoires de ces deux ports sont bien indépendantes et ne sont pas liées directement. Pour les lier, il y a donc dans le noyau une fonction supplémentaire qui est appelée à la fin de chaque fenêtre globale (*major frame*) : `pok_port_flushall`. Cette fonction va regarder les ports en écriture de toutes les partitions et va copier les données écrites dans leur zone mémoire vers les zones mémoires des ports en lecture qui leur sont liés.

Cependant il y a quelques soucis avec cette implémentation. Tout d'abord elle n'est pas complète. En effet, les messages envoyés sont des objets spécifiques et ne contiennent pas juste de la donnée. Ils ont notamment d'autres paramètres comme une durée de validité ou encore une période d'envoi potentiellement apériodique. De plus, bien que cette implémentation marche très bien dans le cas où une seule partition est exécutée à la fois, il faudrait dans le cas de plusieurs partitions en parallèle pouvoir envoyer les données de manière quasi-instantanée et ne pas attendre que la fenêtre globale d'exécution soit terminée. D'autre part, nous discutions aussi de la pertinence d'attendre la fin d'une fenêtre globale pour échanger les informations plutôt que de le faire à chaque changement de partition. N'ayant pas eu le temps de traiter ce problème en profondeur, cela fait donc partie des problèmes qu'il reste à résoudre.

5.2 Problèmes à résoudre

Parmi les problèmes que je n'ai pas eu le temps de traiter, il y a ces différents aspects avec les ports mais ce ne sont pas pas les seuls.

5.2.1 Exécution concurrente de partitions

Cette problématique sera sûrement l'une des plus lourdes à traiter puisque je n'ai découvert cette possibilité qu'après avoir terminé une bonne partie de l'implémentation du multiprocessing. De plus, cet ajout date de la dernière version d'ARINC653, ainsi POK n'a pas été du tout conçu dans cette optique. L'ajout de cette fonctionnalité se heurtera à plusieurs difficultés. Tout d'abord, le choix de la partition courante. Pour l'instant la partition courante est commune à tous les processeurs, ce qui ne sera plus le cas. Ensuite, comment se déroule la configuration des fenêtres d'exécution ? On peut facilement penser à l'équivalent d'un ordonnancement global hors-ligne mais pour les partitions. De plus, ce qui fait la sécurité du partitionnement, c'est la possibilité de n'activer que les segments de la partition courante dans la *Global Descriptor Table*. Or, si plusieurs partitions s'exécutent en parallèle, il faudra activer plusieurs segments, ce qui diminue grandement l'intérêt de cette technique. En effet, dans ce cas un processeur d'une partition A peut exécuter un processus de la partition B par erreur. L'une des solutions serait donc de séparer la GDT et d'en créer une pour chaque processeur.

5.2.2 Problématiques de temps

Il y a également un problème dans l'implémentation et la gestion du budget et de l'échéance des threads. En effet, l'échéance n'est pour l'instant pas prise en compte et le dépassement de cette dernière n'est donc pas considéré comme un problème. Cela est problématique pour un système temps réel critique.

De plus, le budget de chaque thread est exprimé en *quantum*. Et ce budget est décrémenté pour le thread courant à chaque appel de l'ordonnanceur. Or, suite à plusieurs modifications de ma part sur l'ordonnanceur mais aussi sur les objets de synchronisation, l'ordonnanceur n'est plus seulement appelé à la fin d'un *quantum* mais également dès qu'un thread est bloqué ou qu'une ressource est libérée. Ainsi le budget peut être décrémenté trop rapidement.

5.2.3 Le thread d'erreur

Enfin je n'ai jamais pris en considération l'*error thread* lors de mes différents ajouts, ce thread ayant la possibilité de s'exécuter sur tous les processeurs. De plus en fonction des erreurs, il est toujours possible en multiprocesseur de continuer d'exécuter ou non les autres threads sur les autres processeurs.

Chapitre 6

Conclusion

Pour conclure, j'aimerais tout d'abord remercier Samuel Tardieu et Laurent Pautet de m'avoir proposé ce stage et de m'avoir encadré tout au long du projet. Mais je tiens également à remercier toute l'équipe ACES pour leur chaleureux accueil. Ces 6 mois m'ont permis de mettre un pied dans le monde de la recherche académique. C'est quelque chose que j'ai vraiment apprécié, le fait de chercher et de trouver comment implémenter une nouvelle fonctionnalité comme le multiprocessing, de devoir récupérer des informations à droite à gauche et de les comparer afin d'obtenir un code qui fonctionne, mais aussi le fait de restituer à ses collègues les connaissances acquises. De plus, cela m'a permis de découvrir comment fonctionne le développement d'un long projet et notamment les difficultés qu'il peut y avoir en arrivant en cours de route. Le fait de devoir comprendre un code qui n'est pas le sien m'a fait réaliser qu'il est important pour les autres de bien documenter et commenter son code même si ce dernier nous paraît évident sur le moment. Enfin, ce stage m'a permis d'approfondir les notions vues pendant ma scolarité comme la programmation en *bare-metal* ou encore le fonctionnement du temps réel. En effet, n'ayant que les bases dans ces domaines j'ai vraiment compris que ce n'était pas grand-chose comparé à tout ce que j'ai pu apprendre durant ce stage. Cependant, cela m'a aussi rassuré sur mes propres compétences puisque même si ces bases offertes par Télécom Paris ne font pas de moi un expert en Systèmes Embarqués, elles me permettent de pouvoir comprendre des notions plus complexes et de les approfondir par la suite et c'est finalement cette capacité à pouvoir chercher cette information par moi-même, à l'interpréter et la comprendre qui m'a été offerte par ma formation et que ce stage m'a permis de développer.

De plus, j'ai pu découvrir comment fonctionne un système d'exploitation et aussi l'architecture x86 en détails. Ce sont vraiment deux points que j'ai adoré découvrir et développer malgré le fait que je les connaissais peu et qu'ils me paraissaient obscurs au début. Le développement de POK m'a fait réaliser la complexité de nombreuses fonctions que nous utilisons au quotidien comme le `printf`. Le développement du multiprocessing m'a aussi fait découvrir la complexité de l'architecture x86 mais aussi de toutes les informations nécessaires aux processeurs pour fonctionner correctement. J'ai compris que cette complexité qui semble croissante est un frein pour la compréhension dans un premier temps mais les opportunités qu'elle offre une fois maîtrisée valent le coup. Je pense par exemple à la GDT qui est bien plus complexe que la simple segmentation sur 16 bits mais qui offre plus de possibilités. Et même après avoir passé 6 mois sur cette architecture, je sais que je suis encore loin de la maîtriser. C'est finalement ça que le stage m'a fait surtout réaliser, c'est que peu importe ma carrière en tant d'ingénieur, j'aurai toujours de nouvelles choses à apprendre et qu'il y aura toujours quelque chose de nouveau à faire. D'ailleurs, le développement de POK n'est pas terminé et il reste de nombreuses choses à implémenter comme la possibilité d'exécuter des partitions en parallèle ou encore la complétion de l'API d'ARINC653.

Bibliographie

[1] ACES Team. Autonomous and Critical Embedded Systems. https://aces.wp.imt.fr/.

[2] Julien Delange. *Integration of safety and security requirements for the design of safety-critical systems.* Theses, Télécom ParisTech, July 2010.

[3] MEM4CSD Télécom Paris. Refinement of AADL Models for Synthesis of Embedded Systems (RAMSES). https://mem4csd.telecom-paristech.fr/blog/index.php/ramses/.

[4] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn. Software certification of safety-critical avionic systems : Do-178c and its impacts. *IEEE Aerospace and Electronic Systems Magazine*, 30(4) :4–13, 2015.

[5] Laurent Pautet. Integrated Modular Avionic. https://perso.telecom-paristech.fr/pautet/strec/ima.pdf.

[6] Intel. System Programming Guide. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Chapitre 13.1.

[7] OSDEV. Global Descriptor Table. https://wiki.osdev.org/index.php?title=Global_Descriptor_Table&oldid=25156, 2020. [En ligne ; Page disponible le 21-octobre-2020].

[8] OSDEV. GDT Tutorial. https://wiki.osdev.org/index.php?title=GDT_Tutorial&oldid=24026, 2020. [En ligne ; Page disponible le 21-octobre-2020].

[9] Intel. Instruction Set Reference, A-Z. *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[10] OSDEV. MADT. https://wiki.osdev.org/index.php?title=MADT&oldid=25165, 2020. [En ligne ; Page disponible le 19-octobre-2020].

[11] Intel. MultiProcessor Specification, Mai 1997. v1.4.

[12] OSDEV. Symmetric Multiprocessing. https://wiki.osdev.org/index.php?title=Symmetric_Multiprocessing&oldid=18332, 2020. [En ligne ; Page disponible le 19-octobre-2020].

[13] Intel. System Programming Guide. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Chapitre 10.6.

[14] OSDEV. Memory Map (x86). https://wiki.osdev.org/index.php?title=Memory_Map_(x86)&oldid=24684, 2020. [En ligne ; Page disponible le 20-octobre-2020].

Table des figures

Liste des tableaux