Étude comparative de systèmes d'exploitations

dans un contexte critique et temps-réel

Version : git-3181113

Référence CNES : DLA-SF-0000000-211-QGP
Édition 2 - Révision 0

1980-01-01

Table des matières

1 Introduction

L'usage de logiciels dans les systèmes critiques est de nos jours monnaie courante. Ils se retrouvent dans des systèmes critiques de nombreuses industries comme l'aéronautique, l'automobile et le nucléaire. Ainsi, la sûreté des logiciels devient un enjeu crucial et en particulier celle du système d'exploitation sur lequel tourne les logiciels applicatifs. Le développement et la maintenance d'un système d'exploitation étant complexe et coûteux, il est souhaitable d'utiliser une solution informatique sur étagères[1], c'est-à-dire dans le cas présent un système d'exploitation ayant été conçu pour les systèmes critiques.

Ce document est une étude comparative de systèmes d'exploitation utilisés dans un contexte critique ou temps réel. Nous étudions les systèmes suivants[2]: Linux 6.15.2, MirageOS 4.9.0, PikeOS 5.1.3, ProvenVisor, RTEMS 6.1, seL4 13.0.0, Xen 4.20 et XtratuM.

Les systèmes critiques sont exposés à deux types de menaces:
- *Les défaillances*: elles ne sont pas dues à un agent extérieur. L'ensemble des mesures prises pour y remédier relève de la *sûreté* du système.
- *Les attaques*: elles sont causées par une entité malveillante. L'ensemble des mesures prises pour les contrecarrer font parties de la *sécurité* du système.

L'étude met d'abord l'accent sur l'aspect *sûreté*. Toutefois certains des systèmes étudiés sont dédiés à la *sécurité* et des concepts liés à la sécurité seront donc abordés lorsque nous les examinerons. De plus il existe une certaine porosité entre ces deux concepts car des contre-mesures pour la sécurité s'avèrent pertinents aussi du point de vue de la sureté et vice versa.

Avant de plonger plus avant dans les systèmes étudiés, il est important de cerner davantage le sujet et notamment certaines notions de base dans les sous-sections 1.1, 1.2 et 1.3 ci-dessous.

1.1 Qu'est-ce qu'un système d'exploitation?

La diversité des besoins et des systèmes informatiques existant a conduit à un foisonnement de systèmes d'exploitation et en faire une zoologie complète serait hors sujet. Il est en fait difficile de caractériser rigoureusement ce qu'est un système d'exploitation et nous adoptons ici l'approche retenue dans [1], [2], [3] pour définir ce concept. Nous appelons donc *système d'exploitation*[3] un ensemble de routines gérant les ressources matérielles d'un système informatique et s'exécutant dans un mode privilégié du processeur.

Le système en question peut être un serveur, un ordinateur personnel ou un système embarqué. Le rôle principal du système d'exploitation est de fournir une couche d'abstraction logicielle entre le matériel et les logiciels applicatifs. Il permet ainsi de masquer la complexité et la diversité des interfaces matérielles en fournissant des interfaces stables, unifiées et parfois standardisées.

En pratique, la majorité des systèmes d'exploitation fournissent au moins trois services:
- Un ordonnanceur de tâche qui décide quel programme doit être exécuté à un instant donné.
- Une gestion de la mémoire principale.
- Un protocole de communication entre les programmes en cours d'exécution.

[1] On parle parfois de *COTS* pour *Commercial off-the-shelf*.

[2] Nous nous sommes efforcés de fournir des informations valables pour les versions spécifiées. Notez que les entreprises développant ProvenVisor et XtratuM ne communiquent pas de numéros de version pour leurs systèmes d'exploitation.

[3] En anglais *Operating System*, souvent abrégé *OS*.

1.2 Pourquoi utiliser un système d'exploitation?

Bien que l'usage des systèmes d'exploitation dans les composants critiques se généralise, il n'est pas sans alternative. Une autre approche consiste à exécuter l'application directement sur la couche matérielle. On parle alors de programme *bare-metal*.

Néanmoins, l'adoption d'un système d'exploitation procure des avantages considérables, principalement en facilitant la conception et la portabilité des applications. Le tableau Tableau 1 livre quelques éléments de comparaison entre ces deux approches. Notez cependant que les bénéfices apportés par un OS varient considérablement d'un système à l'autre. Comparer ces apports est l'un des enjeux de cette étude.

Caractéristique	Système d'exploitation	Environnement *bare metal*
Portabilité	Élevée, grâce à des interfaces logicielles et des pilotes.	Faible.
Débogage	Facilité par de nombreux outils, parfois intégrés dans l'OS.	Souvent plus complexe.
Isolation en espace/temps	Fourni par l'OS avec différents niveaux de garantie.	Support absent.
Multi-tâche	Souvent supporté via le concept de processus/thread/partition.	Support absent.
Latence	Induite par l'exécution de routines et les basculement de contextes.	Performance maximale offerte par le matériel.
Certification	Facilité dans le cas où l'OS a fait l'objet d'une certification. Dans le cas contraire la tâche peut-être plus complexe encore.	À refaire de zéro. Toutefois le code a certifié peut être considérablement réduit par l'absence de l'OS.

Tableau 1. – Comparaison OS et *bare metal*.

1.3 Criticité et temps réel

Nous étudions les systèmes d'exploitation dans un contexte critique et temps réel. Donnons une définition brève de ces deux qualificatifs.

Un système est dit *critique* si sa défaillance peut entraîner des conséquences indésirables. Ses défaillances varient considérablement en nature et en gravité:

- Elles peuvent se limiter à la simple perte de données, comme dans le cas d'une base de données bancaire.
- Elles peuvent aller jusqu'à des destructions matérielles, comme celles qui peuvent subvenir dans une centrale nucléaire ou une usine.
- Dans les cas les plus graves, elles peuvent engendrer des pertes humaines, comme dans un accident d'avions ou dans la défaillance d'un système médical comme un pacemaker.

La criticité d'un système est généralement évalué lors de sa conception et le choix d'une solution informatique adaptée en est une étape importante.

Un système informatique est qualifié de *temps réel* s'il est capable de piloter un système physique à une vitesse adaptée à l'évolution de ce dernier. Pour parvenir à ce résultat, les logiciels qu'il embarque doivent être capable de répondre à des stimuli dans un temps imparti. L'enjeu n'est donc par la performance mais le respect d'échéances.

1.4 Organisation de l'étude

L'étude est organisée de la façon suivante:

- Un chapitre est dédié à chaque système d'exploitation. Ce chapitre comprend une description succincte du système et une brève note historique. Puis une section est dédiée à chacun des critères de comparaison énumérés en sous-secton 1.5.
- Une série de tableaux comparatifs qui résument les informations détaillés dans les chapitres précédents et de comparer simplement les systèmes.

1.5 Critères de comparaison

Au travers de cette étude, les systèmes d'exploitation ont été étudiés et comparés suivant les critères suivants:

- Type de système d'exploitation
- Architectures supportées
- Support multi-processeur
- Partitionnement spatial
- Partitionnement temporel
- Déterminisme
- Corruption de la mémoire
- Perte du flux d'exécution
- Écosystème
- Gestion des interruptions
- Support *watchdog*
- Programmation *bare-metal*
- Temps de démarrage
- Maintenabilité

Chaque critère est détaillé dans la sous-section ci-dessous. Il est à noter que certains critères n'étaient pas pertinents pour l'ensemble des systèmes, auquel cas la section correspondante pour ce système justifie son élision.

1.5.1 Type de systèmes d'exploitation

Nous classons les systèmes d'exploitation étudiés en quatre grandes catégories:

- Les *systèmes d'exploitation généralistes* *GPOS (General-Purpose Operating System)* constituent la classe la plus connue du grand public. Ils sont le plus souvent exécutés au-dessus de la couche matérielle et offrent un large éventail de services. Leur domaine d'application est particulièrement vaste puisqu'on les retrouve aussi bien sur les ordinateurs personnels, les smartphones que les serveurs et les systèmes embarqués. Parmi les systèmes les plus connus, on peut citer *Linux*, *FreeBSD*, *NetBSD*, *Windows* et *macOS*.
- Les *hyperviseurs* sont des systèmes de virtualisation qui permettent l'exécution de plusieurs systèmes exploitations sur une même machine. Dans cette étude, nous n'examinerons que des *hyperviseurs* de type 1[4], c'est-à-dire des systèmes d'exploitation dédiés à la virtualisation. Parmi les *hyperviseurs* les plus connus, on peut citer *Xen*, *Oracle VM*, *Hyper-V*, *KVM*.
- Les *systèmes d'exploitation temps-réels* (en anglais *RTOS (Real-time Operating System)*) sont des systèmes dédiés au temps réel. Ils offrent de fortes garanties en matière de déterminisme grâce à des ordonnanceurs de tâches et des protocoles de synchronisation spécifiques.

[4] On parle également d'*hyperviseur bare-metal*.

- Les *bibliothèques d'OS* (*LibOS* pour *Library Operating System*) ne sont pas à proprement parler des systèmes d'exploitation mais plutôt des collections de bibliothèques permettant d'exécuter des logiciels sans avoir recours à un *GPOS*. Le développeur lie les modules indispensables à son programme, afin de produire une image appelée un *unikernel*. Celui-ci peut ensuite être exécuté sur un *hyperviseur* ou en *bare-metal*.

Cette nomenclature n'exclut pas qu'un système d'exploitation soit dans plusieurs catégories simultanément. Nous verrons par exemple que de nombreux *hyperviseurs* dédiés à l'embarqué critique proposent nativement des fonctionnalités temps réel. De même beaucoup de *GPOS* intègrent un *hyperviseur* de type 1.

1.5.2 Architectures supportées

Pour chacun des systèmes d'exploitation étudiés, nous donnons une liste des différentes architectures supportées. Afin que cet effort soit tenable, nous avons sélectionné les architectures avec les critères suivants:

- L'architecture doit être utilisée dans de véritables systèmes critiques,
- L'architecture doit être supportée nativement, c'est-à-dire que le système d'exploitation doit pourvoir s'exécuter sur une telle architecture sans avoir recours à un mécanisme d'émulation,
- Certains systèmes ont une longue histoire rendant une documentation exhaustive en pratique très difficile. Nous nous bornons à un sous-ensemble des architectures et renvoyons le lecteur à la documentation officielle pour les architectures plus exotiques,

Avec ces critères à l'esprit, nous avons retenu l'architectures suivantes: `ARM`, `x86`, `PowerPC`, `MIPS`, `RISC-V` et `SPARC`. Notez que ces dernières existent dans des versions 32 bits et 64 bits qui sont listées dans Tableau 2 ci-dessous.

Famille	32 bits	64 bits
`ARM`	`ARMv7-A`	`ARMv8-A`
`x86`	`x86-32`	`x86-64`
`PowerPC`	`PPC 32 bits`	`PPC 64 bits`
`MIPS`	`MIPS32`	`MIPS64`
`RISC-V`	`RV32`	`RV64`
`SPARC`	`SPARC v8`	`SPARC v9`

Tableau 2. – Architectures considérées dans l'étude.

Aparté:

Le support d'une architecture donnée n'est en général pas suffisant pour que le système puisse s'exécuter sur une carte de cette architecture. Cela signifie en général que le programme peut être compilé vers le jeu d'instructions mais il reste un effort important à fournir si l'OS ne fournit pas un *BSP (Board Support Package)* pour la carte considérée. Cet aspect n'est pas abordé en profondeur dans l'étude.

1.5.3 Support multi-processeur

Au début du XXI^{e} siècle, les architectures multi-processeurs se sont imposées dans l'ensemble des secteurs de l'informatique. Jusqu'au milieu des années 2000, la croissance exponentielle de la puissance de calcul était principalement soutenue par l'augmentation rapide des fréquences d'horloges des monoprocesseurs. Cette stratégie a cependant rencontré des limites physiques

(mur thermique, courants de fuite, ...). L'industrie des microprocesseurs s'est alors tournée vers le parallélisme offert par les architectures multi-processeurs pour maintenir la progression de la puissance de calcul.

La diffusion de ces technologie dans les systèmes critiques a été freinée par d'importants défis [4]. En effet, les architectures multi-processeur introduisent de nombreuses sources de non-déterminisme (interférences temporelles, prédiction de branche, ...). In fine, ce non-déterminisme rend les analyses statiques plus complexes et donc la certification de tels systèmes plus difficiles. Ces difficultés sont majorées dans les systèmes critiques mixtes [5].

Toutefois leur usage dans les systèmes critiques est désormais généralisé, principalement motivé par la nécessité d'accroître la puissance de calcul tout en permettant une meilleure intégration et une réduction de poids et de taille des systèmes embarqués, notamment dans les secteur de l'avionique et du spatial.

Il existe deux catégories d'architectures multiprocesseur utilisées dans les systèmes critiques:

- Les architectures *SMP (Symmetric multiprocessing)* sont constituées le plus souvent d'un ensemble de cœurs homogènes. Les cœurs partagent la mémoire principale et la majorité des caches et des bus mémoires. Elles offrent d'excellentes performances, à condition que le système d'exploitation sache en tirer parti. En contrepartie, leur programmation est plus complexe. Par exemple le masquage des interruptions seul ne suffit pas à garantir l'isolation de sections critiques du noyau. En effet, plusieurs cœurs peuvent exécuter en parallèle ces sections, ce qui multiplie les occasions de courses critiques. Il faut alors avoir recours à des mécanisme de synchronisation tels que les *spinlocks* et les verrous atomiques. Ces architectures sont répandues aussi bien sur les serveurs que les ordinateurs personnels mais sont aussi en usage dans des systèmes critiques récents.
- Les architectures *AMP (Asymmetric multiprocessing)* sont constituées le plus souvent d'un ensemble de cœurs hétérogènes. Ces cœurs ne partagent pas leurs caches ou leur bus mémoire. Ces architectures sont conçues pour exécuter des instances distinctes de programmes *bare-metal* sur chaque cœur. Cette isolation des cœurs offre un très bon déterminisme du système et une meilleure isolation des tâches. En contrepartie, les mécanismes de communication interprocesseur sont à la charge du développeur. Ces architectures sont depuis longtemps présentes dans l'embarqué critique, notamment sous la forme de *MPSoC (Multiprocessor System on a chip)*.

Le Tableau 3 récapitule les différences entre ces deux architectures multiprocesseur.

Caractéristique \ Architecture	*SMP*	*AMP*
Nature des cœurs	Le plus souvent homogènes	Le plus souvent hétérogènes
Gestion logicielle	Un unique système d'exploitation gère tous les cœurs et partage dynamique les tâches entre eux	Instance indépendante exécutée sur chaque cœur
Partage de Ressources	Mémoire principale, périphériques et caches partagés	Ressources partitionnées entre les cœurs
Objectif & Performance	Excellent débit	• Déterminisme accru

		• Meilleure isolation des tâches
Domaines d'application	• Ordinateurs personnels • Serveurs	Systèmes embarqués critiques
Prix	Très bon marché	Élevé

Tableau 3. – Différences entre les architectures *SMP* et *AMP*.

1.5.4 Partitionnement

Les systèmes d'exploitation modernes permettent l'exécution de plusieurs tâches simultanément sur une même machine. Les ressources matérielles étant limitées, ces systèmes doivent partager ces ressources de façon sûre et sécurisée entre les tâches en cours d'exécution.

Pour chaque système étudié, nous examinons le partitionnement de deux ressources: la mémoire principale d'une part et le temps *CPU* d'autre part. Pour la mémoire principale, nous parlerons de *partitionnement spatial* et pour le temps *CPU* de *partitionnement temporel*.

Notez que le terme *tâche* doit être compris dans un sens très large et que le vocabulaire varie d'un système à l'autre. Par exemple, un *GPOS* comme *Linux* propose généralement une notion de *thread* famillié des développeurs système, tandis qu'un hyperviseur comme *Xen* parlera de *domaine*. Quant au terme *partition*, il est fréquemment utilisé par la documentation des hyperviseurs.

1.5.4.1 Partitionnement spatial

Le partitionnement spatial désigne le partage de la mémoire principale entre plusieurs tâches en cours d'exécution. On souhaite conserver l'état mémoire de plusieurs tâches dans une même mémoire principale tout en garantissant une forme d'isolation entre elles. Par exemple, on ne veut pas qu'une tâche d'une utilisatrice Alice puisse lire des informations confidentielles actuellement manipulées par un tâche d'un utilisateur Bob. De même, on ne veut pas qu'une instruction erronée exécutée par une tâche d'Alice puisse corrompre accidentellement l'état mémoire d'une tâche de Bob.

Autrefois, le partitionnement spatial était assuré entièrement par une couche logicielle. Ce n'est généralement plus le cas sur les systèmes informatiques actuels qui sont équipés de puces dédiées à cette tâche. Cependant, nous allons voir qu'il y a un compromis à faire entre l'isolation spatiale et le déterminisme du système. Il peut donc être souhaitable de limiter l'usage de ces puces de gestion mémoire.

De nos jours, les ordinateurs personnels et les serveurs disposent d'un microcontrôleur *MMU (Memory Management Unit)*. Ce dernier permet l'utilisation d'adresses virtuelles dans les instructions machines. Lors de l'exécution de telles instructions, ces adresses sont traduites à la volée en adresses physiques. Le *MMU* vérifie également les accès suivant une politique programmable. Ainsi, chaque tâche a l'illusion de disposer de sa propre mémoire principale et les accès frauduleux sont remontés aux systèmes d'exploitation via des interruptions matérielles.

Dans le monde de l'embarqué, en plus des systèmes à *MMU*, coexistent des systèmes à *MPU (Memory Protection Unit)* qui ne font que la protection mémoire et des systèmes sans support matériel pour le partitionnement spatial. On regroupe parfois ces systèmes sous l'appellation *MMU-less*. Le choix d'un système *MMU-less* présente plusieurs avantages. Tout d'abord l'usage d'adresses virtuelles introduit un coût en performance lors de la traduction vers les adresses

physiques. Ce coût est généralement réduit par l'usage d'un cache matériel de type *TLB* (*Translation Lookaside Buffer*) mais son usage rend le système moins déterministe, surtout dans une architecture *SMP* [6].

Pour chaque système, nous examinerons donc le support pour les architectures *MMU* et *MMU-less*.

1.5.4.2 Partitionnement temporel

Le partitionnement temporel désigne le partage du temps *CPU* entre les tâches exécutées. Contrairement à son homologue spatial, le pendant temporel est souvent géré par une couche logicielle: l'ordonnanceur de tâches (*scheduler*). Il a pour rôle de décider quelle tâche doit s'exécuter sur quel processeur et pendant combien de temps.

Un ordonnanceur poursuit des objectifs variés et parfois incompatibles. Parmi les principaux critères, on peut citer:
- *Throughput*: maximiser la quantité de travail accomplie par unité de temps,
- *Latency*: minimiser le délai qui s'écoule entre le réveil d'une tâche et son exécution sur un cœur,
- *Fairness*: répartir équitablement le temps de calcul en tenant compte des priorités des tâches,
- *Déterminisme*: avoir un comportement aussi déterministe que possible.

Par exemple, utiliser un algorithme de décision astucieux aura tendance à augmenter la latence mais peut augmenter le débit. De même, tirer parti du parallélisme d'une architecture *SMP* diminue la latence mais augmente l'indéterminisme du système. Il existe donc une multitude d'ordonnanceurs qui font des compromis différentes entre ces aspects et beaucoup d'autres.

De façon générale un ordonnanceur de *GPOS* cherche à maximiser le débit et être équitable, tandis qu'un ordonnanceur temps réel cherche plutôt à être déterministe et à minimiser la latence.

Pour chacun des systèmes étudiés, nous avons donc décrit les ordonnanceurs disponibles. On examinera en particulier la présence de politiques d'ordonnancement temps réel parmi la liste suivante:
- *Fixed-priority*
- *Rate Monotonic*
- *Deadline Monotonic*
- *Earliest Deadline First*
- *Round Robin*

1.5.4.3 Déterminisme

Comme nous l'avons expliqué dans la sous-section 1.3, les logiciels, et en particulier le système d'exploitation, d'un système critique doivent fournir des garanties sur le temps d'exécution de leurs routines. En informatique usuelle, le temps d'exécution d'un programme ne fait généralement pas parti de sa correction[5]. Ce n'est plus le cas dans un système temps réel où répondre après un délai trop long conduit à un résultat erroné. On souhaite donc que les calculs soient fait suffisamment vite en toute circonstance, tandis qu'en informatique usuelle on cherche généralement à ce que les calculs soient fait le plus vite possible en moyenne.

Afin d'offrir ces garanties temps réel, le système d'exploitation doit être aussi déterministe que possible. Ce déterminisme permet en pratique d'estimer le temps d'exécution de ses routines

[5]Une exception notable est celle des applications multimédia.

dans le pire cas[6]. Le déterminisme est souvent assuré par le caractère préemptible du noyau[7] et des éventuelles autres tâches. En effet, lorsqu'une tâche critique doit commencer son exécution aussi vite que possible, il ne faut pas que celle-ci doive attendre la fin de l'exécution d'une longue routine du noyau ou la fin de la tranche de temps d'une tâche de plus faible priorité. La latence du système d'exploitation est donc une mesure importante pour assurer le respect des échéances.

1.6 Corruption de la mémoire

Nous avons étudié le support logiciel des différents systèmes visant à prévenir la corruption de la mémoire. On distingue deux types d'erreurs:

- Les *soft errors* sont dues à un événement exceptionnel et transitoire qui corrompt des données. Par exemple le rayonnement de fond peut produire un basculement de bits (*bit flips*). Ces erreurs peuvent être souvent corrigées à condition de mettre en places des mesures préventives.
- Les *hard errors* sont dues à un dysfonctionnement matériel au niveau de la puce mémoire. Ces erreurs ne peuvent pas être corrigées et nécessitent un remplaçant de la puce ou, à défaut, une isolation de celle-ci.

Dans cette étude nous nous sommes limités à la mémoire principale et plus précisément aux mémoires *DRAM (Dynamic Random Access Memory)* équipées de puces supplémentaire pour gérer des codes correcteurs. On parle de mémoire *ECC (Error correction code)*.

Aparté: support matériel de l'*ECC*

Les mémoires *ECC* nécessitent un support spécifique par le contrôleur mémoire, le *CPU* et le *BIOS*. Si ce support est rare sur le matériel grand public, il est en revanche commun dans celui destiné aux serveurs.

1.7 Perte du flux d'exécution

La perte du flux d'exécution (*control flow hijacking*) est une vulnérabilité majeure dans les systèmes d'exploitation, où un attaquant modifie le flux d'exécution normal d'un programme pour exécuter du code malveillant. Cette attaque exploite généralement des dépassements de tampon ou d'autres corruptions mémoire pour modifier les adresses de retour ou les pointeurs de fonction.

Les techniques d'attaques sont nombreuses et plusieurs contremesures peuvent être mises en place pour atténuer le risque, notamment:

- Des mécanismes de *Control-Flow Integrity* (*CFI*) [7]. Les *CFI* visent à garantir que le flux d'exécution d'un programme suit uniquement les chemins d'exécution légitimes définis par le graphe de flot de contrôle du programme. Dans les systèmes embarqués et temps-réel, l'application du *CFI* présente des défis particuliers liés aux contraintes de ressources (taille, poids, puissance, coût) et aux exigences temporelles strictes. Les mécanismes de *CFI* doivent minimiser leur surcoût en temps d'exécution tout en offrant des garanties de sécurité robustes.

[6] Ce concept est souvent appelé *WCET* (*Worst Case Execution Time*) dans la littérature.

[7] Nous verrons toutefois avec l'exemple de *seL4* que ce n'est pas toujours la bonne approche pour obtenir le déterminisme.

- La randomisation de l'espace d'adressage (*Adress Space Layout Randomization* (*ASLR*)). Elle consiste à introduire de l'aléa dans les adresses des segments code afin de rendre difficile leur localisation par un attaquant. *ASLR* offre une excellente protection sur les plateforme 64 bits.
- Les *canaris* de pile. Il s'agit d'une protection ajouté à la compilation du programme et prévient les attaques par écrasement de piles en insérant des valeurs aléatoires. Si un attaquant tente de modifier la pile pour modifier une adresse de retour, il écrase un canari et l'attaque peut ainsi être détectée.
- L'utilisation de méthodes comme le *bound checking* pour prévenir les dépassements de tampon.
- Des analyses statiques ou dynamiques pour détecter la présence de vulnérabilités induites par des erreurs de programmation. Certaines de ces analyses peuvent être intégrées directement dans le langage de programmation, par exemple sous la forme d'un *typechecker*. D'autres utilisent des outils externes, voire des assistants à la démonstration.

Nous avons donc examiné la présence de telles contremesures pour chacun des systèmes étudiés.

1.8 Écosystème

Pour chacun des systèmes d'exploitation étudiés, nous avons effectué une revue des outils de son écosystème utiles durant le cycle de vie des applications. Plus précisément, notre analyse s'est concentrée sur l'offre logicielle suivant trois aspects: le *monitoring*, le *profilage* et le *débogage*.

Le *monitoring* vise à surveiller l'activité d'un système informatique. Les outils de *monitoring* permettent le plus souvent la journalisation d'événements. Comme ces outils sont généralement utilisés en production, il est important qu'ils ne grèvent pas la performance ou compromettent la sûreté ou la sécurité du système.

Le *profilage* est une technique utilisée pour mesurer et analyser les performances d'un programme. Elle est le plus souvent employée durant la phase de développement à des fins d'optimisation en permettant de localiser des points chauds. Toute mesure ayant un impact sur l'objet mesuré, il est crucial que cette instrumentation soit faite de la façon la moins intrusive possible.

Le *débogage* est un ensemble de techniques permettant d'analyser un bogue. La technique la plus répandue consiste, via un débogueur, à exécuter le programme pas à pas et explorer l'état de la mémoire et des registres.

1.9 Gestion des interruptions

Une *interruption* est un événement matériel qui altère le flot d'exécution normal d'un programme. Au niveau matériel, elle se manifeste classiquement par un signal électrique émit par un périphérique ou le processeur lui-même et à destination du processeur. Lorsque le processeur reçoit l'interruption, l'exécution courante est suspendue et le contexte est sauvegardé puis une routine du noyau appelée *ISR (Interrupt Service Routine)* est lancée pour gérer l'interruption.

La programmation en présence d'interruptions est rendue difficile par leur nature asynchrone. En effet, rien n'interdit qu'une interruption se déclenche pendant l'exécution de l'*ISR* d'une

autre interruption. C'est même le scénario le plus courant. La présence d'interruption asynchrone induit deux grandes difficultés:

- La correction du noyau repose sur la préservation d'invariants pour ses structures de données. Ainsi, certaines sections de code sont critiques car elles ne peuvent être interrompues sans briser ces invariants.
- La possibilité d'avoir une cascade d'interruption rend difficile l'estimation du temps d'exécution en *espace noyau*. Elle peut même être non bornée dans les pires cas. Cette latence doit être contrôlée pour garantir des bonnes performances et le déterminisme du système.

Une solution consiste à masquer les interruptions lors de l'exécution de sections critiques. À cette fin, les architectures matérielles modernes sont équipées de microcontrôleurs dédiés à la programmation des interruptions. Ainsi les architectures *x86* sont munies de puce *I/O APIC* (*Input/Output Advanced Programmable Interrupt Controller*) pour gérer les interruptions provenant des périphériques et pour les architectures multi-cœur, chaque cœur est muni d'un *Local APIC* pour gérer les interruptions entre cœurs. De même, les architectures *ARM* diposent d'un système dévolue à la programmation des interruptions appelé *GIC* (*Generic Interrupt Controller*). Des contrôleurs similaires existent pour les autres architectures considérées dans la sous-section 1.5.2. Pour simplifier, nous désignons ces puces sous l'appellation *PIC (Programmable Interrupt Controller)*.

Dans tous les cas, un périphérique ou un cœur qui souhaite envoyer une interruption matérielle commence par envoyer une *IRQ (Interrupt ReQest)* à un microcontrôleur *PIC*. Ce dernier décide ou non d'envoyer une interruption au cœur concerné. Cette décision est programmable, le plus souvent sous la forme d'un vecteur stocké dans un registre du *PIC* et accessible par une adresse virtuelle.

Si masquer les interruptions résout la première difficulté de façon drastique, elle doit être fait avec une granularité suffisante pour ne pas grever les performances de l'OS et en particulier la latence. De plus, la désactivation des interruptions rend les sections concernées non-préemptibles puisque la préemption est souvent gérée par une interruption matérielle déclenchée par une horloge matérielle. Masquer les interruptions longtemps n'est donc pas souhaitable non plus dans un scénario temps réel.

Dans le cas d'un hyperviseur, le travail est double puisqu'il doit à la fois gérer les interruptions pour lui-même mais également proposer une interface pour que ses systèmes invités fassent de même de façon contrôlée. Pour atteindre ce but, l'hyperviseur peut adopter trois approches:

- Virtualiser totalement les contrôleurs *PIC*,
- Paravirtualiser les contrôleurs *PIC*,
- Utiliser un support matériel pour la virtualisation des interruptions. Les architectures *x86* et *ARM* modernes disposent d'un support matériel pour la virtualisation de leurs microcontrôleurs *PIC* (*VGIC* pour *ARM*, *Intel APICv* pour *Intel x86* et *AMD AVIC* pour *AMD x86*).

Pour chaque système étudié, nous examinons les aspects suivants:

- Les mécanismes utilisés pour masquer les interruptions en *espace noyau*,
- Les mécanismes de virtualisation des interruptions.

1.10 *Watchdog*

Un *watchdog* est un dispositif matériel ou logiciel conçu pour détecter le blocage d'un système informatique, et de réagir de manière autonome pour ramener ce système dans un état normal. Qu'il s'agisse d'un dispositif matériel ou logiciel, le principe du watchdog consiste le plus

souvent à demander au système surveillé d'envoyer régulièrement un signal à un système surveillant. Le système surveillé dispose d'une fenêtre temporelle pour cette action. S'il n'effectue pas la tâche dans le temps imparti, il est présumé dysfonctionnel. Le système surveillant peut alors tenter de remédier à la situation. Le plus souvent cela consiste à redémarrer la machine.

Les appareils embarqués et les serveurs à haute disponibilité ont souvent recours aux *watchdogs* pour améliorer leur fiabilité. Pour chacun des systèmes nous avons étudiés le support des *watchdog* logiciels et matériels et avons fourni un exemple d'utilisation de l'*API (Applicaton Programming Interface)* lorsque cela était possible.

1.11 Programmation *bare-metal*

La programmation *bare-metal* était et demeure commune dans les systèmes critiques. Toutefois, comme mentionné dans la sous-section 1.2, l'adoption d'un système d'exploitation offre de nombreuses avantages, notamment en permettant l'isolation logicielle de plusieurs tâches critiques. Il est donc fréquent de vouloir porter des applications *bare-metal* existantes vers des architectures virtualisées pour bénéficier de l'isolation et des *API* offertes par ces dernières.

C'est dans cette optique que nous avons examiné les possibilités offertes en matière de programmation *bare-metal* par les hyperviseurs étudiés. Notre analyse est circonscrite aux langages de programmation *Ada*, *C*, *OCaml* et *Rust*.

Le principal défi d'un tel portage est d'adapter un *RTE (RunTime Environment)* de ces langages pour l'hyperviseur donné. Dans le cas de *Ada*, *C* et *Rust* se portage est grandement faciliter par la possibilité de limiter la taille du *RTE* et de se passer de la majorité de la bibliothèque standard ou de la remplacer par une bibliothèque dédiée au *bare-metal*. Pour le langage *OCaml*, la difficulté est plus importante car son *RTE* contient un *ramasse-miette*, ce qui implique de devoir porter un plus grand nombre d'appels systèmes.

1.12 Temps de démarrage

Pour les hyperviseurs, le temps de démarrage des *VM (Virtual Machine)* est une métrique importante de leur performance. En cas de défaillance d'une *VM*, on espère que celle-ci soit relancée aussi rapidement que possible. Un autre usage courant, notamment dans le cloud computing, est de lancer des *VM* à la demande pour s'adapter au mieux aux variations de la charge de travail. Ces *VM* doivent se lancer rapidement pour garantir des temps de réponse acceptables.

1.13 Maintenabilité

L'usage d'un *COTS (Commercial off-the-shelf)* présente le risque d'une rupture de la maintenance du système.

La maintenabilité du système d'exploitation est évalué à travers différents sous-critères:
- La taille du code source.
- La modularité de la base de code et la complexité des invariants de celle-ci.
- L'organisation et le nombres de développeurs.

2 Linux

Linux en bref

- **Type** : GPOS, noyau modulaire + Hyperviseur (KVM) + RTOS (PREEMPT_RT)
- **Langage** : C (98%)
- **Architectures** : 30+ architectures (x86, ARM, RISC-V, PowerPC, MIPS, SPARC, ...)
- **Usage principal** : Serveurs, embarqué, supercalculateurs, desktop
- **Points forts** : Maturité, large écosystème, flexibilité, support matériel étendu, documentation précise
- **Limitations** : Complexité élevée, surface d'attaque importante, déterminisme limité (sans PREEMPT_RT)
- **Licences** : GPL v2 + exception *syscall*

Le noyau *Linux* est un *GPOS* libre de type *UNIX*. Son développement débute en 1991 sous la forme d'un projet personnel de Linus Torvalds, alors étudiant en informatique à l'université d'Helsinki en Finlande. Linus entreprit d'écrire un noyau après avoir constaté qu'il n'existait pas de système *UNIX* bon marché capable d'exploiter pleinement les capacités de son nouveau processeur 32 bits *Intel 80386*. Le projet prend alors rapidement de l'ampleur, notamment après l'adoption en 1992 de la licence *GPLv2* pour distribuer le code source. Ce changement de licence a permis au noyau d'utiliser les outils du projet *GNU (GNU is Not Unix)* afin de fournir un système d'exploitation complet. La première version majeure `1.0` est publiée en 1994 avec un support pour l'interface graphique via le projet *XFree86*. Les distributions *GNU/Linux Red Hat* et *SUSE* publient leur première version majeure en 1994 également. À partir de 1995 avec la version `1.1.85`, le noyau passe d'une architecture *monolithique* à une approche modulaire, permettant le chargement à chaud de modules. La version `2.0` publiée en 1996 propose un support pour les architecture *SMP*. En 2007, la version `2.6.20` intègre un hyperviseur baptisé *KVM*. En 2024, la totalité des patchs du projet *PREEMPT_RT* sont intégrés dans le noyau, faisant de *Linux* un *RTOS*.

De nos jours le noyau *Linux* est développé par une communauté décentralisée de développeurs. De très nombreuses entreprises contribuent au noyau, notamment aux pilotes (*Intel*, *Google*, *Samsung*, *AMD*, ...).

2.1 Tutoriel

Certaines des fonctionnalités présentées dans ce chapitre ne font pas parties des noyaux distribués par défaut par les distributions *GNU/Linux* grand public. Le noyau *Linux* est configuration à la compilation à travers de très nombreuses options. Malheureusement, il ne semble pas exister une méthode standard pour connaître la configuration du noyau en cours d'exécution. Certaines distributions le permettent via la commande:

```
$ zgrep OPTION /proc/config.gz
```

où `OPTION` désigne une option de compilation du noyau. Tandis que d'autres placent un tel fichier dans la partition de démarrage:

```
$ grep OPTION /boot/config-$(uname -r)
```

Nous laissons le soin au lecteur de se reporter à la documentation de sa distribution si les commandes ci-dessus ne fonctionnent pas.

2.2 Architectures supportées

À l'origine, le noyau *Linux* était uniquement développé pour l'architecture *x86-32*. Il a depuis été porté sur de très nombreuses autres architectures [8]. Il fonctionne notamment sur les architectures suivantes: *x86-32*, *x86-64*, *ARM v7*, *ARM v8*, *PowerPC*, *MIPS*, *RISC-V* et *SPARC*.

Quant à l'hyperviseur *KVM*, il supporte la virtualisation assistée par le matériel sur certaines architectures. Sur architecture *x86*, il supporte les extensions de virtualisation *Intel VT-x* et *AMD-V*. Sur architecture *ARM*, il supporte l'extension de virtualisation de *ARM v7* à partir de *Cortex-A15* et de *ARMv8-A*. Enfin il supporte certaines architectures *PowerPC* comme *BookE* et *Book3S*.

2.3 Support multi-processeur

Cette section aborde le support d'architectures multi-processeur sous *Linux*.

2.3.1 Architectures *SMP*

Le support pour les architectures *SMP* est ajoutée dans *Linux 2.0* en 1996. Toutefois les premières versions du noyau supportant les architectures *SMP* avaient recours à un verrou global appelé *BKL (Big Kernel Lock)*. Ce dernier assurait que les sections critiques du noyau ne pouvaient pas s'exécuter en parallèle. Cette technique avait l'avantage d'être simple à mettre en place mais conduisait à des performances médiocres, notamment lorsque le système avait de nombreux cœurs. Le *BKL* a été progressivement remplacé par des mécanismes de synchronisation plus fins comme les *spin locks* et les *mutexes*, puis totalement supprimé à partir de la version *2.6.39*. Le noyau propose aussi depuis la version 2.5 des structures de données synchronisées de type *RCU* (*Read-Copy-Update*) [9] qui permettent la lecture et l'écriture simultanée sans mécanisme de verrouillage pour les lecteurs. Ces structures sont donc particulièrement pertinentes lorsque la majorité des accès sont en lecture. Quant à l'ordonnanceur de tâche, il est conçu pour répartir aussi équitablement que possible le temps *CPU* entre les processus avec une faible latence.

Pour vérifier que votre noyau a été compilé avec ce support, il faut vérifier la présence de l'option `CONFIG_SMP`.

Aparté: `CONFIG_SMP` obligatoire

Le support *SMP* ne sera plus optionnel à partir de *Linux 6.17* pour la majorité des architectures [10]. Les monoprocesseurs sont de plus en plus rares et les primitives introduites pour le support *SMP* n'engendrent qu'un surcoût mineur. Cette décision permettra de simplifier le code source du noyau.

2.3.2 Architectures *AMP*

Depuis la branche `3.x`, le noyau *Linux* offre un support pour les processeurs distants via les sous-systèmes `remoteproc` [11] et `RPMsg` [12]. Vous pouvez vérifier que votre noyau est compilé avec le support pour ces systèmes avec les options respectivement `CONFIG_REMOTEPROC` et `CONFIG_RPMSG`.

Le cas d'usage typique est l'exécution d'un *RTOS* sur un processeur secondaire dans un système embarqué hétérogène sous la forme d'un *MPSoC*. Avant l'apparition de `remoteproc`, le contrôle des processeurs secondaires se faisait via des *API* propriétaires et non standardisées. Quant au système *RPMsg* (*Remote Processor Messaging*), il permet la intercommunication avec un processeur distant via un protocole asynchrone à la *virtio*.

2.4 Partitionnement

Dans cette section, nous décrivons les principaux mécanismes d'isolation de partitionnement des ressources disponibles sous *Linux*. Ces mécanismes sont aujourd'hui utilisés aussi bien pour la virtualisation via *KVM* que pour les conteneurs des logiciels tels que *systemd*, *Docker* ou *Kubernetes*.

2.4.1 Partitionnement spatial

À l'origine *Linux* était conçu uniquement pour s'exécuter en présence d'un *MMU*. Le projet *µCLinux* [13] était une branche modifiée du noyau *Linux* visant à supporter des architectures sans *MMU*. Ce support a finalement été ajouté à la branche officielle du noyau. L'option de compilation `CONFIG_NOMMU` permet d'activer le support sans *MMU* de *Linux*.

2.4.2 Partitionnement temporel

2.4.2.1 Politiques d'ordonnancement

Linux utilise un système sophistiqué pour décider quelle tâche doit s'exécuter sur le *CPU* lorsque le noyau retourne dans l'*espace utilisateur*. Ce mécanisme repose sur une hiérarchie de politiques d'ordonnancement et de priorités.

Chaque tâche se voit attribuer une politique d'ordonnancement et une priorité statique allant de 0 à 99. Les *threads* d'un même processus possèdent la même priorité au lancement d'un processus. La politique et la priorité d'une tâche peuvent être changée via une *API* ou une ligne de commande *POSIX*.

À l'heure actuelle *Linux* implémente six politiques d'ordonnancement, trois temps réel *SCHED_FIFO*, *SCHED_RR*, *SCHED_DEADLINE* et trois normales *SCHED_OTHER*, *SCHED_BATCH* et *SCHED_IDLE*. Les trois politiques *SCHED_OTHER*, *SCHED_FIFO* et *SCHED_RR* font en fait partie de la norme *POSIX* et *Linux* expose également une *API* C *POSIX* pour contrôler les politiques d'ordonnancement. Par défaut, les processus utilisent la politique *SCHED_OTHER*. Une description détaillée de ces politiques est disponible dans la page de manuel `sched` accessible avec la commande:

```
$ man sched
```

Le noyau décide qu'elle tâche doit s'exécuter en suivant les trois règles suivantes:

- S'il y a une tâche prête dont la priorité statique est la plus élevée de toutes les tâches en attente, elle s'exécutera toujours en premier.
- S'il y a plusieurs tâches prêtes de priorité maximale, celle dont la politique est la plus prioritaire est exécutée en premier. L'ordre de priorité entre les politiques est par ordre décroissant: *SCHED_FIFO*, *SCHED_RR*, *SCHED_OTHER*, *SCHED_BATCH* et *SCHED_IDLE*.

- S'il y a plusieurs tâches prêtes de priorité maximale et de même politique, le comportement dépend de la politique en question comme détaillé ci-dessous.
  - *SCHED_DEADLINE*.
  - *SCHED_FIFO* (*First In First Out*) est une politique d'ordonnancement temps réels. Lorsque plusieurs tâches ordonnancées par *SCHED_FIFO* ont la même priorité statique, la première tâche s'exécute jusqu'à relâcher volontairement le *CPU* où qu'une tâche de plus haute priorité arrive.
  - *SCHED_RR* (*Round Robin*) est une politique d'ordonnancement temps réels. Lorsque plusieurs tâches ordonnancées par *SCHED_RR* ont la même priorité statique, elles s'exécutent à tour de rôle pendant un laps de temps configuration.
  - *SCHED_OTHER* et *SCHED_BATCH* sont les politiques d'ordonnancement normales. Elles ont une priorité statique nulle. Autrement dit les tâches temps réel sont toujours plus prioritaires que les tâches normales. Les trois politiques normales sont implémentées grâce à l'ordonnanceur de tâches *CFS* (*Completely Fair Scheduler*) introduit dans *Linux 2.6.23.*. Il s'agit d'un ordonnanceur à priorité dynamique, c'est-à-dire que la priorité d'une tâche dépend de son comportement dans le passé et il est possible d'aider l'ordonnanceur à faire un meilleur choix via un mécanisme de pondération.
  - *SCHED_IDLE* est la politique des tâches qui ne sont exécutées que lorsqu'aucune autre tâche n'est prête.

Aparté: La commande `sched`

Il est possible de déterminer la politique d'un processus via la commande *POSIX* `sched`. Par exemple pour obtenir la politique utilisées pour les *threads* du processus *PID* 1:

```
$ sched -p 1
```

produit la sortie:

```
pid 1's current scheduling policy: SCHED_OTHER
pid 1's current scheduling priority: 0
pid 1's current runtime parameter: 2800000
```

Cette commande permet également de changer la politique d'ordonnancement d'un processus en cours. Par exemple pour utiliser la politique *SCHED_RR* avec le processus 1000 et la priorité statique 10:

```
$ sudo sched -r -p 10 1000
```

2.4.3 Déterminisme

Au tournant du XXI^{e} siècle, des initiatives ont visées à doter *Linux* de capacités temps réel. Le noyau de l'époque avait été développé dans l'optique de maximiser les performances de son ordonnanceur de tâches, au détriment du déterminisme. Les changements pour rendre l'ordonnanceur déterministe étaient donc considérés trop complexes, et des approches alternatives ont émergées. L'une de ces approches consiste à contourner la difficulté en exécutant les tâches temps réel et le noyau *Linux* directement au-dessus d'un micronoyau temps réel. Le noyau devient ainsi une tâche de faible priorité pour ce micronoyau et peut être préempté par ce dernier. On parle alors d'architecture *cokernel* ou *dual kernel*, voir la figure Fig. 1.

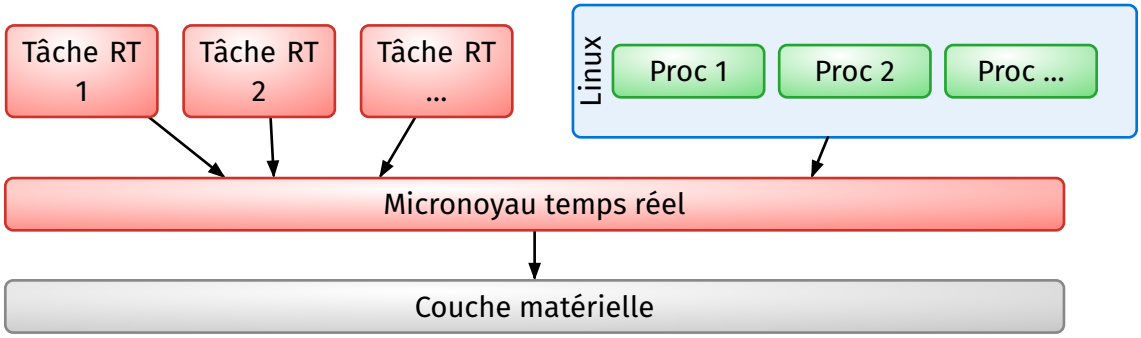

Fig. 1. – Architecture *cokernel*.

En particulier, les projets open-sources *RTLinux*, *RTAI* et *Xenomai* adoptèrent cette approche avec succès. Ces principaux avantages sont ses bonnes garanties quant aux respects des *deadlines* et une latence très faible. En contrepartie, le développeur de l'application temps réel ne peut pas utiliser l'écosystème *UNIX*, rendant le développement plus ardu et coûteux. Par exemple cette architecture conduit à une duplication des pilotes puisque les tâches temps réel ne peuvent pas utiliser les pilotes du noyau *Linux*. Il faut également maintenir une couche d'abstraction dans le noyau *Linux* pour lui permettre d'interagir avec le micronoyau. Cette contrainte a motivé le développement de patchs visant à doter le noyau *Linux* de capacités temps réel. Le plus connu et utilisé est *PREEMPT_RT*.

Le projet *PREEMPT_RT* vise à rendre la majorité des routines du noyau *Linux* préemptibles afin de rendre l'ordonnancement des tâches aussi prédictible que possible. Contrairement aux *cokernels*, cette approche conduit à une modification en profondeur du noyau. Du fait de sa complexité, le projet, initié par Thomas Gleixner et Ingo Molnár en 2005, s'est étalé sur une vingtaine d'années sous la forme d'une succession de patchs. Ces modifications ont été progressivement intégrées à la branche principale du noyau *Linux*, jusqu'aux derniers patchs qui ont été appliqués en septembre 2024. *Linux* est ainsi devenu un *RTOS* complet à partir de sa version *6.12*.

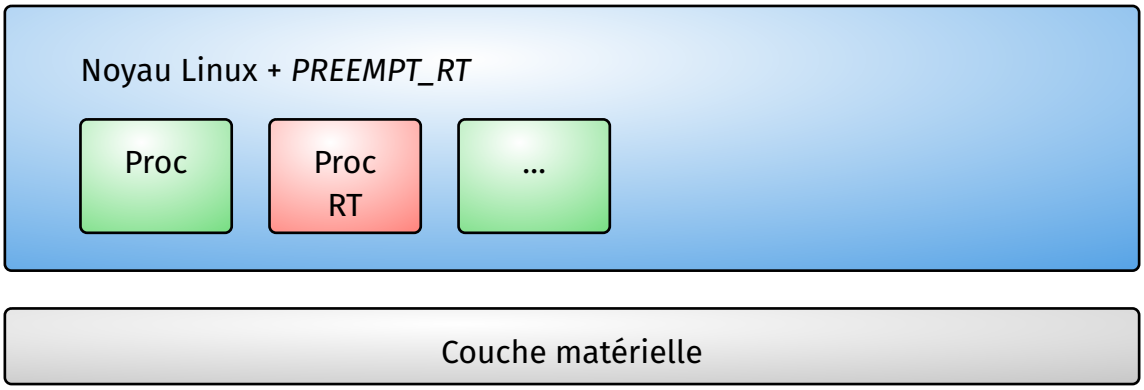

Fig. 2. – Architecture de *Linux* avec *PREEMPT_RT*

Aparté: installation

Bien que *PREEMPT_RT* soit désormais distribué avec la branche principale du noyau, il est nécessaire de compiler ce dernier avec l'option de compilation `CONFIG_PREEMPT_RT` activée pour obtenir un noyau préemptible. Certaines distributions comme *Fedora* ou *Ubuntu* proposent également des noyaux alternatifs avec cette option activée, rendant l'installation de *PREEMPT_RT* plus simple.

Une fois installée, le noyau offre une nouvelle politique d'ordonnancement baptisée *PREEMPT_FULL*, qui comme son nom l'indique, maximise l'ensemble du code préemptible dans le noyau.

Les modifications apportées au noyau par le projet *PREEMPT_RT* sont trop complexes et techniques pour en faire ici une révue détaillée. Toutefois, il est intéressant de comprendre certains de leurs aspects afin de cerner les forces et les limites du temps réel dans ce noyau. Plus d'informations sont disponibles dans la documentation officielle [14].

2.4.3.1 *Timers* hautes résolutions

Historiquement, le noyau *Linux* utilisait une interruption matérielle périodique pour exécuter l'ordonnanceur de tâches. Cette conception impliquait que la précision des *timers* ne pouvaient excéder la durée d'un *tick*, qui était de l'ordre de 1 à 10ms. L'adoption d'une approche *tickless* à partir de la branche 2.6 du noyau a permis d'implémenter des *timers* haute résolution (appelés *hrtimer* dans la terminologie linuxienne pour *High-Resolution Timers*) [15].

Malgré cette amélioration, les *hrtimers* soulèvent toujours des problèmes dans un contexte temps réel. De part la conception actuelle du noyau, les *ISR* appelés lors de l'expiration de ces *timers* sont exécutés avec la plus haute priorité afin de minimiser leur latence. Cela implique qu'une tâche de faible priorité peut retarder l'exécution d'une tâche de plus haute priorité en produisant un grand nombre de *timers*.

Un correctif baptisé *TimerShield* a été proposé dans [16] et implémenté dans un prototype mais n'est pas intégré dans la branche principale du noyau.

2.4.3.2 Mutex temps réels

Chaque fois qu'un processus de faible priorité B à la main sur le *CPU* alors qu'un processus de plus haute priorité A souhaite s'exécuter, on parle d'*inversion de priorité*. Dans le cadre du temps réel, on doit s'assurer que le temps pendant lequel une telle inversion se produit est prédictible. Autrement dit, on doit pouvoir borner cet événement dans le temps. Une vigilance particulière est accordée aux mécanismes de synchronisation, puisqu'une inversion se produit en particulier lorsque que le processus A attend la libération d'une ressource par le processus B. Dans un scénario catastrophe, le processus B n'est jamais ordonnancé, bloquant pendant un temps indéterminé l'exécution de A.

À fin de rendre prédectible la durée de ces inversions de priorité, *PREEMPT_RT* a introduit dans le noyau des *mutex* temps réel (*rt-mutex*). Ceux-ci reposent sur la méthode dîte d'héritage de priorité (*Priority Inheritance*). Dans notre exemple, cela signifie que si le processus B possède une ressource verrouillée par un *mutex* temps réel et que le processus A essaie d'acquérir ce *mutex*, alors la priorité du processus B est augmenté afin qu'il libère cette ressource le plus vite possible.

Plus de détails sur ces *mutex* temps réel sont fournis dans la documentation officielle [17], [18].

2.4.3.3 Gestionnaires d'interruption

Lors de l'exécution d'un *ISR*, il est pratique de désactiver les interruptions car l'*ISR* exécute généralement du code critique et que rien n'empêche d'autres interruptions de subvenir durant son exécution. Cette stratégie a été abondamment utilisée dans le noyau *Linux*. Les *ISR* constituaient donc une partie importante du code non-préemptible du noyau. Afin de réduire la portion de code non-préemptible, l'idée est de diviser en deux étapes le gestionnaire [19]. La première étape (*Top Half*) est exécutée avec les interruptions désactivées et ne fait que le strict nécessaire pour que l'interruption puisse être prise en compte plus tard. La seconde étape (*Bottom Half*) est exécutée dans un *thread* noyau préemptible. Les *threads* noyaux étant planifiés par l'ordonnanceur de tâches, il devient possible d'attribuer des priorités sur l'exécution de ces *threads*.

2.4.3.4 Remplacement de *spinlocks*

En l'absence de *PREEMPT_RT*, une tâche qui attend la libération d'un *spinlock* effectue une attente active. Durant cette attente, la tâche n'est pas préemptible. En présence de *PREEMPT_RT*, ces *spinlocks* sont donc remplacés par des *rt-mutex*.

2.4.3.5 RCU

2.5 KVM

Depuis la version `2.6.20` publiée 2007, *Linux* intègre un hyperviseur baptisé *KVM* (*Kernel-based Virtual Machine*) [20]. Il s'agit d'un hyperviseur de type 1 assisté par le matériel. Il offre également un support pour la paravirtualisation.

2.5.1 Les *control croups*

Les *cgroups* (*control groups*) sont un mécanisme du noyau *Linux* qui permet une gestion fine et configurable des ressources. Il existe deux versions de ce mécanisme dans le noyau actuel:

- La version `v1`, introduite en 2008 dans le noyau *Linux 2.6.24*,
- La version `v2` est une refonte complète de la `v1`, introduite en 2016 dans le noyau *Linux 4.5*. Elle est aujourd'hui la version recommandée.

Dans cette section, nous ne décrivons que le fonctionnement de la version `v2`. Le lecteur intéressé par la première version de l'API pourra se référer à sa documentation [21].

Les *cgroups* forment une structure arborescente et chaque processus appartient à un unique *cgroup*. À leur création, les processus héritent du *cgroup* de leur parent. Ils peuvent par la suite migrer vers un autre *cgroup*, s'ils ont les privilèges adéquates. Ces migrations n'affectent pas leurs enfants déjà existants, mais seulement ceux créés par la suite. Quant aux *threads systèmes*, ils appartiennent généralement au *cgroup* de leur processus mais on peut mettre en place une hiérarchie de *cgroup* pour eux.

La répartition des ressources se fait via des *contrôleurs* spécialisés. Chaque contrôleur permet d'appliquer des restrictions sur un *cgroup* et ses descendants. Une politique appliquée sur un enfant doit être au moins aussi restrictive que celle de son parent.

Les principaux contrôleurs sont:

- `cpu`: contrôle l'utilisation du CPU,
- `memory`: contrôle l'utilisation de la mémoire vive et de la mémoire d'échange,
- `io`: contrôle les opérations d'entrée/sortie sur les périphériques de stockage,
- `pids`: limite le nombre de processus et de threads,

- `cpuset`: affecte un groupe de processus à des cœurs CPU spécifiques,
- `hugetlb`: contrôle l'utilisation des *huge pages*.

Plus d'informations sur les *cgroups* sont disponibles dans la documentation officielle [22].

2.5.1.1 Exemple d'utilisation

La hiérarchie des *cgroups* est accessible dans l'espace utilisateur via un pseudo système de fichiers de type `cgroup2`. Il est généralement monté dans le dossier `/sys/fs/cgroup`. La création et la suppression de *cgroups* se fait alors grâce aux commandes habituelles pour la gestion de fichiers sous UNIX.

Supposons que nous souhaitions limiter la consommation de mémoire d'un processus à 5 Mio. On commence par créer deux[8] *cgroups* `foo` et `bar`:

```
$ sudo mkdir -p /sys/fs/cgroup/foo/bar
$ echo "+memory" | sudo tee /sys/fs/cgroup/foo/cgroup.subtree_control
$ echo "5 * 2^20" | bc | sudo tee /sys/fs/cgroup/foo/bar/memory.max
```

Désormais la mémoire totale occupée par les processus du *cgroup* `bar` ne doit pas excéder les 5 Mio.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void) {
  size_t sz = 0;

  printf("How many bytes do you want to allocate? ");
  if (scanf("%zu", &sz) != 1) {
    printf("Invalid size\n");
    return EXIT_FAILURE;
  }

  char *buf = malloc(sz * sizeof(*buf));
  if (!buf) {
    printf("Cannot allocate %zu bytes\n", sz);
    return EXIT_FAILURE;
  }

  memset(buf, 0, sz);
  free(buf);
  return EXIT_SUCCESS;
}

```

Liste 1. – `limited.c`

À titre d'exemple, compilons et lançons le programme dont le code source est donné dans

```
$ gcc -O0 limited.c -o limited
$ ./limited
```

[8] Il n'est pas possible de le faire avec un seul *cgroup* dû à une règle de l'API appelée «no internal processes».

et dans une autre console, on ajoute le processus au cgroup `bar`:

```
$ pgrep limited | sudo tee /sys/fs/cgroup/foo/bar/cgroup.procs
```

Finalement, on demande plus de mémoire que la limite autorisée et le processus est tué:

```
How many bytes do you want to allocate? 6000000
fish: Job 1, './limited' terminated by signal SIGKILL (Forced quit)
```

Notez que pour obtenir l'erreur escomptée, il faut prendre garde à deux aspects:
- Le message d'erreur `Cannot allocate` ne s'affiche pas car *Linux* n'alloue la mémoire que lorsqu'elle est véritablement utilisée. C'est donc lorsque l'on remplit le tampon de zéros avec `memset` que la mémoire est réclamée.
- Si certaines optimisations sont activées, le compilateur `gcc` supprime l'appel à la fonction `malloc` car il constate qu'on ne lit pas le buffer et donc son contenu est inutile. Il faut donc désactiver ces optimisations avec l'option `-O0`.

2.5.2 Chroot

L'appel système `chroot` permet de changer le dossier racine de l'arborescence vue par le processus appelant. Cette fonction était parfois utilisée pour isoler le système de fichiers d'un démon et ainsi prévenir un accès frauduleux à des fichiers sensibles. De nos jours, cette méthode n'est plus recommandée car cette protection peut être contournée sous certaines conditions. Un exemple d'attaque est détaillé dans sa page de manuel [23]. D'autre part cet appel n'offre pas le même degré d'isolation que les *namespaces* abordés dans la section 2.5.3.

2.5.3 Les namespaces

Les *namespaces* sont des outils permettant d'isoler des ressources pour des processus. Cette isolation permet de créer des environnements sécurisés et indépendants.

Les principaux namespaces sont:
- `PID Namespace`: isole l'arborescence des processus.
- `Network Namespace`: isole la pile réseau, permettant à un conteneur d'avoir ses propres interfaces, tables de routage et règles de pare-feu.
- `Mount Namespace`: isole l'arborescence des fichiers.
- `UTS Namespace` (*Unix Time-sharing System*): isole le nom d'hôte et le nom de domaine.
- `User Namespace`: isole les identifiants utilisateurs et les groupes.

2.5.3.1 Exemple d'utilisation avec `systemd`

Le gestionnaire de services `systemd` intègre l'outil `systemd-nspawn` pour faciliter l'utilisation des *namespaces*. Il constitue une alternative à `chroot` plus sûre. En plus d'isoler l'aborescence des fichiers, cette commande isole celle des processus, le réseau et les utilisateurs. Par exemple, considérons le programme *C* suivant:

```c
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <ctype.h>
#include <stdbool.h>
#include <unistd.h>

static inline bool is_pid(char *name) {
  while (*name && isdigit(*name))
    name++;
  return *name == '\0';
}

int main(void) {
  printf("My pid: %d\n", getpid());

  DIR *dir = opendir("/proc");
  if (!dir)
    return EXIT_FAILURE;

  struct dirent *ent;
  while ((ent = readdir(dir)) != NULL) {
    if (is_pid(ent->d_name))
      printf("%s\n", ent->d_name);
  }

  closedir(dir);
  return EXIT_SUCCESS;
}

```

Liste 2. – `alone.c`

En le compilant puis le liant statiquement à la bibliothèque C, il est possible de le lancer dans un conteneur de *systemd* ainsi:

```
$ gcc -static ./foo/alone.c -o ./foo/alone
$ sudo systemd-nspawn -D ./foo ./alone
```

On obtient alors la sortie suivante:

```
▒ Spawning container foo on /foo.
▒ Press Ctrl-] three times within 1s to kill container.
My pid: 1
1
Container foo exited successfully.
```

révélant que `alone` est le seul processus visible dans le conteneur `foo` et qu'il a le PID 1.

2.5.4 *Capabilities*

Les implémentations UNIX traditionnelles distinguent deux catégories de processus: les processus *privilégiés* et les processus *non privilégiés*. Les processus privilégiés contournent toutes les vérifications de permission du noyau, tandis que les processus non privilégiés sont soumis

à ces vérifications en se basant sur des identifiants associés au processus[9]. Par exemple la commande suivante:

```
$ ps -U root -u root
```

affiche tous les processus ayant pour UID réel ou effectif `root`. Ils constituent l'essentiel des processus privilégiés en cours d'exécution. Vous devriez obtenir une sortie similaire à celle-ci:

```
    PID TTY          TIME CMD
      1 ?        00:00:03 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 pool_workqueue_release
      4 ?        00:00:00 kworker/R-rcu_gp
      5 ?        00:00:00 kworker/R-sync_wq
      6 ?        00:00:00 kworker/R-kvfree_rcu_reclaim
      7 ?        00:00:00 kworker/R-slub_flushwq
      8 ?        00:00:00 kworker/R-netns
     10 ?        00:00:00 kworker/0:0H-events_highpri
     13 ?        00:00:00 kworker/R-mm_percpu_wq
     ...
```

Sans surprise `systemd` et un grand nombre de workers et de threads noyaux sont des processus privilégiés. La commande suivante:

```
$ ps -U $(whoami)
```

vous donnera la liste des processus qui s'exécutent avec l'UID effectif de votre utilisateur. Vous devriez y retrouver vos logiciels. Par exemple, sur mon ordinateur j'obtiens la sortie:

```
    PID TTY          TIME CMD
   2512 ?        00:00:00 systemd
   2514 ?        00:00:00 (sd-pam)
   2523 ?        00:00:00 devmon
   2524 ?        00:00:00 gamemoded
   2530 tty1     00:00:00 fish
   2537 ?        00:00:00 mpd
   2541 ?        00:00:00 dbus-daemon
   2652 ?        00:00:44 pipewire
   2653 ?        00:00:11 wireplumber
   2683 ?        00:00:00 udevil
   2715 ?        00:17:47 niri
  29113 pts/3    00:00:02 typst
   ...
```

Les logiciels `niri`, `fish` et `typst` sont en cours d'exécution avec mes droits utilisateurs. En particulier, ils ne peuvent pas modifier n'importe quel fichier du disque ou faire tous les appels systèmes car ils ont des privilèges limités.

Cette distinction en deux catégories n'offre pas toujours suffisamment de granularité. Il est fréquent de ne vouloir exécuter que quelques appels systèmes avec les privilèges `root` dans

[9]Ces identifiants sont le plus souvent l'UID (*User IDentifier*) effectif, le GID (*Group IDentifier*) effectif ou les groupes supplémentaires du processus.

un processus. Or exécuter un programme avec les droits `root` constitue un risque de sécurité car s'il présente une faille exploitable, un intrus pourrait obtenir les droits `root` à travers lui.

2.5.4.1 SetUID

Les processus peuvent être privilégiés parce qu'ils ont été lancé par l'utilisateur *root* ou via le mécanisme du *setUID* qui permet à processus d'avoir certains des privilèges du propriétaire du binaire plutôt que de l'utilisateur qui l'a lancé. Ainsi le binaire `passwd` appartient à *root* mais permet à n'importe qui de changer son propre mot de passe.

2.6 Corruption de la mémoire

Le noyau *Linux* intègre un sous-système nommé *EDAC* (*Error Detection and Correction*) [24] qui permet la journalisation des erreurs mémoires. La journalisation s'effectue grâce au démon *rasdaemon*.

Certains processeurs AMD nécessitent l'utilisation d'un pilote pour que *EDAC* fonctionne.

Le noyau fournit également une interface logicielle commune [25] via *sysfs*[10] pour les interfaces de pilotage du scrubbing décrites dans le à l'exception de l'interface *ARS* qui utilise son propre pilote.

2.7 Perte du flux d'exécution

Linux peut bénéficier de plusieurs mécanismes de protection du flux d'exécution, notamment via les extensions matérielles modernes comme *Intel CET* (*Control-flow Enforcement Technology*) sur *x86* ou *ARM BTI* (*Branch Target Identification*) sur *ARM*. Ces mécanismes matériels offrent une protection efficace avec un surcoût minimal.

2.8 Écosystème

Linux dispose d'un écosystème riche et mature d'outils de monitoring et d'observabilité [26], [27]. Ces outils permettent de surveiller les performances, l'état du système et d'identifier les problèmes en temps-réel. Parmi les outils de monitoring les plus utilisés:

- *top/htop* [28]: Moniteurs système interactifs affichant l'utilisation du CPU, de la mémoire et des processus en temps réel.
- *netdata* [29]: Solution de monitoring temps-réel légère et performante, collectant automatiquement plus de 5000 métriques sans configuration. Particulièrement adaptée aux environnements embarqués grâce à sa faible empreinte.
- *eBPF* (*Extended Berkeley Packet Filter*) [30] : Technologie moderne permettant l'exécution de code personnalisé dans le noyau sans modification ni ajout de modules. *eBPF* offre une observabilité en temps réel avec un impact minimal sur les performances, devenant l'outil de référence pour le monitoring avancé en 2024.
- *SystemTap* [31]: Permet l'instrumentation dynamique du noyau pour l'analyse approfondie du comportement système.
- *Prometheus/Grafana* [32], [33] : Solutions d'observabilité distribuée largement adoptées pour le monitoring de systèmes critiques.

[10] Le système de fichiers *sysfs* est un pseudo système de fichiers disponible sous Linux. Il permet aux logiciels tournant dans le *user space* de lire et de modifier des paramètres des pilotes et des périphériques via des fichiers. Il est généralement monté dans le dossier */sys*.

- *strace/ptrace*: .
- *perf* [34]: Outil d'analyse de performance intégré dans le noyau *Linux* depuis sa version 2.6.31. À l'origine *perf* permettait de tracer l'activité du *CPU* via des compteurs *PMU (Performance Monitoring Unit)*. Depuis, ses fonctionnalités ont été considérablement étendues et il permet maintenant d'instrumenter avec un faible surcoût aussi bien le noyau que les programmes exécutés dans l'*espace utilisateur*.
- *oprofile* [35]: Outil d'analyse de performance. Il permet le profilage d'une application ou du système tout entier. Il permet également la collecte d'événements via les *PMU*.
- *kgdb/kdb* [36]: *Linux* intègre des interfaces pour déboguer le code du noyau.

Pour les systèmes embarqués, la simplicité et la légèreté des outils sont prioritaires. *Monitorix* est particulièrement adapté à ces contraintes, ayant été conçu pour les serveurs mais utilisable sur dispositifs embarqués grâce à sa taille réduite.

2.8.1 Exemple de profilage

Afin d'illustrer certains outils de profilage, nous allons utiliser le programme suivant qui parcourt des cases d'un tableau d'entiers soit dans de façon séquentielle, soit dans un ordre aléatoire.

```c
#include <stdlib.h>
#include <time.h>
#include <string.h>
#define SIZE 1000000
#define N 100000000

int main(int argc, char **argv) {
  srand(time(NULL));

   int random = argc > 1 && strcmp(argv[1], "random") == 0;
   volatile int arr[SIZE] = {0};
   for (int i = 0; i < N; i++)
     (void)arr[(i + (random ? rand() : 0)) % SIZE];

   return EXIT_SUCCESS;
}

```

Liste 3. – Parcours d'un tableau et *cache misses*

Le mot clé `volatile` sur le tableau `arr` assure que `gcc` ne supprimera pas les accès en lecture sur ce dernier bien que son contenu soit prévisible et jamais utilisé. Vous pouvez le compiler avec la commande `gcc miss.c -o miss`.

Examinons les performances de notre programme Liste 3 à l'aide de la sous-commande `perf stat`. Cette dernière retourne des statistiques issues des registres *PMU* du processeur. En lançant `perf stat ./miss`, on obtient la sortie:

```
Performance counter stats for './miss':

       116.46 msec task-clock:u              #    0.991 CPUs utilized
            0      context-switches:u        #    0.000 /sec
```

```
            0      cpu-migrations:u        #    0.000 /sec
        1,028      page-faults:u           #    8.827 K/sec
  270,371,076      cycles:u                #    2.322 GHz
1,100,144,340      instructions:u          #    4.07  insn per cycle
  100,029,819      branches:u              #  858.891 M/sec
        2,352      branch-misses:u         #    0.00% of all branches
                   TopdownL1               #  25.1 %  tma_backend_bound
                                           #   1.2 %  tma_bad_speculation
                                           #   0.2 %  tma_frontend_bound
                                           #  73.6 %  tma_retiring

  0.117552543 seconds time elapsed

  0.113162000 seconds user
  0.003969000 seconds sys
```

Tandis que parcourir le tableau `arr` dans un ordre aléatoire conduit à un résultat très différent en terme de performance. En effet la commande `perf stat ./miss random` donne la sortie:

```
Performance counter stats for './miss random':

     1,974.28 msec task-clock:u            #    0.999 CPUs utilized
            0      context-switches:u      #    0.000 /sec
            0      cpu-migrations:u        #    0.000 /sec
        2,003      page-faults:u           #    1.015 K/sec
5,945,316,708      cycles:u                #    3.011 GHz
7,896,922,743      instructions:u          #    1.33  insn per cycle
1,600,032,861      branches:u              #  810.440 M/sec
    3,229,770      branch-misses:u         #    0.20% of all branches
                   TopdownL1               #  60.4 %  tma_backend_bound
                                           #   2.7 %  tma_bad_speculation
                                           #   2.8 %  tma_frontend_bound
                                           #  34.1 %  tma_retiring

  1.975546187 seconds time elapsed

  1.968594000 seconds user
  0.001981000 seconds sys
```

Le parcours est nettement plus lent et le nombre de `cache-misses` explose.

2.9 Masquage des interruptions

Le noyau *Linux* propose une *API* en langage C pour masquer les interruptions.

KVM fournit un support pour la virtualisation des interruptions matérielles sur architecture *ARM* via les interfaces *VGIC v2* [37] et *VGIC v3* [38] et sur architecture *x86* via les interfaces *Intel APICv* pour le module *kvm_intel* et *AMD AVIC* pour le module *kvm_amd*.

DRAFT

Le noyau *Linux* propose plusieurs interfaces pour masquer les interruptions, chacune adaptée à des besoins spécifiques :

- `local_irq_disable()` et `local_irq_save()` : désactivent toutes les interruptions au niveau de l'interface IRQ du CPU.

- `disable_irq()` : désactive une interruption spécifique au niveau du contrôleur d'interruptions.

Ces primitives sont essentielles pour protéger les sections critiques du noyau, mais leur usage doit être parcimonieux. Par exemple, des mesures avec le traceur *irqsoff* ont révélé que dans certains cas, la fonction `console_unlock()` pouvait désactiver les interruptions locales pendant environ 10ms, provoquant des dépassements d'échéance pour les *timers* haute résolution (*hrtimer*).

2.9.1 Approche *PREEMPT_RT*

Pour les systèmes temps réel, le projet *PREEMPT_RT* transforme radicalement la gestion des interruptions dans *Linux*. L'objectif principal est de réduire au minimum le code s'exécutant en contexte d'interruption matérielle, en déplaçant la majeure partie du traitement vers le contexte de threads.

PREEMPT_RT introduit le concept d'*interruptions threadées* (*threaded interrupts*). Dans ce modèle, le gestionnaire d'interruption matérielle réel exécuté par le CPU se limite à quelques dizaines de lignes par architecture et se contente de :
- Masquer la ligne d'interruption.
- Acquitter le contrôleur d'interruptions.
- Réveiller le thread correspondant.

Le traitement réel de l'interruption s'effectue ensuite dans un thread noyau normal, qui peut être préempté comme n'importe quel autre thread. Cette approche résout le problème d'*inversion d'interruptions* où une tâche temps réel haute priorité peut être retardée par une cascade d'interruptions.

De plus, *PREEMPT_RT* remplace de nombreux appels à `local_irq_disable()` par des versions « douces » qui n'impactent pas la latence temps réel de la même manière que le masquage matériel des interruptions.

2.9.2 Stratégies d'atténuation de la latence

Pour minimiser l'impact du masquage des interruptions sur la latence des systèmes temps réel, plusieurs stratégies sont employées :
- *Isolation de CPUs* : dédier certains cœurs au traitement des interruptions et d'autres aux processus temps réel. Cette technique est particulièrement efficace sur les architectures multi-cœurs.
- *Minimisation des sections critiques* : réduire au strict minimum la durée pendant laquelle les interruptions sont masquées.
- *Profilage avec irqsoff* : utiliser le traceur *irqsoff* du noyau pour identifier et mesurer les sections où les interruptions sont désactivées trop longtemps. Ce traceur enregistre la trace avec la latence maximale la plus longue.

Ces techniques sont documentées dans la documentation *Red Hat* pour les systèmes temps réel et sont largement utilisées dans l'industrie pour garantir des latences déterministes.

2.9.3 Support KVM des interfaces VGIC sur ARM

L'hyperviseur *KVM* intégré au noyau *Linux* fournit un support pour la virtualisation des interruptions sur l'architecture *ARM* via l'émulation du *GIC* (*Generic Interrupt Controller*). Ce support est implémenté à travers deux interfaces : *VGIC v2* et *VGIC v3*.

2.9.3.1 Interface VGIC v2

L'interface *VGIC v2* (`KVM_DEV_TYPE_ARM_VGIC_V2`) permet de créer un contrôleur d'interruptions virtuel compatible *GICv2* pour les machines virtuelles [37]. Une seule instance de *VGIC* peut être instanciée par machine virtuelle et agit comme contrôleur d'interruptions principal.

Cette interface expose deux régions mémoire :
- Le *Distributor* : accessible via `KVM_VGIC_V2_ADDR_TYPE_DIST`, aligné sur 4 Ko et couvrant 4 Ko.
- L'interface *CPU* virtuelle : accessible via `KVM_VGIC_V2_ADDR_TYPE_CPU`, alignée sur 4 Ko et couvrant 8 Ko.

Le nombre d'interruptions configurables va de 64 à 1024, par incréments de 32. L'implémentation correspond à un *GICv2 sans extensions de sécurité*. La limitation principale de cette interface est le nombre maximal de 8 *CPUs* virtuels par machine virtuelle.

Les systèmes hôtes disposant d'un *GICv3* avec support de compatibilité matérielle peuvent également créer des invités *GICv2* via cette interface.

2.9.3.2 Interface VGIC v3

L'interface *VGIC v3* (`KVM_DEV_TYPE_ARM_VGIC_V3`) permet d'émuler un *GICv3* pour les machines virtuelles [38]. La création d'un invité *GICv3* nécessite un hôte disposant d'un *GICv3*. Il n'est pas possible de créer à la fois un *GICv2* et un *GICv3* sur la même machine virtuelle.

Cette interface expose les régions mémoire suivantes :
- Le *Distributor* : aligné sur 64 Ko et couvrant 64 Ko.
- Les *Redistributors* : deux pages de 64 Ko par *VCPU*, contiguës en mémoire.

Par rapport à la version 2, le *VGIC v3* apporte plusieurs améliorations :
- Support de l'*affinity routing* permettant jusqu'à 512 *VCPUs*.
- Accès via registres système au lieu de *MMIO*.
- Support de l'*ITS* (*Interrupt Translation Service*) pour les interruptions *MSI*.

2.9.3.3 Support de l'ITS

L'*ITS* (*Interrupt Translation Service*) est une extension optionnelle permettant l'injection d'interruptions *MSI(-X)* dans les machines virtuelles [39]. Sa création nécessite un hôte *GICv3* mais ne dépend pas de la présence de contrôleurs *ITS* physiques.

Plusieurs contrôleurs *ITS* peuvent être créés par machine virtuelle, chacun avec une région *MMIO* distincte. L'adresse de base doit être alignée sur 64 Ko et couvrir 128 Ko. L'*ITS* virtuel permet de sauvegarder et restaurer l'état des tables d'interruptions depuis la *RAM* de l'invité, facilitant la migration des machines virtuelles.

2.10 Watchdog

Cette section décrit le support pour des *watchdogs* matériels dans le noyau *Linux* ainsi que le support pour des *watchdogs* logiciels par *systemd*.

2.10.1 API bas niveau

Linux offre une *API* unifiée pour interagir avec les *watchdogs* matériels directement dans l'espace utilisateur [40]. Cette communication se fait via un pseudo-périphérique `/dev/watchdog`. À l'ouverture ce périphérique, le *watchdog* s'active et attend d'être réinitialisé dans

un certain délai de réponse. Une façon simple de le réinitialiser est d'écrire des données quelconques dans le périphérique `/dev/watchdog`. Quant au délai de réponse, il est configurable via l'appel système `ioctl`. Lorsque le périphérique est fermé, le *watchdog* est désactivé. La Liste 4 contient un exemple simple d'utilisation de *watchdog*.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/watchdog.h>

int main(void) {
  int fd = open("/dev/watchdog", O_WRONLY);
  if (fd == -1)
    exit(EXIT_FAILURE);

  // Configure le timeout à 20 secondes.
  int timeout = 20;
  if (ioctl(fd, WDIOC_SETTIMEOUT, &timeout) == -1)
    goto failed;
  printf("Watchdog initialisé\n");

  // Réinitialise le watchdog toutes les 10 secondes.
  while (1) {
    if (write(fd, "\0", 1) != 1)
      goto failed;
    printf("Watchdog rechargé\n");
    sleep(10);
  }

  close(fd);
  return EXIT_SUCCESS;

failed:
  close(fd);
  return EXIT_FAILURE;
}

```

Liste 4. – Exemple d'interaction avec un *watchdog* sous *Linux*.

Toutefois, dans un usage réel, il est souhaitable que le *watchdog* ne puisse pas être désactivé accidentellement. En effet, si par exemple l'appel système `write` échoue dans le code ci-dessus, le descripteur de fichier `fd` sera libéré, ce qui provoquera l'arrêt du watchdog. Pour cette raison, certains pilotes de *watchdogs* permettent de ne pas être désactivables ou seulement par l'écriture d'une séquence de caractères magique sur le périphérique `/dev/watchdog`.

2.10.2 Support dans *systemd*

Pour la plupart des distributions *GNU/Linux* modernes, l'utilisation des watchdogs est simplifiée via le gestionnaire de services *systemd*. Ce dernier permet aussi d'utiliser des *watchdogs* logiciels dans les services. Pour ce faire, il suffit de modifier le démon afin qu'il notifie régulièrement *systemd* via l'appel `sd_notify("WATCHDOG=1")`. Le délai de réponse est quant à lui transmis par la variable d'environnement `WATCHDOG_USEC`. La Liste 5 contient un exemple

d'un démon `/usr/bin/foo` ainsi modifié qui sera automatiquement relancé par *systemd* s'il ne notifie pas ce dernier dans un délai de 30 secondes.

```
1  [Unit]
2  Description=Exemple
3
4  [Service]
5  ExecStart=/usr/bin/foo
6  WatchdogSec=30s
7  Restart=on-failure
8  StartLimitInterval=5min
9  StartLimitBurst=4
10  StartLimitAction=reboot-force
11
```

Liste 5. – Exemple de service *systemd* avec *watchdog*.

2.11 Temps de démarrage

Il existe de nombreuses techniques pour réduire le temps de démarrage d'un système *Linux*. Ces techniques concernent aussi bien le *bootloader*, l'initialisation du noyau ou l'initialisation de l'*espace utilisateur*.

- Pour le *bootloader*, on peut n'initialiser que les périphériques indispensables et optimiser le code assembleur.
- Pour l'initialisation du noyau, on peut utiliser une image non compressée, désactiver les fonctionnalités inutiles pour notre usage et en particulier les outils de profilages.
- Pour l'initialisation de l'*espace utilisateur*

L'initialisation de l'*espace utilisateur* est généralement l'étape la plus longue et donc la phase à optimiser en priorité.

Dans l'article [41], les auteurs étudient des méthodes d'optimisation pour le temps démarrage d'un système *Android* exécuté sur un dispositif embarqué dans une automobile. Ils parviennent à réduire de 65% le temps de démarrage en passant de 29,7s à 10,1s. Sur le noyau *Linux* lui-même, ils obtiennent une amélioration d'un facteur 4.

Dans le mémoire [42], les auteurs comparent et optimisent différents *init systems* à la fois dans un environnement émulé avec *QEMU* et dans une distribution *GNU/Linux* dédiée à l'embarqué. Leur conclusion est qu'une réduction substantielle du temps démarrage de l'*espace utilisateur* est possible via leurs méthodes d'optimisation et que le choix du *init system* est déterminant mais dépendant de l'environnement d'exécution.

(MOVE) Le project Yocto [43] est un projet libre offrant la possibilité de créer sa distribution *Linux* dédiée à l'embarqué.

2.11.1 Profilage de `systemd`

Le programme `systemd` fournit un outil intéressant de profilage baptisé `systemd-analyze`. Il permet d'analyser le temps de démarrage du système et des sessions utilisateurs afin d'identifier des goulots d'étranglement. Détaillons quelques unes des ses commandes:

- `systemd-analyze time`: affiche différents temps relatifs au démarrage du système.
- `systemd-analyze blame`: affiche le temps de démarrage des différents services. Il est à noter que certains services pouvant s'exécuter en parallèle, l'analyse de sa sortie requière une certaine prudence.

- `systemd-analyze dot`: produit un graphe de dépendance des services.
- `systemd-analyze plot`: produit une frise chronologique du démarrage des services.

Par exemple, la commande suivante:

```
$ systemd-analyze time
```

produit une sortie de la forme:

```
Startup finished in 7.274s (firmware) + 3.428s (loader) + 1.007s (kernel)
+ 11.451s (initrd) + 7.587s (userspace) = 30.749s
multi-user.target reached after 7.321s in userspace.
```

Le dernier temps indique le délais écoulé avant que l'*espace utilisateur* ne soit disponible, ce qui correspond en général à l'affichage d'un prompteur pour ouvrir une session. On retrouve aussi d'autres informations intéressantes:

- *Firmware*: Temps de chargement des firmwares via le BIOS.
- *Load*: Temps écoulé dans le *bootloader*.
- *Kernel*: Temps de chargement et d'initialisation du noyau.
- *Initrd*: Temps d'initialisation de la *RAM disk*.
- *Userspace*: Temps écoulé pour lancer tous les services de l'*espace utilisateur*.

2.12 Maintenabilité

Le noyau `Linux` est un logiciel libre distribué sous licence `GPL-2.0` avec l'exception *syscall* qui stipule qu'un logiciel utilisant le noyau `Linux` au travers des appels systèmes n'est pas considéré comme une œuvre dérivée et peut être distribué sous une licence qui n'est pas compatible avec la GPL, y compris une licence propriétaire. Plus d'informations sont disponibles dans le dossier `LICENSES` des sources du noyau `Linux`.

3 MirageOS

MirageOS en bref

- **Type** : LibOS
- **Langage** : OCaml (99%)
- **Architectures** : x86-64, ARM v8, PowerPC[11]
- **Usage principal** : Cloud computing, applications réseau, systèmes embarqués, spatial
- **Points forts** : Sécurité renforcée (surface d'attaque réduite, langage sûr), taille minimale, temps de démarrage rapide, modularité
- **Limitations** : Portabilité limitée sans hyperviseur, débogage complexe, pas d'interface POSIX
- **Licences** : ISC (majoritaire) + LGPLv2 (certaines parties)
- **Projet notable** : SpaceOS (déployé dans l'espace en 2025)

Au tournant des années 2010, le *cloud computing* s'impose comme une solution pour réduire le coût et externaliser la maintenance des services web. Cette innovation a été rendue possible par la démocratisation au début du siècle de la virtualisation sur architecture *x86*. À cette époque, la majorité des *VMs* exécutent un service web dans un *GPOS* complet. Cette approche présente l'avantage de circonscrire au système d'exploitation les modifications nécessaires pour la virtualisation, tout en bénéficiant de l'isolation offerte par l'hyperviseur. En contre partie, la pile logicielle est grandement complexifiée comme l'illustre la Fig. 3.

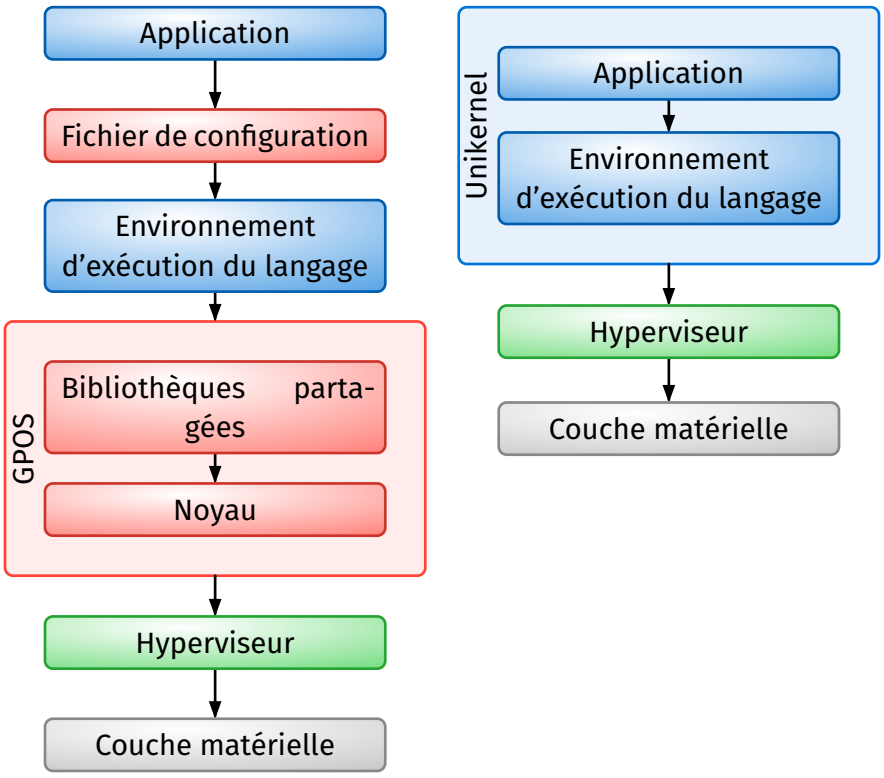

Fig. 3. – Comparaison entre l'approche *GPOS* et l'approche *unikernel*.

En particulier, certains mécanismes d'isolation comme l'ordonnanceur de tâches sont dupliqués entre l'hyperviseur et le noyau exécuté dans la *VM*. De plus, l'introduction d'un *GPOS* augmente considérablement la surface d'attaque *TCB (Trusted computing base)* et les sources de bugs potentiels. Cela est d'autant plus vrai que ces *GPOS* sont souvent écrits dans un langage

[11] Support limité à l'environnement d'exécution *spt* du projet *solo5*.

de programmation[12] n'offrant que peu de garantie du point de vue des types et de la mémoire. C'est de ces deux constats que naît le projet *MirageOS*.

Le projet est initié en 2009 au sein du laboratoire *Computer Laboratory* de l'université de Cambridge sous la houlette de Anil Madhavapeddy [44]. Il est de nos jours maintenu par la *MirageOS Core Team* composée d'universitaires et d'ingénieurs du secteur privé (*Tarides*, *IBM Research*, ...). *MirageOS* fait parti des projets soutenus par le *Xen Project* [45] et un grand nombre de ces contributeurs ont également participé au projet *Xen*.

MirageOS adopte une approche de type *LibOS*. Au lieu de fournir un environnement d'exécution préconfiguré pour les services, *MirageOS* se présentent sous la forme d'une collection de bibliothèques. Ces dernières sont écrites majoritairement en *OCaml*, un langage de programmation de haut niveau offrant la sûreté des types et équipé d'un ramasse-miette. La configuration et l'ensemble des bibliothèques nécessaires au service sont liés durant la compilation pour produire une image appelée *unikernel*. Cet *unikernel* peut alors être exécuté dans divers environnements, voir la sous-section 3.4. Cela conduit à une simplification de la pile logicielle comme illustré dans Fig. 3. L'approche *unikernel* présente de nombreux avantages:

- Une plus petite *TCB* à la fois par la réduction de le taille du code source et l'utilisation d'un langage de programmation sûr,
- Une amélioration des performances et notamment du temps de démarrage,
- Une réduction de la taille des exécutables produits,
- Un profilage simplifié par la suppression d'une couche logicielle volumineuse.

3.1 Tutoriel

Pour faciliter l'exécution des exemples de ce chapitre, une image `docker` est disponible dans le dossier `miragos/` du dépôt. Cette image contient tout le nécessaire pour compiler des images avec MirageOS. Pour installer l'image, tapez:

```
$ make -C irageos setup
```

Vous pouvez accéder au shell du `docker` en tapant:

```
$ make -C irageos shell
```

3.2 Architectures supportées

Pour qu'une architecture soit supportée par *MirageOS*, il est nécessaire qu'elle soit une cible de compilation du compilateur OCaml. Le compilateur *OCaml 4* supporte les architectures suivantes: *x86-32*, *x86-64*, *ARMv7*, *ARMv8*, *PowerPC*, *SPARC* et *MIPS*. Toutefois le support en natif[13] pour les architectures 32-bits a été supprimé à partir d'*OCaml 5*. Il n'est donc pas recommandé d'utiliser *MirageOS* sur de telles plateformes.

En pratique, les *unikernels* produits par *MirageOS* sont rarement exécutés en *bare-metal* mais plutôt dans une partition d'un hyperviseur. Il est donc nécessaire que cet hyperviseur supporte

[12] La vaste majorité de la programmation système est en langage C qui n'offre pratiquement aucune garantie mémoire et dont la sémantique est complexe, notamment sur les architectures *SMP*.

[13] Il subsiste en *bytecode*, ce qui n'est pas pertinent ici puisqu'il faudrait porter la machine virtuelle d'OCaml pour obtenir probablement des performances médiocres.

une des architectures ci-dessus et que l'environnement d'exécution de *MirageOS* ait été porté dessus.

Porter une *LibOS* sur un hyperviseur étant une tâche répétitive, le projet *solo5* vise à mutualiser les efforts en fournissant une couche d'abstraction logicielle entre le *RTE* de *MirageOS* et les différentes *API* d'hyperviseurs et de *GPOS*. À l'heure actuelle, *solo5* semble n'offrit qu'un support pour les systèmes sur *x86-64*, *ARMv8* et *PowerPC* (limité à l'environnement d'exécution *spt*).

Nous considérons donc que seuls ces architectures sont officiellement supportées par le projet *MirageOS*.

3.3 Support multi-processeur

Les *unikernels* de *MirageOS* étant souvent exécutés au-dessus d'un hyperviseur, la question du support d'architectures multi-processeur revient à déterminer si ces images peuvent tirer parti du parallélisme de ces processeurs. En premier lieu, le *RTE* d'*OCaml* doit permettre le parallélisme du code *OCaml*.

Jusqu'à *OCaml 4*, le *runtime OCaml* utilisait un verrou global assurant que le code *OCaml* ne puisse jamais être exécuté en parallèle. Ce verrou permettait de garantir la préservation d'invariants internes, notamment au niveau du ramasse-miette. Un moyen de bénéficier malgré tout du parallélisme était d'écrire le code à paralléliser en C puis de l'interfacer avec le code *OCaml*.

Cette solution n'a pas été retenue par les développeurs de *MirageOS* qui souhaitaient bénéficier de la sûreté des types du langage *OCaml*. Les services web étant généralement des applications *I/O bound*[14], une approche à base de programmation asynchrone a été retenue pour permettre l'entrelacement de fils d'exécutions. *MirageOS* a donc été implémenté avec la bibliothèque de *threads* coopératifs *Lwt* [46], [47].

[14]Une application est qualifiée *I/O bound* si son temps d'exécution est dominé la vitesse de traitement des entrées/sorties.

Aparté: Bibliothèque *Lwt*

Lwt est une bibliothèque de *threads* coopératifs écrite en *OCaml* dans le cadre du projet *Ocsigen* [48]. Elle simplifie la programmation asynchrone en proposant un style de programmation monadique via des promesses.

Par exemple, dans le code suivant, deux *threads* légers sont lancés pour afficher un message après un décompte grâce à la fonction `Mirage_sleep.ns`:

```
open Lwt.Infix

let start () =
  Lwt.join
    [
      ( Mirage_sleep.ns (Duration.of_sec 1) >|= fun () →
        Logs.info (fun m → m "Heads") );
      ( Mirage_sleep.ns (Duration.of_sec 2) >|= fun () →
        Logs.info (fun m → m "Tails") );
    ]
  >|= fun () → Logs.info (fun m → m "Finished")
```

La fonction `Lwt.join` crée une promesse qui ne sera résolue que lorsque les deux *threads* auront terminé leur travail. Lorsque cette promesse est résolue, on peut afficher le message *Finished* dans le terminal.

Lorsque le parallélisme est vraiment nécessaire, par exemple si un service doit effectuer une tâches lourdes en calcul, une solution est d'exécuter plusieurs instances du même *unikernel* et de les synchroniser via les *IPC (Inter-Process Communication)* de l'hyperviseur sous-jacent. Cette solution a été mise en pratique sur l'hyperviseur *Xen* [49], [50].

La version 5 d'*OCaml* introduit deux nouvelles fonctionnalités utiles pour *MirageOS*:

- D'une part le verrou global du *RTE* d'*OCaml* a été supprimé. L'exécution en parallèle de code *OCaml* est donc possible via le concept de *domaine* [51]. Cet ajout s'est fait aux prix d'une complexification du modèle mémoire d'OCaml mais néanmoins maîtrisé [52],
- D'autre part l'introduction des effets algébriques facilite la création d'une bibliothèque de *threads* coopératifs. Il est désormais possible de réaliser une telle bibliothèque sans utiliser un style monadique, ni allouer des clôtures sur le tas pour représenter les piles d'exécution des *threads*.

Un effort est en cours pour porter *MirageOS* sur *OCaml 5* [53], ce qui devrait conduire à une amélioration des performances des *unikernels* sur les architectures *SMP* et à une simplification de leur architecture lorsque le parallélisme est nécessaire pour des raisons de performance.

3.4 Environnements d'exécution

Les *LibOS* souffrent généralement d'un problème de portabilité car elles doivent être adaptées à chaque environnement matériel spécifique. Cette problématique est largement atténuée par l'usage d'un hyperviseur qui offre une couche d'abstraction matérielle standardisée, facilitant ainsi le déploiement des *unikernels* sur différentes plateformes.

Les *unikernels* produits par *MirageOS* peuvent tourner sur les hyperviseurs *Xen*, *KVM* de *Linux*, *BHyve* de *FreeBSD* et *OpenBSD VMM*. Il supporte également le noyau de séparation *Muen*. Finalement, il est possible d'exécuter une image dans un environnement *UNIX* comme une distribution *GNU/Linux* ou *macOS*, ce qui est particulièrement utile pour débogage.

À l'origine *MirageOS* n'était supporté que par *Xen*. Lors de son portage vers *KVM* par *IBM Research*, le projet *solo5* a été lancé afin de mutualiser les efforts d'un tel portage. Aujourd'hui *solo5* propose un environnement d'exécution standardisé pour différentes *LibOS* et ciblant différents *hyperviseurs*.

Tutoriel: choisir l'environnement d'exécution

Le choix de l'environnement d'exécution se fait au moment de la configuration du projet via la commande:

```
$ mirage configure -t ENV
```

où `ENV` peut désigner les valeurs suivantes: `unix`, `macosx`, `xen`, `virtio`, `hvt`, `muen`, `qubes`, `genode`, `spt`, `unikraft-firecracker` ou `unikraft-qemu`.

- L'option `xen` permet l'exécution dans un domaine de *Xen*. Il s'agit de l'environnement original du projet *MirageOS*. En production, on exécute généralement le noyau minimaliste *Mini-OS* dans le *dom0* de *Xen* [54].
- L'option `unix` permet d'exécuter l'*unikernel* dans *KVM*.
- Les options `unix` et `macosx` permettent d'exécuter l'*unikernel* dans une distribution *GNU/Linux*, respectivement *macOS*. C'est un atout précieux pour le débogage et le profilage de l'application mais ne correspond généralement pas à l'environnement d'exécution en production.
- Les options `unikraft-firecracker` et `unikraft-qemu` ont été ajoutées récemment au projet [55]. Elles permettent d'exécuter l'*unikernel* dans un environnement *unikraft*.

Dans les sections suivantes, nous exécuterons les exemples dans l'hyperviseur *Xen*. Ce choix est motivé par le fait qu'il s'agit aujourd'hui du cas d'usage fréquent.

3.5 Partitionnement

MirageOS n'offre pas de partitionnement temporel ou spatial. Cette tâche incombe à un noyau de séparation dans lequel l'*unikernel* est exécuté, typiquement un hyperviseur comme *Xen*. Lorsque l'on souhaite isoler plusieurs services *MirageOS*, l'usage est d'exécuter ces services dans des *unikernels* distincts et de les faire communiquer via les *IPC* de l'hyperviseur.

En particulier, si vous utiliser *Xen* comme noyau de séparation, vous pouvez utiliser la bibliothèque *ocaml-vchan* [56] de *MirageOS* pour communiquer entre deux *unikernels*.

À notre connaissance, *MirageOS* n'a jamais été utilisé dans un contexte temps réel. Le principal obstacle vient du ramasse-miette d'OCaml qui n'offre pas de garanties déterministes. Quant à la bibliothèque *Lwt*, elle n'a pas été conçue pour cet usage puisque les tâches doivent rendre la main volontairement à l'ordonnanceur *Lwt*. Si vous souhaitez exécuter une tâche temps réel en parallèle d'un *unikernel MirageOS*, il faudra avoir recours à un hyperviseur temps réel et exécuter cette tâche dans une partition distincte.

3.6 Corruption de la mémoire

La gestion de la corruption de la mémoire est généralement déléguée à l'environnement d'exécution de l'*unikernel*.

Pour la journalisation des événements, il suffirait de se reposer sur le support offert par le système exécuté dans *dom0*. Par exemple, dans le cas de *Xen* avec un noyau *Linux* dans *dom0*,

on pourrait utiliser les sous-systèmes décrits dans la sous-section 2.6 et les *IPC* de *Xen* pour récupérer ces informations dans l'*unikernel*.

3.7 Perte du flux d'exécution

Le *typechecker OCaml* garantit à la compilation la sûreté des types. Une donnée d'un type A ne peut pas être accédée par une série d'instructions attendant une donnée d'un autre type B. En particulier la représentation machine des données est compatible avec celle attendue par les instructions qui y accèdent.

Le *runtime OCaml* garantie à l'exécution l'absence de dépassement de tampon en insérant des vérifications aux endroits appropriés. En cas d'erreur, une exception est levée par le *runtime* et met généralement fin à l'exécution du programme.

Ces deux garanties éliminent une grande partie des vecteurs d'attaques de la pile d'exécution. Bien sûr, d'autres stratégies d'atténuation peuvent être mise en place au niveau de l'hyperviseur.

3.8 Écosystème

Le profilage et le débogage d'un *unikernel* dépend fortement de l'environnement dans lequel il est exécuté. Pour *MirageOS*, le cas le plus favorable est celui d'une distribution *GNU/LINUX*, puisqu'il y existe pléthore d'outils, voir la sous-section 2.8. Le manuel *OCaml* contient également un guide pour le profilage avec *perf* de programmes *OCaml* [57].

Il existe aussi quelques outils spécifiques à *MirageOS* ou au langage *OCaml*:

- *mirage-monitoring* [58]: Outil de monitoring pour les *unikernels* produits par *MirageOS*. Il supporte le *dashboard Telegraph* de *Grafana*,
- *memtrace* [59]: Profileur mémoire pour le langage *OCaml* développé par l'entreprise *Janestreet*. Il permet de générer une trace compacte de l'utilisation de la mémoire par un programme *OCaml*. La trace produite peut ensuite être explorée avec *memtrace_viewer* [60]. Il existe une bibliothèque *MirageOS memtrace-mirage* [61] qui offre un support pour cet outil dans un *unikernel*,
- *memtrace_viewer* [60]: Outil d'exploration de traces produites par *memtrace*,
- *mirage-profile* [62]: Profileur pour les programmes *OCaml* utilisant la bibliothèque *Lwt* et en particulier les *unikernels* de *MirageOS*. Sa conception et des exemples d'utilisation sont exposés dans un article de blog [63]. Le projet ne semble plus être maintenu,
- *mirage-trace-viewer* [64]: Outil de visualisation des traces produites par *mirage-profile* ou *mirage-trace-dump-xen*.

3.8.1 Profilage mémoire avec `memtrace-mirage`

L'exemple Liste 7 illustre l'utilisation de `memtrace-mirage` dans un *unikernel*. La fonction `start` est le point d'entrée de l'*unikernel*. Cette fonction commence par établir un socket `TCP` à l'adresse `10.0.0.1:24`[15]. Lorsqu'un client établit une connexion, `memtrace` est lancé jusqu'à ce que la connexion soit interrompue. La fonction `alloc` est exécutée de façon concurrentielle afin de produire un grand nombre d'allocations. L'exécution de l'*unikernel* se termine après 100 secondes.

[15] L'adresse `10.0.0.1` est l'adresse *IP* par défaut utilisée par la bibliothèque `mirage-tcp-ip`.

```
open Mirage

let main =
  main "Unikernel.Make"
    ~packages:[
      package "duration";
      package "memtrace-mirage"
    ]
    (stackv4v6 @→ job)

let stackv4v6 = generic_stackv4v6 default_network

let () = register "memtrace" [ main $ stackv4v6 ]

```

Liste 6. – Configuration de l'*unikernel*

```
open Lwt.Infix

module Make (S : Tcpip.Stack.V4V6) = struct
  module Memtrace = Memtrace.Make (S.TCP)

  let rec alloc i =
    if i < 0 then Lwt.return_unit
    else
      let a = Array.init 1_000_000 (fun i → i * i) in
      Array.sort Int.compare a;
      Lwt.pause () >>= fun () → alloc (i - 1)

  let start s =
    S.TCP.listen (S.tcp s) ~port:1234 (fun f →
      match Memtrace.Memprof_tracer.active_tracer () with
      | Some _ → S.TCP.close f
      | None →
        let tracer =
          Memtrace.start_tracing ~context:None ~sampling_rate:1e-4 f
        in
        Lwt.async (fun () →
          S.TCP.read f >|= fun _ →
          Memtrace.stop_tracing tracer);
        Lwt.return_unit);
    alloc 10
end


```

Liste 7. – Exemple d'utilisation de `memtrace-mirage`

Voyons comment exécuter cet exemple pas à pas. On commence par créer l'unikernel à l'aide de l'image docker, puis on lance cet unikernel dans un domaine de *Xen*:

```
$ make -C mirageos build-memtrace
$ cd mirageos/unikernels/memtrace
$ sudo xl create memtrace.xl -c
```

On peut alors récupérer la trace produite par `memtrace` en établissant dans autre terminal une connexion sur `10.0.0.2:1234`:

```
$ nc 10.0.0.2 1234 > mirageos/trace
```

Finalement, on peut lancer une instance de `memtrace-view`:

```
$ make -C mirageos memtrace-viewer
```

Cette commande lance un serveur web écoutant sur l'adresse `localhost:8080`.

> **Attention: incompatibilité avec OCaml 5**
>
> Le module `Gc.Memprof` nécessaire à `memtrace` ne fonctionne plus en OCaml 5 car le fonctionnement du ramasse-miette a été changé en profondeur. Des efforts sont en cours pour restaurer cette fonctionnalité dans une version ultérieure du compilateur OCaml.

3.9 Masquage des interruptions

Il ne semble pas exister de bibliothèque MirageOS pour le masquage d'interruption. Cela n'est pas étonnant puisque le masquage des interruptions est plutôt utilisé dans un contexte multi-thread avec de la préemption. Or le modèle de concurrence de MirageOS est souvent mono-thread et sans préemption. La préemption est assurée au niveau de l'hyperviseur sur lequel l'*unikernel* s'exécute.

3.10 Watchdog

MirageOS ne semble pas offrir d'*API* en OCaml pour interagir avec un *watchdog*. Le support est donc dépendant de l'environnement dans lequel l'image est exécutée.

Dans le cas de l'hyperviseur *Xen*, il suffit d'appeler les fonctions C de la bibliothèque *xencontrol* comme illustré dans la Liste 10 à travers des *bindings* en OCaml. De tels *bindings* existent déjà dans le dossier `tools/ocaml/` du dépôt *Xen*.

3.11 Temps de démarrage

Le temps de démarrage de *MirageOS* est un enjeu important pour ses applications dans le *cloud computing*. Ainsi le temps de démarrage d'un *unikernel* produit par *MirageOS* a fait l'objet de plusieurs études.

Dans l'article fondateur [65], les auteurs ont étudiés le temps de démarrage d'*unikernels MirageOS* et d'un serveur *Apache* sous *Debian* virtualisés dans des partitions *Xen*. Les *unikernels* démarraient deux fois plus vite que la combinaison *Debian/Apache*. Un gain substantiel vient de l'optimisation de la *toolstack* de *Xen*, permettant aux *unikernels* de démarrer en seulement 50ms.

Dans [49], les auteurs ont étudié le temps de démarrage d'*unikernels MirageOS* sur un hyperviseur *Xen* avec une pile logicielle optimisée. Ils sont parvenus sur des temps de démarrage à froid inférieurs à 350 ms sur *ARM* et 30ms sur *x86*.

En conclusion, le temps de démarrage de *MirageOS* peut être rendu très faible et négligeable face du démarrage de la partition elle-même. L'enjeu est donc d'optimiser le temps de démarrage de cette dernière.

3.12 Qualifications et certifications

À notre connaissance, *MirageOS* n'a pas fait l'objet de certifications. L'objectif premier de *MirageOS* est davantage la sécurité que la sûreté de fonctionnement, il pourrait faire donc l'objet de certifications comme celles décrites dans les Critères Communs (niveau *EAL*) mais rien de tel semble avoir été entrepris jusqu'à présent.

3.13 Maintenabilité

MirageOS est en majorité écrit en OCaml, un langage de haut niveau qui offre de bonne garantie du point de vue de la sûreté des types et de la mémoire. À ce jour le dépôt https://github.com/mirage/mirage est constitué à 99% de code OCaml pour un total de 9075 *SLOC*.

Toutefois la totalité de l'unikernel ne provient pas de la compilation de codes *OCaml*. Il subsiste plusieurs parties en langage C et notamment:
- L'environnement d'exécution du langage OCaml est écrit en C. Cela inclut en particulier son ramasse-miette,
- Quelques bibliothèques en C comme *GMP* au travers de *Zarith*. Leur réécriture en OCaml est théoriquement possible mais nécessiterait un effort considérable en pratique,
- Les pilotes doivent être écrits dans un langage bas niveau comme le langage C.

Le code de *MirageOS* est publié sous la licence *ISC* avec certaines parties sous licence *LGPLv2*. L'utilisation d'une licence open-source permissive comme *ISC* est nécessaire car l'*unikernel* produit par *MirageOS* est lié statiquement avec les bibliothèques. Grâce à cette licence, vous n'avez pas les contraintes des licences *GPL* lorsque vous distribuez le binaire de votre *unikernel*.

MirageOS a été utilisé dans plusieurs projets d'envergures. Récemment l'entreprise *Tarides* a développé le système d'exploitation *SpaceOS* pour des applications spatiales et satellitaires [66], [67]. Il s'agit d'une solution sécurisée et efficace pour les satellites multi-utilisateurs et multi-missions, construite sur la technologie des *unikernels*.

SpaceOS a été conçu en partenariat avec plusieurs organisations du secteur spatial (L'*ESA* (*European Space Agency*), Le *CNES*, *Thales Alenia Space* ,*OHB* , *Eutelsat*, Le *Singapore Space Agency*).

Le 15 mars 2025, *OCaml* a été lancé dans l'espace à bord de la mission *Transporter-13*. *DPhi Space* a embarqué son ordinateur *Clustergate* sur ce vol, et l'équipe *SpaceOS* a déployé un logiciel basé sur *OCaml 5* sur le satellite. Cette mission a démontré la viabilité des *unikernels MirageOS* pour les applications spatiales en conditions réelles.

Les principaux avantages de *SpaceOS* incluent notamment:
- Une réduction de taille d'un facteur 20 par rapport à un déploiement basé sur des conteneurs *Linux*,
- Une sécurité accrue grâce à l'utilisation d'un langage à gestion mémoire sûre (*OCaml*),
- Une architecture modulaire permettant de compiler uniquement les fonctionnalités nécessaires du système d'exploitation.

Ces résultats ont valu à *SpaceOS* une reconnaissance industrielle significative, notamment le prestigieux *Airbus Innovation Award* lors de la *Paris Space Week* 2024.

4 PikeOS

PikeOS en bref

- **Type** : RTOS + Hyperviseur type 1 (inspiré du micronoyau L4)
- **Langage** : C
- **Architectures** : x86-64, ARMv7/v8, PowerPC, RISC-V, SPARC
- **Usage principal** : Systèmes critiques (aéronautique, automobile, défense, médical)
- **Points forts** : Conçu pour la certification, paravirtualisation + HVM, support multi-criticité
- **Limitations** : Propriétaire, coût de licence
- **Licences** : Propriétaire (SYSGO/Thalès)
- **Certifications** : Kits disponibles pour DO-178B/C, IEC 61508, ISO 26262

PikeOS est un hyperviseur temps réel dédié à l'embarqué.

Depuis la fin des années 90, l'entreprise *SYSGO* développait son propre micronoyau baptisé *P4* et inspiré du noyau *L4* de Jochen Liedtke [68]. À cette époque, l'usage de micronoyaux dans l'embarqué est envisagé du fait de l'augmentation des performances et du besoin croissant de fiabilité dans les logiciels embarqués. Contrairement à la majorité des implémentations du micronoyau *L4* de l'époque, *P4* était donc conçu pour l'embarqué et était totalement préemptif pour des usages temps réel. Cette expérience a permis aux ingénieurs de *SYSGO* d'identifier des limites dans la conception du noyau *P4*, principalement héritées de l'*API* de *L4*. Ces limites concernaient notamment l'isolation temporelle et spatiale.

Les ingénieurs de *SYSGO* ont alors développé un nouveau micronoyau *PikeOS* avec pour objectif une meilleure isolation afin qu'il soit utilisable dans les systèmes de criticité mixte. L'idée était de développer un hyperviseur pour assurer l'isolation de partition. La plateforme a également été pensée pour faciliter la certification. La première version est commercialisée en 2005.

En 2006, *SYSGO* ajoute le support de l'architecture *ARM*.

En 2008, l'avioneur *Airbus* choisit *PikeOS* comme plateforme de référence *DO-178B* pour le système *FSA-NG* de l'*A350*.

En 2012, l'entreprise *Thales* acquière *SYSGO*.

En 2022, *PikeOS 5.1.3* obtient la certification de sécurité *Critères Communs* au niveau *EAL 5+* pour les architectures *x86-64*, *ARMv8* et *PowerPC* [69].

4.1 Architectures supportées

PikeOS supporte les architectures suivantes: *x86-64*, *ARMv7*, *ARMv8*, *PowerPC*, *RISC-V* et *SPARC*. Le support pour l'architecture *ARM* existe depuis 2006. En particulier, *PikeOS* supporte les architectures *SPARC LEON3* et *LEON4* utilisés dans le spatial.

Quant à son hyperviseur, il propose un support pour la virtualisation matérielle sur les architectures *ARM v7* [70] et *x86-64* [71].

Le support matériel se fait via des *BSP* et *SYSGO* propose le développement de nouveau *BSP* à la demande.

4.2 Support multi-processeur

4.2.1 Architectures *SMP*

PikeOS dispose d'un support les architectures *SMP*.

Le support *SMP* a été amélioré dans *PikeOS 4.2*. Afin de garantir un déterminisme suffisant et des estimations *WCET (Worst Case Execution Time)* suffisamment fines, *PikeOS* avait recours à un verrou global similaire au *BKL* de *Linux* (voir la sous-section 2.3.1). Ce verrou assurait que les sections critiques du micronoyau ne pouvaient pas s'exécuter en parallèle. Depuis la version *4.2*, *PikeOS* utilise un système de verrouillage plus fin qui permet aux appels systèmes de s'exécuter en parallèle s'ils n'utilisent pas des ressources en commun [72].

PikeOS peut également être configuré pour invalider les caches et la *TLB (Translation Lookaside Buffer)* lorsqu'il bascule d'une partition temporelle à une autre [73].

4.2.2 Architectures *AMP*

L'entreprise *SYSGO* propose une version spéciale de *PikeOS* baptisée *PikeOS for MPU*. Cette édition de l'hyperviseur est dédiée aux plateformes *MPSoC* équipées de *MPU* et offre en particulier un support pour des architecture *AMP*. Il supporte les architectures *ARMv7-R*, *ARMv8-R* et dispose de *BSP* les *MPSoC NG-Ultra* et *AMD Zynq Ultrascale+*.

4.3 Partitionnement

PikeOS offre une solide isolation spatiale et temporelle de ses partitions. Sa conception est inspirée de la norme avionique *ARINC 653* pour les systèmes temps réel. Toutefois *PikeOS* ne se conforme pas à celle-ci.

4.3.1 Partitionnement spatial

Le noyau de séparation de *PikeOS* garantit une isolation stricte de la mémoire entre les partitions [74]. Cette isolation a fait l'objet d'une vérification formelle au niveau du code source [75]. La séparation mémoire a été prouvée formellement en décomposant les exigences de haut niveau en propriétés fonctionnelles du gestionnaire de mémoire sous forme d'assertions vérifiables.

La version standard de *PikeOS* utilise un *MMU* pour créer des espaces d'adressages virtuels et contrôler les accès mémoires. Pour les systèmes embarqués dépourvus de *MMU*, *SYSGO* propose *PikeOS for MPU*. Cette version fonctionne sur des architectures *ARM* de type *MPSoC*. À notre connaissance, il n'y a pas de support pour des architectures dépourvues de tout contrôle mémoire.

4.3.2 Partitionnement temporel

PikeOS utilise un ordonnanceur hiérarchique breveté à double niveau inspiré de la norme *ARINC 653* [73]. Cette architecture combine un partitionnement temporel strict au niveau des partitions avec une flexibilité d'ordonnancement au niveau des threads.

4.3.2.1 Ordonnancement au niveau des partitions

Le premier niveau de l'ordonnanceur assure l'isolation temporelle stricte entre les partitions. Il distribue statiquement le temps *CPU* selon un schéma cyclique de type *TDM* (*Time Division Multiplexing*). Le cycle complet, appelé *major time frame* dans la terminologie *ARINC 653*, est subdivisé en plusieurs fenêtres temporelles (*partition windows* ou *minor time frames*) de durées variables, chacune allouée à une partition spécifique.

Cette approche permet de garantir qu'une partition reçoit toujours son temps d'exécution alloué de manière déterministe. La configuration de ces fenêtres temporelles est entièrement statique et définie lors de la phase de configuration du système, garantissant ainsi un comportement prévisible. La granularité des fenêtres temporelles peut descendre jusqu'à 250 µs, offrant une précision de l'ordre de la microseconde pour l'ordonnancement des partitions.

4.3.2.2 Ordonnancement au niveau des threads

Le second niveau de l'ordonnanceur opère au sein de chaque partition pendant sa fenêtre temporelle. *PikeOS* propose plusieurs politiques d'ordonnancement pour les threads:

- *Fixed-priority preemptive scheduling*: il s'agit de la politique principale. L'ordonnanceur sélectionne toujours le thread avec la plus haute priorité parmi les threads prêts à s'exécuter dans la partition active. Un thread de haute priorité peut immédiatement préempter un thread de plus faible priorité,
- *Earliest Deadline First* (*EDF*): *PikeOS* supporte également des politiques d'ordonnancement *EDF*, permettant de prioriser dynamiquement les threads selon leurs échéances.

Cette combinaison d'un ordonnancement temporel strict au niveau des partitions et d'un ordonnancement flexible au niveau des threads permet à *PikeOS* de supporter simultanément des applications de criticités différentes tout en garantissant l'isolation temporelle nécessaire pour les systèmes temps réel.

4.3.3 Déterminisme

PikeOS a été conçu dès le départ pour offrir un comportement déterministe adapté aux exigences du temps réel.

4.3.4 Draft

Son hyperviseur support la virtualisation assisté par le matériel (*HW-Virt*).

4.3.5 Déterminisme

PikeOS a été conçu dès l'origine pour offrir un comportement déterministe adapté aux exigences temps réel dur. Plusieurs mécanismes architecturaux garantissent la prévisibilité et le contrôle des latences. Contrairement aux premiers micronoyaux de la famille *L4*, *PikeOS* est totalement préemptif [68].

4.3.5.1 Préemption du micronoyau

Le micronoyau *PikeOS* est totalement préemptif [68]. Cette caractéristique est fondamentale pour minimiser les latences et respecter les échéances temps réel. Contrairement à son prédécesseur *P4*, *PikeOS* a été spécifiquement pensé pour éliminer les sections non préemptibles qui pouvaient introduire de la gigue et empêcher une estimation précise du *WCET*.

L'ordonnancement au sein des partitions utilise un mécanisme préemptif à priorités fixes. Cela garantit qu'une tâche de haute priorité peut immédiatement interrompre une tâche de plus faible priorité pour respecter ses contraintes temporelles. *PikeOS* supporte également des politiques d'ordonnancement de type *EDF* (*Earliest Deadline First*), permettant de prioriser dynamiquement les tâches selon leurs échéances.

4.3.5.2 Latences bornées

Le déterminisme de *PikeOS* repose sur des latences bornées, inhérentes à sa conception microkernel. En limitant les fonctionnalités du noyau aux services essentiels, *PikeOS* réduit les surcoûts et élimine les éléments non déterministes tels que l'allocation dynamique de mémoire

dans les chemins d'exécution critiques. Cette approche minimaliste garantit des temps de réponse prévisibles.

Le partitionnement temporel applique des allocations strictes de temps *CPU*. Chaque partition reçoit des fenêtres d'exécution dédiées définies par des tranches de temps, ce qui assure un ordonnancement déterministe avec une précision de l'ordre de la microseconde. Cette isolation temporelle est essentielle pour garantir qu'une partition ne puisse pas accaparer le processeur au détriment des autres.

4.3.5.3 Estimation du *WCET*

Les temps d'exécution dans le pire cas sont contrôlés via plusieurs mécanismes. Comme mentionné dans la sous-section 4.2.1, *PikeOS* utilise depuis la version *4.2* un système de verrouillage à granularité fine qui prévient la contention dans les configurations multiprocesseurs. Cette évolution a permis d'améliorer significativement les estimations *WCET* en permettant l'exécution parallèle d'appels systèmes n'accédant pas aux mêmes ressources.

PikeOS fournit également des mécanismes de timers haute résolution pour la gestion précise des événements, facilitant ainsi l'implémentation de modèles d'exécution déclenchés par le temps (*time-triggered*).

Enfin, comme évoqué précédemment, *PikeOS* peut être configuré pour invalider les caches et la *TLB* lors des changements de partition temporelle. Bien que cette option réduise les performances, elle élimine les interférences entre partitions et améliore la prévisibilité des temps d'exécution, ce qui peut être nécessaire pour des analyses *WCET* très conservatrices.

4.3.6 OS invités supportés

L'hyperviseur de *PikeOS* supporte les OS invités[16] suivants:

- *ELinOS* est supporté par *PikeOS*. Il s'agit d'une distribution *Linux* pour l'embarqué temps réel développé par *SYSGO* [76]. Elle peut être exécutée aussi bien dans une partition paravirtualisée qu'une partition virtualisée par le matériel. *SYSGO* offre un support pour chaque version d'une durée de 5 ans extensible,
- *RTEMS* est supporté par *PikeOS* depuis 2010 [77]. Il est en particulier possible de l'utiliser sur la plateforme *LEON*,
- Les systèmes qui se conforment à *POSIX* comme *Linux* ou *Android*,
- *Windows* est supporté dans les partitions assistées par le matériel sur *x86* [78].

4.4 Corruption de la mémoire

PikeOS ne semble pas fournir d'*API* unifiée pour la gestion des erreurs mémoire *ECC* au niveau du noyau de séparation. La documentation publique de *SYSGO* ne détaille pas de mécanisme centralisé de journalisation des erreurs mémoire comparable à *EDAC* sous *Linux* ou au *Health Monitor* d'*XtratuM* pour les erreurs mémoire.

La gestion de la corruption de la mémoire dans *PikeOS* s'effectue à plusieurs niveaux:

- Le *BSP* (*Board Support Package*) peut intégrer un support pour les contrôleurs mémoire *ECC* spécifiques à la plateforme. Par exemple, pour les processeurs *LEON SPARC* que *PikeOS* supporte [77], les versions durcies comme le *LEON3FT* et le *LEON5* intègrent des mécanismes matériels de correction d'erreurs et de *scrubbing* automatique de la mémoire cache pour prévenir l'accumulation d'erreurs dues aux radiations spatiales [79]. Ces fonctionnalités sont accessibles via l'*API* du *BSP LEON* de *PikeOS*.

[16] Ils sont appelés *GuestOS* dans la documentation.

- Les systèmes d'exploitation invités (*GuestOS*) peuvent implémenter leur propre gestion des erreurs mémoire. Ainsi, lorsque *PikeOS* exécute *ELinOS* (une distribution *Linux* embarqué développée par *SYSGO*) dans une partition, ce système invité peut utiliser le sous-système *EDAC* de *Linux* pour détecter et journaliser les erreurs mémoire *ECC*, comme décrit dans 2.6.
- Le *Health Monitor* de *PikeOS* permet la détection d'erreurs et la gestion des fautes au niveau des partitions [80]. Bien que les détails techniques ne soient pas publiquement documentés, ce mécanisme peut être configuré pour réagir aux erreurs matérielles, y compris potentiellement les erreurs mémoire critiques.

Cette approche décentralisée est cohérente avec l'architecture de noyau de séparation de *PikeOS*, où chaque partition est isolée et peut avoir des exigences différentes en matière de gestion d'erreurs selon son niveau de criticité.

4.5 Écosystème

PikeOS est livré avec un ensemble d'outils de développement, incluant un *IDE (Integrated Development Environment)*, des outils de débogage et de simulation. *SYSGO* a également noué un certain nombre de partenariats avec d'autres entreprises pour développer des outils pour *PikeOS* [81]. Nous avons ainsi pu identifier les outils suivants: *CODEO*, *RVS* et *TRACE32*.

PikeOS étant propriétare, son écosystème est limité à cette offre logicielle.

4.5.1 *CODEO*

La société *SYSGO* propose un *IDE* baptisé *CODEO* [82], [83] et basé sur l'*IDE* Eclipse. Il permet entre autres de:

- Développer une application en *C* ou *C++* pour *PikeOS*,
- Configurer les partitions et la politique d'ordonnancement statiquement,
- Visualiser les partitions en cours d'exécution et les activer ou les désactiver,
- Débogueur une application tournant sur *PikeOS* à distance via un support du débogueur du plugin *CDT* d'Eclipse,
- Tracer des événements provenant du noyau *PikeOS* ou de l'application utilisateur,
- Émulation via *QEMU* pour le prototypage.

L'outil est également compatible avec *ELinOS* [84].

4.5.2 *Rapita Verification Suite*

La suite logicielle *RVS* (*Rapita Verification Suite*) développée par *Rapita Systems* permet la vérification de logiciels embarqués critiques directement sur la cible. En particulier, la suite comprend des analyses pour:

- Le temps d'exécution *WCET*,
- La couverture d'un jeu de tests,
- Couplage des données et du flot de contrôles.

Il produit des preuves pour les certifications *DO-178C* et *ISO 26262*. À ce titre *SYSGO* propose un partenariat avec *RVS* afin de faciliter ces certifications [85], [86].

4.5.3 *TRACE32*

L'outil *TRACE32* développé par l'entreprise *Lauterbach* comprend un débogueur et un traceur [87]. Il permet un débogage de l'intégralité de la pile logicielle, de l'application utilisateur jusqu'au pilotes. L'outil supporte *PikeOS* depuis plus de 15 ans.

4.6 Gestion des interruptions

4.7 Perte du flux d'exécution

PikeOS n'utilise pas les contremesures classiques (*ASLR*, *DEP*, *stack canaries*, *CFI*) car il adopte une approche de sécurité par isolation architecturale via un noyau de séparation certifié *Common Criteria EAL 5+* [74], [88]. La protection repose sur:

- La séparation spatiale et temporelle stricte via des partitions logicielles [89],
- L'utilisation de *MMU*/*MPU* pour l'isolation matérielle,
- Une politique de liste blanche pour la communication inter-partition,
- La prévention de propagation d'erreurs entre partitions.

4.8 *Watchdog*

4.9 Programmation *bare-metal*

Il est possible d'exécuter des applications *bare-metal* dans les partitions de *PikeOS* à condition d'adapter un *RTE* du langage de programmation pour l'*API PikeOS native*. Cette *API* n'étant pas librement accessible, de tels *RTE* n'existent que via des projets internes à *SYSGO* ou des partenariats avec d'autres entreprises. Nous avons pu recenser les *RTE* suivants:

- Les langages *C* et *C++* sont supportés via respectivement les *RTE CENV* et *CPPENV*. Ces environnements sont livrés avec *CODEO*.
- Le langage *Ada* est supporté via des *RTE* développés en partenariat avec d'autres entreprises [90], [91]. Le *RTE* développé par *AdaCore* supporte le profile *Ravenscar*. Il s'agit d'un sous-ensemble strict du langage *Ada* pour le temps réel, limitant le parallélisme afin de permettre des analyses plus fines [91].
- Le langage *Rust* est supporté [92].
- Le langage *SCADE* est supporté via un partenariat avec l'entreprise *Ansys* [93].

Nous n'avons pas trouvé d'information concernant *OCaml* sur *PikeOS* et ce support n'existe probablement pas.

4.10 Temps de démarrage

4.11 Qualifications et certifications

Le noyau *PikeOS* a été conçu pour faciliter la qualification et la certification des systèmes l'utilisant. Il propose de nombreux kits de certification en matière de sûreté:

- Pour l'aéronautique et le spatial avec *RTCA DO-178C* jusqu'au plus haut niveau *DAL A* (*Design Assurance Level A*),
- Pour le ferroviaire avec *EN 50128* et *EN 50657* jusqu'au plus haut niveau *SIL 4* (*Safety Integrity Level 4*),
- Pour l'automobile avec *ISO 26262* jusqu'au plus haut niveau *ASIL D* (*Automotive Safety Integrity Level D*),
- Pour l'industrie médicale avec *IEC 61508* jusqu'au niveau *SIL 3* (*Safety Integrity Level 3*).

Normes:

- Critères communs (quel niveau?)
- SAR

Il est possible d'avoir une certification pour une partition spécifique.

4.12 Maintenabilité

PikeOS est un logiciel propriétaire aux sources fermées. Nous n'avons pas pu évaluer la taille de sa base de code faute d'informations communiquées par *SYSGO*. Toutefois, au vu du nombres de certifications et de sa conception de type *L4*, le noyau est sans doute de petite taille, c'est-à-dire de l'ordre de quelques dizaines de milliers de lignes de code.

L'entreprise ne semble pas davantage communiquer sur ses licences. Les modalités et les coûts de ces licences sont certainement à négociés avec *SYSGO* en fonction de chaque projet.

4.13 Draft

- Un noyau de séparation permettant la criticité mixte
- Ce noyau de séparation répond au standard MILS (*Multiple Independent Levels of Security*)

5 ProvenVisor

ProvenVisor en bref

- **Type** : Hyperviseur type 1 vérifié formellement (+ micronoyau ProvenCore)
- **Langage** : C
- **Architectures** : ARM v8-A (avec support MMU)
- **Usage principal** : Systèmes embarqués critiques, IoT sécurisé, isolation forte
- **Points forts** : TCB minimal, vérification formelle, intégration ProvenCore (micro-noyau prouvé), conteneurs sécurisés
- **Limitations** : Propriétaire, ARM uniquement
- **Licences** : Propriétaire (ProvenRun)
- **Certifications** : Processus de certification en cours

ProvenVisor est un hyperviseur de type 1 développé par l'entreprise *ProvenRun*. Il se place comme un concurrent de *Xen* avec pour différence d'avoir un *TCB* plus réduit et d'être vérifié grâce à des méthodes formelles. Sa cible est le marché de l'*IdO (internet des objets)* sur des microprocesseurs *ARM*.

ProvenVisor a été développé pour être combiné avec *ProvenCore*. *ProvenCore* est un noyau sécurisé et prouvé. *TEE (Trusted execution environment)*

À ce titre *ProvenVisor* est comparable à *seL4* car tous les deux cherchent à offrir le plus petit *TCB* possible.

ProvenCore est un micronoyau qui cherche à la fois à minimiser la taille du code et la surface d'attaque (les deux allant souvent de pair). Il propose des containeurs sécurisé avec la possibilité de communiquer de façon sécuriser entre eux. Il a fait l'objet d'une vérification formelle [94].

ProvenVisor est développé par l'entreprise *ProvenRun* qui est spécialisée dans la sécurité et les systèmes embarqués critiques.

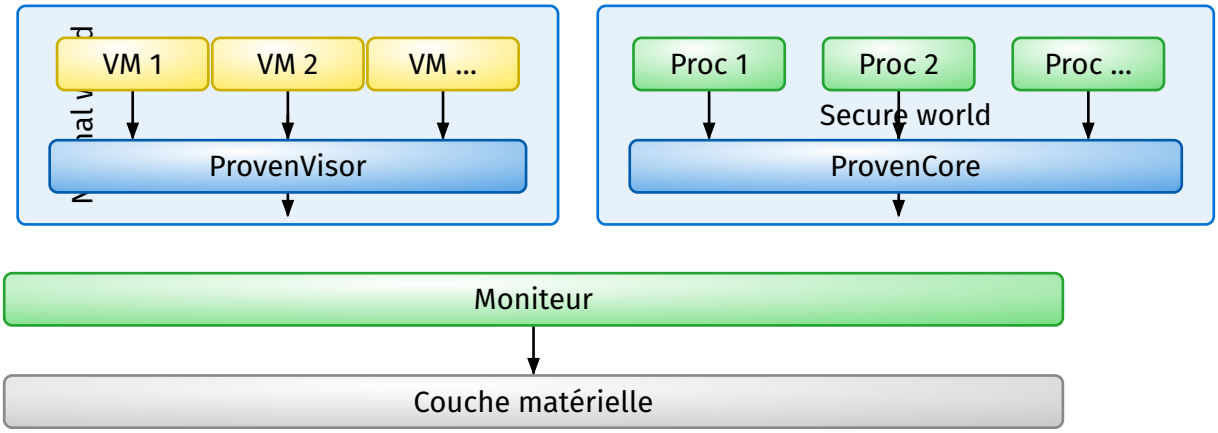

Fig. 4. – Architecture de *ProvenVisor*.

5.1 Architectures supportées

L'hyperviseur est disponible sur l'architecture *ARM v8-A*. Il offre un support pour le *MMU* sur cette architecture.

ProvenCore est conçu pour fonctionner avec le *TEE TrustZone* de l'architecture *ARM*.

5.2 Certifications

ProvenVisor a été conçu dès le départ avec les certifications de haut niveau en tête. L'hyperviseur bénéficie d'une vérification formelle, ce qui facilite grandement le processus de certification pour les niveaux les plus exigeants des *Critères Communs*.

ProvenVisor permet d'atteindre le niveau *EAL5* (*Evaluation Assurance Level 5*) des *Critères Communs* [95]. Ce niveau nécessite une conception semi-formelle et une analyse rigoureuse de la sécurité. Le fait que *ProvenVisor* soit vérifié formellement le place dans une position favorable pour les certifications de haut niveau.

Il est important de noter que *ProvenCore*, le micronoyau prouvé qui accompagne *ProvenVisor*, a obtenu la certification *Common Criteria EAL7* [96], le niveau d'assurance le plus élevé. Cette certification démontre la maturité des méthodes de vérification formelle employées par *ProvenRun* et renforce la confiance dans l'ensemble de leur écosystème de sécurité

5.3 Licences

ProvenVisor est un produit propriétaire développé et commercialisé par l'entreprise française *ProvenRun*. Le code source n'est pas disponible publiquement et l'utilisation de l'hyperviseur nécessite une licence commerciale. Cette approche propriétaire permet à *ProvenRun* de maintenir un contrôle strict sur la qualité et la sécurité du code, tout en finançant les efforts de vérification formelle et de certification.

Les conditions exactes de licence et les modèles de tarification ne sont pas publiquement documentés et doivent être négociés directement avec *ProvenRun*.

5.4 Support multi-processeur

ProvenVisor est conçu pour fonctionner sur des architectures *ARM* modernes qui incluent généralement plusieurs cœurs de processeur. L'hyperviseur tire parti des extensions de virtualisation de l'architecture *ARMv8-A* qui sont présentes sur les processeurs multi-cœurs.

Bien que les détails techniques spécifiques du support multi-processeur ne soient pas largement documentés publiquement, l'architecture *ARM Cortex-A* ciblée par *ProvenVisor* supporte nativement les configurations *SMP*. L'hyperviseur utilise les capacités du *GICv2* ou *GICv3* (*Generic Interrupt Controller*) avec extensions de virtualisation [95], qui sont essentielles pour la gestion des interruptions dans un environnement multi-cœur.

Le support du *PSCI* (*Power State Coordination Interface*) version 1 indique également que *ProvenVisor* peut gérer l'allumage et l'extinction des cœurs individuels, une fonctionnalité importante pour les systèmes multi-processeurs [95].

5.5 Partitionnement

Le partitionnement est au cœur de la conception de *ProvenVisor*. En tant qu'hyperviseur de type 1, son rôle principal est de fournir une isolation forte entre les différentes machines virtuelles qu'il héberge.

5.5.1 Partitionnement spatial

ProvenVisor assure un partitionnement spatial strict entre les machines virtuelles grâce à plusieurs mécanismes:

Le *SMMU* (*System Memory Management Unit*) d'*ARM* est supporté par *ProvenVisor* [95]. Le *SMMU* fournit une traduction d'adresses en deux étapes et des protections système supplémentaires. La traduction en deux étapes permet aux systèmes d'exploitation invités d'utiliser la mémoire virtuelle avec traduction accélérée matériellement (étape 1), tandis que l'hyperviseur conserve un contrôle complet sur la façon dont l'espace d'adressage de chaque invité est mappé vers le périphérique physique (étape 2). Cela simplifie l'isolation des espaces d'adressage des différents invités par l'hyperviseur.

Le *MMU* standard est également supporté, offrant les mécanismes de base pour la virtualisation de la mémoire. Les extensions de virtualisation matérielle de l'architecture *ARMv8-A* permettent à *ProvenVisor* d'isoler efficacement les machines virtuelles sans surcoût de performance significatif.

L'architecture illustrée dans la Fig. 4 montre clairement la séparation entre le *Normal world* (où s'exécutent les VMs sous *ProvenVisor*) et le *Secure world* (où s'exécutent les processus sous *ProvenCore*). Cette séparation s'appuie sur *TrustZone*, la technologie de sécurité matérielle d'*ARM*, qui fournit un environnement d'exécution de confiance complètement isolé.

Une caractéristique importante de *ProvenVisor* est que les communications inter-invités ne sont autorisées qu'après autorisation explicite [95]. Cette politique garantit qu'aucune fuite d'information ne peut se produire entre les VMs sans que cela soit intentionnellement configuré.

Le *TCB* minimal de *ProvenVisor* contribue également à la robustesse du partitionnement spatial. Plus le *TCB* est petit, moins il y a de code susceptible de contenir des bugs qui pourraient compromettre l'isolation.

5.5.2 Partitionnement temporel

Les informations publiquement disponibles sur le partitionnement temporel de *ProvenVisor* sont limitées. Cependant, en tant qu'hyperviseur conçu pour les systèmes embarqués critiques, il est raisonnable de supposer qu'il offre des mécanismes de contrôle du temps CPU alloué à chaque machine virtuelle.

L'ordonnancement des machines virtuelles est typiquement géré au niveau de l'hyperviseur, qui décide quand chaque VM peut s'exécuter. Les détails spécifiques de l'algorithme d'ordonnancement utilisé par *ProvenVisor* ne sont pas publiquement documentés. Il est probable que l'hyperviseur offre des options de configuration pour garantir que les VMs critiques reçoivent le temps CPU nécessaire à leur bon fonctionnement, mais les spécificités de ces mécanismes relèvent de la documentation propriétaire.

5.6 Déterminisme

Le déterminisme dans *ProvenVisor* découle de sa conception minimale et de sa vérification formelle. Un *TCB* réduit signifie moins de chemins d'exécution possibles dans le code de l'hyperviseur, ce qui facilite l'analyse du comportement temporel.

La vérification formelle de *ProvenVisor* contribue également au déterminisme. En prouvant mathématiquement les propriétés du système, on s'assure que le comportement de l'hyperviseur est prévisible et conforme à sa spécification. Cela est particulièrement important pour les systèmes critiques où un comportement non déterministe pourrait conduire à des défaillances catastrophiques.

L'utilisation des extensions de virtualisation matérielle d'*ARM* aide également à maintenir un comportement déterministe. Ces extensions permettent aux VMs de s'exécuter avec un minimum d'interventions de l'hyperviseur, réduisant ainsi les sources de variabilité temporelle.

Cependant, il convient de noter que les informations publiques sur les analyses *WCET* ou les garanties temps réel spécifiques de *ProvenVisor* sont limitées. Pour des applications nécessitant des garanties temps réel strictes, il serait nécessaire de consulter la documentation technique détaillée de *ProvenRun*.

5.7 Corruption de la mémoire

ProvenVisor intègre plusieurs mécanismes pour prévenir la corruption de la mémoire:

La vérification formelle de *ProvenVisor* est le premier rempart contre la corruption de la mémoire. En prouvant mathématiquement l'absence de certaines classes de bugs, on élimine les erreurs de programmation qui pourraient conduire à des corruptions mémoire, telles que les dépassements de tampon, les utilisations après libération (*use-after-free*), ou les double libérations.

Le *TCB* minimal réduit la surface d'attaque. Moins il y a de code, moins il y a d'opportunités pour des bugs de corruption mémoire. Cette approche minimaliste est un principe fondamental de la conception de *ProvenVisor*.

Les mécanismes matériels de protection mémoire de l'architecture *ARM* sont pleinement exploités. Le support du *MMU* et du *SMMU* permet de configurer des permissions d'accès granulaires pour chaque région mémoire. Les tentatives d'accès non autorisées déclenchent des exceptions qui peuvent être interceptées par l'hyperviseur.

L'isolation fournie par *TrustZone* ajoute une couche supplémentaire de protection. Le *Secure world* et le *Normal world* ont des espaces d'adressage physiques séparés, et le code s'exécutant dans le *Normal world* ne peut pas accéder à la mémoire du *Secure world*. Cela protège les composants critiques exécutés sous *ProvenCore* contre les corruptions provenant des VMs moins fiables.

Le langage de développement utilisé pour *ProvenVisor* est le *C*. Bien que ce langage soit sujet aux erreurs de gestion mémoire, la vérification formelle compense cette faiblesse en prouvant que le code implémenté ne contient pas ces erreurs.

5.8 Perte du flux d'exécution

La protection contre la perte du flux d'exécution dans *ProvenVisor* repose sur plusieurs piliers:

La vérification formelle garantit que le code de l'hyperviseur ne contient pas de vulnérabilités permettant de détourner le flux d'exécution. Cela inclut les bugs classiques comme les dépassements de tampon qui pourraient être exploités pour écraser l'adresse de retour sur la pile.

Les extensions de sécurité de l'architecture *ARMv8-A* offrent plusieurs protections matérielles contre le détournement du flux d'exécution. Le bit *NX* (*No-eXecute*), également connu sous le nom de bit *XN* (*eXecute Never*) sur *ARM*, permet de marquer des pages mémoire comme non exécutables. Cela empêche l'exécution de code injecté dans des zones de données.

Le *PAN* (*Privileged Access Never*) sur *ARMv8.1-A* empêche le code privilégié d'accéder accidentellement ou malicieusement à la mémoire de l'utilisateur. Bien que cette fonctionnalité

soit principalement destinée aux noyaux de systèmes d'exploitation, elle peut également être pertinente pour un hyperviseur.

Le *Pointer Authentication* introduit dans *ARMv8.3-A* fournit une protection cryptographique contre la falsification des pointeurs. Chaque pointeur est signé cryptographiquement lorsqu'il est stocké et vérifié lorsqu'il est utilisé. Toute tentative de modifier un pointeur sans connaître la clé cryptographique sera détectée.

Le *TCB* minimal réduit également le risque de vulnérabilités exploitables. Moins il y a de code, moins il y a de surface d'attaque pour un adversaire cherchant à détourner le flux d'exécution.

L'isolation stricte entre les VMs signifie qu'un compromis dans une VM ne permet pas directement d'attaquer l'hyperviseur ou les autres VMs. Un attaquant devrait d'abord s'échapper de la VM compromise, ce qui nécessiterait d'exploiter une vulnérabilité dans *ProvenVisor* lui-même – une tâche rendue extrêmement difficile par la vérification formelle.

5.9 Écosystème

L'écosystème de *ProvenVisor* est intrinsèquement lié à celui de *ProvenCore*. Les deux produits sont conçus pour fonctionner ensemble et forment un système de sécurité complet pour les applications embarquées critiques.

5.9.1 Intégration avec *ProvenCore*

ProvenCore est un micronoyau formellement vérifié qui s'exécute dans le *Secure world* de *TrustZone* [94]. Il fournit des conteneurs sécurisés avec la possibilité de communiquer de façon sécurisée entre eux. *ProvenVisor* et *ProvenCore* collaborent pour offrir une solution complète de sécurité et d'isolation.

L'architecture illustrée dans la Fig. 4 montre comment *ProvenVisor* gère les VMs dans le *Normal world* tandis que *ProvenCore* gère les processus sécurisés dans le *Secure world*. Un moniteur de sécurité coordonne les interactions entre ces deux mondes.

5.9.2 Compatibilité avec les systèmes d'exploitation invités

ProvenVisor est conçu pour s'intégrer de manière transparente avec les systèmes d'exploitation généralistes courants [95]. Il supporte notamment:

- *Linux*: le système d'exploitation le plus répandu dans l'embarqué peut s'exécuter

comme VM sous *ProvenVisor*.
- *Android*: important pour les applications IoT et les dispositifs connectés.
- D'autres systèmes d'exploitation compatibles *ARM* peuvent également être hébergés.

Cette compatibilité permet de réutiliser des logiciels existants tout en bénéficiant de l'isolation et de la sécurité fournies par *ProvenVisor*.

5.9.3 Partenariats et support matériel

ProvenRun collabore avec plusieurs acteurs de l'industrie des semi-conducteurs. Leur partenariat avec *STMicroelectronics* démontre l'intégration de leurs solutions dans des plateformes matérielles concrètes [97]. Des démonstrations sur des modules *Toradex* ont également été réalisées [98], montrant la portabilité de la solution sur différentes plateformes *ARM*.

5.9.4 Documentation et support

Étant un produit commercial propriétaire, la documentation détaillée et le support technique de *ProvenVisor* sont réservés aux clients de *ProvenRun*. Les informations publiquement disponibles se limitent principalement aux documents marketing et aux présentations techniques de haut niveau.

5.10 Gestion des interruptions

La gestion des interruptions dans *ProvenVisor* s'appuie sur les capacités de virtualisation des contrôleurs d'interruptions *ARM*.

ProvenVisor requiert un *GICv2* ou *GICv3* (*Generic Interrupt Controller*) avec extensions de virtualisation [95]. Ces extensions permettent à l'hyperviseur de présenter des interruptions virtuelles aux VMs invitées sans intervention logicielle pour chaque interruption, ce qui améliore considérablement les performances.

Le *GIC* avec extensions de virtualisation fournit:

- Des interruptions virtuelles qui peuvent être injectées directement dans les VMs par le matériel, réduisant la latence.
- La capacité de router les interruptions physiques vers les VMs appropriées.
- L'isolation des interruptions entre les VMs, garantissant qu'une VM ne peut pas voir ou manipuler les interruptions d'une autre VM.

L'hyperviseur peut configurer quelles interruptions physiques sont routées vers quelles VMs. Certaines interruptions peuvent être réservées à l'hyperviseur lui-même, tandis que d'autres sont déléguées aux VMs invitées.

Le support du *PSCI* (*Power State Coordination Interface*) permet également de gérer les événements liés à l'alimentation et à la gestion des cœurs, qui sont souvent déclenchés par des interruptions ou des événements similaires.

5.11 Support *watchdog*

Les informations spécifiques sur le support *watchdog* intégré dans *ProvenVisor* ne sont pas largement documentées publiquement. Cependant, plusieurs aspects de l'architecture suggèrent comment les mécanismes *watchdog* peuvent être utilisés avec cet hyperviseur.

Dans un environnement virtualisé, le support *watchdog* peut s'organiser à plusieurs niveaux:

5.11.1 *Watchdog* matériel

Les plateformes *ARM* modernes intègrent généralement des *watchdogs* matériels. Ces dispositifs peuvent être configurés pour réinitialiser le système si le logiciel ne les actualise pas régulièrement. Dans le contexte de *ProvenVisor*, l'hyperviseur lui-même pourrait être responsable de la gestion du *watchdog* matériel principal, assurant que le système reste opérationnel même si une VM invitée se bloque.

5.11.2 *Watchdog* virtualisé

ProvenVisor pourrait potentiellement présenter des *watchdogs* virtualisés aux VMs invitées, permettant à chaque VM de surveiller ses propres applications critiques. La virtualisation des *watchdogs* est une fonctionnalité commune dans les hyperviseurs modernes, bien que les détails d'implémentation spécifiques de *ProvenVisor* ne soient pas publiquement documentés.

5.11.3 Surveillance par l'hyperviseur

L'hyperviseur lui-même peut agir comme un *watchdog* pour les VMs qu'il héberge. En surveillant l'activité de chaque VM, *ProvenVisor* pourrait détecter les VMs qui ne répondent plus et prendre des mesures appropriées, comme les redémarrer ou notifier le système de supervision.

5.11.4 Intégration avec *ProvenCore*

Dans l'architecture combinée *ProvenVisor*/*ProvenCore*, des mécanismes de surveillance sophistiqués peuvent être mis en place. *ProvenCore*, s'exécutant dans le *Secure world*, pourrait surveiller l'hyperviseur et les VMs depuis un environnement de confiance isolé, offrant une couche supplémentaire de résilience.

Pour des informations détaillées sur les capacités *watchdog* spécifiques de *ProvenVisor* et sur la façon de les configurer pour des applications critiques, il serait nécessaire de consulter la documentation technique propriétaire de *ProvenRun*.

5.12 Programmation *bare-metal*

ProvenVisor étant un hyperviseur de type 1, il s'exécute directement sur le matériel nu (*bare-metal*) sans nécessiter un système d'exploitation hôte. Cependant, il n'est pas conçu pour héberger directement des applications *bare-metal*. Son rôle est plutôt de fournir un environnement virtualisé dans lequel des systèmes d'exploitation invités peuvent s'exécuter.

Pour des applications nécessitant un accès *bare-metal* au matériel, plusieurs approches sont possibles:

- *Passthrough* de périphériques: *ProvenVisor* peut potentiellement permettre à une VM

d'accéder directement à certains périphériques matériels sans intervention de l'hyperviseur. Cela donnerait à cette VM un accès quasi-*bare-metal* à ces périphériques spécifiques.
- Utilisation de *ProvenCore*: pour des applications critiques nécessitant un accès

direct et contrôlé au matériel, *ProvenCore* dans le *Secure world* pourrait être une solution plus appropriée.
- VM dédiée: une VM peut être configurée avec un accès privilégié à certaines

ressources matérielles, s'approchant d'un fonctionnement *bare-metal* pour les parties critiques.

Il est important de noter que l'ajout d'un hyperviseur comme *ProvenVisor* introduit nécessairement une certaine latence et complexité par rapport à une exécution purement *bare-metal*. Cependant, les bénéfices en termes d'isolation et de sécurité compensent généralement ce surcoût dans les applications pour lesquelles *ProvenVisor* est conçu.

5.13 Temps de démarrage

Les informations publiquement disponibles sur les temps de démarrage spécifiques de *ProvenVisor* sont limitées. Cependant, plusieurs facteurs suggèrent que l'hyperviseur devrait avoir un temps de démarrage relativement rapide:

Le *TCB* minimal implique moins de code à initialiser au démarrage. Un hyperviseur de petite taille peut généralement démarrer plus rapidement qu'un système d'exploitation complet ou qu'un hyperviseur aux fonctionnalités étendues.

La documentation indique que *ProvenVisor* nécessite moins de 1 Mo de mémoire flash et RAM pour une configuration simple [95]. Cette empreinte mémoire réduite suggère un système épuré qui devrait démarrer rapidement.

Le temps de démarrage total du système dépendra également du temps nécessaire pour démarrer les VMs invitées. *ProvenVisor* lui-même pourrait démarrer rapidement, mais si une VM *Linux* doit ensuite être lancée, le temps de démarrage perçu par l'utilisateur inclurait le temps de démarrage de *Linux*.

Pour des applications critiques nécessitant des temps de démarrage garantis, il serait nécessaire de consulter la documentation technique détaillée de *ProvenRun* pour obtenir des chiffres précis et des garanties formelles.

5.14 Maintenabilité

La maintenabilité de *ProvenVisor* présente des caractéristiques distinctives liées à sa nature propriétaire et à son approche basée sur la vérification formelle.

5.14.1 Maintenance par *ProvenRun*

En tant que produit propriétaire, la maintenance de *ProvenVisor* est assurée exclusivement par *ProvenRun*. Les clients n'ont pas accès au code source et dépendent donc de l'éditeur pour les corrections de bugs, les mises à jour de sécurité et les nouvelles fonctionnalités.

Cette approche présente des avantages et des inconvénients:
- *Avantages*: *ProvenRun* maintient un contrôle strict sur la qualité du code, ce qui

est crucial pour un composant aussi critique qu'un hyperviseur sécurisé. La vérification formelle est maintenue à travers les versions successives.
- *Inconvénients*: les clients sont dépendants d'un seul fournisseur et ne peuvent pas

effectuer leurs propres modifications ou corrections urgentes. La pérennité du produit est liée à la santé de l'entreprise *ProvenRun*.

5.14.2 Impact de la vérification formelle

La vérification formelle a un impact significatif sur la maintenabilité de *ProvenVisor*:

D'une part, elle garantit une qualité de code exceptionnelle. Les bugs qui passent généralement à travers les tests traditionnels sont détectés et éliminés grâce aux preuves mathématiques. Cela devrait réduire le nombre de bugs découverts en production et donc le besoin de correctifs urgents.

D'autre part, la vérification formelle complique les modifications du code. Toute modification nécessite de re-prouver les propriétés du système, ce qui peut être un processus long et coûteux. Cela peut ralentir l'ajout de nouvelles fonctionnalités ou l'adaptation à de nouvelles plateformes matérielles.

5.14.3 Support à long terme

Pour les systèmes embarqués critiques, le support à long terme est essentiel. Les produits peuvent rester en service pendant des décennies. Les modalités de support à long terme de *ProvenVisor* doivent être négociées avec *ProvenRun* et dépendent du contrat de licence.

La stabilité du noyau est un atout pour le support à long terme. Un système formellement vérifié devrait nécessiter moins de mises à jour correctives, ce qui peut réduire les risques et les coûts associés au maintien de systèmes en production sur de longues périodes.

5.14.4 Documentation

La documentation technique détaillée de *ProvenVisor* n'est pas publiquement disponible. Les clients de *ProvenRun* ont accès à la documentation nécessaire pour intégrer et utiliser l'hyperviseur dans leurs produits. La qualité et la complétude de cette documentation sont des facteurs importants pour la maintenabilité des systèmes basés sur *ProvenVisor*.

6 RTEMS

RTEMS en bref

- **Type** : RTOS libre
- **Langage** : C (96%)
- **Architectures** : 20+ architectures (ARM, PowerPC, RISC-V, SPARC, x86, MIPS, ...)
- **Usage principal** : Systèmes embarqués temps-réel (spatial, militaire, médical, industriel)
- **Points forts** : Libre (GPL/BSD), API POSIX, support SMP, modulaire, mature, utilisé par NASA/ESA
- **Limitations** : Documentation parfois limitée, configuration complexe
- **Licences** : BSD 2-clause + exceptions (permettant code propriétaire)
- **Missions notables** : Galileo, James Webb Space Telescope, Mars rovers

RTEMS (*Real-Time Executive for Multiprocessor Systems*) est un *RTOS* libre conçu pour les systèmes embarqués.

6.1 Histoire et développement

Le projet *RTEMS* trouve son origine en 1988, lorsque le *Research, Development, and Engineering Center* de l'*U.S. Army Missile Command* commandite une étude évaluant les exécutifs temps réel pour applications *Ada* distribuées [99], [100]. L'objectif est de développer un système d'exploitation temps-réel basé sur des normes ouvertes et exempt de redevances, en réponse aux limitations des solutions commerciales de l'époque qui offraient un support insuffisant pour le multitraitement et les standards ouverts [101]. L'entreprise *OAR Corporation* (*On-Line Applications Research Corporation*) est contractée pour effectuer la recherche et le développement initial.

À cette époque, le système est destiné à un usage militaire, en particulier dans les systèmes de missiles, d'où son nom initial : *Real-Time Executive for Missile Systems* [100]. Dès 1989, une version alpha du système est développée, supportant le processeur *Motorola MC68020* et incluant un *BSP* pour la carte *Motorola MVME136* [99]. En 1990, les premières versions internes destinées à l'armée américaine émergent, intégrant le support du processeur *Intel i386* et du *BSP FORCE386*, marquant ainsi les premiers progrès vers les capacités multiprocesseur [99].

En 1992, *RTEMS* franchit une étape importante en dehors du domaine militaire : le *Superconducting Super Collider* (*SSC*) devient la première organisation non-militaire à recevoir *RTEMS*, après l'avoir évalué favorablement face à *pSOS+*, un système d'exploitation temps réel commercial concurrent [99]. Cette adoption marque le début de l'expansion du système vers des applications civiles.

La première version publique de *RTEMS* (version 3.0) est distribuée sur bande magnétique en 1993, suivie de la version 3.1 rendue disponible via FTP anonyme en 1994 [99], [100]. Cette mise à disposition publique facilite l'adoption du système par la communauté scientifique et industrielle. À partir de 1995, la gestion du projet est entièrement confiée à *OAR Corporation*, qui assure depuis lors la maintenance et le développement de *RTEMS*, ainsi que la maintenance de son infrastructure web [101].

Pendant les années 1990, *RTEMS* commence à être utilisé dans le domaine civil, notamment par la *NASA* et l'*ESA (European Space Agency)*, pour des applications spatiales. Cette évolution

vers des usages civils et multiprocesseurs conduit au renommage du système en *Real-Time Executive for Multiprocessor Systems*, conservant l'acronyme *RTEMS* tout en reflétant mieux son champ d'application élargi [100].

De nos jours, *RTEMS* est utilisé dans de nombreuses missions spatiales de premier plan. Du côté de la *NASA*, le système équipe notamment le *Mars Reconnaissance Orbiter*, les sondes *Dawn*, *Fermi*, *Magnetospheric Multiscale*, *Solar Dynamics Observatory*, *Parker Solar Probe*, *Juno* et le rover *Perseverance* sur Mars [102], [103]. *RTEMS* orbite actuellement autour de cinq planètes et du Soleil, et se trouve à la surface de Mars dans le cadre de la mission *Curiosity Rover* [102]. Du côté de l'*ESA*, le système a été déployé sur des missions clés comme *Herschel*, *Planck*, *BepiColombo* (mission vers Mercure) et *ExoMars* [102]. L'ensemble de la constellation européenne de navigation par satellite *Galileo* utilise également *RTEMS* comme système d'exploitation temps réel [102].

6.2 Tutoriel

Les exemples de ce chapitre ont été réalisés sur une carte *Raspberry PI 4*. En plus de cette carte, vous aurez sans doute besoin d'un adaptateur *UART* vers *USB* afin d'interagir avec le noyau installé sur la carte via ses pins `TX`, `RX` et `Ground`.

> **Attention:**
>
> Prenez garde à ce que l'adapteur *USB/UART* fonctionne avec un voltage de 3,3V, sans quoi vous détruirez votre *Raspberry*.

Vous pouvez générer la chaîne de compilation *RTEMS* pour *Raspberry* à partir d'un fichier *Docker* en tapant la commande suivante:

```
$ make setup -C ./rtems
```

ce qui prend environ une demie heure pour terminer la compilation. Finalement, notez que les images produites par cette chaîne de compilation nécessite un *bootloader*. Le plus simple est de réutiliser le *bootloader* de *Raspberry OS lite* et de remplacer le fichier `/boot/kernel8.img` par l'image produite.

Après avoir branché le *Raspberry* sur votre ordinateur et avant de le mettre sous tension, vous pouvez lancez la commande suivante afin d'interagir avec l'interface *UART*:

```
$ minicom -D /dev/ttyUSB0
```

Notez que le nom de l'interface *TTY* peut varier suivant l'adaptateur utilisé.

6.3 Architectures supportées

Du fait de sa longue histoire, *RTEMS* a supporté et supporte encore aujourd'hui un grand nombre d'architectures. Nous nous concentrons ici sur les architectures énumérées dans la sous-section 1.5.2.

D'après [104], *RTEMS* supporte les familles d'architectures suivantes: *x86-32*, *x86-64 ARMv7*, *ARMv8*, *PowerPC*, *MIPS*, *RISC-V* et *SPARC*.

Le support se fait via des *BSP*. En particulier, le projet distribue un *BSP* pour les processeurs *LEON2* et *LEON3* ayant pour architectures *SPARC v8* et conçus pour des applications dans le spatial.

6.4 Support multi-processeur

Cette section aborde le support d'architectures multi-processeur sous *RTEMS*. *RTEMS* offre à la fois un support pour les architectures *SMP* [105], mais également pour les architectures *AMP*.

6.4.1 Architectures *SMP*

Depuis la version *4.11.0*, *RTEMS* offre un support pour les architectures *SMP* des processeurs *x86*, *ARM*, *PowerPC*, *RISC-V* et *SPARC*. Ce support est toutefois relatif à chaque *BSP*. Il est par exemple disponible pour les processeurs *LEON3* et *LEON4*.

Ce support inclut entre autres:
- Des ordonnanceurs de tâches dédiés comme *EDF* (*Earliest Deadline First*) qui tient compte d'échéances via le mécanisme de *deadline* (voir la sous-section 6.5.3) et se comporte comme un ordonnanceur à priorité fixe en l'absence de *deadlines*.
- La possibilité de circonscrire une tâche à un sous-ensemble de *CPU* via un mécanisme d'affinité.
- Un support pour la migration de tâches.
- Des mécanismes de synchonisations fins via des protocoles de verrouillage comme *OMIP* et *MrsP*.

Aparté: Activation *SMP*

Le support *SMP* n'est pas activé par défaut. Il requière d'être activé durant la phase de compilation du noyau via l'option `--enable-smp`.

Le support *SMP* a fait l'objet de vérifications et de pré-qualifications comme nous le verrons dans la sous-section 6.11.

6.4.2 Architectures *AMP*

Il est possible d'utiliser *RTEMS* sur des *MPSoC*. Par exemple, il existe un *BSP* pour le *MPSoC Xilinx Zynq UltraScale+* [106].

6.5 Partitionnement

RTEMS fournit un partitionnement temporel temps réel avec de nombreux ordonnanceurs et des protocoles de synchronisation fins. En revna

6.5.1 Partitionnement spatial

Contrairement à un *GPOS* ou un hyperviseur, *RTEMS* n'offre pas de séparation traditionnelle entre *espace utilisateur* et *espace noyau*. Cette absence de séparation permet de meilleures performances et un meilleur déterminisme au prix d'une isolation spatiale faible. Ce choix de conception résulte d'un compromis entre le caractère déterministe du noyau et l'isolation spatiale.

Les versions récentes de *RTEMS* prennent en charge les *MMU* ou *MPU* de processeurs récents. Le support est relatif à chaque *BSP* et il est possible de pas activer le *MMU* ou le *MPU* et de disposer d'un modèle mémoire plat (*flat memory model*) afin de bénéficier des avantages

précités. En présence d'un microcontrôleur de gestion de la mémoire, son initialisation est à la charge du *BSP* [107], [108].

Lorsqu'une isolation spatiale est requise, l'usage est d'exécuter *RTEMS* dans une partition d'un noyau de séparation, typiquement dans l'hyperviseur de *XtratuM*.

6.5.2 Partitionnement temporel

RTEMS est distribué avec quatre ordonnanceurs différents:

- *Deterministic Priority Scheduler* est un ordonnanceur préemptif basé sur des niveaux de priorités (jusqu'à 256 niveaux). Il offre de très bonnes performances mais il peut requérir trop de mémoire pour les configurations les plus minimaliste. Il existe une version de cet ordonnanceur pour les multi-processeur [109].
- *Simple Priority* est un ordonnanceur préemptif similaire au précédent mais avec un compromis temps/mémoire différent. Il privilégie une empreinte mémoire plus petite au prix de structures de données plus lentes. Il existe une version de cet ordonnanceur pour les multi-processeur [109].
- *EDF* (*Earliest Deadline First*) est un ordonnanceur préemptif basé sur les échéances (*deadlines*). L'échéance la plus proche est prioritaire. Cet ordonnanceur existe pour les architectures *SMP* [109].
- *CBS* (*Constant Bandwidth Server*) est un ordonnanceur préemptif orienté sur l'isolation temporelle. Chaque tâche dispose d'un budget strict et ne peut pas influencer l'exécution des autres.

Il est possible d'implémenter son propre ordonnanceur et de rendre non-préemptible une tâche.

6.5.3 Déterminisme

RTEMS est avant tout un *RTOS*. Sa conception est donc guidée par le souci de préserver le déterminisme autant que possible.

6.5.4 Protocols de synchonisation

Afin d'assurer la synchronisation des sections critiques et de prévenir les inversions de priorité, *RTEMS* propose quatre protocoles de verrouillage:

- *ICPP* (*Immediate Ceiling Priority Protocol*) pour les architectures monoprocesseur,
- *PIP* (*Priority Inheritance Protocol*) pour les architectures monoprocesseur. La prorité d'une tâche est élevée au niveau de la plus haute priorité d'une tâche qui attend un verrou. C'est une approche similaire aux verrous *rt-mutex* de *Linux* vu en sous-section 2.4.3,
- *MrsP* (*Multiprocessor Resource Sharing Protocol*) pour les architectures *SMP*. Il généralise les verrous *ICPP* aux multiprocesseur *SMP* et utilise de l'attente active,
- *OMIP* (*O(m) Independence Preserving Protocol*) pour les architectures *SMP*. Il généralise le protocol *PIP* aux architectures *SMP*.

6.5.5 Rate Monotonic Manager

Rate Monotonic Manager permet la gestion de tâches périodiques via l'algorithme *RMS* (*Rate Monotinic Scheduling*). C'est un algorithme qui attribut statiquement une priorité à chaque tâche périodique inversement proportionnel à la période. Cette politique d'attribution des priorités peut être combiné avec les ordonnanceurs décrits dans la sous-section précédente.

6.6 Corruption de la mémoire

RTEMS ne semble pas fournir d'*API* unifié pour journaliser les erreurs mémoires ou gérer le *scrubbing*.

Dans le cas du *scrubbing*, le support est relatif au *BSP* utilisé. Par exemple, il existe une *API* pour gérer le *scrubbing* dans le fichier `bsps/include/grlib/memscrub.h`. Ce dernier fait parti du *BSP* pour les microprocesseurs *LEON*. L'usage du *scrubbing* étant rendu nécessaire par le rayonnement inhérent aux missions spatiales, il est très probable qu'un tel support est développé dans un *BSP* chaque fois que celui-ci est nécessaire pour une telle mission.

6.7 Perte du flux d'exécution

RTEMS étant principalement écrit en langage *C*, il est exposés aux vulnérabilités classiques par corruption de la mémoire afin de détourner le flux d'exécution.

Nous n'avons pas trouvé de mécanisme de protection face à ce type d'attaques dans *RTEMS*. Cette absence s'explique par le fait que le noyau n'offre pas d'isolation spatiale forte. Les applications s'exécutant dans *RTEMS* doivent donc être de confiance et leur développement nécessite une attention accrue pour éviter les erreurs de programmation. À défaut, il faut recourir à un hyperviseur pour assurer l'isolation spatiale.

6.8 Écosystème

Les développeurs de *RTEMS* offrent plusieurs outils de monitoring et de profilage:

- *profreport* [110]: Outil de profilage pour le noyau. Il nécessite que ce dernier soit compilé avec l'option `RTEMS_PROFILING` afin d'activer le profilage du noyau lui-même [111]. Il produit un rapport au format *XML* sur sa sortie standard.
- *rtrace* [112]: Outil pour afficher et sauvegarder les traces produites par le sous-système *RTEMS Tracing Framework*. Ce dernier est conçu pour minimiser l'impact sur le système. Il permet de tracer des événements de l'ordonnanceur de tâches (création/destruction d'une tâche, basculement de contexte, ...), des *IPC*, des protocoles de synchronisation et des appels systèmes en général.
- *RTEMS shell* [113]: Shell qui comprend de nombreuses commandes affichant des informations sur les tâches en cours d'exécution [114] et permet l'exploration de l'état mémoire [115].
- *CPU Usage Statistics* [116]: *API* de *RTEMS* permettant de collecter des statistiques de l'usage *CPU* par tâche. La granularité des mesures peut être de l'ordre de la nanoseconde sur les versions récentes de *RTEMS* et à condition que le *BSP* le supporte.

RTEMS ne semble pas offrir d'outil ou d'*API* unifiée pour suivre les registres *PMU*. Un support existe dans certains *BSP*.

6.9 Gestion des interruptions

6.10 *Watchdog*

RTEMS ne fournit pas à notre connaissance d'API unifiée pour gérer les *watchdogs* matériels. Le support est implémenté au niveau du *BSP*. Ce support est disponible pour le *Raspberry PI 4* comme nous l'illustrons dans la sous-section 6.10.1.

D'après la documentation, il est également possible d'implémenter un *watchdog* logiciel via le *Timer Manager*. Plus précisément, on peut mettre en place un *timer* avec la fonction `rtems_timer_fire_after` pour implémenter la logique d'un tel *watchdog*.

6.10.1 Watchdog matériel avec un *Raspberry PI 4*

Le dossier `./rtems/examples/watchdog` contient un exemple d'interaction avec le watchdog d'un Raspberry.

```c
#include <rtems.h>
#include <stdio.h>
#include <stdlib.h>
#include <bsp/watchdog.h>

rtems_task Init(rtems_task_argument ignored) {
  raspberrypi_watchdog_init();

  // Configure le timeout à 10 secondes.
   raspberrypi_watchdog_start(10 * 1000);
   printf("\nWatchdog initialisé\n");

   // Réinitialise le watchdog toutes les 5 secondes.
   while (1) {
     raspberrypi_watchdog_reload();
     printf("Watchdog rechargé\n");
     rtems_task_wake_after (5 * rtems_clock_get_ticks_per_second());
   }
}

#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT
#include <rtems/confdefs.h>

```

Liste 8. – Interaction avec un *watchdog* sur un *Raspberry PI 4*.

La commande suivante compile et produit une image dans `./rtems/artifacts/watchdog.img`.

```
$ make  -C ./rtems/watchdog watchdog
```

6.11 Qualifications & certifications

Du fait de son usage dans le spatial, il est nécessaire de pouvoir qualifier *RTEMS* afin de l'intégrer dans des missions.

6.11.1 Kit de qualifications par l'ESA

L'*ESA* offre un kit de *qualification* pour des applications de *RTEMS* dans le domaine spatial [117]. Il est disponible pour les cartes *Cobham Gaisler GR712RC* (architecture *LEON3* double-cœur) et *GR740* (architecture *LEON4* quadri-cœur). Ce kit est disponible sous licence *Creative Common Attribution-ShareAlike 4.0*.

L'ESA (*European Space Agency*) offre un kit de *qualification* pour des applications de *RTEMS* dans le spatial [117] dans sa version *SMP*.

- QDP kit de préqualification.
- Le kit est sous licence Creative Common Attribution-ShareAlike 4.0.
- Plateforme supportée Cobham Gaisler GR712RC (double-cœur LEON3) et GR740 (quadri-cœur LEON4).
- Utilise GCC (v10.2.1) et la bibliothèque mathématique pour les systèmes critiques (libmcs).
- L'application est liée statiquement à RTEMS. Il faut donc une qualification conjointe de l'application et de RTEMS.
- Conformité *ECSS (European Cooperation for Space Standardization)*

Il y a une qualification de RTEMS dans un cadre mono-cœur par Edisoft.

Un effort important a été livré pour appliquer des méthodes formelles sur RTEMS. C'est une activité sponsorisé par *ECSS* afin de s'assurer de la fiabilité de RTEMS dans un cadre *SMP*. Ils ont utilisé Promela/SPIN [118], un model-checker. Edisoft a encore contribué sur cette version.

- Promela est le langage de formalisation tandis que SPIN est le model checker.

6.11.2 Model checking avec *Promela/SPIN*

La correction fonctionnelle de certaines parties de l'*API* de *RTEMS* ont été vérifié par *model checking* [118] grâce à un financement de l'*ESA*. Cette vérification concerne en particulier les primitives de synchronisation utilisées sur les architectures *SMP*.

L'approche retenue fut la génération automatique de tests en utilisant le langage de spécification *Promela* (*PROtocol MEta LAnguage*) et le vérificateur de modèles *SPIN* pour vérifier la correction et générer les tests à partir de la spécification en *Promela*. Les auteurs envisagent d'étendre cette vérification aux ordonnanceurs de *RTEMS*.

Plus d'informations sont disponibles dans la documentation officielle [119].

6.12 Programmation *bare-metal*

RTEMS n'étant un hyperviseur, nous n'avons pas examiné ce critère.

6.13 Temps de démarrage

Nous n'avons pas trouvé d'informations précises sur le temps de démarrage du noyau *RTEMS*.

6.14 Maintenabilité

Le projet *RTEMS* est développé et maintenu depuis plus de 30 ans. Il est écrit en langage C à plus de 96% pour 1 990 023 *SLOC*.

RTEMS est un logiciel libre distribué sous une multitude de licences libres et open-sources avec pour licence principale *BSD 2-Clause*. Le point commun de ces licences est qu'elles autorisent l'utilisateur a lié son programme avec le code source de *RTEMS* sans devoir redistribuer son propre code source [120].

6.15 Draft

- Il offre un support pour les architectures *SMP* et *AMP*.
- Il permet le cross-développement via d'autres OS: distributions GNU/Linux, Windows, BSD, Solaris, MacOS.

- Il est utilisé dans l'industrie spatiale, notamment chez les acteurs européens.
- ARINC 653 RTEMS
- Il existe un support commercial pour les entreprises européennes ou américaines et la communauté offre bien sûr un support gratuit sans garantie.

6.16 Partitionnement

RTEMS est un système à espace d'adressage unique. Le noyau et les tâches partagent le même espace d'adressage et s'exécute en mode noyau (vérifier). Par conséquent *RTEMS* n'offre pas les mêmes niveaux de sûreté qu'un noyau de séparation comme un hyperviseur. C'est la raison pour laquelle il est parfois exécuté au-dessus d'un hyperviseur.

Il y a un support pour les MPU (memory protection unit) qui sont des version simplifiées des MMU. Vérifier si cette info est valable.

RTEMS propose aussi des mécanismes de partitionnement en mémoire.

RTEMS propose un ordonnanceur en *cluster* (*clustered scheduling*). Cet ordonnanceur permet de partitionner l'ensemble des cœurs en des sous-ensembles appelés *cluster*. L'objectif de cette conception est de limiter les migrations de tâches entre cœur pour des raisons de performances[17] tout en préservant un bon contrôle sur la latence dans le pire cas (*worst-case latencies*). *RTEMS* propose également des primitives de synchronisation inter-clusters. En utilisant des clusters et des mécanismes de synchronisation adéquate, il est possible d'avoir des tâches temps réels et des tâches maximisant le *throughput*.

[17] La migration excessive de tâche conduit à une invalidation des caches des cœurs.

7 seL4

seL4 en bref

- **Type** : Micronoyau temps-réel + Hyperviseur type 1 (3ème génération L4)
- **Langage** : C
- **Architectures** : ARM (32/64-bit), x86 (32/64-bit), RISC-V
- **Usage principal** : Systèmes critiques (défense, médical, automobile, aérospatial)
- **Points forts** : Vérifié formellement (Isabelle/HOL), capabilities, partitions mixtes, certifiable Critères Communs EAL7, absence prouvée de bugs critiques
- **Limitations** : Courbe d'apprentissage élevée, écosystème limité, documentation technique avancée
- **Licences** : GPL v2 (noyau), code utilisateur sous licence libre au choix
- **Caractéristique unique** : Seul OS avec correction prouvée formellement du code C au binaire

Le noyau *seL4* est un micronoyau temps-réel de troisième génération de la famille *L4*. Il intègre également un hyperviseur de type 1. Sa conception débute en 2006 à l'institut de recherche *NICTA*[18], aujourd'hui connu sous le nom de Trustworthy Systems. C'est un noyau orienté sécurité dont l'un des premiers objectifs était d'être entièrement vérifié à l'aide de méthodes formelles. Grâce à ces efforts, il peut aujourd'hui être certifié avec le niveau le plus exigent dès Critères communs.

Le noyau *seL4* est un micronoyau de troisième génération. Il inclut un hyperviseur de type 1 et un *RTOS*. Sa conception a débuté en 2006 à l'institut de recherche *NICTA*[19]. L'objectif était de créer un système d'exploitation capable de satisfaire les exigences de sécurité et de sûreté des *CC (Critères communs)*. À ce titre, les contraintes induites par la vérification formelle du noyau ont été prises en compte dès le départ du projet. Comme son nom le suggère, dans son design, *seL4* est fortement inspiré du micronoyau de seconde génération *L4*. Ainsi, il fournit des abstractions pour la mémoire virtuelle, les *threads* et la communication inter-processus. Toutefois, contrairement à la majorité des autres micronoyaux de la famille *L4*, il fournit également des *capabilities* pour gérer les autorisations.

7.1 Tutoriel

Le site de *seL4* fournit un tutoriel détaillé et une image *docker* contenant tout le nécessaire pour tester le micronoyau dans une machine virtuelle. En supposant que vous avez installé *docker* sur votre machine, il vous suffit de récupérer l'image docker de la façon suivante:

```
$ git clone https://github.com/seL4/seL4-CAmkES-L4v-dockerfiles.git
$ cd seL4-CAmkES-L4v-dockerfiles
$ make user
```

7.2 Architectures supportées

Le développement initial de *seL4* s'est fait uniquement sur l'architecture *ARM v7*. Le projet a depuis été porté sur les plateformes *x86* et *RISC-V*. La dernière version du micronoyau supporte les architectures suivantes: *x86-32*, *x86-64*, *ARM v7*, *ARM v8* et *RISC-V*.

[18] Acronyme pour *National Information and Communications Technology Australia.*
[19] Acronyme pour *National Information and Communications Technology Autralia*)

Sur l'architecture *x86*, il est possible d'utiliser les instructions *VT-X* pour la virtualisation assistée par le matériel.

Plus d'informations sur le support des différentes plateformes sont disponibles sur leur site [121].

7.3 Support multi-processeur

seL4 offre un support aussi bien pour les architectures multiprocesseur *SMP* que *AMP*.

7.4 Support *SMP*

Le noyau *seL4* dispose d'un support *SMP* pour les architectures *x86* et *ARM*. La fonctionnalité pour *x86* semble avoir été ajouté lors de l'introduction du support *x86-64*. Quant à *ARM v7*, le support *SMP* date de la version majeure 6.0.0 et était d'abord limité à la plateforme *MPSoC Sabre* avec au plus quatre cœurs.

Le micronoyau utilise un verrou global *BKL* de type *CLH* comme mécanisme de i synchronisation [122]. Cela signifie que deux sections critiques du noyau ne peuvent pas s'exécuter en parallèle. Cette approche a été adoptée pour sa simplicité et comme une étape intermédiaire avant de mettre en œuvre des mécanismes de synchronisation plus fins. De plus les appels systèmes de *seL4* étant courts, cela n'engendre pas une dégradation des performances comme c'était le cas dans le noyau *Linux*. Toutefois l'utilisation d'un tel verrou donne des estimations du *WCET* pessimistes [123].

Aparté:

Le support *SMP* n'est pas activé par défaut.

Quelque soit la configuration utilisée, le support *SMP* n'a pas fait l'objet de vérification formelle [122]. Cette vérification est d'autant plus difficile que les architectures *SMP* modernes ont des modèles de mémoire faible [122], [124]. C'est le cas notamment de l'architecture *ARM* qui reste l'architecture de référence pour le développement *seL4* du fait de son omniprésence dans l'embarqué. L'implémentation des verrous *CLH* a fait récemment l'objet de vérification formelle [124].

7.5 Support *AMP*

seL4 offre un support pour des architectures *AMP* sur *MPSoC*. L'avantage de cette approche est de bénéficier de la vérification formelle dans ce cas contrairement aux architectures *SMP*.

Il y a également un support pour *OpenAMP*.

Quelques projets qui utilisent *seL4* sur des architectures *AMP*: [125].

7.6 Les capacités

Une particularité importante de la conception de *seL4* au sein de la famille des noyaux *L4* est l'utilisation de capacités (*capabilities*) pour rendre compte des droits d'accès.

Les capacités sont des jetons donnant à leur possesseur des droits d'accès à des ressources du systèmes [126]. De façon plus concrète, une capacité se présente sous la forme d'un pointeur immutable qui combine un objet du noyau et des informations de contrôle d'accès. La possession d'une capacité constitue en elle-même l'autorisation d'accès à la ressource associée.

Ainsi, une capacité adéquate doit être passée à chaque appel système afin que le micronoyau accepte de l'exécuter.

À chaque tâche (*thread*) est associée un ensemble de capacités (*CSpace* pour *Capability-Space*) qui représente les ressources accessibles de la tâche.

Il existe plusieurs types de capacités. Certaines représentes des structures de données du noyau ou encore des ressources abstraites, tandis que d'autres représentent des zones mémoires contiguës inutilisées [127].

7.7 Partitionnement spatial

Le micronoyau *seL4* se distingue par une approche radicalement différente pour le partitionnement spatial. Habituellement, la gestion de la mémoire est effectuée en *espace noyau*. Une tâche en *espace utilisateur* n'a pas accès à la mémoire physique et effectue des allocations dynamiques via des appels systèmes sans pouvoir décider de l'agencement des zones allouées dans son propre espace d'adressage virtuel. À l'inverse l'*API* de *seL4* n'expose que quelques primitives pour manipuler les structures de pagination du *MMU*, laissant le soin aux applications de gérer leur propre espace d'adressage [126], [128].

Le mécanisme des capacités assure l'isolation mémoire entre les tâches. Il y a une correspondance biunivoque entre la mémoire utilisée par une tâche A et les capacités qu'elle possède pour manipuler cette mémoire. Une autre tâche B ne peut accéder à une certaine plage mémoire de A sans posséder également cette capacité. Ainsi chaque tâche peut implémenter sa propre politique de gestion de la mémoire. Cette propriété a été formellement vérifiée [129].

Au démarrage la majorité des capacités sont confiées à une tâche spéciale *root*. Cette tâche peut en créer des nouvelles et leur céder une partie de ses capacités. Ce système est rendu possible par l'absence d'allocation dynamique et facilite également la vérification formelle.

À notre connaissance, *seL4* nécessite la présence d'un *MMU* complet pour fonctionner car son modèle mémoire repose entièrement sur la possibilité de virtualiser la mémoire physique.

7.8 Partitionnement temporel

seL4 implémente une approche à deux niveaux pour le partitionnement temporel, combinant un ordonnanceur hiérarchique basé sur les domaines avec un ordonnanceur de threads à priorités [128]. Cette architecture permet d'obtenir à la fois un partitionnement temporel strict au niveau supérieur et de la flexibilité au niveau des threads.

7.8.1 Ordonnanceur standard

L'ordonnanceur de base de *seL4* est un ordonnanceur préemptif basé sur les priorités avec une politique *round-robin* pour les threads de même priorité. Le noyau supporte 256 niveaux de priorités, allant de 0 à 255, où 255 représente la priorité maximale. À chaque tick d'horloge, le noyau sélectionne le *thread* de plus haute priorité parmi ceux qui sont prêts à s'exécuter. Lorsque plusieurs *threads* de même priorité sont prêts, ils sont ordonnancés selon une politique *round-robin* avec des tranches de temps (*timeslices*) configurables [128], [130].

Le noyau alloue le temps *CPU* en quanta de temps fixes appelés *ticks*. Chaque *thread* possède un champ *timeslice* qui représente le nombre de *ticks* durant lesquels le *thread* peut s'exécuter avant d'être préempté. Le pilote de timer du noyau est configuré pour générer des interruptions périodiques qui marquent chaque *tick*.

7.8.2 Ordonnancement hiérarchique par domaines

seL4 fournit un ordonnanceur hiérarchique de haut niveau qui permet l'ordonnancement statique et cyclique de partitions d'ordonnancement appelées *domaines*. Les domaines sont configurés statiquement lors de la compilation avec un ordonnancement cyclique et sont non préemptibles, ce qui garantit un ordonnancement entièrement déterministe des domaines [128].

Cette approche à deux niveaux fonctionne de la façon suivante:

- Au niveau supérieur, les domaines sont ordonnancés de manière cyclique selon un planning déterministe et fixe. Chaque domaine reçoit une tranche de temps allouée durant laquelle tous ses *threads* peuvent s'exécuter.
- Au sein d'un domaine, les *threads* sont ordonnancés selon le mécanisme de priorités décrit précédemment.

Les *threads* ne peuvent être ordonnancés que lorsque leur domaine est actif. Les communications *IPC* entre domaines sont différées jusqu'au prochain basculement de domaine, et l'appel système `seL4_Yield` n'est pas possible entre domaines différents.

7.8.3 Extensions mcs (*Mixed-Criticality System*)

Les extensions mcs (*Mixed-Criticality System*) de *seL4* introduisent le concept de contexte d'ordonnancement (*scheduling context*) qui offre une isolation temporelle renforcée et des garanties temps réel plus strictes [131]. Ces extensions sont actuellement en cours de vérification formelle [132].

7.8.3.1 Contextes d'ordonnancement

Un contexte d'ordonnancement est un nouvel objet noyau qui contient les paramètres d'ordonnancement, notamment un budget et une période. Ces paramètres représentent une borne supérieure sur le temps d'exécution alloué: le noyau garantit qu'un *thread* ne peut pas s'exécuter plus de *budget* microsecondes sur une période de *period* microsecondes [133].

Il existe deux types de contextes d'ordonnancement:

- *Contextes complets* (*Full scheduling contexts*): lorsque le budget est égal à la période, le contexte accorde un accès à 100% du temps *CPU*. Le contexte agit alors comme une simple tranche de temps et le *thread* auquel il est lié est traité en *round-robin*.
- *Contextes partiels* (*Partial scheduling contexts*): lorsque le budget est inférieur à la période, le contexte limite l'accès au *CPU* proportionnellement au ratio budget/période. Par exemple, un contexte avec un budget de 900 000 microsecondes et une période de 1 000 000 microsecondes accorde environ 90% du temps processeur.

7.8.3.2 Isolation temporelle par serveur sporadique

L'isolation temporelle est assurée par une implémentation de l'algorithme de serveur sporadique (*sporadic server*) [131]. Pour les contextes partiels, *seL4* impose la borne supérieure d'exécution en garantissant la contrainte de fenêtre glissante: durant toute période, le budget ne peut pas être dépassé.

Le suivi du budget est réalisé en suivant le budget éligible en morceaux appelés *replenishments* (ou *refills* dans l'*API*). Un *replenishment* est simplement une quantité de temps accompagnée d'un horodatage à partir duquel ce temps peut être consommé [133]. Lorsqu'un *thread* se bloque ou cède volontairement le processeur, le noyau programme de nouveaux *replenishments* pour les périodes futures tout en déduisant le temps consommé des allocations actuelles.

7.8.3.3 Comportement de l'ordonnanceur

L'ordonnanceur mcs sélectionne le *thread* de plus haute priorité, non bloqué, avec un contexte d'ordonnancement configuré disposant d'un budget disponible. Le comportement de base de l'ordonnanceur reste similaire à la version standard: l'ordonnancement basé sur les priorités est conservé tout en imposant des contraintes temporelles strictes.

7.8.3.4 Serveurs passifs et donation de contexte

Les extensions mcs permettent également le partage de ressources à la manière des rpc. Les serveurs peuvent devenir « passifs » en se déliant de leur contexte d'ordonnancement, ce qui permet aux *threads* clients de donner (*donate*) leur contexte d'ordonnancement durant les appels de procédure à distance.

7.9 Déterminisme

Le noyau *seL4* se distingue par une approche originale pour garantir un bon niveau de déterminisme. Alors que la majorité des *RTOS* privilégient un noyau entièrement préémptible afin de minimiser la latence, *seL4* adopte une approche événementielle largement non préemptible.

Un autre aspect intéressant est que *seL4* a fait l'objet d'une analyse approfondie du *WCET* [134], [135]. Cette analyse fournit une estimation formellement vérifiée du temps d'exécution de l'ensemble des routines du micronoyau. Il semble que *seL4* soit à ce jour le seul noyau en mode protégé disposant d'une analyse *WCET* complète et vérifiée formellement.

Nous approfondissons ces deux aspects dans les sous-sections suivantes.

7.9.1 Approche événementielle

Le noyau *seL4* s'exécute avec les interruptions matérielles désactivées. C'est une approche héritée de la famille des noyaux *L4* qui simplifie à la fois la conception en éliminant la concurrence au sein du noyau et simplifie sa vérification formelle. La désactivation des interruptions implique que le noyau n'est pas préemptible puisque l'ordonnanceur de tâche ne peut être invoqué via une interruption matérielle programmée.

Pour qu'un tel noyau fournisse de bonnes garanties temps réel, il est impératif que les appels systèmes aient un temps d'exécution borné. Pour que la latence soit faible, il faut que ces appels soient aussi court que possible. Le noyau *seL4* fournit de telles garanties en évitant les boucles infinies et l'allocation dynamique. Lorsqu'il n'est pas possible de borner un appel système, des point d'interruption sont introduits manuellement afin de permettre au noyau de suspendre la tâche.

7.9.2 Analyse du *WCET*

DRAFT

7.9.3 Analyse *WCET* formellement vérifiée

En plus de disposer d'un ordonnanceur déterministe, *seL4* a fait l'objet d'une analyse de son *WCET* approfondie [134], [135]. Autrement dit, pour un certain nombre de configurations, on dispose d'une estimation vérifiée du temps d'exécution de toutes les routines du micronoyau. *seL4* est à ce jour le seul noyau en mode protégé disposant d'une analyse *WCET* complète et rigoureuse.

Cette analyse a été réalisée sur l'architecture *ARM11* et a produit des bornes supérieures certifiées pour tous les appels système. Les résultats montrent que la borne supérieure sûre est environ quatre fois le pire temps observé expérimentalement. Cette estimation conservative

s'explique principalement par la modélisation des caches: le processeur *ARM11* dispose d'un cache L1 à 4 voies avec remplacement aléatoire, et une analyse sûre de la résidence en cache nécessite de le modéliser comme un cache à correspondance directe d'un quart de capacité.

L'approche la plus aboutie exploite le cadre de validation de traduction de *seL4* pour lier formellement le binaire (sujet de l'analyse *WCET*) à l'environnement *C* riche en sémantique et à tous les théorèmes et invariants prouvés sur le code *C* dans le contexte de la preuve de correction fonctionnelle.

Toutefois cette analyse a été faite sur ARMv6 et ARM ne fournit pas les informations nécessaires pour réitérer cette analyse sur les nouvelles architectures. Cette limitation est problématique car *ARM* reste l'architecture de référence pour *seL4* du fait de son omniprésence dans l'embarqué. Il semble qu'il y ait un projet pour une telle analyse sur *RISC-V*, dont les spécifications ouvertes facilitent ce type d'analyse.

7.9.4 Approche événementielle non préemptible

Le noyau tourne avec les interruptions matérielles désactivées. Ce choix simplifie grandement la conception et la vérification formelle. Cette décision de conception, héritée des premiers micronoyaux de la famille *L4*, offre plusieurs avantages significatifs:

- Simplification de l'implémentation du noyau en éliminant la nécessité de gérer la concurrence interne,
- Facilitation de la vérification formelle en réduisant l'espace des états possibles,
- Amélioration des performances en cas moyen en évitant les coûts liés à la sauvegarde et restauration de contexte lors d'interruptions imbriquées.

Le modèle événementiel évite également la complexité de la suspension d'exécution (*yielding*), qui rendrait le raisonnement formel sur la correction beaucoup plus difficile. C'est précisément pour cette raison que *seL4* a adopté cette approche: simplifier le raisonnement sur la correction du système.

7.9.5 Conception pour borner le temps d'exécution

Les appels systèmes sont généralement courts. Ceux qui sont trop longs sont préemptibles à des points clés ajoutés par les développeurs. Pour qu'un noyau non préemptible puisse offrir de bonnes garanties temps réel, il est essentiel que tous les appels système aient un temps d'exécution borné. *seL4* respecte ce principe de conception en:

- Évitant les boucles non bornées dans le noyau,
- Éliminant toute allocation dynamique de mémoire,
- Gardant la plupart des appels système courts par conception,
- Introduisant des points de préemption explicites dans les rares appels système longs, permettant au noyau de suspendre l'opération et de traiter les interruptions en attente avant de reprendre.

Cette architecture garantit que même si les interruptions sont désactivées pendant l'exécution dans le noyau, le temps maximal pendant lequel une interruption peut être masquée reste borné et raisonnablement court.

7.9.6 Garanties temps réel

Les analyses *WCET* de *seL4* permettent d'obtenir des bornes supérieures déterministes pour les appels système et les latences d'interruption. Plus précisément, *seL4* fournit un temps de réponse aux interruptions garanti d'environ 500 microsecondes sur une plateforme *BeagleBoard-xM* équipée d'un processeur *ARM Cortex-A8*.

Il semblerait qu'être préemptible ne soit pas un prérequis pour offrir de faibles latences. Des travaux de recherche ont démontré qu'un noyau largement non préemptible peut atteindre des latences d'interruption dans un facteur deux de celles d'un noyau entièrement préemptible, tout en offrant des garanties formellement vérifiées et une meilleure simplicité conceptuelle.

Les extensions mcs (*Mixed-Criticality System*) décrites dans la sous-section 7.8.3 renforcent encore les capacités temps réel de *seL4* en introduisant des contextes d'ordonnancement avec budgets et périodes. Ces extensions permettent une isolation temporelle stricte par l'algorithme de serveur sporadique, offrant ainsi des garanties de type temps réel strict (*hard real-time*).

7.9.7 Temps réel avec isolation spatiale

Il permet de faire du temps réel tout en ayant l'isolation spatiale, ce qui n'est pas le cas de nombreux *RTOS* comme *RTEMS* (voir la sous-section 6.5.1). Le système de capacités et le partitionnement spatial rigoureux de *seL4* n'affectent pas ses capacités temps réel, permettant ainsi de concevoir des systèmes à criticité mixte avec de fortes garanties de sécurité et de sûreté. Cette combinaison unique de déterminisme temporel et d'isolation spatiale fait de *seL4* un choix particulièrement adapté pour les systèmes critiques à criticité mixte.

7.10 Corruption de la mémoire

seL4 étant un micronoyau qui se concentre sur l'isolation, il ne semble pas proposer d'*API* ou de logiciel de journalisation pour les erreurs mémoires. Ce support doit être implémenté par pilote en espace utilisateur.

7.11 Perte du flux d'exécution

Bien que le noyau *seL4* soit écrit en *C*, il est moins exposé aux attaques par corruption de mémoire qu'un programme *C* usuel. En effet, sa vérification formelle a permis de prouver l'absence d'un certains nombres d'erreurs de programmation [136]. Certaines de ces erreurs, comme par exemple le dépassement de tampon, sont des vecteurs d'attaques très commun et notamment de détournement du flux d'exécution.

7.12 Écosystème

Le micronoyau *seL4* offre un écosystème limité. Les outils de haut niveau classiques pour la surveillance et le profilage des applications ne semblent pas être disponibles. Toutefois, il existe un kit de développement *seL4 Microkit* [137] offrant une couche logicielle au-dessus du micronoyau et visant à simplifier le développement sur cette plateforme. En particulier ce kit inclut:

- Un moniteur qui journalise les erreurs systèmes comme les accès mémoire erroné ou les instructions invalides. Il est possible d'implémenter son propre moniteur.
- Une console de débogage permettant une communication avec le système embarqué via une interface série *UART*,
- Un mode *benchmark* qui permet de collecter des informations via les registres *PMU*.

7.13 Gestion des interruptions

7.13.1 Masquage des interruptions

Comme évoqué dans la sous-section 7.9, le noyau *seL4* adopte une approche radicalement différente de la plupart des systèmes d'exploitation : il s'exécute avec les interruptions maté-

rielles désactivées [138]. Cette décision de conception, héritée de la famille des micronoyaux *L4*, simplifie drastiquement la vérification formelle du noyau en éliminant la concurrence au sein de l'espace noyau.

Pour qu'un noyau non préemptible offre de bonnes garanties temps réel, tous les appels système doivent avoir un temps d'exécution borné. *seL4* respecte cette contrainte en évitant les boucles non bornées et toute allocation dynamique de mémoire. Pour les rares appels système longs, des points de préemption explicites permettent au noyau de suspendre temporairement l'opération afin de traiter les interruptions en attente.

Cette approche garantit que le temps maximal pendant lequel une interruption peut être masquée reste borné. Les analyses du *WCET* de *seL4* [134], [135] montrent qu'un temps de réponse aux interruptions d'environ 500 microsecondes peut être atteint sur une plateforme *BeagleBoard-xM* équipée d'un processeur *ARM Cortex-A8*.

7.13.2 Gestion des interruptions en espace utilisateur

seL4 gère les interruptions via son système de *capabilities* [138]. La tâche racine reçoit au démarrage la *capability* `seL4_CapIRQControl`, une capacité unique et non duplicable permettant de dériver des *capabilities* pour tous les numéros d'*IRQ* du système. Cette capacité peut toutefois être transférée entre espaces de capacités (*CSpace*).

À partir de `seL4_CapIRQControl`, il est possible de créer des *capabilities* `IRQHandler` pour des numéros d'*IRQ* spécifiques. Contrairement à `IRQControl`, les `IRQHandler` peuvent être dupliqués et déplacés selon la politique du système. Les invocations pour obtenir ces *capabilities* dépendent de l'architecture : *x86* supporte `GetIOAPIC` et `GetMSI`, tandis qu'*ARM* supporte `GetTrigger`.

Les interruptions sont reçues par les applications en associant un objet de notification (*notification*) à un `IRQHandler` via l'invocation `seL4_IRQHandler_SetNotification`. Lors d'une interruption matérielle, *seL4* délivre un signal à l'objet notification associé. Pour différencier plusieurs sources d'interruptions, le système utilise le *badging* : le badge de la *capability* notification liée à chaque `IRQHandler` est combiné en OU logique avec le mot de données de la notification.

seL4 implémente un mécanisme de masquage implicite : le micronoyau ne livrera aucune interruption supplémentaire après qu'une *IRQ* soit levée jusqu'à ce que l'`IRQHandler` correspondant ait été acquitté via l'invocation `seL4_IRQHandler_Ack`. Ce mécanisme garantit qu'une même interruption ne peut être délivrée plusieurs fois avant d'avoir été traitée.

Les applications peuvent attendre ou sonder les interruptions respectivement avec `seL4_Wait` ou `seL4_Poll`, qui délivrent le mot de données de l'objet notification comme badge du message.

7.13.3 Virtualisation des interruptions

Pour les scénarios de virtualisation, *seL4* exploite les extensions matérielles de virtualisation des interruptions lorsqu'elles sont disponibles. Sur les architectures *ARM* équipées de *KernelArmHypervisorSupport*, le micronoyau tire parti du *VGIC* (*Virtual Generic Interrupt Controller*), qui permet l'injection d'interruptions virtuelles dans les machines virtuelles sans piégeage systématique vers l'hyperviseur.

Actuellement, *seL4* supporte le *GICv2* virtuel. Le support du *GICv3* virtuel est en cours de développement [138]. Le *VGIC* matériel permet aux machines virtuelles d'acquitter et de com-

pléter des interruptions virtuelles directement, sans intervention de l'hyperviseur, améliorant ainsi les performances.

Pour les interruptions *PPI* (*Private Peripheral Interrupts*) virtuelles, notamment les interruptions de timer virtuel, *seL4* utilise un mécanisme de *fault*. Ces interruptions sont délivrées aux *VCPU* via des faults de type `seL4_Fault_VPPIEvent` et doivent être acquittées par l'invocation `seL4_ARM_VCPU_AckVPPI` sur l'objet *VCPU* concerné. Le noyau peut suivre l'état des *IRQ* pour chaque *VCPU* et délivrer les événements d'interruption comme des faults du *VCPU*.

La bibliothèque `libsel4vm` fournit une interface de plus haut niveau pour la virtualisation des interruptions. Elle offre notamment les fonctions `vm_inject_irq` pour injecter une interruption dans le contrôleur d'une machine virtuelle, `vm_set_irq_level` pour ajuster le niveau du signal d'interruption, et `vm_register_irq` pour associer des callbacks d'acquittement à des interruptions spécifiques.

Sur *x86*, les mécanismes diffèrent : les interruptions *MSI* (*Message Signaled Interrupts*) peuvent être obtenues via `seL4_IRQControl_GetMSI`. Toutefois, contrairement à *ARM*, il n'est généralement pas possible d'obtenir deux gestionnaires *IOAPIC* pour la même *IRQ*, ce qui peut compliquer le partage d'interruptions entre machines virtuelles.

7.14 Watchdog

seL4 n'offre pas une *API* unifiée pour la gestion de *watchdog* matériel. Toutefois nous avons trouvé un support pour de tels *watchdog* dans certains *BSP*.

7.15 Programmation *bare-metal*

La documentation de *seL4* détaille les manipulations nécessaires pour exécuter du code C ou Rust en *bare-metal*.

Il y a une crate Rust [139].

Le *seL4 Microkit* offre également une *API* pour les langages *C* et *Rust* [137].

7.16 Temps de démarrage

7.17 Draft

7.18 Partitionnement

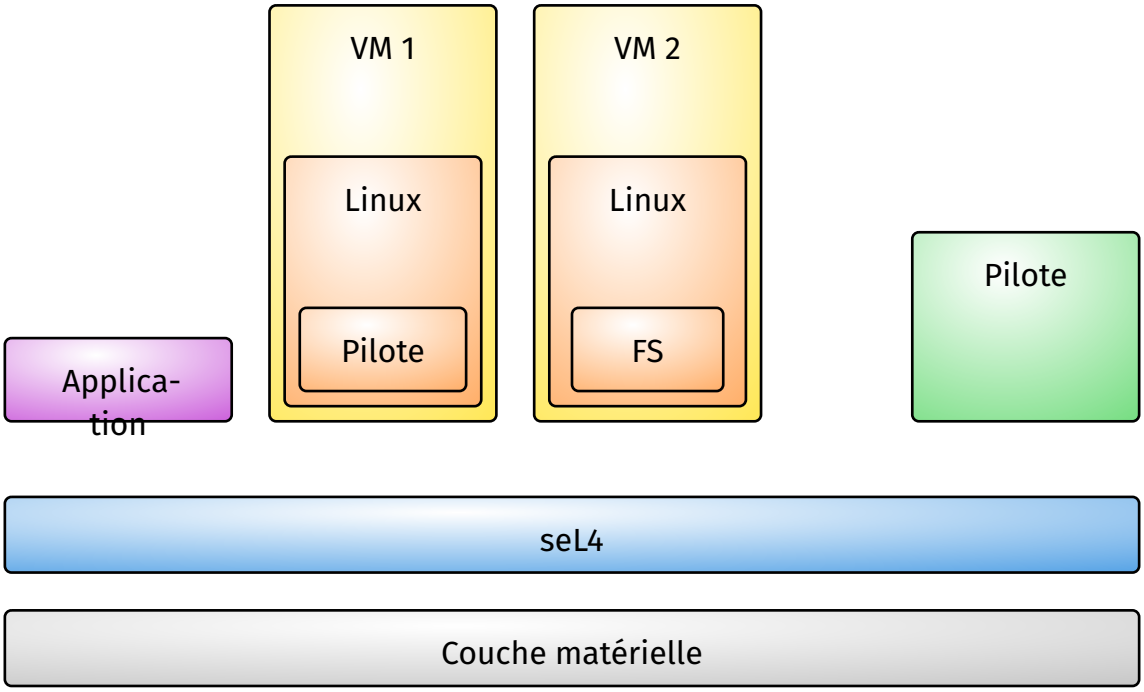

Fig. 5. – Architecture de l'hyperviseur *seL4*.

Lorsqu'il est utilisé en tant qu'hyperviseur, *seL4* s'exécute dans le mode d'exécution *hyperviseur*.

7.19 Vérifications formelles

Le noyau *seL4* a fait l'objet d'une vérification formelle profonde. L'approche suppose la correction du compilateur, du code assembleur et du matériel mais démontre la conformité du code C avec ses spécifications.

7.20 Licences & brevets

Le noyau de `seL4` est un logiciel libre distribué principalement sous licence `GNU General Public License version 2 only (GPL-2.0)`. Le code utilisateur et les pilotes peuvent être distribués sous n'importe quelle licence [140].

7.21 draft

Il a l'avantage de supporté les partitions mixtes [141].

`seL4` a fait l'objet d'une spécification et d'une vérification formelle à l'aide de l'assistant de preuve *Isabelle/HOL*. La correction[20] de l'implémentation a été démontrée pour plusieurs configurations et il a été également démontré que le code binaire est correct pour les architectures *ARM* et *RISC-V* [142]. Cette vérification formelle implique en particulier que `seL4` est dépourvu de certaines erreurs de programmation classiques [143]. Il est notamment dépourvu

[20]La correction d'un algorithme signifie qu'il a été démontré que cet algorithme respecte sa spécification.

de débordements de tampon, de déréférencements de pointeurs nuls, de fuites mémoire et de dépassements d'entier.

8 Xen

Xen en bref

- **Type** : Hyperviseur type 1
- **Langage** : C (majoritaire)
- **Architectures** : x86 (32/64-bit), ARM (32/64-bit)
- **Usage principal** : Cloud computing, hébergement, data centers, virtualisation d'entreprise
- **Points forts** : Mature et éprouvé, paravirtualisation + HVM, largement adopté (AWS, Citrix), dom0less pour boot rapide, stub domains pour sécurité
- **Limitations** : TCB important, complexité, dépendance au dom0 (Linux)
- **Licences** : GPL v2 (majoritaire) avec exceptions pour compatibilité
- **Utilisateurs notables** : Amazon AWS, Citrix, OVH, Rackspace

Xen est un hyperviseur de type 1 développé par le consortium d'entreprises Xen Project. Ces un pionnier de la *paravirtualisation* mais il offre aussi un support étendu pour la virtualisation assistée par le matériel. Il est aujourd'hui très utilisé dans le monde de l'hébergement et du cloud computing.

L'histoire de *Xen* est étroitement liée à l'évolution de la virtualisation et du cloud computing. Elle débute en 1999 avec le projet de recherche *XenoServers* à l'université de Cambridge. Le chercheur Ian Pratt, entouré de plusieurs étudiants, propose une infrastructure pour exécuter plusieurs services sur des *VMs* Java. L'idée fondatrice était de garantir l'isolation des services, même lorsqu'ils n'étaient pas dignes de confiance et d'assurer équité quant à la répartition des ressources.

En 2003, une première version de l'hyperviseur *Xen* est publié sous licence libre. Contrairement à son prédécesseur *XenoServers*, il permet d'exécuter n'importe quelle application dans une *VM* tournant sur un noyau *Linux* modifié. Ces modifications contournent les limites de performances de la virtualisation complète sur architecture *x86* en permettant au noyau virtualisé de collaboré avec l'hyperviseur. C'est la naissance de la *paravirtualisation*.

En 2005, le support pour la virtualisation assistée par le matériel est ajoutée en étroite collaboration avec Intel qui développait alors sa technologie *Intel VT-X*. Cette technologie permet la virtualisation de systèmes d'exploitation à sources fermées comme *Windows*.

Cette même année, la société *XenSource Inc* est fondée pour continuer le développement de *Xen* et faire face à la concurrence. Elle est racheté en 2007 par *Citrix* qui propose toujours une version commerciale de *Xen* baptisé *Citrix Hypervisor*.

Aujourd'hui le développement de *Xen* se concentre sur le support d'autres architectures que *x86*, et notamment *ARM* (voir la sous-section 8.2) et l'utilisation combinée de la *paravirtualisation* et de la virtualisation assistée par le matériel (voir la sous-section 8.6).

8.1 Tutoriel

Les exemples de cette section ont été lancé sur une machine *x86* avec *Xen*. L'installation de *Xen* est grandement simplifié par son support dans certaine distribution *GNU/Linux*. Il vous suffit d'installer les paquets appropriés puis de redémarrer en choisissant l'hyperviseur *Xen* au démarrage.

Afin de pouvoir illustrer certaines fonctionnalités de *Xen*, cette section explique comment mettre en place une machine virtuelle faisant tourner la distribution *GNU/Linux Alpine*. Nous partons du principe que vous êtes parvenu à installer correctement *xen* et *qemu* sur votre machine. Le fichier ci-dessous donne un exemple de configuration d'une VM en paravirtualisation:

```
1  name='alpine'
2  memory='2048'
3  vcpus=2
4  type='pvh'
5  #kernel='/usr/lib/grub-xen/grub-x86_64-xen.bin'
6  kernel='/nix/store/vpvvndr4sfi2xapk3zlhy9hcfc6w087k-pvgrub-image/lib/
grub-xen/grub-x86_64-xen.bin'
7  disk=[ './alpine.qcow2,qcow2,hda,w' ]
8  boot='d'
9  #vif = [ 'mac=00:16:3e:00:00:00,bridge=xenbr0' ]
10  # Cette option est requise à cause d'un bug dans qemu-xen
11  #device_model_override='/run/current-system/sw/bin/qemu-system-i386'
12
```

Liste 9. – Configuration d'une VM Alpine

Plus d'options sont documentées dans la page de manuel `xl.cfg`. Téléchargez l'image d'*Alpine* sur son site officiel:

```
$ wget https://dl-cdn.alpinelinux.org/alpine/v3.22/releases/x86_64/alpine-
standard-3.22.1-x86_64.iso
```

et extrayez les deux fichiers `/boot/vmlinuz-lts` et `/boot/initramfs-lts` de l'image iso:

```
$ mkdir iso
$ mount -t iso9660 -o ro ./alpine-standard-3.22.1-x86_64.iso ./iso
$ cp ./iso/boot/vmlinuz-lts ./iso/initramfs-lts .
$ umount iso
$ rm iso
```

Il vous faut également créer un disque virtuel à l'aide de l'outil `qemu-img`:

```
$ qemu-img create -f qcow2 ./alpine.qcow2 50G
```

Finalement vous pouvez lancer la VM avec la commande suivante:

```
$ sudo xl create alpine.cfg -c
```

Le login par défaut est `root` sans mot de passe. Pour quitter la console de la VM, tapez `CTRL-]`.

8.2 Architectures supportées

Attention:

Dans cette section nous utiliserons les abréviations *PV*, *HVM* et *PVH* qui désignent des types de partitions sous *Xen*. Ces notions sont détaillées dans la section 8.6.

À l'origine *Xen* ne supportait que l'architecture *x86* pour des partitions de type *PV*. Par la suite, la virtualisation assistée par le matériel a été ajoutée pour les technologies *Intel VT-X* puis *AMD V* sous la forme de partitions de type *HVM*.

L'hyperviseur *Xen* supporte les architectures suivantes: *x86-32* à partir de la version P6[21], *x86-64*, *ARM v7* et *ARM v8*. *Xen* a également supporté l'architecture *IA64* jusqu'à la version 4.2. Il existe des travaux en cours pour supporter les architectures *PowerPC* et *RISC-V*. Un support préliminaire de ces architectures est disponibles depuis *Xen 4.20* [144]. Quant à la virtualisation assistée par le matériel de type *HVM (Hardware Virtual Machine)*, elle nécessite les extensions de virtualisation *Intel VT-X* ou *AMD-V* sur *x86* et les *Virtualization Extensions* sur *ARM* [145].

Architecture	PV	HVM	PVH
x86-32	$\geq$ P6		
x86-64		+ *Intel VT-X*	
ARMv7			+ *Virtualization Extensions*
ARMv8			+ *Virtualization Extensions*
PowerPC	*Xen* $\geq$ 4.20		
RISC-V	*Xen* $\geq$ 4.20		

Tableau 4. – Récapitulatif des architectures supportées par l'hyperviseur *Xen*

8.3 Support multi-processeur

Xen offre un support pour les architectures *SMP* et *AMP*. Ce support se fait via l'abstraction offerte pour les *CPU* virtuels et des ordonnanceurs adaptés aux architectures multi-processeur.

8.4 Support *SMP*

Xen offre un support *SMP* pour toutes les architectures vues en sous-section 8.2. En particulier les ordonnanceurs *Credit Scheduler* et *Credit2 Scheduler* sont capables de répartir automatiquement la charge sur les différents cœurs.

Lorsqu'un système invité supporte lui-même les architectures *SMP*, il peut en tirer parti dès lors qu'il dispose de plusieurs *vCPU*.

8.5 Support *AMP*

Nous n'avons pas d'informations précises sur l'usage de *Xen* sur des plateformes *AMP*. Toutefois la documentation de *RTEMS* mentionne l'usage de *Xen* sur un *MPSoC Xilinix Zynq UltraScale+* [146].

8.6 Partitionnement

Xen propose trois types de partitions différentes:

- Les partitions de type *PV* permettent la *paravirtualisation* totale du système invité. Elles nécessitent une adaptation de ce dernier mais aucun support matériel n'est *a priori* requis. Ces partitions offrent de bonnes performances. Il s'agit du mode originel de *Xen* pour l'architecture *x86*.

[21] Cette version correspond à l'introduction des processeurs *Intel Pro* en 1995.

- Les partitions de type *HVM* permettent la virtualisation assistée par le matériel. Elles nécessitent des extensions matérielles (voir la sous-section 8.2) mais aucune modification du système d'exploitation hôte[22]. Les performances sont généralement moindre que pour les partitions de type *PV*.
- Les partitions *PVH* cherchent à offrir le meilleur des deux types de partitions décrits ci-dessus. Certaines parties du systèmes (les entrées/sorties par exemples) sont paravirtualisées et d'autres (comme le *CPU*) reposent sur de la virtualisation assisté par le matériel. Ce type de partition offre souvent de meilleures performances que les partitions *PV* et *HVM* sans avoir besoin de modifiant autant le noyau hôte.

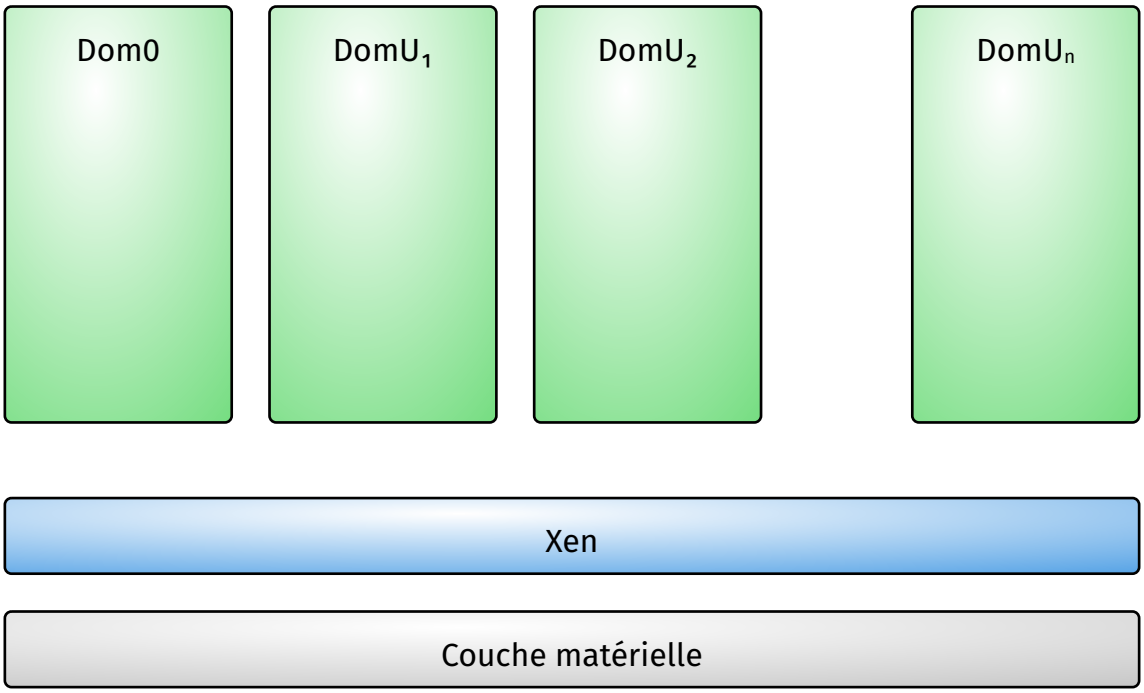

Fig. 6. – Architecture de *Xen*

Xen utilise le terme de *domaine* pour qualifier les conteneurs des machines virtuelles en cours d'exécution. Il existe trois types de domaines:

- Le domaine 0 (abrégé *dom0*) désigne un domaine privilégié qui est automatiquement lancé au démarrage de l'hyperviseur. Le système d'exploitation hôte est généralement une distribution *Linux* modifiée (voir la section 8.6.5).
- Les domaines utilisateurs (abrégé *domU*) sont les domaines qui contiennent les OS invités. Il existe deux types de tels domaines. Les domaines de paravirtualisation et les domaines *HVM*.
- *dom0less*.

8.6.1 Stub domains

Aparté: Qu'est-ce qu'un stub domain?

Un *stub domain* (ou *stubdomain*) est une mini-machine virtuelle légère dédiée à exécuter un seul service isolé dans Xen (comme l'émulateur QEMU pour un invité). Au lieu d'exécuter ces services dans le dom0 privilégié où une faille compromettrait tout le système, on les isole dans des stub domains avec des privilèges minimaux. Cela améliore considérablement la sécurité.

[22]Ce dernier point est crucial pour support des systèmes d'exploitation à sources fermées, comme par exemple *Windows*

Un *stub domain* (ou *stubdomain*) est un domaine système spécialisé utilisé pour désagréger le domaine de contrôle (*dom0*) [147], [148]. Il s'agit d'un domaine léger dédié à l'exécution de services ou de pilotes spécifiques, notamment le modèle de périphérique *QEMU* associé à un domaine HVM.

L'avantage principal des *stub domains* réside dans l'amélioration de la sécurité par isolation. Traditionnellement, *QEMU* et d'autres services critiques s'exécutent dans le *dom0* avec des privilèges élevés. En cas de vulnérabilité de sécurité dans *QEMU*, un attaquant pourrait obtenir un accès privilégié au *dom0* et compromettre l'ensemble du système. En exécutant *QEMU* dans un *stub domain*, ce dernier est automatiquement déprivilégié (via `XEN_DOMCTL_set_target`) de sorte qu'il n'a de privilèges que sur le domaine HVM spécifique auquel il est associé.

La plupart des *stub domains* sont basés sur le système d'exploitation minimaliste *Mini-OS* [54], bien que des travaux aient été menés sur des *stub domains* basés sur *Linux*.

8.6.2 Partitionnement spatial

L'hyperviseur *Xen* assure l'isolation mémoire entre les domaines en s'appuyant sur plusieurs techniques de gestion de la mémoire qui varient selon le type de virtualisation utilisé et les capacités matérielles disponibles. L'objectif principal est de garantir qu'aucun domaine ne puisse accéder à la mémoire d'un autre domaine sans autorisation explicite, tout en maintenant des performances acceptables.

Pour les domaines paravirtualisés, *Xen* a introduit une technique innovante baptisée *direct paging* [149]. Dans ce modèle, le système d'exploitation invité est conscient de la distinction entre les adresses physiques (*machine addresses*) réelles et les adresses pseudo-physiques qu'il manipule. Les tables de pages du système invité mappent directement les adresses virtuelles vers les adresses machines physiques. Cependant, pour maintenir les invariants de sécurité, *Xen* impose que toutes les mises à jour des tables de pages passent par des hypercalls tels que `HYPERVISOR_update_va_mapping` et `HYPERVISOR_mmuext_op`. Ces hypercalls permettent à *Xen* de vérifier que chaque domaine ne modifie que les pages qui lui appartiennent et ne crée pas de mappages de pages de tables inscriptibles. Le système d'exploitation invité dispose d'un accès en lecture seule aux véritables tables de pages mais doit obligatoirement utiliser ces hypercalls pour toute modification.

Pour les domaines HVM (*Hardware Virtual Machine*) qui exécutent des systèmes d'exploitation non modifiés, *Xen* peut utiliser deux approches selon les capacités du processeur. Sur les processeurs plus anciens sans support matériel de virtualisation mémoire, *Xen* utilise des *shadow page tables*. Cette technique consiste à maintenir dans l'hyperviseur des copies des tables de pages des invités qui traduisent directement les adresses virtuelles invitées vers les adresses machines physiques. Chaque modification des tables de pages par l'invité provoque une interception par l'hyperviseur qui met à jour les tables fantômes correspondantes. Cette approche impose un coût de performance significatif en raison du nombre élevé de sorties *VM* (*VM exits*) générées.

Sur les processeurs récents équipés de support matériel pour la virtualisation mémoire, *Xen* exploite les fonctionnalités *EPT* (*Extended Page Tables*) sur *Intel* ou *NPT* (*Nested Page Tables*) sur *AMD*. Ces technologies implémentent une traduction d'adresses en deux étapes : le système invité maintient ses propres tables de pages qui traduisent les adresses virtuelles vers les adresses pseudo-physiques, puis le matériel utilise une seconde table gérée par l'hyperviseur pour traduire les adresses pseudo-physiques vers les adresses machines réelles. Cette approche élimine le besoin de maintenir des tables fantômes et réduit drastiquement le

nombre d'interventions de l'hyperviseur, améliorant les performances jusqu'à 48% pour les charges de travail intensives en gestion mémoire selon certaines études.

Sur l'architecture *ARM*, *Xen* s'appuie sur la traduction d'adresses en deux étapes (*2-stage translation*) fournie par le matériel depuis *ARMv7* avec les extensions de virtualisation [150]. Cette fonctionnalité permet à l'hyperviseur de contrôler la vue mémoire de chaque invité tout en laissant le système d'exploitation invité gérer librement son propre espace d'adressage virtuel.

Pour les systèmes critiques nécessitant un partitionnement spatial strict, *Xen* propose depuis sa version *4.12* un mode *dom0less* qui permet de démarrer plusieurs domaines en parallèle directement depuis l'hyperviseur au moment du boot, sans intervention du domaine privilégié *dom0* [151]. Cette approche facilite la mise en œuvre d'un partitionnement statique où l'ensemble du système (invités et communications) est défini statiquement, garantissant un comportement déterministe et une cohérence après redémarrage. Chaque domaine dispose d'un accès direct au matériel protégé par l'iommu et bénéficie d'une isolation stricte vis-à-vis des autres domaines. Des fonctionnalités avancées comme le *cache coloring* peuvent être activées pour assurer un partitionnement complet du cache, où chaque machine virtuelle se voit allouer ses propres entrées de cache sans partage, permettant ainsi aux applications temps réel d'atteindre une latence d'interruption déterministe.

L'isolation mémoire de *Xen* a été conçue avec une architecture en micronoyau où l'hyperviseur lui-même reste minimal et délègue la gestion des pilotes et l'accès matériel au domaine privilégié *dom0*. Cette séparation des préoccupations renforce la surface d'attaque réduite de l'hyperviseur. Le projet *Xen* a également entrepris un effort de conformité avec les règles *MISRA C* pour faciliter la certification dans les environnements critiques où la sûreté et la sécurité sont primordiales [152].

8.6.3 Partitionnement temporel

Xen offre une abstraction des processeurs physiques appelée *vCPU* (*Virtual CPU*). La correspondance entre processeur physique et *vCPU* est souple puisqu'il n'est pas nécessaire qu'un *vCPU* corresponde toujours au même processeur physique, ni même qu'il y ait un processeur physique disponible pour chaque *vCPU* à un instant donné. En particulier, il est possible d'avoir davantage de *vCPU* que de processeurs physiques. On parle alors d'*oversubscription*. Toutefois une *VM* ne peut pas avoir plus de *vCPU* que le nombre de processeurs physiques disponibles.

L'allocation des *vCPU* sur les processeurs physiques est géré par un ordonnanceur similaire à ceux utilisés pour des processus dans un *GPOS*. La distribution officielle de *Xen* offre deux ordonnanceurs généralistes:

- *Credit Scheduler* est l'ordonnanceur historique du projet *Xen* et il est encore à ce jour l'ordonnanceur par défaut [153]. Il permet une répartition juste des processeurs physiques entre les *VM*. Plus précisément chaque *VM* dispose d'une fraction du temps *CPU* total qui est proportionnel à un poids configuré à l'avance. De plus l'ordonnanceur garantit qu'un processeur physique ne restera pas inactif s'il y a une tâche pouvant y être exécutée[23].
- *Credit2 Scheduler* est une évolution de *Credit Scheduler* [154]. Il est conçu pour être plus juste et offrir de meilleures performances sur les serveurs dotés d'un grand nombres de processeurs. Il est disponible depuis la version *4.8* de *Xen*.

[23] On dit que l'ordonnanceur est *work-conserving*.

Depuis sa version *4.5*, *Xen* distribue également un ordonnanceur temps réel baptisé *RTDS*. Nous donnons plus d'informations sur ce dernier dans la sous-section 8.6.4.

8.6.4 Déterminisme

À ses débuts *Xen* a été conçu pour le *cloud computing*. En particulier, le projet met l'accent sur les garanties que les ressources louées par des clients seront effectivement disponibles lorsque leurs *VMs* les requerront. Les ordonnanceurs que nous avons vus dans la sous-section 8.6.3 cherchent donc à être aussi juste que possibles. Cette garantie est en contradiction avec les besoins du temps réel. En effet une tâche critique peut avoir soudainement besoin de beaucoup de ressources, si ce n'est la totalité des ressources.

Le projet *RT-Xen* visait à dôter *Xen* d'un ordonnanceur temps réel pour ces *vCPU*. À l'origine le projet est développé à partir d'un *fork* de la branche *4.3* de *Xen*. Il a été intégré dans *Xen* à partir de la version *4.5* sous la forme d'un nouvel ordonnanceur baptisé *RTDS* (*Real-Time Deferrable Server*) [155]. L'objectif de *RTDS* est d'assurer que les *VMs* reçoivent un temps *CPU* minimal garanti. Cet ordonnanceur supporte les plateformes *SMP*.

Il est donc possible d'exécuter dans une *VM* un *RTOS*. Par exemple *RTEMS* peut être exécuté dans une telle configuration sur architecture *ARM*.

8.6.5 OS invités supportés

Xen étant un paravirtualisateur, il nécessite un support spécifique des OS invités, que ce soit pour les *VM* s'exécutant dans le domaine privilégié *dom0* ou les *VM* s'exécutant dans les domaines *domU*. Pour le domaine *dom0*, il offre un support pour de nombreuses distributions *GNU/Linux* ainsi que quelques autres noyaux de type *UNIX*. Plus d'informations sont disponibles. Pour le domaine *domU*, *Xen* offre aussi un large support pour les OS invités.

8.7 Corruption de la mémoire

L'hyperviseur *Xen* ne dispose pas d'un système de journalisation des erreurs mémoires. En revanche, il transmet ces erreurs au système d'exploitation exécuté dans le domaine privilégié *Dom0*. Il est alors possible d'utiliser les outils livrés avec ce système pour journaliser ces erreurs. Il est par exemple possible d'exécuter un noyau *Linux* dans le domaine *Dom0* et d'utiliser les fonctionnalités de pilotage de la mémoire *ECC* décrites en sous-section 2.6.

8.8 Perte du flux d'exécution

Comme pour les autres systèmes d'exploitation, *Xen* est susceptible aux attaques visant à détourner le flux d'exécution. La protection contre ces attaques repose principalement sur:

- L'utilisation de langages de programmation sûrs pour certaines composantes,
- Les mécanismes de protection matériels (*Intel CET*, *ARM BTI*) lorsqu'ils sont disponibles sur la plateforme cible,
- Les pratiques de développement sécurisé et les revues de code approfondies.

La nature critique de l'hyperviseur *Xen* en fait une cible privilégiée pour les attaquants. Une compromission du flux d'exécution au niveau de l'hyperviseur peut potentiellement affecter tous les domaines hébergés, d'où l'importance cruciale de ces mécanismes de protection [7].

8.9 Monitoring et profilage

Xen propose plusieurs outils pour le monitoring des performances et de l'état du système [156], [157]:

- **xentop**: Utilitaire similaire à *top* pour afficher des informations sur tous les domaines s'exécutant sur un système *Xen*. Il permet d'identifier les domaines responsables des charges les plus élevées en I/O ou en traitement.
- **xenmon**: Outil utile pour surveiller les performances des domaines *Xen*, particulièrement pour identifier les domaines responsables des charges I/O ou processeur les plus importantes.
- **RRD (*Round Robin Databases*)**: *Xen* expose des métriques de performance via des bases de données RRD. Ces métriques peuvent être interrogées via HTTP ou à travers l'outil *RRD2CSV*. *XenCenter* utilise ces données pour produire des graphes de performance système affichant l'utilisation du CPU, de la mémoire, du réseau et des I/O disque.
- **Intégration avec des outils tiers**: *Xen* supporte l'intégration avec des outils de monitoring via *NRPE* (*Nagios Remote Plugin Executor*) et *SNMP* (*Simple Network Management Protocol*), permettant l'utilisation de solutions de monitoring tierces.

L'écosystème libre et gratuit pour monitorer *Xen* semble assez limité. Il est possible que les grands acteurs du *cloud computing* aient développé leurs propres outils en interne.

8.9.1 L'outil *xl*

L'outil *xl* est livré avec *Xen* et offre des fonctionnalités basiques pour observer l'état des domaines en cours d'exécution.

La couche logicielle introduite par la virtualisation peut introduire des régressions de performance dans les logiciels applicatifs par rapport à une exécution directement sur un OS *bare-metal*. Dans ce contexte, il est nécessaire d'utiliser des outils de profilage dédiés à l'hyperviseur. Dans cette section, on présente trois outils de profilage pour *Xen*: *Xenprof*, *XenTune* et *xentrace*.

8.9.2 *Xenoprof*

8.9.3 *XenTune*

8.9.4 Le traceur *xentrace*

Le logiciel *xentrace* [158] est un outil distribué dans *Xen*. Il permet de tracer l'activité des CPU virtuels et ainsi de savoir ce que fait une machine virtuelle sur un CPU donné. Ces données sont collectées grâce à des *tracepoints* positionnés à des endroits clés du code de *Xen*. Ils sont activés via *xentrace* lorsqu'il est exécuté dans le domaine *dom0*. Ce dernier produit alors un fichier binaire qui peut ensuite être analysé par *xenanalyze*[24].

- La commande *xl* livrée avec *Xen* offre des fonctionnalités basiques de monitoring via ses méta-options `top` et `list`.
- *Xen Orchestra* est un solution de monitoring pour l'écosystème XenServer et *XCP-ng*.
- *Zabbix* est un système de monitoring open source qui peut surveiller les performances des domaines.
- *Netdata* peut être utilisé avec *Xen*.
- *Xenoprof*

[24]Contrairement à *xentrace*, *xenanalyze* n'est pas distribué avec *Xen*.

8.10 Programmation *bare-metal*

Il est possible d'exécuter des applications *bare-metal* dans les domaines de *Xen* pour les langages *C*, *Rust* et *OCaml*.

Le projet *MirageOS* a développé un environnement d'exécution complet pour le langage de programmation *OCaml* sur des partitions de type *pvh*.

8.11 Watchdog

Xen permet la mise en place d'un *watchdog* dans *dom0* ou dans des domaines utilisateurs. L'exemple ci-dessous met en place un *watchdog* qui doit être réinitialisé d'en un laps de temps de 30 secondes:

```c
#include <stdlib.h>
#include <stdio.h>
#include <xenctrl.h>

int main(void) {
  xc_interface *h = xc_interface_open(NULL, NULL, 0);
  if (!h)
    return EXIT_FAILURE;

  // Configure le timeout à 30 secondes.
  int timeout = 30;
  int id = xc_watchdog(h, 0, timeout);
  if (id <= 0)
    goto failed;
  printf("Watchdog initialisé\n");

  // Réinitialise le watchdog toutes les 15 secondes.
  while (1) {
    sleep(15);
    if (xc_watchdog(h, id, timeout))
      goto failed;
    printf("Watchdog rechargé\n");
  }

failed:
  xc_interface_close(h);
  return EXIT_FAILURE;
}

```

Liste 10. – Exemple d'interaction avec un *watchdog* sous *Xen*.

Pour compiler ce programme, tapez:

```
$ make watchdog -C ./xen
```

et pour lancer le programme dans le domaine utilisateur, tapez:

```
$ ./xen/artifacts/watchdog
```

Il suffit alors de fermer ce programme avec `CTRL-C` pour cesser de réinitialiser le *watchdog*. Par défaut, *Xen* terminera le domaine utilisateur. Ce comportement peut être changé avec l'option `on_watchdog` du fichier de configuration de *xenlight*. Par exemple, l'option `on_watchdog='reboot'` provoquera le redémarrage du domaine.

Xen distribue un service *xenwatchdogd* pour lancer les *watchdogs* [159]. Le service est lancé en précisant un *timeout* et un *sleep* ainsi:

```
$ xenwatchdogd 30 15
```

Notez que l'outil `xl` permet de choisir quelle stratégie appliquer lorsque le *watchdog* est déclenché via le paramètre `on_watchdog`. Plus d'informations sont disponibles dans la page de manuelle `xl.cfg`.

Linux dispose d'un pilote *xen_wdt* pour le *watchdog* virtuel de *Xen* qui implèmente l'API décrit dans la section 2.10.1.

8.12 Masquage des interruptions

Les mécanismes de gestion des interruptions dépendent du type de partitions considérées.

Dans les partitions *PV*, les interruptions matérielles sont virtualisées via le concept d'*event PIRQ* [160]. Plus précisément des *upcall* sont utilisés en guise de notifications, c'est-à-dire que l'hyperviseur appelle une routine du système invité pour l'informer de la survenue d'un événement. Il est possible de masquer ces événements [160].

Dans le cas d'une partition *HVM*, le système invité a un accès direct aux interruptions matérielles. Tout ce qui est possible en *bare-metal* devrait être également possible dans une telle partition.

Finalement dans le cas d'une partition *PVH*, *Xen* utilise le support matériel pour la majorité des interruptions matérielles comme en *HVM* mais il est toujours possible d'utiliser des périphériques paravirtualisés et les *events PIRQ* pour gérer les interruptions matérielles [161].

8.13 Maintenabilité

Xen est écrit à 93% en langage *C* pour un total de 581 193 *SLOC* dont 45 220 *SLOC* pour les pilotes. Ces chiffres inclus toutes les architectures, ce qui ne tient pas compte d'une grande disparité entre les parties les plus anciennes pour les partitions *PV* sur *x86* et les parties plus récentes pour les partitions *PVH* sur *ARM*.

Xen est réputé pour avoir un *TCB* plus important que d'autres hyperviseurs, notamment dû à la taille importante de ces sources. Il est toutefois important de souligner que le volume de code varie d'un facteur 10 entre les des architectures les mieux supportées, à savoir *x86* et *ARM*.

Un autre facteur important qui augmente la *TCB* est l'usage d'un noyau *Linux* dans le *dom0*. La compromission de ce noyau compromettant tout le système, il ne peut en être exclu. Il est possible de réduire l'influence du *dom0* sur la *TCB* en utilisant des noyaux personnalisés.

8.14 Licences

Xen est un logiciel libre distribué majoritairement sous licence `GPLv2`. Toutefois certaines parties sont distribuées sous des licences plus permissives afin de pas contraindre les licences

des logiciels applicatifs ou des *OS* portés sur *Xen*. Ces exceptions sont spécifiées dans les entêtes des fichiers concernés. Plus d'informations sont disponibles dans le fichier `COPYING` du dépôt git [162].

8.15 Temps de démarrage

Aparté: Qu'est-ce que dom0less?

Le mode *dom0less* est une fonctionnalité *Xen* permettant de démarrer des machines virtuelles invitées directement depuis l'hyperviseur, sans attendre que le *dom0* soit complètement initialisé. Cela réduit drastiquement le temps de démarrage (de plusieurs secondes à moins d'une seconde) pour les systèmes embarqués et temps-réel où chaque milliseconde compte.

Le mode *dom0less* est une fonctionnalité de *Xen* permettant d'accélérer significativement le démarrage des domaines [151], [163]. Cette optimisation répond à un besoin critique dans les systèmes embarqués et temps-réel où le temps de démarrage est déterminant.

Traditionnellement, le démarrage d'un domaine depuis l'initialisation du système nécessite plusieurs étapes séquentielles prenant plusieurs secondes:
1. Démarrage de l'hyperviseur *Xen*
2. Démarrage du noyau *dom0*
3. Initialisation de l'espace utilisateur du *dom0*
4. Disponibilité de l'outil `xl` pour créer les domaines

Avec *dom0less*, *Xen* démarre les domaines sélectionnés directement depuis l'hyperviseur au moment du boot, en parallèle sur différents cœurs physiques. Cette approche permet d'obtenir des temps de démarrage sous-secondes pour les systèmes temps-réel. Le temps de démarrage total devient approximativement égal à: temps_xen + temps_domU, éliminant ainsi le surcoût du démarrage du *dom0* et de son espace utilisateur.

Cette fonctionnalité est particulièrement adaptée aux systèmes à partitionnement statique où plusieurs domaines doivent démarrer rapidement lors de l'initialisation de l'hôte. Elle s'intègre désormais dans le projet *Hyperlaunch* qui généralise cette approche.

9 XtratuM

XtratuM en bref

- **Type** : Hyperviseur temps-réel type 1 qualifié ECSS
- **Langage** : C
- **Architectures** : x86-32, SPARC/LEON (LEON2/3/4), ARM v7/v8
- **Usage principal** : Spatial, aéronautique (IMA - Integrated Modular Avionics)
- **Points forts** : Qualifié ECSS catégorie B, 1000+ satellites déployés, ordonnancement cyclique ARINC-653, isolation temporelle et spatiale forte, health monitoring
- **Limitations** : Version NG propriétaire, écosystème limité, principalement orienté spatial/aéro
- **Licences** : GPL v2 (version libre) + version propriétaire XtratuM/NG (fentISS)
- **Missions spatiales** : Galileo, JUICE, SWOT, PLATiNO, MERLIN, SVOM

XtratuM est un hyperviseur temps-réel de type 1 qualifié pour un usage dans le spatial. Le projet est initié en 2004 au sein de l'institut *Automática e Informática Industrial* (ai2) de l'*Universidad Politécnica* de Valence en Espagne [164], [165]. Ces travaux universitaires ont abouti à la création de l'entreprise *fentISS* [166] en 2010 avec le soutien du *CNES* et du groupe *Airbus* [165]. L'hyperviseur *XtratuM* est désormais maintenu et développé par *fentISS*. *XtratuM* a été conçu pour être exécuté sur de l'embarqué critique en donnant de fortes garanties quant à l'isolation spatiale et temporaire de ses partitions [164]. L'entreprise

Le succès de *XtratuM* dans le spatial est remarquable: son hyperviseur temps-réel est désormais déployé dans plus d'un millier de satellites et engins spatiaux [167], [168], en faisant l'un des logiciels système les plus largement adoptés en orbite. Cette présence massive témoigne de la maturité et de la fiabilité du système dans des environnements opérationnels critiques.

9.1 Architectures supportées

Les premières versions de *XtratuM* ont supporté les architectures *x86-32*, *PowerPC* et *SPARC* (*LEON2*). Toutefois les brochures récentes ne mentionnent plus les architectures *x86* et *PowerPC*, ce qui laisse à penser que leur support n'a pas été maintenu et que le support pour l'architecture *x86-64* n'a jamais existé.

D'après les dernières brochures [169], *XtratuM* supporte les architectures suivantes: *ARM-v7*, *ARM-v8*, *SPARC*, *RISC-V*. En particulier, il supporte les architectures *SPARCv8*, *LEON3* et *LEON4* qui sont utilisées dans des missions spatiales. Le support se fait via des *BSP*.

9.2 Support multi-processeur

XtratuM était originellement développé sur des architectures monoprocesseur *x86* et *LEON*. Le support multi-cœur a donc nécessité de profondes modifications. C'est une approche à base de *spinlock* qui fut adopter pour assurer la synchronisation des sections critiques du noyau [170]. Chaque partition peut allouer plusieurs cœurs via une couche d'abstraction sous forme de *CPU* virtuels [170].

9.3 Partitionnement

Le partitionnement de *XtratuM* est conforme à la norme *ARINC-653*. Il est donc conçu pour assurer une excellente isolation temporelle et spatiale de ses partitions. Chaque partition peut contenir un système d'exploitation ou une application *bare-metal*.

9.3.1 Partitionnement spatial

Les partitions *XtratuM* sont exécutés en mode utilisateur.

La plateforme *LEON2* ne disposait pas nécessairement de *MMU*. Ce n'est plus le cas des architectures *LEON3* et *LEON4* qui incluent un tel dispositif. *XtratuM* intègre donc un support pour ce dispositif. En plus d'améliorer l'isolation spatiale en prévenant les lectures non autorisées, les *MMU* permettent d'implémenter des *IPC* plus rapides [171].

9.3.2 Partitionnement temporel

XtratuM implémente une politique d'ordonnancement cyclique conforme à la norme *ARINC-653*. Dans le domaine temporel, *XtratuM* alloue le CPU aux partitions selon un plan défini lors de la configuration [172]. Il s'agit d'un ordonnancement cyclique statique (*static cyclic scheduling*) où tous les intervalles d'exécution des partitions sont déterminés avant l'exécution.

Chaque partition est ordonnancée pour un créneau temporel (*time slot*) défini par un temps de démarrage et une durée. Durant ce créneau, *XtratuM* alloue les ressources système à la partition. Lorsque le créneau de la partition est écoulé, *XtratuM* force un changement de contexte vers la partition suivante selon le plan cyclique défini.

Le système utilise un ordonnancement hiérarchique à deux niveaux: *XtratuM* gère l'ordonnancement des partitions au niveau supérieur, tandis que chaque partition peut exécuter son propre ordonnanceur pour ses tâches internes.

9.3.3 Communication inter-partition

XtratuM fournit un mécanisme de communication inter-partition (*IPC*) conforme à la norme *ARINC-653* [172], [173]. Ce mécanisme permet l'échange de messages entre les partitions *ARINC* s'exécutant sur la même carte.

Un *canal* (*channel*) est un lien logique entre une partition source et une ou plusieurs partitions de destination. Les partitions peuvent envoyer et recevoir des messages via plusieurs canaux à travers des points d'accès définis appelés *ports*. *XtratuM* utilise deux interruptions virtuelles pour notifier les partitions de la disponibilité de nouveaux messages dans les ports de destination.

La norme *ARINC-653* définit deux modes de transfert de messages:

- **Mode échantillonnage** (*sampling mode*): Supporte les messages multicast envoyés d'une seule source vers plusieurs destinations. La transmission d'un message sur un canal copie le message du port d'échantillonnage source vers les tampons de tous les ports d'échantillonnage de destination. Ce mode convient aux données périodiques où seule la dernière valeur est pertinente.
- **Mode file d'attente** (*queuing mode*): Ne supporte que les messages unicast. Les messages sont mis en file d'attente et traités séquentiellement. Ce mode convient aux événements sporadiques qui nécessitent un traitement ordonné.

9.3.4 Déterminisme

9.3.5 OS invités supportés

L'hyperviseur de *XtratuM* offre un support pour les *OS* suivants:

- Le *RTOS* LithOS développé par *fentISS* et conforme *ARINC-653* [172], [174],
- Le *RTOS RTEMS*, notamment sur les plateformes *LEON* [175].

- Le noyau *Linux* [175],
- Le micronoyau temps réel *ORK+* (*Open Ravenscar Kernel*). Il a été porté sur *XtratuM* en 2011 [176]. Il permet le développement d'applications en *Ada* avec le profile *Ravenscar*.

9.3.6 Programmation *bare-metal*

Il est possible d'exécuter des applications *bare-metal* dans les partitions de *Xtratum* à condition d'adapter un *RTE* du langage de programmation pour l'*API* de *Xtratum*. Il est possible de programmer en *bare-metal* avec les langages suivants:

- *C* grâce au *RTE XRE* (*XUL Runtime Environment*) développé par *fentISS*,
- *Ada* avec le profile *Ravenscar*,
- *Rust* peut être utilisé en *bare-metal* en interfaçant avec l'*ABI (Application Binary Interface) C* de *XtratuM*. Une *crate* est disponible [177].

9.4 Corruption de la mémoire

Le *Health Monitor* d'*XtratuM* est capable de journaliser les *MCE (Machine Check Exception)* en cas d'erreur de mémoire non corrigible. Comme pour les autres erreurs, il est possible de configurer un paliatif.

9.5 Outils

fentISS propose une suite d'outils pour faciliter le développement avec *XtratuM*:

- **XPM** : plugin Eclipse pour la gestion de projets *XtratuM*
- **Xoncrete** : analyse et génération d'ordonnancement
- **Xcparser** : configuration de l'hyperviseur
- **Xtraceview** : support d'observabilité
- **SKE** : simulateur *XtratuM* sur serveurs

9.5.1 *Health monitoring*

XtratuM intègre un service de *health monitoring* conforme à la norme *ARINC-653* [172]. Ce service permet la détection et la gestion des défaillances des partitions.

Lorsqu'un événement de *health monitoring* est déclenché, le système peut entreprendre des actions correctives telles que des changements de mode. Par défaut, le Plan 1 (mode maintenance) est le plan exécuté lorsqu'un événement sélectionne un changement de mode comme action.

Le principe d'isolation des partitions garantit que la défaillance d'une partition n'affecte pas les autres partitions. Cependant, bien qu'une partition ne puisse pas affecter les autres partitions, la défaillance peut toujours se produire et potentiellement conduire à une défaillance du système global. Le service de *health monitoring* permet de limiter ces risques par une détection précoce et des actions de récupération appropriées.

9.6 Support de *watchdog*

9.7 Temps de démarrage

9.8 Qualifications et certifications

ECSS-Qualified?

XtratuM est qualifié selon la norme *ECSS* (*European Cooperation for Space Standardization*) catégorie B [178]. Cette qualification en fait un hyperviseur adapté aux missions spatiales critiques.

L'hyperviseur a été qualifié initialement pour les processeurs *SPARC-Leon* et *ARM Cortex-R4/R5* et *A9*. L'entreprise *fentISS* continue de travailler sur la qualification de nouvelles versions, notamment *XtratuM Next Generation* pour lequel un processus de qualification ECSS niveau B est en cours.

9.9 Masquage des interruptions

9.10 Licences

À l'origine *XtratuM* était un projet open source sous licence *GPLv2* [179]. Cette version ne semble plus être développée et *fentISS* distribue une réécriture du noyau appelée *XNG* (*XtratuM Next Generation*) sous licence propriétaire. L'entreprise *fentISS* ne semble pas communiquer sur ses licences ou sa politique tarifaire. Les modalités et les coûts des licences sont négociés avec *fentISS* en fonction des besoins du projet.

9.11 Draft

XtratuM virtualises la mémoire, les timers et les interruptions.

XtratuM fait parti du projet *SAFEST* [180]. Il s'agit d'un projet visant à faire collaborer différents acteurs du secteur aérospatial européen afin d'améliorer les performances et de réduire les coûts.

IMA (*Integrated Modular Avionics*) est une tendance dans l'avionique à ramener au niveau de calculateurs modulaires identiques des fonctions logicielles auparavant prises en charge par des calculateurs dédiés.

XtratuM/NG(abrégé *XNG*) est une version plus récente de l'hyperviseur qui offre un meilleur support multi-cœur.

10 Tableaux comparitifs

Type d'OS	Avantages	Inconvénients
Unikernel	• Petite surface d'attaque • Petite empreinte mémoire • Faible temps de démarrage	• Débogage difficile • Recompilation & déploiement pour chaque changement
Hyperviseur	• Optimisation des resources • Isolation	
Classique	• Support matériel • Outil de débogage	

10.1 Architectures supportées

Le tableau suivant résume le support de ces architectures de processeur pour les systèmes d'exploitation de cette étude. Lorsque l'OS est un hyperviseur, il s'agit du support pour le matériel surlequel est exécuté l'hyperviseur.

OS	x86-32	x86-64	ARM v7	ARM v8	PowerPC	MIPS	RISC-V	SPARC
Linux [8]	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
KVM [181]	Oui	Oui	Oui	Oui	Oui	Non	Non	Non
MirageOS [182], [183]	Non	Oui	Non	Oui	Non	Non	Non	Non
PikeOS [184]	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui
ProvenVisor [95]	Non	Non	Non	Oui	Non	Non	Non	Non
RTEMS 6.3	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
seL4 [121]	Oui	Oui	Oui	Oui	Non	Non	Oui	Non
Xen	Non	Non	Oui	Non	Oui	Non	Non	Non
XtratuM	Non	Non	Oui	Non	Oui	Non	Non	Non

10.2 SLOC

Une première mesure simple de la complexité d'un programme est donné par la métrique *SLOC* (pour *source lines of code*) qui mesure la taille d'un programme informatique en nombres de lignes dans son code source. Les sources de `PikeOS`, `ProvenVisor` and `XtratuM` étant fermées et n'ayant pas trouvé de données concernant le *SLOC* pour ces OS, nous les excluons de cette section.

Pour les OS open-sources, nous avons utilisé l'outil `SLOCCount`[185] pour effectuer ces mesures. Cet outil ne compte pas les commentaires.

OS	Total (SLOC)	Pilotes (SLOC)	Langage
Linux & KVM	26 927 724	18 920 036	C (98%)
MirageOS	9,075	?	OCaml (99%)
PikeOS	?	?	C (?)
ProvenVisor	?	?	C (?)
RTEMS	1 990 023	71,238	C (96%)

seL4	68 175	1 086	C (87%)
Xen	581 193	45 220	C (93%)
XtratuM	?	?	C (?)

10.3 Politiques d'ordonnancement

Le tableau suivant présente les politiques d'ordonnancement disponibles pour chaque système d'exploitation étudié, en se basant sur les six politiques d'ordonnancement implémentées par Linux (*SCHED_FIFO*, *SCHED_RR*, *SCHED_DEADLINE*, *SCHED_OTHER*, *SCHED_BATCH*, *SCHED_IDLE*). Le tableau indique également le support du mode tick-less et la capacité à exécuter simultanément des tâches temps réel et normales.

OS	FIFO	Round-Robin	Dead-line	CFS/ Équitable	Batch	Idle	Tick-less	Mixte TR/ Normal
Linux	Oui[25]	Oui[26]	Oui[27]	Oui[28]	Oui[29]	Oui[30]	Oui[31]	Oui
MirageOS	Non	Non	Non	Non	Non	Non	N/A[32]	Non
PikeOS	Non	Non	Non	Non	Non	Non	?	Oui[33]
ProvenVisor	?	?	?	?	?	?	?	?
RTEMS	Non	Non	Oui[34]	Non	Non	Non	Non	Oui[35]
seL4	Non	Oui[36]	Non	Non	Non	Non	Non[37]	Limité[38]
Xen	Non	Non	Oui[39]	Oui[40]	Non	Non	Oui[41]	Oui[42]
XtratuM	Non	Non	Non	Non	Non	Non	Non[43]	Limité[44]

10.4 Support *watchdog*

Le tableau suivant résume le support des mécanismes *watchdog* pour chaque système d'exploitation étudié. Les *watchdogs* matériels désignent la capacité d'interagir avec des périphériques *watchdog* physiques, tandis que les *watchdogs* logiciels permettent de surveiller l'état des processus ou services sans matériel dédié.

[25] *SCHED_FIFO*
[26] *SCHED_RR*
[27] *SCHED_DEADLINE*
[28] *SCHED_OTHER* via *CFS*
[29] *SCHED_BATCH*
[30] *SCHED_IDLE*
[31] Support via `CONFIG_NO_HZ`
[32] Pas d'ordonnancement préemptif, uniquement coopératif via *Lwt*
[33] Partitionnement temporel adaptatif breveté
[34] *EDF* (*Earliest Deadline First*)
[35] Priorité fixe + *EDF* + *CBS*
[36] Pour tâches de même priorité
[37] Utilise des ticks périodiques
[38] Domaines temporels isolés
[39] *RTDS* (*Real-Time Deferrable Server*)
[40] *Credit* et *Credit2*
[41] Support pour VMs invitées
[42] Peut mixer VMs temps réel et best-effort
[43] Ordonnancement cyclique statique *ARINC-653*
[44] Possible au sein de chaque partition

OS	Watchdog matériel	Watchdog logiciel	Notes
Linux	Oui	Oui	API unifiée via `/dev/watchdog`. Support logiciel via *systemd*.
MirageOS	Indirect	Non	Pas d'API native. Dépend de l'environnement (ex: *xencontrol* pour Xen).
PikeOS	?	?	Information non disponible publiquement.
ProvenVisor	Probable	Probable	Support non documenté publiquement. Watchdog matériel ARM, virtualisation possible, surveillance par hyperviseur.
RTEMS	Oui	Oui	Support au niveau BSP. Watchdog logiciel via *Timer Manager*.
seL4	Partiel	Oui	Pas d'API unifiée. Support dans certains BSP. Watchdog logiciel via *Timer Manager*.
Xen	Oui	Oui	Support dans *dom0* et domaines utilisateurs. Service *xenwatchdogd* disponible. API compatible Linux via *xen_wdt*.
XtratuM	?	?	Information non disponible publiquement.

10.5 OS invités supportés par hyperviseur

Le tableau Tableau 5 récapitule les systèmes d'exploitation invités supportés par les différents hyperviseurs étudiés dans ce document.

Hyperviseur	**OS invités supportés**
KVM	• GNU/Linux (toutes distributions majeures) • Windows • BSD (FreeBSD, OpenBSD, NetBSD) • Solaris
PikeOS	• ELinOS (distribution Linux embarqué temps réel de SYSGO) • RTEMS • Systèmes conformes POSIX (Linux, Android) • Windows (dans partitions HVM sur x86)
ProvenVisor	• Information non disponible dans la documentation consultée
Xen	• GNU/Linux (nombreuses distributions) • Noyaux de type UNIX • Windows (via HVM) • BSD
XtratuM	• LithOS (RTOS conforme ARINC-653, développé par fentISS) • RTEMS • Linux • ORK+ (Open Ravenscar Kernel, micronoyau temps réel pour Ada)

Tableau 5. – OS invités supportés par hyperviseur

11 Glossaire

ABI. Application Binary Interface. interface binare-programme 97

AMP. Asymmetric multiprocessing. Architectures multiprocesseur dont chaque cœur exécute son propre programme. Les ressources mémoires et périphériques sont statiquement partagées entre les cœurs. Les cœurs peuvent être identiques (homogène) ou différents (hétérogène). Ces architectures visent à garantir un comportement déterministe et une bonne isolation des tâches. 7, 16, 46, 65, 69, 72, 85

API. Applicaton Programming Interface. Interface de programmation applicative. 13, 17, 28, 37, 42, 45, 48, 50, 67, 69, 73, 74, 77, 79, 97

BKL. Big Kernel Lock. todo 16, 46, 72

BSP. Board Support Package. todo 6, 45, 46, 48, 63, 65, 66, 67, 79, 95

CC. Critères communs. Ensemble de normes internationales spécifiant des critères à suivre pour évaluer la sécurité de systèmes d'information. Les certificats suivant ces normes distinguent sept niveaux d'assurance de EAL1 à EAL7. 71

COTS. Commercial off-the-shelf. todo 13

DRAM. Dynamic Random Access Memory. Type de barette de mémoire utilisée massivement comme mémoire principale sur les serveurs et ordinateurs personnels. 10

ECC. Error correction code. Code utilisé pour permettre la détection et la correction d'erreur dans des données. 10, 48, 49, 89

ECSS. European Cooperation for Space Standardization. Ensembles de normes utilisées dans le spatial. 69

ESA. European Space Agency. todo 63, 64, 68, 69

GNU. GNU is Not Unix. todo 15

GPOS. General-Purpose Operating System. Système d'exploitation généraliste offrant un large éventaille de services. 5, 6, 8, 9, 15, 35, 37, 65, 88

HVM. Hardware Virtual Machine. todo 85

IDE. Integrated Development Environment. Environnement de développement comprenant en général un éditeur de texte, un débogueur et un support pour des logiciels de gestion de versions comme git. 49

IPC. Inter-Process Communication. todo 38, 39, 40, 67, 74, 96

IRQ. Interrupt ReQest. Requête d'interruption envoyée à un PIC afin d'interrompre le processeur. 12, 78, 79

ISR. Interrupt Service Routine. Gestionnaire d'interruption exécutée lorsqu'une interruption matérielle ou logicielle survient. 11, 20, 21

IdO. internet des objets. Réseau d'objets embarqués et connectés au travers d'Internet. 53

MCE. Machine Check Exception. Erreur émise par le matériel qui est généralement fatale. 97

MMU. Memory Management Unit. Microcontrôleur dédié à la traduction des adresses virtuelles en adresses physiques. 8, 9, 17, 46, 65, 73

MPSoC. Multiprocessor System on a chip. Système embarqué complet sur un seul circuit intégré comprenant plusieurs processeurs généralement hétérogènes. 7, 17, 46, 65, 72, 85

MPU. Memory Protection Unit. Microcontrôleur chargé de vérifier les accès en mémoire. 8, 65

PIC. Programmable Interrupt Controller. Ensemble de microcontrôleurs dédiés à la programmation des interruptions matérielles et en particulier à leur masquage. 12

PMU. Performance Monitoring Unit. Registres matériels intégrés dans les microprocesseurs modernes afin de compter des événements bas niveau dans un but de profilage. 27, 77

RTE. RunTime Environment. todo 13, 37, 38, 50, 97

RTOS. Real-time Operating System. Système d'exploitation dédié pour le temps réel. Il offre des garanties fortes en matière de déterminisme. 5, 75, 77, 96

SMP. Symmetric multiprocessing. Architectures multiprocesseur dont les cœurs partagent les mêmes ressources et sont gérés par un seul système d'exploitation. Ces architectures visent à maximiser la charge de travail traitée. 7, 9, 15, 16, 36, 38, 46, 54, 65, 66, 69, 72, 85, 89

TCB. Trusted computing base. La base de calcul approuvé fait référence à l'ensemble des composants matériels et logiciels d'un système informatique dont la compromission conduit à celle du système entier. Plus cette base est petite et à fait l'objet de vérification, plus la sécurité du système est élevée. 35, 36, 53, 55, 56, 57, 59, 92

TEE. Trusted execution environment. Un environnement d'exécution de confiance est une zone sécurisée et isolée des autres environnements d'exécution. Cette zone est située dans le processeur. Elle garantit la confidentialité des données qui y sont stockées. 53

TLB. Translation Lookaside Buffer. Mémoire cache du processeur utilisée dans le but d'accélérer la traduction des adresses virtuelles en adresses physiques. 46, 48

VM. Virtual Machine. todo 13, 35, 83, 88, 89

WCET. Worst Case Execution Time. todo 46, 47, 48, 49, 56, 72, 75, 76, 78

bare-metal. Un environnement d'exécution bare-metal est un environnement dépourvu de système d'exploitation. On parle également de logiciel bare-metal lorsqu'il est possible de l'exécuter dans un tel environnement. 4, 5, 6, 7, 13, 36, 50, 59, 69, 79, 91, 92, 95, 97

bootloader. Programme prenant le relais du BIOS afin d'initialiser le matériel, puis de localiser et charger l'image d'un noyau. Il passe ensuite le relais au système d'exploitation qu'il vient d'initialiser. 32, 33, 64

espace noyau. todo 12, 65, 73

espace utilisateur. todo 17, 27, 32, 33, 65, 73

hard error. Corruption de la mémoire due à un dysfonctionnement matériel au niveau de la puce mémoire. 10

hyperviseur. Plate-forme de virtualisation permettant l'exécution de plusieurs systèmes d'exploitations simultanément sur une même machine physique. 5, 6

init system. Premier programme exécuté sur un système d'exploitation de type _UNIX_. 32

monolithique. todo 15

paravirtualisation. todo 83, 85

qualification. todo 68, 69

ramasse-miette. TODO 13

runtime. Environnement d'exécution. Dans le cas d'un langage de programmation managé, cela inclut son garbage collector. 37, 40

soft error. Corruption de la mémoire due à un événement exceptionnel et transitoire. Par exemple le rayonnement de fond peut produire un basculement de bits. 10

spinlock. todo 7, 21, 95

sécurité. Mesure prise pour prévenir les attaques d'une entité malveillante. 3

sûreté. Mesure prise pour veiller au bon fonctionnement d'un système et en cas de défaillance pour limiter les dégât occasionné. 3

Bibliographie

[1] A. S. Tanenbaum et H. Bos, *Modern operating systems*. Pearson Education, Inc., 2015.

[2] A. S. Tanenbaum, A. S. Woodhull, et others, *Operating systems: design and implementation*, vol. 68. Prentice Hall Englewood Cliffs, 1997.

[3] A. Silberschatz, P. B. Galvin, et G. Gagne, *Operating system concepts essentials*. Wiley Publishing, 2013.

[4] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, et B. D. de Dinechin, « The shift to multicores in real-time and safety-critical systems », in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2015, p. 220-229.

[5] A. Burns et R. I. Davis, « A survey of research into mixed criticality systems », *ACM Computing Surveys (CSUR)*, vol. 50, nº 6, p. 1-37, 2017.

[6] V.-A. Paun, B. Monsuez, et P. Baufreton, « On the determinism of multi-core processors », in *French Singaporean Workshop on Formal Methods and Applications*, 2013.

[7] R. J. Walls *et al.*, « Survey of Control-Flow Integrity Techniques for Embedded and Real-Time Embedded Systems », *ACM Transactions on Embedded Computing Systems*, vol. 21, nº 4, 2022, Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://dl.acm.org/doi/full/10.1145/3538275

[8] « Linux Documentation – CPU Architectures ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/v6.15/arch/index.html

[9] « What is RCU? ». Consulté le: 18 octobre 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/next/RCU/whatisRCU.html

[10] « Linux 6.17 Now Makes Multi-Core/SMP Support Unconditional ». Consulté le: 16 octobre 2025. [En ligne]. Disponible sur: https://www.phoronix.com/news/Linux-6.17-Unconditional-SMP

[11] « Remote Processor Framework ». Consulté le: 3 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/staging/remoteproc.html

[12] « Remote Processor Messaging ». Consulté le: 3 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/staging/rpmsg.html

[13] « µCLinux - page Wikipédia ». Consulté le: 25 novembre 2026. [En ligne]. Disponible sur: https://fr.wikipedia.org/wiki/%CE%9CCLinux

[14] « Real-time preemption ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/next/core-api/real-time/index.html

[15] T. Gleixner et D. Niehaus, « Hrtimers and beyond: Transforming the linux time subsystems », in *Proceedings of the Linux symposium*, 2006, p. 333-346.

[16] P. Patel, M. Vanga, et B. B. Brandenburg, « Timershield: Protecting high-priority tasks from low-priority timer interference (outstanding paper) », in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, p. 3-12.

[17] « RT-mutex implementation design ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/locking/rt-mutex-design.html

[18] « RT-mutex subsystem with PI support ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/locking/rt-mutex.html

[19] « Threaded interrupt handler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/threadirq

[20] « Linux KVM website ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://linux-kvm.org/

[21] « Control Groups v1 ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html

[22] « Control Group v2 ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/admin-guide/cgroup-v2.html

[23] « chroot(2) – Linux manuall page ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://man7.org/linux/man-pages/man2/chroot.2.html

[24] « The Linux Kernel Documentation: EDAC subsystem ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/edac/index.html

[25] « The Linux Kernel Documentation: Scrubbing ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/edac/scrub.html

[26] B. Gregg, « Linux Performance ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.brendangregg.com/linuxperf.html

[27] « The Best Linux Monitoring Tools for 2024 ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://last9.io/blog/the-best-linux-monitoring-tools-for-2024/

[28] « Htop ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://htop.dev/

[29] « Netdata ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://www.netdata.cloud/

[30] « eBPF ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://ebpf.io/

[31] « SystemTrap ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/SystemTap

[32] « Prometheus ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://prometheus.io/

[33] « Grafana ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://grafana.com/

[34] « xenwatchdog man page ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://perfwiki.github.io/main/

[35] « OProfile ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://oprofile.sourceforge.io/about/

[36] « Using kgdb, kdb and the kernel debugger internals ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/next/dev-tools/kgdb.html

[37] « ARM Virtual Generic Interrupt Controller v2 (VGIC) — The Linux Kernel documentation ». Consulté le: 22 novembre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/virt/kvm/devices/arm-vgic.html

[38] « ARM Virtual Generic Interrupt Controller v3 and later (VGICv3) — The Linux Kernel documentation ». Consulté le: 22 novembre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/virt/kvm/devices/arm-vgic-v3.html

[39] « ARM Virtual Interrupt Translation Service (ITS) — The Linux Kernel documentation ». Consulté le: 22 novembre 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/v5.9/virt/kvm/devices/arm-vgic-its.html

[40] « The Linux Watchdog driver API ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/watchdog/watchdog-api.html

[41] G. Singh, K. Bipin, et R. Dhawan, « Optimizing the boot time of Android on embedded system », in *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)*, 2011, p. 503-508.

[42] A. Al Abduallah et A. Hedberg, « Decreasing boot time in an embedded Linux environment », Bachelor Thesis, Örebro, Sweden, 2023.

[43] « Yocto Project ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://www.yoctoproject.org/

[44] A. Madhavapeddy, « Unikernels ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://anil.recoil.org/projects/unikernels

[45] « Xen Project: MirageOS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://xenproject.org/projects/mirage-os/

[46] J. Vouillon, « Lwt: a cooperative thread library », in *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, 2008, p. 3-12.

[47] « Lwt manual ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: https://ocsigen.org/lwt/latest/manual/manual

[48] « Ocsigen website ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: https://ocsigen.org/home/intro.html

[49] A. Madhavapeddy *et al.*, « Jitsu:{Just-In-Time} summoning of unikernels », in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, p. 559-573.

[50] « Vchan: Low-latency inter-VM communication channels ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://mirage.io/blog/update-on-vchan

[51] K. Sivaramakrishnan *et al.*, « Retrofitting parallelism onto OCaml », *Proc. ACM Program. Lang.*, vol. 4, nº ICFP, août 2020, doi: 10.1145/3408995.

[52] « Memory model: The hard bits - OCaml manual ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: https://ocaml.org/manual/5.4/memorymodel.html

[53] « MirageOS on OCaml 5 ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: https://tarides.com/blog/2025-02-06-mirageos-on-ocaml-5/

[54] « Mini-OS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Mini-OS

[55] « MirageeOS on Unikraft ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://discuss.ocaml.org/t/mirageos-on-unikraft/16975

[56] « GitHub ocaml-vchan ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/mirage/ocaml-vchan

[57] « OCaml Profiling ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://ocaml.org/manual/5.4/profil.html

[58] « Git mirage-monitoring ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://git.robur.coop/robur/mirage-monitoring

[59] « Memtrace ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: https://github.com/janestreet/memtrace

[60] « Memtrace Viewer ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: https://github.com/janestreet/memtrace_viewer

[61] « Github memtrace-mirage ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/robur-coop/memtrace-mirage

[62] « GitHub mirage-profile ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/mirage/mirage-profile

[63] T. Leonard, « Visualising an asynchronous monad ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://roscidus.com/blog/blog/2014/10/27/visualising-an-asynchronous-monad/

[64] « GitHub mirage-trace-viewer ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/talex5/mirage-trace-viewer

[65] A. Madhavapeddy et D. J. Scott, « Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? », *Queue*, vol. 11, nº 11, p. 30-44, 2013.

[66] Tarides, « OCaml in Space - Welcome SpaceOS! ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://tarides.com/blog/2023-07-31-ocaml-in-space-welcome-spaceos/

[67] Tarides, « OCaml in Space: SpaceOS is on a Satellite! ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://tarides.com/blog/2025-04-03-ocaml-in-space-spaceos-is-on-a-satellite/

[68] R. Kaiser et S. Wagner, « Evolution of the PikeOS microkernel », in *First international workshop on microkernels for embedded systems*, 2007.

[69] « SYSGO: PikeOS achieves Common Criteria (CC) level EAL5+ Security Certification ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/press-releases/sysgos-pikeos-achieves-common-criteria-eal-5-security-certification

[70] « PikeOS Support of ARM Cortex-A15 based Hardware Virtualization ». Consulté le: 20 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/press-releases/sysgo-announces-pikeos-support-of-arm-cortex-a15-based-hardware-virtualization

[71] « PikeOS: HwVirt Support on x86 Architecture ». Consulté le: 20 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/blog/article/pikeos-hwvirt-support-on-x86-architecture

[72] « Avec PikeOS 4.2, Sysgo renforce le support des applications à certifier sur architectures multicœurs ». Consulté le: 29 octobre 2025. [En ligne].

Disponible sur: https://www.lembarque.com/article/avec-pikeos-4-2-sysgo-renforce-le-support-des-applications-a-certifier-sur-architectures-multicoeurs

[73] O. M. Motzkus Andreas, « PikeOS Safe Real-Time Scheduling ». 2016.

[74] « Security Target PikeOS Separation Kernel v5.1.3 ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://commoncriteriaportal.org/files/epfiles/1146b_pdf.pdf

[75] E. Alkassar, M. A. Hillebrand, W. Paul, et S. Petrova, « Proving Memory Separation in a Microkernel by Code Level Verification ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://formal.kastel.kit.edu/~bormer/pub/amics2011.pdf

[76] « ELinOS - Embedded Linux Distribution ». Consulté le: 20 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/elinos

[77] « SYSGO's PikeOS now supports LEON Processor and RTEMS Operating System ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/press-releases/sysgos-pikeos-now-supports-leon-processor-and-rtems-operating-system

[78] « Windows as PikeOS Guest OS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/windows

[79] « LEON5 - Gaisler ». Consulté le: 5 décembre 2025. [En ligne]. Disponible sur: https://www.gaisler.com/products/leon5

[80] « D2.11 Final user manual for the industry-ready RTOS multicore solutions ». Consulté le: 5 décembre 2025. [En ligne]. Disponible sur: https://cordis.europa.eu/docs/projects/cnect/5/611085/080/deliverables/001-D211.pdf

[81] « Technology Partner Alliances ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/technology-alliances

[82] « Codeo ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/codeo

[83] « Codeo for PikeOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/popups/codeo-for-pikeos

[84] « Codeo for ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/popups/codeo-for-elinos

[85] « SYSGO and Rapita Systems announce Partnership Deal ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/press-releases/sysgo-and-rapita-systems-announce-partnership-deal

[86] « Rapita Systems ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/technology-alliances/rapita

[87] « OS-and Hypervisor-aware Debugging ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.lauterbach.com/features/os-awareness

[88] « PikeOS ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/PikeOS

[89] « Security Target for the PikeOS Separation Kernel ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://www.commoncriteriaportal.org/nfs/ccpfiles/files/epfiles/1185b_pdf.pdf

[90] « Ada on PikeOS - A fortunate Combination ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/press-releases/ada-on-pikeos-a-fortunate-combination

[91] « AdaCore adapte son environnement de développement à l'OS temps réel PikeOS de SYSGO ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.lembarque.com/article/adacore-adapte-son-environnement-de-developpement-a-los-temps-reel-pikeos-de-sysgo

[92] « Rust for PikeOS & ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/rust

[93] « Ansys Certified Model-based Development meets Safety-critical Execution ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/technology-alliances/ansys

[94] S. Lescuyer, « ProvenCore: Towards a Verified Isolation Micro-Kernel. », in *MILS@ HiPEAC*, 2015.

[95] « ProvenRun homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://provenrun.com/provenvisor/

[96] « ProvenCore secure OS achieves EAL7 Common Criteria certification ». Consulté le: 4 décembre 2025. [En ligne]. Disponible sur: https://provenrun.com/provencore-secure-os-achieves-eal7-common-criteria-certification/

[97] « ProvenRun - STMicroelectronics Partner Program ». Consulté le: 4 décembre 2025. [En ligne]. Disponible sur: https://www.st.com/content/st_com/en/partner/partner-program/partnerpage/ProvenRun.html

[98] « ProvenCore Secure OS and ProvenVisor Hypervisor on Toradex Modules ». Consulté le: 4 décembre 2025. [En ligne]. Disponible sur: https://www.toradex.com/videos/provencore-os-provenvisor-hypervisor-on-toradex-modules-ew-2019

[99] J. Sherrill, « OAR RTEMS Past Present and Future ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://slidetodoc.com/oar-rtems-past-present-and-future-joel-sherrill/

[100] « RTEMS - Wikipedia ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/RTEMS

[101] « About OAR and RTEMS ». Consulté le: 16 octobre 2025. [En ligne]. Disponible sur: https://www.rtems.com/aboutoarandrtems

[102] « RTEMS - European Space Agency ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS

[103] « NASA's Perseverance Rover reaches Mars with the help of the open-source real-time operating system RTEMS! ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://spaceanddefense.io/nasas-perseverance-rover-reaches-mars-with-the-help-of-the-open-source-real-time-operating-system-rtems/

[104] « RTEMS: Architectures ». Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: https://www.rtems.org/architectures/

[105] « RTEMS Symmetric Multiprocessing ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/symmetric_multiprocessing_services.html

[106] « RTEMS Xilinx Zyna UltraScale+ BSP ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/bsps/arm/xilinx-zynqmp.html

[107] « System Initialization - RTEMS documentation ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: https://docs.rtems.org/docs/6.1/bsp-howto/initilization_code.html

[108] « Memory Model - RTEMS documentation ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: https://docs.rtems.org/docs/6.1/cpu-supplement/port.html

[109] « SMP Schedulers ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/scheduling-concepts/smp-schedulers.html

[110] « profreport - print a profiling report ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/rtems_specific_commands.html#profreport-print-a-profiling-report

[111] « BSP Build System ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/bld/index.html

[112] « Tracing ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/tracing/index.html

[113] « RTEMS Shell Guide ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/index.html

[114] « RTEMS Specific Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/rtems_specific_commands.html

[115] « Memory Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/memory_commands.html

[116] « CPU Usage Statistics ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/cpu_usage_statistics.html

[117] « RTEMS SMP QDP ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://rtems-qual.io.esa.int/

[118] A. Butterfield et F. Tuong, « Applying formal verification to an open-source real-time operating system », *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*. Springer, p. 348-366, 2023.

[119] « Formal Verification Overview ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/eng/fv/overview.html

[120] « RTEMS Licenses ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://gitlab.rtems.org/rtems/rtos/rtems/-/blob/main/LICENSE.md?ref_type=heads

[121] « seL4 Supported platforms ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Hardware/

[122] K. McLeod, « Multiprocessing on seL4 with verified kernels ». [En ligne]. Disponible sur: https://sel4.org/Summit/2022/slides/d1_07_Multiprocessing_on_seL4_with_verified_kernels_Kent_Mcleod.pdf

[123] G. Heiser, « State of seL4-related Research at Trustworthy Systems ». [En ligne]. Disponible sur: https://sel4.org/Summit/2022/slides/d1_02_State_of_seL4-related_research_Gernot_Heiser.pdf

[124] R. J. Colvin, I. J. Hayes, S. Heiner, P. Höfner, L. Meinicke, et R. C. Su, « Practical Rely/Guarantee Verification of an Efficient Lock for seL4 on Multicore Architectures ». [En ligne]. Disponible sur: https://arxiv.org/abs/2407.20559

[125] « Solox AMP Rust ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://github.com/jonlamb-gh/solox-amp-rust

[126] « The seL4 Capability System ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/workshops/pdfs/20160423-sel4-capabilities.pdf

[127] « Capabilities - Tutorial seL4 ». Consulté le: 2 décembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Tutorials/capabilities.html

[128] « The seL4 Microkernel - An Introduction ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://sel4.systems/About/seL4-whitepaper.pdf

[129] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, et G. Klein, « Verified Protection Model of the seL4 Microkernel », *Electronic Proceedings in Theoretical Computer Science*, 2011, Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://www.researchgate.net/publication/221160526_Verified_Protection_Model_of_the_seL4_Microkernel

[130] « Threads - seL4 docs ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Tutorials/threads.html

[131] A. Lyons, K. McLeod, C. Lewis, D. Kurth, et G. Heiser, « Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time », *Proceedings of the Eurosys Conference*, 2018, Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://www.researchgate.net/publication/324641091_Scheduling-context_capabilities_a_principled_light-weight_operating-system_mechanism_for_managing_time

[132] « 10.1.1 Mcs - seL4 docs ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/releases/sel4/10.1.1-mcs.html

[133] « MCS - seL4 docs ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Tutorials/mcs.html

[134] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, et G. Heiser, « Timing analysis of a protected operating system kernel », in *2011 IEEE 32nd real-time systems symposium*, 2011, p. 339-348.

[135] T. Sewell, F. Kam, et G. Heiser, « Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis », in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, p. 1-11.

[136] G. Klein *et al.*, « seL4: Formal verification of an OS kernel », in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, p. 207-220.

[137] « Microkit User Manual ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/projects/microkit/manual/latest/

[138] « seL4: Interrupts tutorial ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Tutorials/interrupts.html

[139] « rust-sel4 ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://github.com/seL4/rust-sel4

[140] « seL4 Licensing ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://sel4.systems/Legal/license.html

[141] S. H. VanderLeest, « The open source, formally-proven seL4 microkernel: considerations for use in avionics », in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, p. 1-9.

[142] « seL4 FAQ Verification ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://sel4.systems/About/FAQ.html#verification

[143] « seL4 What the Proof Implies ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://sel4.systems/Verification/implications.html

[144] « Xen 4.20 release notes ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: https://xenproject.org/blog/xen-project-4-20-oss-virtualization/

[145] « Xen ARM with Virtualization Extensions ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions

[146] R. VanVossen, « XEN ON THE ZYNQ ULTRASCALE+ MPSOC ».

[147] « Xen: Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/StubDom

[148] « Xen: Device Model Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains

[149] « X86 Paravirtualised Memory Management ». Consulté le: 3 décembre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management

[150] S. L. Software, « Deep Dive: MMU Virtualization with Xen on ARM ». Consulté le: 3 décembre 2025. [En ligne]. Disponible sur: https://www.starlab.io/blog/deep-dive-mmu-virtualization-with-xen-on-arm

[151] « Xen: dom0less ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/unstable/features/dom0less.html

[152] « Xen Project Releases Version 4.17 with Enhanced Security, Higher Performance, Improved Embedded Static Configuration and Speculative Mitigation Support ». Consulté le: 3 décembre 2025. [En ligne]. Disponible sur: https://xenproject.org/blog/xen-project-releases-version-4-17-with-enhanced-security-higher-performance-improved-embedded-static-configuration-and-speculative-mitigation-support/

[153] « Credit Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit_Scheduler

[154] « Credit2 Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit2_Scheduler

[155] S. Xi *et al.*, « Real-time multi-core virtual machine scheduling in xen », in *Proceedings of the 14th International Conference on Embedded Software*, 2014, p. 1-10.

[156] « Xen Monitoring Tools and Techniques ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.virtuatopia.com/index.php?title=Xen_Monitoring_Tools_and_Techniques

[157] Citrix, « Monitor and manage your deployment ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://docs.xenserver.com/en-us/xenserver/8/monitor-performance.html

[158] « xentrace man page ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/4.20-testing/man/xentrace.8.html

[159] « xenwatchdog man page ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/4.20-testing/man/xenwatchdogd.8.html

[160] « Xen: Event Channel Internals ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Event_Channel_Internals

[161] « PVH - Xen 4.21 documentation ». Consulté le: 25 novembre 2024. [En ligne]. Disponible sur: https://xenbits.xenproject.org/docs/4.21-testing/misc/pvh.html

[162] « XEN Licensing ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://xenbits.xenproject.org/gitweb/?p=xen.git;a=blob;f=COPYING;h=824c3aa353b47507241831f4753590f86a162014;hb=refs/heads/staging-4.20

[163] « True static partitioning with Xen Dom0-less ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/

[164] M. Masmano, I. Ripoll, et A. Crespo, « An overview of the XtratuM nanokernel », in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2005.

[165] « Red 5G espacial ». Consulté le: 12 septembre 2025. [En ligne]. Disponible sur: https://www.upv.es/noticias-upv/noticia-11299-red-5g-espacia-es.html

[166] « fentISS ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://www.fentiss.com/

[167] « fentISS software now powers over 1,000 spacecraft ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.satelliteevolution.com/post/milestone-in-the-space-industry-fentiss-software-now-powers-over-1-000-spacecraft

[168] « Space Missions ». Consulté le: 3 novembre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/missions/

[169] « Safety Runs on XtratuM/NG ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://storage.googleapis.com/b2match-as-1/QFAPVNvf2nGRmR9JyyQhuYuZ

[170] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, et A. Crespo, « XtratuM hypervisor redesign for LEON4 multicore processor », *SIGBED Rev.*, vol. 11, nº 2, p. 27-31, sept. 2014, doi: 10.1145/2668138.2668142.

[171] M. Masmano, I. Ripoll, A. Crespo, et S. Peiro, « Xtratum for leon3: an open source hypervisor for high integrity systems », in *ERTS2 2010, Embedded Real Time Software & Systems*, 2010.

[172] A. Crespo, M. Masmano, et I. Ripoll, « LithOS: a ARINC-653 guest operating for XtratuM ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.osadl.org/fileadmin/dam/rtlws/12/Crespo.pdf

[173] M. Aldea Rivas et M. González Harbour, « ARINC-653 Inter-partition Communications and the Ravenscar Profile ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://oa.upm.es/42418/1/INVE_MEM_2015_228287.pdf

[174] « LithOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/lithos/

[175] « XtratuM homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.fentiss.com/xtratum/

[176] Á. Esquinas, J. Zamorano, J. A. De la Puente, M. Masmano, I. Ripoll, et A. Crespo, « ORK+/XtratuM: An open partitioning platform for Ada », in *International Conference on Reliable Software Technologies*, 2011, p. 160-173.

[177] « xng-rs ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://github.com/aeronautical-informatics/xng-rs

[178] fentISS, « XtratuM Hypervisor ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/xtratum/

[179] « XtratuM code source ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://github.com/lfd/XtratuM

[180] « SAFEST Project ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://safest-project.eu/

[181] « seL4: Interrupts tutorial ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://www.linux-kvm.org/page/Processor_support

[182] « MirageOS on ARM64 ». Consulté le: 16 juin 2025. [En ligne]. Disponible sur: https://mirage.io/docs/arm64

[183] « MirageOS Installation ». Consulté le: 16 juin 2025. [En ligne]. Disponible sur: https://mirage.io/docs/install

[184] « PikeOS homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.sysgo.com/pikeos

[185] « SLOCCount website ». Consulté le: 3 juillet 2025. [En ligne]. Disponible sur: https://dwheeler.com/sloccount/