



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

11/12/2025



Table des matières

1	Introduction	3
2	Linux	15
3	MirageOS	33
4	PikeOS	43
5	ProvenVisor	49
6	RTEMS	55
7	seL4	63
8	Xen	69
9	XtratuM	79
10	Tableaux comparitifs	85
11	Glossaire	89
	Bibliographie	93

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 tournent 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 étagères¹, 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 suivants²: 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 sûreté 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 existants 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.

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

²Nous 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.

³En 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 <i>bare metal</i>
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.	Nécessite un support du <i>RTE</i> (<i>RunTime Environment</i>).
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 pas 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-section 1.5.
- Une série de tableaux comparatifs qui résument les informations détaillées 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
- Corruption 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 1⁴, 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.

⁴On 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, prédition 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* (*Multi-processor System on a chip*).

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

Caractéristique \ Architecture	<i>SMP</i>	<i>AMP</i>
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	<ul style="list-style-type: none"> 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* familié 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 est le partage du temps *CPU* entre les tâches en cours d'exécution. Contrairement à son homologue spatial, ce partage est le plus souvent géré par une couche logicielle appelée *ordonnanceur de tâches* (*scheduler*). Le rôle principal de l'ordonnanceur est de décider qu'elle tâche doit maintenant s'exécuter et sur quel processeur.

Un ordonnanceur de tâches poursuit des objectifs variés et parfois incompatibles. Il existe de nombreux critères pour qualifier et quantifier les qualités et défauts d'un ordonnanceur. Nous ne retenons que les suivants:

- *Throughput*: quantité de travail accomplie par unité de temps,
- *Latency*: délai qui s'écoule entre le réveil d'une tâche et son exécution sur un cœur,
- *Fairness*: équité quant au temps de calcul en tenant compte des priorités des tâches,
- *Déterminisme*: capacité à prédire l'ordre d'exécution des tâches.

Il est difficile de concilier toutes ces qualités dans un même ordonnanceur. 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.

Généralement un ordonnanceur de tâches d'un *GPOS* cherchera à maximiser le *throughput* tout en restant équitable. Quant à un ordonnanceur de tâches d'un *RTOS*, il cherchera à être aussi déterministe que possible et à minimiser la latence, même si cela réduit parfois le *throughput*.

Il existe donc une multitude d'ordonnanceurs faisant des compromis dans la poursuite des qualités ci-dessus. Les systèmes d'exploitation offrent souvent plusieurs ordonnanceurs afin de pouvoir s'adapter aux différents usages. 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*
- *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 correction⁵. 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.

⁵Une exception notable est celle des applications multimédia.

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 dans le pire cas⁶. Le déterminisme est souvent assuré par le caractère préemptible du noyau⁷ 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 mémoire

Nous avons étudié le support logiciel des différents systèmes visant à prévenir la corruption 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 place 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 supplémentaires 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:

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

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

- 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.
- La randomisation de l'espace d'adressage (*Address 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 plateformes 64 bits.
- Les *canaris* de pile. Il s'agit d'une protection ajoutée 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 dépassemens 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 émis

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. Pour les architectures multi cœur, chaque cœur est muni d'un *Local APIC* pour gérer les interruptions entre coeurs. De même, les architectures *ARM* disposent d'un système dévolu 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 ReQuest)* 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 physique ou virtuelle.

Si masquer les interruptions résout drastiquement la première difficulté, 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-préemp-tibles puisque la préemption est souvent gérée par une interruption matérielle levée par une horloge. Masquer les interruptions longtemps n'est donc pas non plus souhaitable 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*, c'est-à-dire capturer les requêtes du systèmes invités en insérant des interruptions dans son flot d'exécution puis en émulant ces requêtes lorsqu'elles sont autorisées,
- Paravirtualiser les contrôleurs *PIC*, c'est-à-dire exposer une *API (Applicaton Programming Interface)* aussi proche du matériel que possible pour que le système invité puisse implémenter un pilote pour celle-ci,

- 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 chacun des systèmes étudiés, nous avons examinés les mécanismes de virtualisation et de masquage des interruptions matérielles.

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* lorsque cela était possible.

1.11 Programmation *bare-metal*

La programmation *bare-metal* demeure commune dans les systèmes critiques. Cependant, l'adoption d'un *COTS* (*Commercial off-the-shelf*) pour le système d'exploitation offre beaucoup d'avantages. Il est donc fréquent de vouloir porter une application *bare-metal* vers un hyperviseur.

C'est dans cette optique que nous avons examiné les solutions proposées en matière de programmation *bare-metal* pour les différents hyperviseurs étudiés. Notre analyse se borne aux langages de programmation *Ada*, *C*, *OCaml* et *Rust*. Le principal défi d'un tel portage est d'adapter un *RTE* de ces langages pour l'hyperviseur donné.

Dans le cas de *Ada*, *C* et *Rust* le portage est grandement facilité par la taille du *RTE* de ces langages et de se passer de la majorité de la bibliothèque standard ou encore 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

La *maintenabilité* d'une solution informatique est un facteur essentiel à analyser afin de faire un choix éclairé. Si l'usage d'un *COTS* offre des avantages indéniables durant la phase

développement, il introduit une dépendance technologique significative sur le long terme. La maintenance du *COTS* peut cesser ou ses conditions d'utilisation peuvent changées, entraînant une transition vers une solution alternative, souvent coûteuses et complexes.

Nous avons comparé la maintenabilité des différents systèmes avec les critères suivants:

- *Licence*: une licence libre ou *Open Source* présente l'avantage d'avoir un accès facile au code source. Même si le *COTS* n'est plus maintenu, il est toujours possible de le faire évoluer et d'appliquer des correctifs,
- *Taille de l'écosystème*: un grand écosystème facilite le développement et assure que d'autres utilisateurs contribuent à son maintien,
- *Taille du code source*: la taille du code source n'est pas toujours gage de complexité mais cela reste un indice important,
- *Ancienneté*: un *COTS* ancien est souvent plus mature mais il embarque aussi davantage de dette technique,
- *Support commercial*: la présence d'un support commercial assure de recevoir une aide, au moins sur le moyen terme.

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 architectures *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 coeurs. 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 *RPMmsg* [12]. Vous pouvez vérifier que votre noyau est compilé avec le support pour ces systèmes avec les options respectivement `CONFIG_REMOTEPROC` et `CONFIG_RPMMSG`.

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.

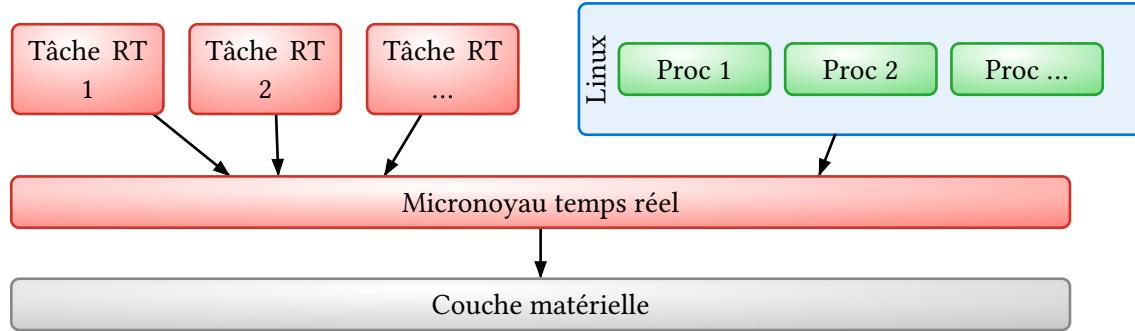


Fig. 1. – Architecture *cokernel*.

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.

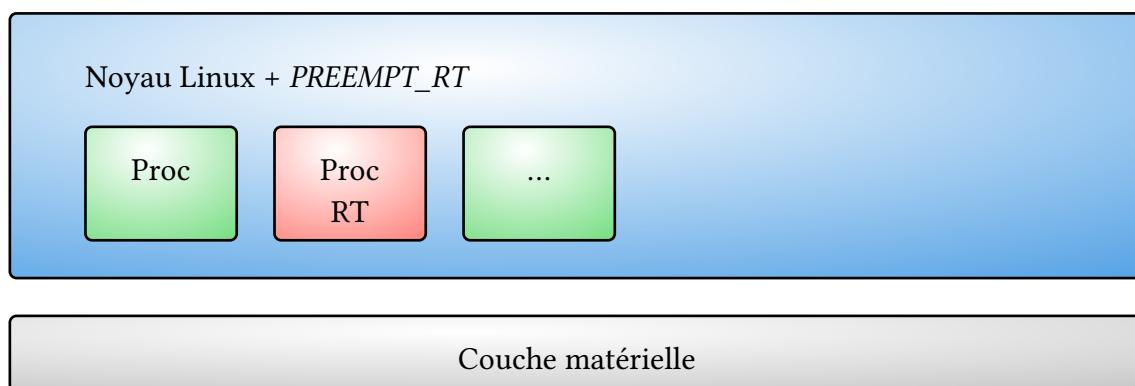


Fig. 2. – Architecture de *Linux* avec *PREEMPT_RT*

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 revue 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 ordonné, bloquant pendant un temps indéterminé l'exécution de A.

À fin de rendre prédictible 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 dite 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*.

2.5 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 priviléges 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 souhaitons limiter la consommation de mémoire d'un processus à 5 Mio. On commence par créer deux⁸ *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.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(void) {
6     size_t sz = 0;
7
8     printf("How many bytes do you want to allocate? ");
9     if (scanf("%zu", &sz) != 1) {
10         printf("Invalid size\n");
11         return EXIT_FAILURE;
12     }
13
14     char *buf = malloc(sz * sizeof(*buf));
15     if (!buf) {
16         printf("Cannot allocate %zu bytes\n", sz);
17         return EXIT_FAILURE;
18     }
19
20     memset(buf, 0, sz);
21     free(buf);
22     return EXIT_SUCCESS;
23 }
24
```

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
```

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

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

```
$ 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'arborescence des fichiers, cette commande isole celle des processus, le réseau et les utilisateurs. Par exemple, considérons le programme C suivant:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dirent.h>
4 #include <ctype.h>
5 #include <stdbool.h>
6 #include <unistd.h>
7
8 static inline bool is_pid(char *name) {
9     while (*name && isdigit(*name))
10         name++;
11     return *name == '\0';
12 }
13
14 int main(void) {
15     printf("My pid: %d\n", getpid());
16
17     DIR *dir = opendir("/proc");
18     if (!dir)
19         return EXIT_FAILURE;
20
21     struct dirent *ent;
22     while ((ent = readdir(dir)) != NULL) {
23         if (is_pid(ent->d_name))
24             printf("%s\n", ent->d_name);
25     }
26
27     closedir(dir);
28     return EXIT_SUCCESS;
29 }
30

```

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 processus⁹. 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	gammemode
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 priviléges limités.

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

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 priviléges `root` dans 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 priviléges 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 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. Cette interface permet le pilotage du *scrubbing*.

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.

¹⁰Le 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_`.

- *Prometheus/Grafana* [32], [33] : Solutions d’observabilité distribuée largement adoptées pour le monitoring de systèmes critiques.
- *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.

```

1 #include <stdlib.h>
2 #include <time.h>
3 #include <string.h>
4 #define SIZE 1000000
5 #define N 100000000
6
7 int main(int argc, char **argv) {
8     srand(time(NULL));
9
10    int random = argc > 1 && strcmp(argv[1], "random") == 0;
11    volatile int arr[SIZE] = {0};
12    for (int i = 0; i < N; i++)
13        (void)arr[(i + (random ? rand() : 0)) % SIZE];
14
15    return EXIT_SUCCESS;
16 }
17

```

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 [37]. Les principales primitives sont `local_irq_disable`, `local_irq_save` et `disable_irq` qui permettent de désactiver toutes les interruptions du *CPU* ou une interruption spécifique au niveau du contrôleur.

Pour les systèmes temps réel, le projet *PREEMPT_RT* introduit le concept d'*interruptions threadées* où le gestionnaire matériel se limite à masquer l'interruption, acquitter le contrôleur et réveiller un thread noyau qui effectue le traitement réel de façon préemptible. Cette approche résout les problèmes d'*inversion d'interruptions* et réduit la latence.

KVM fournit un support pour la virtualisation des interruptions matérielles via les interfaces *VGIC v2/v3* [38], [39] sur *ARM* et *Intel APICv/AMD AVIC* sur *x86*. Le *VGIC v3* offre notamment le support de l'*affinity routing* (jusqu'à 512 *VCPUs*) et de l'*ITS (Interrupt Translation Service)* pour les interruptions *MSI*.

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*.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <sys/ioctl.h>
6 #include <linux/watchdog.h>
7
8 int main(void) {
9     int fd = open("/dev/watchdog", O_WRONLY);
10    if (fd == -1)
11        exit(EXIT_FAILURE);
12
13    // Configure le timeout à 20 secondes.
14    int timeout = 20;
15    if (ioctl(fd, WDIOC_SETTIMEOUT, &timeout) == -1)
16        goto failed;
17    printf("Watchdog initialisé\n");
18
19    // Réinitialise le watchdog toutes les 10 secondes.
20    while (1) {
21        if (write(fd, "\0", 1) != 1)
22            goto failed;
23        printf("Watchdog rechargeé\n");
24        sleep(10);
25    }
26
27    close(fd);
28    return EXIT_SUCCESS;
29
30 failed:
31    close(fd);
32    return EXIT_FAILURE;
33 }
34

```

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 Programmation *bare-metal*

KVM permet l'exécution d'applications *bare-metal* sous forme d'*unikernels* dans ses machines virtuelles. Ces *unikernels* regroupent l'application et un *RTE* minimal, sans système d'exploitation complet.

En *Rust*, *RustyHermit* [41] est un *unikernel* qui peut s'exécuter sur *KVM* via l'hyperviseur *uhyve* ou *QEMU* [42].

Pour *OCaml*, *MirageOS* supporte l'exécution d'*unikernels* sur *KVM/QEMU* via *Solo5* [43].

Nous n'avons pas trouvé de solution documentée permettant d'exécuter des applications *Ada* en *bare-metal* dans une partition *KVM*.

2.12 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* (initialisation uniquement des périphériques indispensables, optimisation du code assembler, ...), l'initialisation du noyau (image non compressée, désactivation des fonctionnalités inutiles, désactivation du profilage, ...) ou l'initialisation de l'*espace utilisateur* qui est généralement l'étape la plus longue.

Dans l'article [44], 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 [45], 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.

2.12.1 Profilage de *systemd*

Comme expliqué ci-dessus, l'initialisation de l'*espace utilisateur* est souvent l'étape la plus longue au démarrage.

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. Détailons 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*.

2.13 Maintenabilité

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

Le développement du noyau a débuté en 1991, ce qui fait plus de 34 ans d'existence. Cette longue histoire a permis à *Linux* et à son écosystème d'atteindre un niveau de maturité exceptionnel. Des milliers de développeurs contribuent au noyau au travers de centaines d'entreprises (*Intel*, *Google*, *Samsung*, *AMD*, *Red Hat*, ...).

Toutefois avec plus de 26 millions de lignes de code (et 18,9 millions de plus pour les pilotes seulement), le noyau *Linux* est une d'une complexité rarement atteinte dans l'histoire du développement logiciel. Cette taille colosse constitue un défi pour la vérification du noyau ou la certification de solutions logicielles intégrant *Linux*.

De nombreuses entreprises proposent un support commercial pour *Linux*, notamment *Red Hat*, *SUSE* et *Canonical*. Ce support couvre aussi bien les distributions grand public que les versions temps réel pour l'embarqué.

3 MirageOS

MirageOS en bref

- **Type** : LibOS
- **Langage** : OCaml (99%)
- **Architectures** : x86-64, ARM v8, PowerPC¹¹
- **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) + GPLv2 (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.

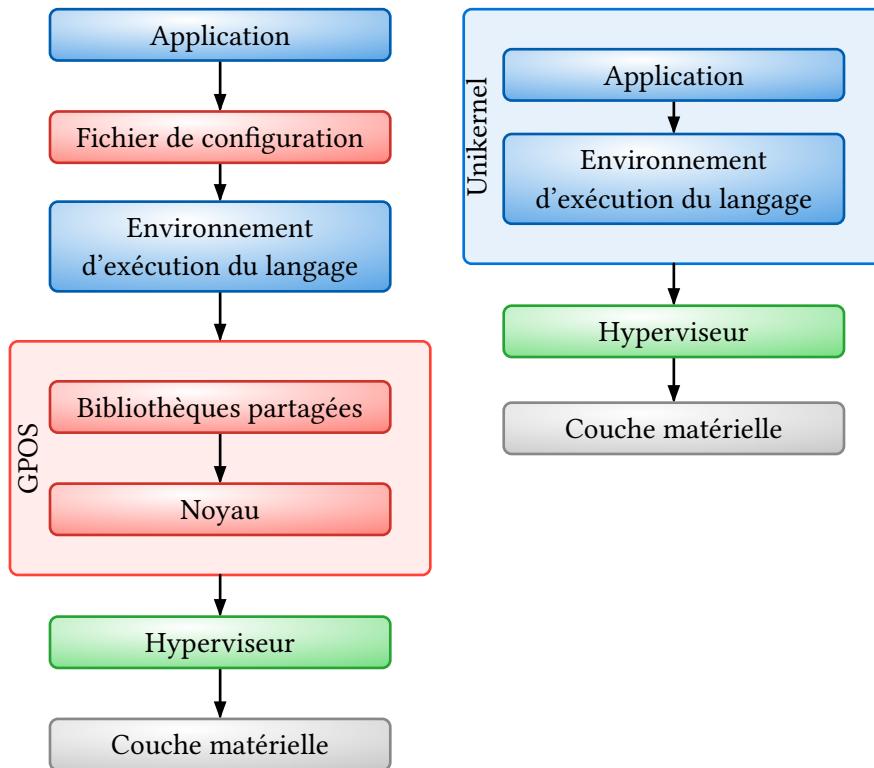


Fig. 3. – Comparaison entre l'approche *GPOS* et l'approche *unikernel*.

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

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

de programmation¹² 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 [46]. 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* [47] 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 la 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 mirageos setup
```

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

```
$ make -C mirageos 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 natif¹³ 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

¹²La 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*.

¹³Il 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'offrir 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* [48], [49].

¹⁴Une 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* [50]. 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* [51], [52].

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* [53]. Cet ajout s'est fait aux prix d'une complexification du modèle mémoire d'*OCaml* mais néanmoins maîtrisé [54],
- 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* [55], 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* [56].
- 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 [57]. 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* [58] 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 mémoire

La gestion de la corruption 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* [59].

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

- *mirage-monitoring* [60]: Outil de monitoring pour les *unikernels* produits par *MirageOS*. Il supporte le *dashboard Telegraph* de *Grafana*,
- *memtrace* [61]: 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* [62]. Il existe une bibliothèque *MirageOS memtrace-mirage* [63] qui offre un support pour cet outil dans un *unikernel*,
- *memtrace_viewer* [62]: Outil d'exploration de traces produites par *memtrace*,
- *mirage-profile* [64]: 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 [65]. Le projet ne semble plus être maintenu,
- *mirage-trace-viewer* [66]: 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

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

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.

```

1  open Mirage
2
3  let main =
4      main "Unikernel.Make"
5      ~packages:[
6          package "duration";
7          package "memtrace-mirage"
8      ]
9      (stackv4v6 @-> job)
10
11 let stackv4v6 = generic_stackv4v6 default_network
12
13 let () = register "memtrace" [ main $ stackv4v6 ]
14

```

Liste 6. – Configuration de l'*unikernel*

```

1 open Lwt.Infix
2
3 module Make (S : Tcpip.Stack.V4V6) = struct
4     module Memtrace = Memtrace.Make (S.TCP)
5
6     let rec alloc i =
7         if i < 0 then Lwt.return_unit
8         else
9             let a = Array.init 1_000_000 (fun i -> i * i) in
10            Array.sort Int.compare a;
11            Lwt.pause () >= fun () -> alloc (i - 1)
12
13     let start s =
14         S.TCP.listen (S.tcp s) ~port:1234 (fun f ->
15             match Memtrace.Memprof_tracer.active_tracer () with
16             | Some _ -> S.TCP.close f
17             | None ->
18                 let tracer =
19                     Memtrace.start_tracing ~context:None ~sampling_rate:1e-4 f
20                 in
21                 Lwt.async (fun () ->
22                     S.TCP.read f >|= fun _ ->
23                         Memtrace.stop_tracing tracer);
24                 Lwt.return_unit);
25             alloc 10
26     end
27
28

```

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-viewer`:

```
$ 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 [67], 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 [51], 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

Le code de *MirageOS* est publié sous la licence *ISC* avec certaines parties sous licence *GPLv2*. 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, l'utilisateur n'a pas les contraintes des licences *GPL* lorsqu'il distribue son *unikernel*.

Le projet a été initié en 2009, ce qui en fait un projet relativement jeune avec environ 16 ans d'existence.

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*.

Cette valeur doit être considérée avec prudence puisqu'un véritable projet devra certainement utiliser d'autres bibliothèques de *MirageOS* pour fonctionner.

De plus la totalité de l'*unikernel* produit 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*.

MirageOS bénéficie d'un support commercial de l'entreprise *Tarides*, qui maintient activement le projet et propose des services de conseil et de développement autour de la technologie *MirageOS*. La *MirageOS Core Team*, composée d'universitaires et d'ingénieurs du secteur privé, assure la maintenance continue du projet.

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 [68], [69]. 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 [70]. 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* [71].

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* [72] et *x86-64* [73].

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 [74].

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 [75].

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 [76]. Cette isolation a fait l'objet d'une vérification formelle au niveau du code source [77]. 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'adresses 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* [75]. Cette architecture combine un partitionnement temporel strict au niveau des partitions avec une flexibilité d'ordonnancement au niveau des threads.

Au premier niveau, l'ordonnanceur distribue statiquement le temps *CPU* selon un schéma cyclique *TDM* (*Time Division Multiplexing*). Le cycle complet est subdivisé en fenêtres temporelles de durées variables, chacune allouée à une partition spécifique. Cette configuration statique garantit un ordonnancement déterministe avec une granularité jusqu'à 250 µs.

Au second niveau, *PikeOS* propose plusieurs politiques d'ordonnancement pour les threads au sein de chaque partition:

- *Fixed-priority preemptive scheduling*: l'ordonnanceur sélectionne toujours le thread avec la plus haute priorité et permet la préemption,
- *Earliest Deadline First (EDF)*: permet de prioriser dynamiquement les threads selon leurs échéances.

Cette architecture à deux niveaux permet 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 l'origine pour offrir un comportement déterministe adapté aux exigences temps réel dur. Contrairement aux premiers micronoyaux de la famille *L4*, le micronoyer *PikeOS* est totalement préemptif [70], éliminant les sections non préemptibles qui pouvaient introduire de la gigue et empêcher une estimation précise du *WCET*.

Le déterminisme de *PikeOS* repose sur des latences bornées, inhérentes à sa conception microkernel. En limitant les fonctionnalités du noyau aux services essentiels, *PikeOS* élimine les éléments non déterministes tels que l'allocation dynamique de mémoire dans les chemins d'exécution critiques. Le partitionnement temporel [75] garantit un ordonnancement déterministe avec une précision de l'ordre de la microseconde.

L'ordonnancement au sein des partitions utilise un mécanisme préemptif à priorités fixes et supporte également des politiques d'ordonnancement de type *EDF* (*Earliest Deadline First*).

Enfin, *PikeOS* peut être configuré pour invalider les caches et la *TLB* lors des changements de partition temporelle [75] afin d'éliminer les interférences entre partitions, ce qui améliore la prévisibilité des temps d'exécution pour des analyses *WCET* très conservatrices.

4.3.4 OS invités supportés

L'hyperviseur de *PikeOS* supporte les OS invités¹⁶ suivants:

- *ELinOS* est supporté par *PikeOS*. Il s'agit d'une distribution *Linux* pour l'embarqué temps réel développé par *SYSGO* [78]. 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 [79]. 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* [80].

4.4 Corruption 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 mémoire dans *PikeOS* s'effectue à plusieurs niveaux:

¹⁶Ils sont appelés *GuestOS* dans la documentation.

- 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 [79], 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 [81]. Ces fonctionnalités sont accessibles via l'*API* du *BSP LEON* de *PikeOS*.
- 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 [82]. 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* [83]. Nous avons ainsi pu identifier les outils suivants: *CODEO*, *RVS* et *TRACE32*.

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

4.5.1 CODEO

La société *SYSGO* propose un *IDE* baptisé *CODEO* [84], [85] 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* [86].

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 [87], [88].

4.5.3 TRACE32

L'outil *TRACE32* développé par l'entreprise *Lauterbach* comprend un débogueur et un traceur [89]. 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+* [76], [90]. La protection repose sur:

- La séparation spatiale et temporelle stricte via des partitions logicielles [71], [91],
- L'utilisation de *MMU/MPU* pour l'isolation matérielle [76],
- Une politique de liste blanche pour la communication inter-partition [91],
- La prévention de propagation d'erreurs entre partitions [76].

4.8 Watchdog

Nous n'avons pas trouvé d'informations sur un support *watchdog* dans *PikeOS*. Il est probable que ce support est relatif à chaque *BSP* et présent pour certains d'entre eux étant donné le domaine d'application de ce système d'exploitation.

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 [92], [93]. 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 [93].
- Le langage *Rust* est supporté [94].
- Le langage *SCADE* est supporté via un partenariat avec l'entreprise *Ansys* [95].

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 pour différents secteurs industriels critiques.

4.11.1 Certifications de sûreté

PikeOS propose des kits de certification pour différents domaines :

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

PikeOS version 5.1 est certifié selon la norme *DO-178C* au niveau *DAL-A* ainsi que selon les normes *IEC 61508*, *EN 50128*, *EN 50657* et *ISO 26262*.

SYSGO a été la première entreprise au monde à atteindre le niveau de sûreté *SIL 4* (le plus élevé) selon la norme *IEC 61508* pour un système d'exploitation temps réel et hyperviseur multi-coeur.

Chaque kit de certification contient un hyperviseur *PikeOS* conforme aux standards, une aide documentaire complète pour le développement et les tests, ainsi qu'un manuel de sûreté avec des directives d'utilisation dans des conceptions de systèmes critiques.

4.11.2 Certification de sécurité

Le noyau de séparation *PikeOS* version 5.1.3 a obtenu la certification *Common Criteria (ISO 15408)* au niveau *EAL 5+* pour les architectures *ARMv8*, *x86-64* et *PowerPC* [71].

Le niveau *EAL 5+* garantit que *PikeOS* a été conçu de manière semi-formelle et vérifié contre des vulnérabilités complexes. Basé sur l'architecture *MILS (Multiple Independent Levels of Security)*, *PikeOS* est présenté comme le seul système d'exploitation offrant les standards de certification les plus élevés en matière de sûreté et de cybersécurité pour le même produit et la même base de code.

SYSGO fournit un kit de certification complet conforme au niveau *EAL 5+* pour faciliter la certification de sécurité des dispositifs embarqués basés sur *PikeOS*. Les clients peuvent utiliser ces artefacts pour leurs processus de certification en aéronautique (*DO-356A/ED-203A*), automobile (*ISO/SAE 21434*), ferroviaire (*CLC/TS 50701*) et défense.

4.12 Maintenabilité

PikeOS est un logiciel propriétaire aux sources fermées. L'entreprise *SYSGO* ne semble pas communiquer sur ses licences. Les modalités et les tarifs des licences sont certainement à négociés au cas par cas.

PikeOS a été commercialisé pour la première fois en 2005, ce qui en fait un système avec environ 20 ans d'existence.

Nous n'avons pas pu évaluer la taille de la base de code, faute d'informations librement accessibles. Toutefois, au vu des nombreuses certifications et de sa conception inspirée du micronoyau *L4*, nous spéculons que le noyau de *PikeOS* est probablement de petite taille, c'est-à-dire de l'ordre quelques dizaines de milliers de lignes de code.

Le système d'exploitation bénéficie d'un écosystème solide dans les domaines critiques, notamment l'aéronautique (avec *Airbus* qui l'a choisi comme plateforme de référence pour l'*A350*), l'automobile, le ferroviaire et le médical.

Le support commercial de *PikeOS* est assuré par *SYSGO*, filiale du groupe *Thales* depuis 2012. Cette acquisition par un acteur majeur de la défense et de l'aérospatial est un gage de pérennité du produit.

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 (micronoyau 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 conteneurs sécurisé avec la possibilité de communiquer de façon sécuriser entre eux. Il a fait l'objet d'une vérification formelle [96].

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

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* [97]. 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* [98], 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 Support multi-processeur

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

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 [97], qui sont essentielles pour la gestion des interruptions dans un environnement multi-coeur.

Le support du *PSCI* (*Power State Coordination Interface*) version 1 indique également que *ProvenVisor* peut gérer l'allumage et l'extinction des coeurs individuels, une fonctionnalité importante pour les systèmes multi-processeurs [97].

5.4 Partitionnement

5.4.1 Partitionnement spatial

ProvenVisor assure un partitionnement spatial strict entre ces machines virtuelles en se reposant sur les mécanismes matériels des architectures ARM. L'hyperviseur supporte les *MMU* de ARM [97].

La technologie *TrustZone* d'ARM assure la séparation entre le *Normal world* (VMs sous *ProvenVisor*) et le *Secure world* (processus sous *ProvenCore*), fournissant un environnement d'exécution de confiance complètement isolé.

Les communications inter-invités ne sont autorisées qu'après autorisation explicite [97], garantissant qu'aucune fuite d'information ne peut se produire entre les VMs sans configuration intentionnelle.

5.4.2 Partitionnement temporel

Les informations publiquement disponibles sur le partitionnement temporel de *ProvenVisor* sont très limitées. Étant un hyperviseur, il est certain qu'il dispose de mécanismes pour partager le temps d'exécution entre ses partitions.

5.4.3 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*.

5.5 Corruption mémoire

ProvenVisor intègre plusieurs mécanismes pour prévenir la corruption 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 minimalistre 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.

5.6 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.

5.7 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.

5.7.1 Intégration avec *ProvenCore*

ProvenCore est un micronoyau formellement vérifié qui s'exécute dans le *Secure world* de *TrustZone* [96]. 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 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.

5.7.2 Compatibilité avec les systèmes d'exploitation invités

ProvenVisor est conçu pour s'intégrer de manière transparente avec les systèmes d'exploitation généralistes courants [97]. 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*.

5.7.3 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 [99]. Des démonstrations sur des modules *Toradex* ont également été réalisées [100], montrant la portabilité de la solution sur différentes plateformes *ARM*.

5.7.4 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.

5.8 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 [97]. 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 coeurs, qui sont souvent déclenchés par des interruptions ou des événements similaires.

5.9 Support *watchdog*

Les informations spécifiques sur le support *watchdog* de *ProvenVisor* ne sont pas documentées publiquement. Dans un environnement virtualisé, le support *watchdog* peut toutefois s'organiser à plusieurs niveaux. Les plateformes ARM modernes intègrent généralement des *watchdogs* matériels que l'hyperviseur pourrait gérer pour assurer la disponibilité du système. *ProvenVisor* pourrait potentiellement présenter des *watchdogs* virtualisés aux VMs invitées, une fonctionnalité commune dans les hyperviseurs modernes, ou agir lui-même comme *watchdog* en surveillant l'activité des VMs. Dans l'architecture combinée *ProvenVisor/ProvenCore*, des mécanismes de surveillance sophistiqués depuis le *Secure world* pourraient offrir une couche supplémentaire de résilience. Pour des informations détaillées, il serait nécessaire de consulter la documentation technique propriétaire de *ProvenRun*.

5.10 Programmation *bare-metal*

ProvenVisor étant un hyperviseur de type 1, il s'exécute directement sur le matériel nu (*bare-metal*) mais n'est pas conçu pour héberger directement des applications *bare-metal*. Son rôle est de fournir un environnement virtualisé pour des systèmes d'exploitation invités. Pour des applications nécessitant un accès *bare-metal* au matériel, plusieurs approches sont possibles: le *passthrough* de périphériques permettant à une VM d'accéder directement à certains périphériques, l'utilisation de *ProvenCore* dans le *Secure world* pour un accès direct et contrôlé, ou une VM dédiée avec accès privilégié aux ressources matérielles. L'ajout d'un hyperviseur introduit une certaine latence par rapport à une exécution purement *bare-metal*, mais les bénéfices en termes d'isolation et de sécurité compensent généralement ce surcoût.

5.11 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 [97]. 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.

5.12 Maintenabilité

ProvenVisor est un produit propriétaire développé par l'entreprise française *ProvenRun*. Le code source n'est pas disponible publiquement et nécessite une licence commerciale dont les conditions doivent être négociées directement avec l'éditeur.

L'écosystème de *ProvenVisor* est ciblé et spécialisé, concentré principalement sur le marché de l'*IdO* et des systèmes embarqués critiques nécessitant une sécurité renforcée, avec un focus sur les architectures *ARM v8-A*.

La taille exacte du code source n'est pas communiquée publiquement, mais est vraisemblablement de petite taille en raison de la conception minimale et du focus sur un *TCB* réduit, caractéristique typique des hyperviseurs formellement vérifiés.

Le support est assuré exclusivement par *ProvenRun*, créant une dépendance à un fournisseur unique. Les clients ne peuvent pas effectuer leurs propres modifications ou corrections urgentes. Le support à long terme doit être négocié dans le contrat de licence.

La vérification formelle a un double impact sur la maintenabilité. D'une part, elle garantit une qualité de code exceptionnelle avec très peu de bugs découverts en production, réduisant les besoins de correctifs urgents. D'autre part, toute modification nécessite de re-prouver les propriétés du système, ralentissant l'évolution et l'adaptation du produit.

La pérennité du produit est directement liée à la santé de l'entreprise *ProvenRun*.

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.

Le projet est initié en 1988 par l'entreprise *OAR (On-Line Applications Research Corporation)* sous contrat de l'*U.S. Army Missile Command* [101]. Cette dernière voulait un système d'exploitation temps-réel basé sur des normes libres et exempt de redevances [102]. À cette époque, le système est destiné à un usage militaire, en particulier dans des missiles¹⁷. En 1993, une première version du projet est rendue publique. À partir de 1995, la gestion du projet est entièrement confiée à *OAR* qui assure la maintenance et le développement de *RTEMS*, ainsi que la maintenance de son infrastructure web. Pendant les années 90, *RTEMS* commence à être utilisé dans le civil, notamment par la *NASA* et l'*ESA (European Space Agency)*. Le projet est alors renommé *Real-Time Executive for Multiprocessor Systems* pour souligner ce changement ainsi que le support des systèmes multiprocesseurs. De nos jours, il est utilisé dans des missions spatiales et notamment la constellation de satellites *Galileo*.

6.1 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'adaptateur *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

¹⁷Le sigle *RTEMS* signifiait alors *Real-Time Executive for Missile Systems*.

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 lancer 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.

6.2 Architectures supportées

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

D'après [103], *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.

6.3 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* [104], mais également pour les architectures *AMP*.

6.3.1 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 ordonneurs de tâches dédiés comme *EDF* (*Earliest Deadline First*) qui tient compte d'échéances via le mécanisme de *deadline* (voir la sous-section 6.4.3) et se comporte comme un ordonneur 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 synchronisations fins via des protocoles de verrouillage comme *OMIP* et *MrsP*.

Aparté: Activation *SMP*

Le support *SMP* n'est pas activé par défaut. Il requiert 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 6.10.

6.3.2 Architectures AMP

Il est possible d'utiliser *RTEMS* sur des *MPSoC*. Par exemple, il existe un *BSP* pour le *MPSoC Xilinx Zynq UltraScale+* [105].

6.4 Partitionnement

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

6.4.1 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* [106], [107].

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*.

6.4.2 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 minimalistes. Il existe une version de cet ordonnanceur pour les multi-processeur [108].
- *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 [108].
- *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* [108].
- *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.

6.4.3 Déterminisme

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

6.4.4 Protocols de synchronisation

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 priorité 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*.

6.4.5 Rate Monotonic Manager

Rate Monotonic Manager permet la gestion de tâches périodiques via l'algorithme *RMS (Rate Monotonic 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.

6.5 Corruption 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 `bsp/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.

6.6 Perte du flux d'exécution

RTEMS étant principalement écrit en langage *C*, il est exposés aux vulnérabilités classiques par corruption 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.

6.7 Écosystème

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

- *profreport* [109]: 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 [110]. Il produit un rapport au format *XML* sur sa sortie standard.
- *rtrace* [111]: 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* [112]: Shell qui comprend de nombreuses commandes affichant des informations sur les tâches en cours d'exécution [113] et permet l'exploration de l'état mémoire [114].
- *CPU Usage Statistics* [115]: *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*.

6.8 Gestion des interruptions

RTEMS fournit un *Interrupt Manager* qui offre les primitives nécessaires pour masquer les interruptions et installer des gestionnaires d'interruption [116], [117].

6.8.1 Masquage des interruptions

RTEMS propose plusieurs directives pour masquer les interruptions. En configuration uniprocesseur, `rtems_interrupt_disable()` et `rtems_interrupt_enable()` désactivent et restaurent les interruptions masquables. Pour garantir la portabilité entre configurations uniprocesseur et *SMP*, les directives `rtems_interrupt_local_disable()` et `rtems_interrupt_local_enable()` sont disponibles dans toutes les configurations [118].

En configuration *SMP*, le masquage local des interruptions ne garantit pas l'exclusion mutuelle à l'échelle du système. *RTEMS* fournit donc des *interrupt locks*, implémentés sous forme de *ticket locks*, pour assurer la synchronisation entre coeurs tout en garantissant l'équité *FIFO* nécessaire à la prévisibilité des systèmes temps réel [119], [120]. Dans les configurations uniprocesseur, ces verrous se réduisent à de simples séquences de désactivation/activation d'interruptions.

RTEMS permet également de contrôler des vecteurs d'interruption individuels via `rtems_interrupt_vector_enable()` et `rtems_interrupt_vector_disable()` [118].

6.8.2 Gestionnaires d'interruption

Les directives `rtems_interrupt_catch()` et `rtems_interrupt_handler_install()` permettent d'installer des *ISR* pour des vecteurs d'interruption spécifiques. *RTEMS* sauvegarde automatiquement le contexte et garantit un ordonnancement correct des tâches à la sortie d'une interruption [117].

La latence d'interruption est bornée en minimisant la durée des sections critiques. Des mesures sur processeur *SPARC Leon 2* montrent des latences typiques de 10 à 40 µs [121]. En configuration *SMP*, les interruptions peuvent être affectées à des coeurs spécifiques pour gérer la latence sous charge [122].

6.9 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 6.9.1.

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*.

6.9.1 Watchdog matériel avec un *Raspberry PI 4*

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

```

1 #include <rtems.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <bsp/watchdog.h>
5
6 rtems_task Init(rtems_task_argument ignored) {
7     raspberrypi_watchdog_init();
8
9     // Configure le timeout à 10 secondes.
10    raspberrypi_watchdog_start(10 * 1000);
11    printf("\nWatchdog initialisé\n");
12
13    // Réinitialise le watchdog toutes les 5 secondes.
14    while (1) {
15        raspberrypi_watchdog_reload();
16        printf("Watchdog rechargeé\n");
17        rtems_task_wake_after (5 * rtems_clock_get_ticks_per_second());
18    }
19 }
20
21 #define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
22 #define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
23 #define CONFIGURE_MAXIMUM_TASKS 1
24 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
25 #define CONFIGURE_INIT
26 #include <rtems/confdefs.h>
27

```

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
```

Il suffit alors de copier cette image sur le *Raspberry* puis de lancer `minicom` pour écouter le port série et observer les messages émis par le noyau.

6.10 Qualifications & certifications

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

6.10.1 Kit de qualifications par l’ESA

L’*ESA* offre un kit de *qualification* pour des applications de *RTEMS* dans le domaine spatial [123]. 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 Commons Attribution-ShareAlike 4.0*.

L'ESA (*European Space Agency*) offre un kit de *qualification* pour des applications de *RTEMS* dans le spatial [123] 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 [124], un model-checker. Edisoft a encore contribué sur cette version.

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

6.10.2 Model checking avec *Promela/SPIN*

La correction fonctionnelle de certaines parties de l'*API* de *RTEMS* ont été vérifié par *model checking* [124] 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 ordonneurs de *RTEMS*.

Plus d'informations sont disponibles dans la documentation officielle [125].

6.11 Programmation *bare-metal*

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

6.12 Temps de démarrage

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

6.13 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 à lier son programme avec le code source de *RTEMS* sans devoir redistribuer son propre code source [126].

RTEMS bénéficie d'un écosystème particulièrement important dans le domaine spatial, où il est utilisé par la *NASA* et l'*ESA* dans de nombreuses missions, notamment la constellation de satellites *Galileo*, le *James Webb Space Telescope*, et les rovers martiens. Cette adoption par des agences spatiales de premier plan témoigne de la maturité et de la fiabilité du système.

Le support commercial de RTEMS est assuré principalement par l'entreprise *OAR (On-Line Applications Research Corporation)*, qui maintient et développe le projet depuis 1995. Un support est disponible pour les entreprises européennes et américaines. La communauté active du projet offre également un support gratuit, bien que sans garantie formelle.

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 exigeant 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*. 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*.

¹⁸Acronyme pour *National Information and Communications Technology Australia*.

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 [127].

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 coeurs.

Le micronoyau utilise un verrou global *BKL* de type *CLH* comme mécanisme de synchronisation [128]. 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 [129].

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 [128]. Cette vérification est d'autant plus difficile que les architectures *SMP* modernes ont des modèles de mémoire faible [128], [130]. 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 [130].

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* comme expliqué dans la section précédente.

Il y a également un support pour *OpenAMP*.

Quelques projets qui utilisent *seL4* sur des architectures *AMP*: [131].

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 [132]. De façon plus concrète, une capacité se présente sous la forme d'un pointeur immuable 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 représentent des structures de données du noyau ou encore des ressources abstraites, tandis que d'autres représentent des zones mémoires contigües inutilisées [133].

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 [132], [134].

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 [135].

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 propose deux modes de partitionnement temporel: un mode standard et des extensions *MCS* (*Mixed-Criticality System*).

L'ordonnanceur standard de *seL4* combine deux niveaux d'ordonnancement [134]:

- Un ordonnanceur hiérarchique de haut niveau basé sur des *domaines*: les domaines sont ordonnancés de manière cyclique et déterministe selon un planning statique configuré à la compilation. Chaque domaine reçoit une tranche de temps fixe et les communications *IPC* entre domaines sont différées.
- Un ordonnanceur de *threads* au sein de chaque domaine: ordonneur préemptif à priorités fixes (256 niveaux) avec *round-robin* pour les threads de même priorité.

Les extensions *MCS* [136] introduisent les *contextes d'ordonnancement* qui permettent d'allouer un budget de temps *CPU* à chaque *thread* sur une période donnée. L'isolation temporelle est assurée par un algorithme de serveur sporadique. Ces extensions offrent des garanties temps réel renforcées et sont en cours de vérification formelle [137].

7.9 Déterminisme

seL4 se distingue par une approche originale pour garantir le déterminisme nécessaire aux systèmes temps réel. Alors que la plupart des *RTOS* privilégient un noyau entièrement préemptible pour minimiser la latence, *seL4* adopte une approche événementielle largement non préemptible. Le noyau s'exécute avec les interruptions matérielles désactivées, ce qui simplifie sa conception et sa vérification formelle. Pour offrir de bonnes garanties temps réel malgré cette non-préemptibilité, tous les appels système ont un temps d'exécution borné: *seL4* évite les boucles infinies, élimine l'allocation dynamique et introduit des points de préemption explicites dans les rares appels longs.

Le déterminisme de *seL4* est renforcé par une analyse *WCET* formellement vérifiée [138], [139]. C'est à ce jour le seul noyau en mode protégé disposant d'une telle analyse complète et rigoureuse, fournissant des bornes supérieures certifiées pour tous les appels système. Cette analyse permet d'estimer précisément le temps d'exécution des routines du noyau dans le pire cas, garantissant ainsi le respect des échéances temporelles critiques. Combiné aux extensions *MCS* qui offrent une isolation temporelle stricte, *seL4* peut fournir des garanties de temps réel strict tout en maintenant l'isolation spatiale, ce qui n'est pas le cas de nombreux autres *RTOS*.

7.10 Corruption 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 [140]. 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* [141] 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

Les interruptions sont gérées en espace utilisateur via le système de *capabilities* [142]. La tâche racine reçoit *seL4_CapIRQControl* permettant de dériver des *capabilities IRQHandler* pour des *IRQ* spécifiques. Les applications reçoivent les interruptions via des objets notification

associés aux `IRQHandler`. Le masquage est implicite : aucune interruption n'est livrée jusqu'à l'acquittement explicite via `seL4_IRQHandler_Ack`.

Pour la virtualisation, `seL4` exploite les extensions matérielles comme le *VGIC* sur *ARM* (support *GICv2*, *GICv3* en développement) permettant l'injection d'interruptions virtuelles sans piégeage systématique. La bibliothèque `libsel4vm` fournit une interface de haut niveau avec des fonctions comme `vm_inject_irq` et `vm_register_irq`.

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*

`seL4` dispose d'un support pour la programmation *bare-metal* en langage *C* et *Rust*. À notre connaissance, il n'existe pas de support officiel pour les langages *Ada* et *OCaml* sur `seL4`.

Pour le langage *C*, le projet fournit la bibliothèque `libsel4` [143] pour interagir avec le micro-noyau et un *RTE* `sel4runtime` [144] minimaliste.

Le langage *Rust* bénéficie d'un support officiel et documenté pour `seL4` [145], [146].

7.16 Temps de démarrage

Nous n'avons pas trouvé d'informations précises sur le temps de démarrage du noyau `seL4`.

7.17 Maintenabilité

Le noyau `seL4` est un logiciel libre distribué principalement sous licence *GPLv2*. Le code utilisateur et les pilotes peuvent être distribués sous n'importe quelle licence [147].

Le projet a débuté en 2006, ce qui en fait un projet d'environ 19 ans.

La base de code est écrite en langage *C* à 87% pour un total de 68 175 *SLOC*. Cette petite taille est le gage d'une *TCB* réduite. Cependant, ce chiffre ne tient pas compte de la difficulté importante à maintenir des démonstrations tout en continuant à faire évoluer le logiciel.

`seL4` bénéficie d'un écosystème actif dans les domaines critiques et son code librement accessible facilite le développement de nouveaux logiciels.

Le support commercial de `seL4` est assuré par plusieurs organisations, dont la *seL4 Foundation* et *Trustworthy Systems*.

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 prédecesseur *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 collaborer 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 8.4).

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 certaines distributions *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/vpvvndr4sfi2xapk3zlh9hcfc6w087k-pvgrub-image/lib/
7 grub-xen/grub-x86_64-xen.bin'
8 disk=[ './alpine.qcow2,qcow2,hda,w' ]
9 boot='d'
10 #vif = [ 'mac=00:16:3e:00:00:00,bridge=xenbr0' ]
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 8.4.

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*. Aujourd'hui *Xen* supporte les architectures suivantes: *x86-32*, *x86-64*, *ARM v7*, *ARM v8*. Il existe aussi un support préliminaire pour les architectures *PowerPC* et *RISC-V* [148]. *Xen* supporte également la virtualisation assistée par le matériel sur *ARM* [149].

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 ordonneurs adaptés aux architectures multi-processeur.

8.3.1 Support *SMP*

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

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

8.3.2 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 Xilinx Zynq UltraScale+* [150].

8.4 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*.
- 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 hôte¹⁹. 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 exemple) 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.

¹⁹Ce dernier point est crucial pour support des systèmes d'exploitation à sources fermées, comme par exemple *Windows*

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 ou *Mini-OS*.
- 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*.

8.4.1 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 domaines avec des priviléges minimaux. Cela améliore considérablement la sécurité.

Un *stub domain* (ou *stubdomain*) est un domaine système spécialisé utilisé pour désagréger le domaine de contrôle (*dom0*) [151], [152]. 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 priviléges 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 priviléges 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* [56], bien que des travaux aient été menés sur des *stub domains* basés sur *Linux*.

8.4.2 Partitionnement spatial

Xen assure l'isolation mémoire entre les domaines en utilisant différentes techniques selon le type de virtualisation. Pour les domaines paravirtualisés, le système *direct paging* impose que les mises à jour des tables de pages passent par des hypercalls vérifiés par l'hyperviseur [153]. Pour les domaines *HVM*, *Xen* utilise soit des *shadow page tables* sur les processeurs anciens, soit une traduction en deux étapes (*EPT/NPT*) sur les processeurs récents, offrant de meilleures performances. Sur *ARM*, *Xen* s'appuie sur la traduction d'adresses en deux étapes du matériel [154]. Pour les systèmes critiques, le mode *dom0less* permet un partitionnement statique et une isolation stricte [155]. L'hyperviseur suit une architecture minimale et poursuit une conformité *MISRA C* pour la certification [156].

8.4.3 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 [157]. 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 exécutée²⁰.
- *Credit2 Scheduler* est une évolution de *Credit Scheduler* [158]. 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*.

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 8.4.4.

8.4.4 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 8.4.3 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 doter *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*) [159]. 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*.

8.5 Corruption 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.

8.6 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,

²⁰On dit que l’ordonnanceur est *work-conserving*.

- 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].

8.7 Écosystème

Xen propose plusieurs outils pour le monitoring des performances et de l'état du système [160], [161]:

- *xentop* [162]: 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* [163]: 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.
- *xl* [164]: Outil livré avec *Xen* qui offre des fonctionnalités basiques pour observer l'état des domaines en cours d'exécution.
- *xentrace* [165]: Outil distribué dans *Xen* qui 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*²¹.
- *Xen Orchestra* est une 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.

8.8 Programmation *bare-metal*

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

Pour le langage *C*, *Mini-OS* [56] est un noyau minimaliste distribué avec *Xen* qui fournit une base pour le développement d'applications *bare-metal*. Il est principalement utilisé pour les *stub domains* [151], des domaines légers dédiés à l'exécution de composants isolés.

Pour le langage *Rust*, il existe des crates *xen* et *xen-sys*.

²¹Contrairement à *xentrace*, *xenanalyze* n'est pas distribué avec *Xen*.

Le projet *MirageOS* a développé un environnement d'exécution complet pour le langage de programmation *OCaml* sur des partitions de type *PVH* [47].

8.9 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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <xenctrl.h>
4
5 int main(void) {
6     xc_interface *h = xc_interface_open(NULL, NULL, 0);
7     if (!h)
8         return EXIT_FAILURE;
9
10    // Configure le timeout à 30 secondes.
11    int timeout = 30;
12    int id = xc_watchdog(h, 0, timeout);
13    if (id <= 0)
14        goto failed;
15    printf("Watchdog initialisé\n");
16
17    // Réinitialise le watchdog toutes les 15 secondes.
18    while (1) {
19        sleep(15);
20        if (xc_watchdog(h, id, timeout))
21            goto failed;
22        printf("Watchdog rechargeé\n");
23    }
24
25 failed:
26     xc_interface_close(h);
27     return EXIT_FAILURE;
28 }
29

```

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* [166]. 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.

8.10 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* [167]. 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 [167].

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 [168].

8.11 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 [155], [169]. 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 coeurs 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.

8.12 Maintenabilité

Xen est un logiciel libre distribué majoritairement sous licence GPLv2. Certaines portions sont distribuées sous des licences plus permissives afin de ne pas contraindre le choix de licence pour l'utilisateur. Ces exceptions sont spécifiées dans les en-têtes des fichiers concernés. Plus d'informations sont disponibles dans le fichier COPYING du dépôt git de *Xen* [170].

Il 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 et modes de virtualisation (*PV*, *HVM* et *PVH*). *Xen* est donc réputé avoir une *TCB* importante du fait de ce volume de code. Cependant, il faut noter que la taille du code varie beaucoup suivant l'architecture et le mode de virtualisation considéré. Ainsi, l'implémentation du mode *PVH* pour *ARM* est d'une taille nettement plus réduite.

Le projet *Xen* a débuté en 1999, ce qui en fait un système d'environ 26 ans. *Xen* bénéficie d'un écosystème large dans le domaine du cloud computing et de l'hébergement.

Le support commercial de *Xen* est assuré principalement par *Citrix* à travers sa distribution *Citrix Hypervisor*. Le *Xen Project*, hébergé par la *Linux Foundation*, coordonne le développement Open Source et fédère la communauté.

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, RISC-V
- **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 et avionique
- **Licences** : GPL v2 (version libre) + version propriétaire XtratuM/NG (fentISS)

XtratuM est un hyperviseur temps-réel qualifié pour un usage dans l'avionique ou 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 [171], [172]. Ces travaux universitaires ont abouti à la création de l'entreprise *fentISS* [173] en 2010 avec le soutien du *CNES* et du groupe *Airbus* [172]. 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 [171].

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 [174], [175], [176], 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 *LEON2*. Toutefois les documents récentes ne mentionnent plus les architectures *x86* et *PowerPC*, ce qui laisse à penser que leur support n'a pas été maintenu et nous n'avons pas trouvé de mention de l'architecture *x86-64*.

D'après les dernières brochures [177], *XtratuM* supporte les architectures suivantes: *ARMv7*, *ARMv8*, *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 [178]. Chaque partition peut allouer plusieurs cœurs via une couche d'abstraction sous forme de *CPU virtuels* [178].

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 exécutées 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 [179].

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 [180]. 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 [180], [181]. 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* [180], [182],
- Le *RTOS* RTEMS, notamment sur les plateformes LEON [183],
- Le noyau *Linux* [183],
- Le micronoyau temps réel ORK+ (*Open Ravenscar Kernel*). Il a été porté sur *XtratuM* en 2011 [184]. Il permet le développement d'applications en *Ada* avec le profile *Ravenscar*.

9.4 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 [185].

9.5 Corruption 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.

9.6 Écosystème

L'entreprise *fentISS* commercialise une suite d'outils pour faciliter le développement avec *XtratuM*. Cette suite inclus les outils suivants:

- *SKE* [186]: simulateur *XtratuM* sur serveurs,
- *XPM* [183]: plugin Eclipse pour la gestion de projets *Xtratum*,
- *Xoncrete* [183]: analyse et génération d'ordonnancement,
- *Xmcparser* [187]: configuration de l'hyperviseur,
- *Xtraceview* [188]: support d'observabilité.

9.6.1 *Health monitoring*

XtratuM intègre un service de *health monitoring* conforme à la norme *ARINC-653* [180]. 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.

9.7 Support de *watchdog*

Nous n'avons pas trouvé d'informations sur le support de *watchdog*. Toutefois étant donné le domaine d'application de cet hyperviseur, il est très probable qu'un tel support existe.

9.8 Temps de démarrage

Nous n'avons pas trouvé d'informations sur le temps de démarrage de l'hyperviseur *XtratuM*.

9.9 Qualifications et certifications

ECSS-Qualified?

XtratuM est qualifié selon la norme *ECSS (European Cooperation for Space Standardization)* catégorie B [189]. 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.

9.10 Masquage des interruptions

XtratuM virtualise les interruptions matérielles pour garantir l'isolation spatiale et temporelle des partitions conformément à la norme *ARINC-653*. Les partitions ne peuvent pas accéder directement à la table des interruptions ni interagir avec les interruptions matérielles natives. Toute tentative d'activation ou de désactivation directe des interruptions matérielles déclenche un *trap* géré par l'hyperviseur [190], [191].

9.10.1 Virtualisation des interruptions

L'hyperviseur définit ses propres interruptions virtuelles qui sont délivrées aux partitions [190]. Toutes les interruptions matérielles sont directement gérées par *XtratuM* et, lorsque nécessaire, propagées aux partitions qui définissent leur propre table d'interruptions virtuelles [192]. Le sous-système d'interruptions virtuelles a été réécrit de zéro dans la version 3.1 de *XtratuM*.

Durant l'exécution d'une partition, seules les interruptions allouées à cette partition sont activées au niveau matériel, ce qui minimise les interférences entre partitions grâce à un mécanisme d'isolation matériel [193]. Lorsqu'une partition est dans l'état suspendu, son *CPU* virtuel n'est pas ordonné et les interruptions ne sont pas délivrées. Les interruptions survenues pendant cet état restent en attente [191].

9.10.2 API de masquage des interruptions

XtratuM fournit une *API* basée sur des *hypercalls* pour permettre aux partitions de masquer et démasquer les interruptions virtuelles [194]. Les principaux *hypercalls* disponibles sont:

- *XM_mask_irq()* : masque une interruption virtuelle spécifique,
- *XM_unmask_irq()* : démasque une interruption virtuelle.

Ces *hypercalls* vérifient que le numéro d'interruption est dans les plages autorisées (interruptions matérielles ou interruptions étendues) et utilisent des opérations atomiques (*XMAatomicSetMask* et *XMAatomicClearMask*) pour modifier les masques d'interruption dans la *Partition Control Table* (pct) [194]. Une partition ne peut masquer que les interruptions

matérielles qui lui ont été allouées, ce qui fait partie de la stratégie de *XtratuM* pour maintenir l’isolation temporelle et spatiale entre les partitions dans les systèmes critiques [194].

9.10.3 Health Monitor

XtratuM intègre un module appelé *Health Monitor* qui gère de façon étendue les *traps*, les interruptions et les erreurs [193]. Ce module permet la détection et la gestion centralisée des événements liés aux interruptions, contribuant à la robustesse du système dans un contexte critique.

9.11 Maintenabilité

À l’origine *XtratuM* était un projet open source sous licence *GPLv2* [195]. Cette version ne semble plus être développée et *fentISS* distribue une réécriture du noyau appelée *XNG* (*XtratuM Next Generation*) sous licence propriétaire [196], [197]. L’entreprise *fentISS* ne semble pas communiquer sur ses licences. Les modalités sont donc probablement à négocier avec *fentISS* au cas par cas.

Nous n’avons pas pu évaluer la taille de la base de code, faute d’informations librement accessibles. *XtratuM* étant un noyau de séparation certifiable, sa base de code doit être réduite afin de rendre la certification possible.

Le projet *XtratuM* a été initié en 2004, ce qui en fait un système d’environ 21 ans. Ayant été réécrit au alentour de , sa base de code aurait en fait .

Le projet *XtratuM* a été initié en 2004 au sein de l’institut *Automática e Informática Industrial* (ai2) de l’*Universidad Politécnica* de Valence en Espagne, ce qui en fait un système d’environ 21 ans. Le projet a connu une transition majeure avec le passage de la version open-source originale à la version propriétaire *XNG* (*XtratuM Next Generation*) développée par *fentISS*.

XtratuM s’inscrit dans l’écosystème aérospatial européen, notamment à travers le projet *SAFEST* qui vise à faire collaborer différents acteurs du secteur pour améliorer les performances et réduire les coûts. Le système cible particulièrement l’*IMA* (*Integrated Modular Avionics*), une approche visant à consolider des fonctions logicielles auparavant dispersées sur des calculateurs dédiés.

Le support commercial de *XtratuM* est assuré par l’entreprise *fentISS* qui développe et maintient la version *XNG*.

10 Tableaux comparatifs

Ce dernier chapitre regroupe des tableaux récapitulatifs et comparatifs suivant les différents critères de l'étude.

10.1 Type de système d'exploitation

OS	Temps réel	Hyperviseur	LibOS	GPOS	RTOS	Noyau de sép.
Linux						
MirageOS						
PikeOS						
ProvenVisor						
RTEMS						
seL4						
Xen						
XtratuM						

Tableau 4. – Type de systèmes d'exploitation

10.2 Architectures supportées

Le tableau ci-dessous résume le support des architectures par système d'exploitation.

OS	x86-32	x86-64	ARMv7	ARMv8	PowerPC	MIPS	RISC-V	SPARC
Linux								
MirageOS	OCaml < 5		OCaml < 5		spt			
PikeOS								
ProvenVisor								
RTEMS								
seL4								
Xen								
XtratuM	?	?						

Tableau 5. – Architectures supportées par système d'exploitation.

Quelques remarques pour l'interprétation de ces informations:

- Pour les hyperviseurs, il faut un support matériel supplémentaire pour tirer le meilleur du matériel. Le lecteur peut se rapporter à la section du système concerné pour trouver ces informations,
- Pour les hyperviseurs, il faut que le système invité supporte également l'architecture, à moins d'avoir recours à la virtualisation complète,
- Le support de l'architecture ne suffit pas à supporter n'importe quelle carte de cette architecture. *PikeOS*, *ProvenVisor*, *RTEMS*, *seL4* et *XtratuM* publient des *BSP* pour des cartes et des architectures spécifiques. En dernier terme, seul l'existence de ce *BSP* fait foi pour le support d'une carte donnée.

10.3 Partitionnement temporel

10.4 Programmation *bare-metal*

Hyperviseur	Ada	C	OCaml	Rust
KVM (Linux)				
PikeOS	Ravenscar			
ProvenVisor				
seL4				
Xen				
XtratuM	Ravenscar			

10.5 Maintenabilité

Le tableau ci-dessous rassemble les informations relatives à la maintenabilité des différents systèmes d'exploitation de l'étude.

OS	Licence(s)	Écosystème	Taille (SLOC)	Doc	Support commercial	Ancienneté (années)
Linux	GPLv2	Très large	27M	Excellente	Red Hat, SUSE, Canonical, ...	34
MirageOS	ISC/ LGPLv2	Moyen	< 10k	Bonne	Tarides	16
PikeOS	Propriétaire	Moyen		Limitée	SYSGO	20
ProvenVisor	Propriétaire	Limité			ProvenRun	10
RTEMS	BSD 2-Clause	Large	2M	limitée	OAR	32
seL4	GPLv2	Moyen	70k	Technique	seL4	19
Xen	GPLv2	Large	500k	Datée	Citrix, Xen Project	22
XtratuM	Propriétaire	Limité			fentISS	21

Tableau 7. – Comparaison de la maintenabilité des systèmes d'exploitation.

Quelques remarques pour l'interprétation de ces données:

- Les tailles des bases de code ont été estimées en utilisant le logiciel `SLOCCount` [198]. Ce dernier utilise le *SLOC* (*source line of code*) comme unité et n'inclut pas les commentaires dans le comptage. De plus, nous n'avons pas pris les pilotes en compte dans la mesure. Cela explique certaines divergences avec des chiffres trouvables sur le web. Par exemple *Linux* distribue une quantité colossale de pilotes pour un total de l'ordre de dizaines de millions de *SLOC*. La majorités de ces lignes de code n'aboutissent pas dans l'exécutable final et il serait injuste de les prendre en compte,
- Pour *MirageOS*, la taille de la base de code ne concerne que le noyau irréductible de l'OS. En pratique, il faut utiliser d'autres bibliothèques du projet *MirageOS* pour développer son application,

- Pour *seL4*, le tableau comparatif ne prend pas en compte la maintenance des preuves qui constituent en réalité une part importante des efforts de développement,
- Si l'ancienneté est souvent un gage de stabilité, elle entraîne aussi une dette technique plus importante. On constate que les systèmes les plus anciens sont également ceux ayant la base de code la plus volumineuse. En contrepartie, ils sont un écosystème plus développé et mature.

11 Glossaire

ABI. Application Binary Interface. interface binare-programme	81
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, 44, 56, 57, 64, 71
API. Applicaton Programming Interface. Interface de programmation applicative.	12, 13, 17, 28, 35, 40, 43, 45, 46, 47, 58, 59, 61, 65, 66, 67, 81, 82
BKL. Big Kernel Lock. todo	16, 44, 64
BSP. Board Support Package. todo	6, 43, 44, 46, 47, 56, 57, 58, 59, 67, 79, 85
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.	63
COTS. Commercial off-the-shelf. todo	13, 14
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, 45, 46, 73
ECSS. European Cooperation for Space Standardization. Ensembles de normes utilisées dans le spatial.	61
ESA. European Space Agency. todo	55, 60, 61
GNU. GNU is Not Unix. todo	15
GPOS. General-Purpose Operating System. Système d'exploitation généraliste offrant un large éventail de services.	5, 6, 8, 9, 15, 33, 35, 57, 73
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.	46
IPC. Inter-Process Communication. todo	36, 37, 38, 58, 65, 80
IRQ. Interrupt ReQuest. Requête d'interruption envoyée à un PIC afin d'interrompre le processeur.	12, 66
ISR. Interrupt Service Routine. Gestionnaire d'interruption exécutée lorsqu'une interruption matérielle ou logicielle survient.	12, 20, 21, 59
IdO. internet des objets. Réseau d'objets embarqués et connectés au travers d'Internet.	49, 54
MCE. Machine Check Exception. Erreur émise par le matériel qui est généralement fatale.	81
MMU. Memory Management Unit. Microcontrôleur dédié à la traduction des adresses virtuelles en adresses physiques.	8, 9, 17, 44, 50, 57, 65
MPSoC. Multiprocessor System on a chip. Système embarqué complet sur un seul circuit intégré comprenant plusieurs processeurs généralement hétérogènes.	7, 17, 44, 57, 64, 71

MPU. Memory Protection Unit. Microcontrôleur chargé de vérifier les accès en mémoire.	8, 57
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, 66
RTE. RunTime Environment. todo	4, 13, 31, 35, 36, 47, 67, 81
RTOS. Real-time Operating System. Système d'exploitation dédié pour le temps réel.	5, 9, 66, 81
Il offre des garanties fortes en matière de déterminisme.	
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, 34, 36, 44, 50, 56, 57, 58, 59, 61, 64, 71, 73
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.	33, 34, 49, 50, 51, 52, 54, 67, 77
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.	49
TLB. Translation Lookaside Buffer. Mémoire cache du processeur utilisée dans le but d'accélérer la traduction des adresses virtuelles en adresses physiques.	44, 45
VM. Virtual Machine. todo	13, 33, 69, 73
WCET. Worst Case Execution Time. todo	44, 45, 46, 51, 64, 66
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, 31, 34, 47, 53, 61, 67, 74, 76, 80, 81, 86
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.	31, 32, 55, 56
espace noyau. todo	12, 57, 65
espace utilisateur. todo	17, 27, 31, 32, 57, 65
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_.	31
monolithique. todo	15
paravirtualisation. todo	69, 71
qualification. todo	60, 61

ramasse-miette. TODO	13
runtime. Environnement d'exécution. Dans le cas d'un langage de programmation managé,	35, 38
cela inclut son garbage collector.	
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, 79
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, n° 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, n° 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/rpmmsg.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] « Linux generic IRQ handling ». Consulté le: 9 décembre 2025. [En ligne]. Disponible sur: <https://www.kernel.org/doc/html/latest/core-api/genericirq.html>

-
- [38] « 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>
 - [39] « 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>
 - [40] « The Linux Watchdog driver API ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: <https://docs.kernel.org/watchdog/watchdog-api.html>
 - [41] « RustyHermit ». Consulté le: 9 décembre 2025. [En ligne]. Disponible sur: <https://hermit-os.github.io/tutorial/>
 - [42] « Running RustyHermit ». Consulté le: 9 décembre 2025. [En ligne]. Disponible sur: <https://github.com/hermit-os/hermit-rs/wiki/Running-RustyHermit>
 - [43] « Run Mirage Unikernels on KVM/QEMU with Solo5 ». Consulté le: 9 décembre 2025. [En ligne]. Disponible sur: <https://mirageos.org/blog/introducing-solo5>
 - [44] 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.
 - [45] A. Al Abdulla et A. Hedberg, « Decreasing boot time in an embedded Linux environment », Bachelor Thesis, Örebro, Sweden, 2023.
 - [46] A. Madhavapeddy, « Unikernels ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: <https://anil.recoil.org/projects/unikernels>
 - [47] « Xen Project: MirageOS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: <https://xenproject.org/projects/mirage-os/>
 - [48] J. Vouillon, « Lwt: a cooperative thread library », in *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, 2008, p. 3-12.
 - [49] « Lwt manual ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: <https://ocsigen.org/lwt/latest/manual/manual>
 - [50] « Ocsigen website ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: <https://ocsigen.org/home/intro.html>
 - [51] 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.
 - [52] « Vchan: Low-latency inter-VM communication channels ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: <https://mirage.io/blog/update-on-vchan>
 - [53] K. Sivaramakrishnan *et al.*, « Retrofitting parallelism onto OCaml », *Proc. ACM Program. Lang.*, vol. 4, n° ICFP, août 2020, doi: 10.1145/3408995.
 - [54] « Memory model: The hard bits - OCaml manual ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: <https://ocaml.org/manual/5.4/memorymodel.html>
 - [55] « MirageOS on OCaml 5 ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: <https://tarides.com/blog/2025-02-06-mirageos-on-ocaml-5/>

-
- [56] « Mini-OS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: <https://wiki.xenproject.org/wiki/Mini-OS>
 - [57] « MirageeOS on Unikraft ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: <https://discuss.ocaml.org/t/mirageos-on-unikraft/16975>
 - [58] « GitHub ocaml-vchan ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://github.com/mirage/ocaml-vchan>
 - [59] « OCaml Profiling ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://ocaml.org/manual/5.4/profil.html>
 - [60] « Git mirage-monitoring ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://git.robur.coop/robur/mirage-monitoring>
 - [61] « Memtrace ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: <https://github.com/janestreet/memtrace>
 - [62] « Memtrace Viewer ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: https://github.com/janestreet/memtrace_viewer
 - [63] « Github memtrace-mirage ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://github.com/robur-coop/memtrace-mirage>
 - [64] « GitHub mirage-profile ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://github.com/mirage/mirage-profile>
 - [65] 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/>
 - [66] « GitHub mirage-trace-viewer ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: <https://github.com/talex5/mirage-trace-viewer>
 - [67] 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, n° 11, p. 30-44, 2013.
 - [68] 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/>
 - [69] 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/>
 - [70] R. Kaiser et S. Wagner, « Evolution of the PikeOS microkernel », in *First international workshop on microkernels for embedded systems*, 2007.
 - [71] « 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>
 - [72] « 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>

-
- [73] « 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>
 - [74] « 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>
 - [75] O. M. Motzkus Andreas, « PikeOS Safe Real-Time Scheduling ». 2016.
 - [76] « Security Target PikeOS Separation Kernel v5.1.3 ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: https://commoncriteriaproduct.org/files/epfiles/1146b_pdf.pdf
 - [77] 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>
 - [78] « ELinOS - Embedded Linux Distribution ». Consulté le: 20 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/elinos>
 - [79] « 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>
 - [80] « Windows as PikeOS Guest OS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/windows>
 - [81] « LEON5 - Gaisler ». Consulté le: 5 décembre 2025. [En ligne]. Disponible sur: <https://www.gaisler.com/products/leon5>
 - [82] « 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>
 - [83] « Technology Partner Alliances ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/technology-alliances>
 - [84] « Codeo ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/codeo>
 - [85] « Codeo for PikeOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/popups/codeo-for-pikeos>
 - [86] « Codeo for ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/popups/codeo-for-elinos>
 - [87] « 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>
 - [88] « Rapita Systems ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/technology-alliances/rapita>
 - [89] « OS-and Hypervisor-aware Debugging ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.lauterbach.com/features/os-awareness>

-
- [90] « PikeOS ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: <https://en.wikipedia.org/wiki/PikeOS>
 - [91] « 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
 - [92] « 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>
 - [93] « 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>
 - [94] « Rust for PikeOS & ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.sysgo.com/rust>
 - [95] « 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>
 - [96] S. Lescuyer, « ProvenCore: Towards a Verified Isolation Micro-Kernel. », in *MILS@HiPEAC*, 2015.
 - [97] « ProvenRun homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: <https://provenrun.com/provenvisor/>
 - [98] « 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/>
 - [99] « 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
 - [100] « 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>
 - [101] 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-j Joel-Sherrill/>
 - [102] « About OAR and RTEMS ». Consulté le: 16 octobre 2025. [En ligne]. Disponible sur: <https://www.rtems.com/aboutoarandrtems>
 - [103] « RTEMS: Architectures ». Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: <https://www.rtems.org/architectures/>
 - [104] « RTEMS Symmetric Multiprocessing ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/symmetric_multiprocessing_services.html

-
- [105] « RTEMS Xilinx Zyna UltraScale+ BSP ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/user/bsp/arm/xilinx-zynqmp.html>
 - [106] « 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
 - [107] « Memory Model - RTEMS documentation ». Consulté le: 25 novembre 2027. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/6.1/cpu-supplement/port.html>
 - [108] « SMP Schedulers ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/c-user/scheduling-concepts/smp-schedulers.html>
 - [109] « 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
 - [110] « BSP Build System ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/user/bld/index.html>
 - [111] « Tracing ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/user/tracing/index.html>
 - [112] « RTEMS Shell Guide ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/shell/index.html>
 - [113] « RTEMS Specific Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/rtems_specific_commands.html
 - [114] « Memory Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/memory_commands.html
 - [115] « CPU Usage Statistics ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/cpu_usage_statistics.html
 - [116] « RTEMS: Interrupt Manager ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/6.1/c-user/interrupt/index.html>
 - [117] « RTEMS Interrupt Manager: Introduction ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/c-user/interrupt/introduction.html>
 - [118] « RTEMS Interrupt Manager: Directives ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/branches/master/c-user/interrupt/directives.html>
 - [119] « RTEMS Symmetric Multiprocessing (SMP) ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/symmetric_multiprocessing_services.html
 - [120] « RTEMS: Interrupt Locks ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://ftp.rtems.org/pub/rtems/releases/5/5.1/docs/html/doxygen/group__ClassicINTRLocks.html
 - [121] « RTEMS core analysis for space applications ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://www.researchgate.net/publication/269304137_RTEMS_core_analysis_for_space_applications

-
- [122] « RTEMS Symmetric Multiprocessing (SMP) ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/symmetric_multiprocessing_services.html
 - [123] « RTEMS SMP QDP ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: <https://rtems-qual.io.esa.int/>
 - [124] 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.
 - [125] « Formal Verification Overview ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://docs.rtems.org/docs/main/eng/fv/overview.html>
 - [126] « 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
 - [127] « seL4 Supported platforms ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/Hardware/>
 - [128] 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
 - [129] 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
 - [130] 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>
 - [131] « Solox AMP Rust ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: <https://github.com/jonlamb-gh/solox-amp-rust>
 - [132] « The seL4 Capability System ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: <https://www.cl.cam.ac.uk/research/security/ctsrn/cheri/workshops/pdfs/20160423-sel4-capabilities.pdf>
 - [133] « Capabilities - Tutorial seL4 ». Consulté le: 2 décembre 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/Tutorials/capabilities.html>
 - [134] « The seL4 Microkernel - An Introduction ». Consulté le: 30 novembre 2025. [En ligne]. Disponible sur: <https://sel4.systems/About/seL4-whitepaper.pdf>
 - [135] 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
 - [136] 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
- [137] « 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>
- [138] 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.
- [139] 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.
- [140] 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.
- [141] « Microkit User Manual ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/projects/microkit/manual/latest/>
- [142] « seL4: Interrupts tutorial ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/Tutorials/interrupts.html>
- [143] « seL4 API Reference ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/projects/sel4/api-doc.html>
- [144] « The seL4 run-time ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/projects/sel4runtime/>
- [145] « Rust | seL4 docs ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://docs.sel4.systems/projects/rust/>
- [146] « rust-sel4 ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: <https://github.com/seL4/rust-sel4>
- [147] « seL4 Licensing ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: <https://sel4.systems/Legal/license.html>
- [148] « Xen 4.20 release notes ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: <https://xenproject.org/blog/xen-project-4-20-oss-virtualization/>
- [149] « Xen ARM with Virtualization Extensions ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions
- [150] R. VanVossen, « XEN ON THE ZYNQ ULTRASCALE+ MPSOC ».
- [151] « Xen: Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: <https://wiki.xenproject.org/wiki/StubDom>
- [152] « Xen: Device Model Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains
- [153] « X86 Paravirtualised Memory Management ». Consulté le: 3 décembre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management

-
- [154] 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-mmuvirtualization-with-xen-on-arm>
 - [155] « Xen: dom0less ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/unstable/features/dom0less.html>
 - [156] « 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/>
 - [157] « Credit Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit_Scheduler
 - [158] « Credit2 Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit2_Scheduler
 - [159] 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.
 - [160] « 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
 - [161] 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>
 - [162] « xentop man page ». Consulté le: 11 décembre 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/unstable/man/xentop.1.html>
 - [163] « xenmon man page ». Consulté le: 11 décembre 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/unstable/man/xenmon.1.html>
 - [164] « xl man page ». Consulté le: 11 décembre 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/unstable/man/xl.1.html>
 - [165] « xentrace man page ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/4.20-testing/man/xentrace.8.html>
 - [166] « xenwatchdog man page ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: <https://xenbits.xen.org/docs/4.20-testing/man/xenwatchdog.8.html>
 - [167] « Xen: Event Channel Internals ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Event_Channel_Internals
 - [168] « 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>
 - [169] « 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/>

-
- [170] « 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>
 - [171] 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.
 - [172] « 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>
 - [173] « fentISS ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/>
 - [174] « 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>
 - [175] « Space Missions ». Consulté le: 3 novembre 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/missions/>
 - [176] « FENTISS Celebrates 1,000+ Spacecraft Deployments of XtratuM ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://embeddedcomputing.com/technology/software-and-os/fentiss-celebrates-1000-spacecraft-deployments-of-xtratum>
 - [177] « Safety Runs on XtratuM/NG ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://storage.googleapis.com/b2match-as-1/QFAPVNvf2nGRmR9JyyQhuYuZ>
 - [178] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, et A. Crespo, « XtratuM hypervisor redesign for LEON4 multicore processor », *SIGBED Rev.*, vol. 11, n° 2, p. 27-31, sept. 2014, doi: [10.1145/2668138.2668142](https://doi.org/10.1145/2668138.2668142).
 - [179] 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.
 - [180] 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>
 - [181] 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
 - [182] « LithOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/lithos/>
 - [183] « XtratuM homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/xtratum/>
 - [184] 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.
 - [185] « xng-rs ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://github.com/aeronautical-informatics/xng-rs>

-
- [186] fentISS, « SKE - XtratuM Emulation Environment ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/ske/>
- [187] ESA - European Space Agency, « XtratuM Hypervisor for LEON4 Volume 2: XtratuM User Manual ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <http://microelectronics.esa.int/gr740/xm-4-usermanual-047d.pdf>
- [188] fentISS, « The XtratuM Hypervisor as Key Enabling Technology for New Space: A Success Story ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: https://archives.ecrts.org/fileadmin/WebsitesArchiv/ecrts2021/wp-content/uploads/2021/07/20210709_TheXMHypAsKITyforNewSpaceSuccessStory_final.pdf
- [189] fentISS, « XtratuM Hypervisor ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: <https://www.fentiss.com/xtratum/>
- [190] « XtratuM Hypervisor for LEON4 Volume 2: XtratuM User Manual ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: <http://microelectronics.esa.int/gr740/xm-4-usermanual-047d.pdf>
- [191] M. Masmano, I. Ripoll, A. Crespo, et J. Real, « XtratuM: An Open Source Hypervisor for TSP Embedded Systems in Aerospace ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/wp-content/uploads/2022/11/XM_for_TSP_systems_based_in_aerospace.pdf
- [192] M. Masmano, I. Ripoll, A. Crespo, J. Real, et J. Alonso, « XtratuM: a Hypervisor for Safety Critical Embedded Systems ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/wp-content/uploads/2022/11/XM_for_Safety_Critical_EMBEDDED_Systems.pdf
- [193] M. Masmano, I. Ripoll, A. Crespo, et S. Peiro, « XtratuM for LEON3: an Open Source Hypervisor for High Integrity Systems ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: <https://hal.science/hal-02267841/document>
- [194] « XtratuM Library - hypervisor.c ». Consulté le: 7 décembre 2025. [En ligne]. Disponible sur: <https://github.com/lfd/XtratuM/blob/master/user/libxm/common/hypervisor.c>
- [195] « XtratuM code source ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: <https://github.com/lfd/XtratuM>
- [196] « Learn more about the hypervisor used in De-RISC Project: XtratuM Next Generation (XNG) by fentISS ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://derisc-project.eu/xtratum-hypervisor-by-fentiss/>
- [197] « Leveraging virtualization with XtratuM/NG in METASAT ». Consulté le: 8 décembre 2025. [En ligne]. Disponible sur: <https://metasat-project.eu/leveraging-virtualization-with-xtratum-ng-in-metasat/>
- [198] « SLOCCount website ». Consulté le: 3 juillet 2025. [En ligne]. Disponible sur: <https://dwheeler.com/sloccount/>