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

2025-11-17

Table des matières

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Pourquoi utiliser un système d'exploitation?

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

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

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

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

1.3 Criticité et temps réel

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

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

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

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

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

1.4 Organisation de l'étude

L'étude est organisée de la façon suivante:

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

1.5 Critères de comparaison

Au travers de cette étude, les systèmes d'exploitation ont été étudiés et comparés suivant les critères détaillés ci-dessous. Il est noté que certains critères n'étaient pas pertinent pour l'ensemble des systèmes, auquel cas la section corresponde justifie son élision.

1.5.1 Type de systèmes d'exploitation

Dans ce document, 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 directement 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*, *Windows* et *macOS*.
- Les *hyperviseurs*[4] sont des systèmes d'exploitation dédiés à la virtualisation, c'est-à-dire à l'exécution d'OS invités au-dessus d'une couche logicielle. On les retrouve fréquemment sur des serveurs exécutant simultanément plusieurs OS invités. Parmi les systèmes les plus utilisés, on peut citer *VMware vSphere*, *Hyper-V*, *KVM*, *VirtualBox* ou encore QEMU.
- Les *systèmes d'exploitation temps-réels* (*RTOS* pour *Real-Time Operating System*) sont des systèmes d'exploitation donnant des garanties sur le temps d'exécution.
- 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*, c'est-à-dire directement sur la couche matérielle.

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 utilisé 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,
- Certain système 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,

[4] On parle également de *Virtual Machine Monitor* abrégé *VMM*.

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

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

Tableau 2. – Architectures considérées dans l'étude.

Aparté:

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

1.5.3 Support multi-processeur

Au début du XXI^{e} siècle, les architectures multi-processeurs se sont imposées dans l'ensemble des secteurs de l'informatique. Jusqu'au milieu des années 2000, la croissance exponentielle de la puissance de calcul était principalement soutenue par l'augmentation rapide des fréquences d'horloges des monoprocesseurs. Cette stratégie a cependant rencontré des limites physiques (mur thermique, courants de fuite, ...). L'industrie des microprocesseurs s'est alors tournée vers le parallélisme offert par les architectures multi-processeurs pour maintenir la progression de la puissance de calcul.

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

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

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

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

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

Caractéristique \ Architecture	*SMP*	*AMP*
Nature des cœurs	Le plus souvent homogènes	Le plus souvent hétérogènes
Gestion logicielle	Un unique système d'exploitation gère tous les cœurs et partage dynamique les tâches entre eux	Instance indépendante exécutée sur chaque cœur
Partage de Ressources	Mémoire principale, périphériques et caches partagés	Ressources partitionnées entre les cœurs
Objectif & Performance	Excellent débit	• Déterminisme accru • Meilleure isolation des tâches
Domaines d'application	• Ordinateurs personnels • Serveurs	Systèmes embarqués critiques
Prix	Très bon marché	Élevé

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

1.5.4 Partitionnement

Le partitionnement des ressources est un mécanisme fondamental des systèmes d'exploitation modernes. Il vise à permettre l'exécution simultanée de plusieurs tâche sur une même machine physique. On parle alors de système *multi-tâche*. En effet, les ressources matérielles étant le plus souvent insuffisantes pour exécuter chaque tâche sur sa propre machine, il est nécessaire de partager ces ressources entre les programmes. Dans ce contexte, l'isolation des tâches en cours d'exécution devient nécessaire afin de s'assurer qu'un programme malveillant ou défec-

tueux ne puisse compromettre l'ensemble du système. Ce partage peut être opéré à plusieurs niveaux, notamment:

- Au niveau des *processus* s'exécutant sur un système d'exploitation.
- Au niveau des OS invités s'exécutant sur un hyperviseur.

Dans cette section, nous examinons ce partitionnement pour deux ressources: la mémoire principale et le processeur.

1.5.5 Partitionnement spatial

Le partitionnement en mémoire vise à partager la mémoire principale entre plusieurs tâches en cours d'exécution. Ce partage est crucial car il permet de conserver en mémoire tout ou une partie des données de plusieurs processus, améliorant les performances du système.

Dans le cas des processus, la méthode la plus courante pour gérer ce partage s'appuie sur la *mémoire virtuelle*. Au lieu de faire référence à des adresses physiques directement, les instructions utilisent des adresses virtuelles qui sont traduites à la volée vers des adresses physiques par une puce dédiée: le *MMU* (*Memory Management Unit*). Ainsi, chaque processus a l'illusion de disposer de la totalité de la mémoire principale.

Lorsqu'une instruction tente d'accéder à une adresse virtuelle qui ne figure pas dans le table du processus en cours d'exécution, un *page fault* est émis sous la forme d'une interruption matérielle et permet au système d'exploitation de réagir en conséquence.

Un autre aspect important est la *pagination*. L'espace d'adressage est subdivisée en des pages de tailles fixes. Cela permet de n'avoir qu'une portion des données d'un processus en mémoire et de charger les pages manquantes à la demande.

1.5.6 Partitionnement temporel

Les systèmes d'exploitation moderne permettent l'exécution de programmes dans un contexte multi-tâches. Cette exécution peut être *concurrentielle* ou *parallèle*. Dans cette section, une tâche peut aussi bien désigner un programme, un *thread* ou même un OS invité.

L'*ordonnanceur de tâche* (*scheduler*) est un des composants principales d'un système d'exploitation. Son rôle est de décider quelle tâche doit être exécuté à un instant donné sur le CPU. Un *scheduler* peut poursuivre des objectifs différents et parfois incompatibles. Il peut notamment chercher à:

- Maximiser la quantité de travail accomplie par unité de temps. En anglais, on parle souvent du *throughtput*.
- Minimiser la *latence* (*latency*), c'est-à-dire
- Être *équitable* (*fairness*) en donnant des tranches de temps en proportion de la priorité et de la charge de travail d'une tâche.

L'ordonnanceur de tâches d'un *RTOS* cherche à maximiser le nombre de tâches pouvant respecter leurs *deadlines* simultanément. À cette fin, la *latence*.

L'ordonnanceur de tâches d'un *GPOS* cherche le plus souvent à maximiser la quantité de travail accomplie par unité de temps[5]

Un *cœur* est un ensemble de circuits intégrés capable d'exécuter des instructions de façon autonome. Un microprocesseur embarquant plusieurs cœurs est qualifié de *processeur multi-cœur*.

[5]Cette quantité est souvent désigner par *throughput* en anglais.

De nos jours, certains fabricants comme Intel ou ARM proposent des processeurs où les cœurs ne sont plus identiques. L'intérêt principal de ces architectures hybrides est de faire un compromis entre la puissance de calcul et l'efficacité énergétique. Ainsi on y trouve généralement deux types de cœurs:

- Les cœurs performances: ces unités sont dédiées aux tâches lourdes mais sont gourmandes en énergie. On peut citer les cœurs *P-cores* chez Intel et *big* chez ARM.
- Les cœurs économes: moins performantes que les cœurs de la première catégorie mais consomment nettement moins d'énergie et dissipent moins de chaleur. On peut citer les cœurs *E-cores* chez Intel et *LITTLE* chez ARM).

1.5.6.1 Déterminisme

Comme nous l'avons expliqué dans la sous-section 1.3, les logiciels, et en particulier le système d'exploitation, d'un système critique doivent fournir des garanties sur le temps d'exécution de leurs routines. En informatique usuelle, le temps d'exécution d'un programme ne fait généralement pas parti de sa correction[6]. Ce n'est plus le cas dans un système temps réel où répondre après un délai trop long conduit à un résultat erroné. On souhaite donc que les calculs soient fait suffisamment vite en toute circonstance, tandis qu'en informatique usuelle on cherche généralement à ce que les calculs soient fait le plus vite possible en moyenne.

Afin d'offrir ces garanties temps réel, le système d'exploitation doit être aussi déterministe que possible. Ce déterminisme permet en pratique d'estimer le temps d'exécution de ses routines dans le pire cas[7]. Le déterminisme est souvent assuré par le caractère préemptible du noyau[8] et des éventuelles autres tâches. En effet, lorsqu'une tâche critique doit commencer son exécution aussi vite que possible, il ne faut pas que celle-ci doive attendre la fin de l'exécution d'une longue routine du noyau ou la fin de la tranche de temps d'une tâche de plus faible priorité. La latence du système d'exploitation est donc une mesure importante pour assurer le respect des échéances.

1.6 Corruption de la mémoire

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

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

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

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

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

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

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 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.

1.8 Masquage des interruptions

La gestion des interruptions est l'une des tâches primordiales d'un système d'exploitation. Une *interruption* est un événement matériel qui altère le flot d'exécution normal d'un programme. Au niveau matériel, elles se manifestent par des signaux électriques pouvant être émis à tout moment par:

- Un périphérique (clavier, disque, carte PCI, …).
- Le CPU lui-même.
- Dans une architecture multi-cœur, des signaux sont émis entre les cœurs.

Lorsqu'une interruption est déclenchée, l'exécution courante est suspendue. Dans ce cas, un gestionnaire d'interruption prend le relais. Il est important de noter qu'une interruption peut subvenir à n'importe quel moment, y compris pendant l'exécution d'un gestionnaire d'interruption. Cela pose plusieurs difficultés:

- Il n'est pas toujours possible d'interrompre l'exécution d'une routine, notamment dans une section critique. C'est un scénario courant dans un noyau.
- Dans un programme temps réel et suivant le niveau d'exigence, la latence induite par ces interruptions doit ou non être prise en compte dans les contraintes temporelles.

1.8.1 Interruptions programmables

Les architectures modernes permettent généralement la programmation des interruptions grâce à des puces dédiées réparties entre la carte mère et le CPU:

- Sur les architectures Intel et AMD, cette tâche est répartie entre la puce *I/O APIC*[9] qui gère les interruptions émises par les périphériques et des circuits intégrés dans chaque cœur appelés *Local APIC* qui gèrent les interruptions entre les cœurs.
- Sur les architectures ARM, cette tâche est dévolue au *GIC* (*Generic Interrupt Controller*).

L'émetteur de l'interruption envoie une requête d'interruption (*IRQ* pour *Interrupt ReQuest*) à l'une de ces puces qui décide ensuite d'envoyer ou non l'interruption au destinataire (TODO: vérifier).

1.8.2 Masquage des interruptions

Une solution pour gérer les interruptions est de *masquer*, c'est-à-dire bloquer, temporairement certaines d'entre elles.

Les architecture moderne embarque généralement plusieurs puces dédiées à la gestion des requêtes d'interruption (*IRQ* pour *Interrupt ReQuest*). Par exemple, sur les architectures Intel et AMD, cette tâche est accomplie par le sous-système *APIC* (*Advanced Programmable Interruption Controller*). Sur les architectures ARM, elle est dévolue au *GIC* (*Generic Interrupt Controller*).

Les processeurs multi-cœur disposent aussi de puce *APIC* par cœur, permettant la gestion des interruptions entre cœurs (*Inter-Processor Interrupt* IPI).

Les contrôleurs d'interruption permettent également de mettre des niveaux de priorité sur les interruptions.

1.9 Support de *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 de temps 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 lorsque cela était possible.

1.10 Support de langages de programmation en baremetal

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

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

[9] *APIC* est un abréviation pour *Advanced Programmable Interrupt Controller*

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

1.11 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.

1.12 Maintenabilité

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

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

2 Linux

Linux en bref

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

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'interfaces graphiques via le projet *XFree86*. Les distributions *GNU/Linux Red Hat* et *SUSE* publient leur première version majeure en 1994 également. À partir de 1995 avec la version `1.1.85`, le noyau passe d'une architecture *monolithique* à une approche modulaire, permettant le chargement à chaud de modules. La version `2.0` publiée en 1996 propose un support pour les architecture *SMP*. En 2007, la version `2.6.20` intègre un hyperviseur baptisé *KVM*. En 2024, la totalité des patchs du projet *PREEMPT_RT* sont intégrés dans le noyau, faisant de *Linux* un *RTOS*.

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

2.1 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 [6]. 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*.

2.2 Support multi-processeur

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

2.2.1 Architectures *SMP*

Le support pour les architectures *SMP* est ajoutée dans *Linux 2.0* en 1996. Toutefois les premières versions du noyau supportant les architectures *SMP* avaient recours à un verrou global appelé *BKL (Big Kernel Lock)*. Ce dernier assurait que les sections critiques du noyau ne pouvaient pas s'exécuter en parallèle. Cette technique avait l'avantage d'être simple à mettre en place mais conduisait à des performances médiocres, notamment lorsque le système avait de nombreux cœurs. Le *BKL* a été progressivement remplacé par des mécanismes de synchronisation plus fins comme les *spin locks* et les *mutexes*, puis totalement supprimé à partir de la version *2.6.39*. Le noyau propose aussi depuis la version 2.5 des structures de données synchronisées de type *RCU* (*Read-Copy-Update*) [7] 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 en cours d'exécution a été compilé avec ce support, tapez la commande suivante:

```
$ zcat /proc/config.gz | grep CONFIG_SMP
```

Aparté: `CONFIG_SMP` obligatoire

Le support *SMP* ne sera plus optionnel à partir de *Linux 6.17* pour la majorité des architectures [8]. 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.

2.2.2 Architectures *AMP*

Depuis la branche `3.x`, le noyau *Linux* offre un support pour les processeurs distants via les sous-systèmes `remoteproc` [9] et `RPMsg` [10]. Vous pouvez vérifier que votre noyau est compilé avec le support pour ces systèmes via respectivement les commandes:

```
$ zcat /proc/config.gz | grep CONFIG_REMOTEPROC
$ zcat /proc/config.gz | grep CONFIG_RPMSG
```

Le cas d'usage typique est l'exécution d'un *RTOS* sur un processeur secondaire dans un système embarqué hétérogène sous la forme d'un *MPSoC*. Avant l'apparition de `remoteproc`, le contrôle des processeurs secondaires se faisait via des *API (Applicaton Programming Interface)* 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*.

2.3 Partitionnement

2.3.1 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*.

2.3.2 Partitionnement spatial

2.3.3 Partitionnement temporel

2.3.3.1 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 actuel *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
```

2.3.4 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.

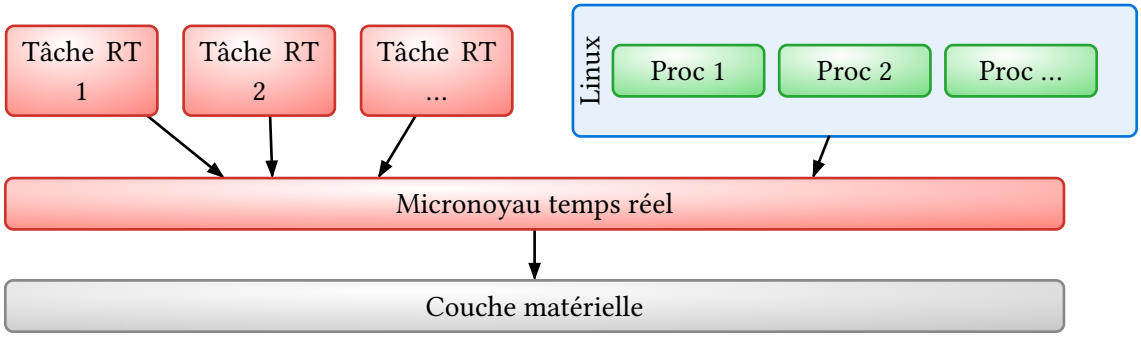

Fig. 1. – Architecture *cokernel*.

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*.

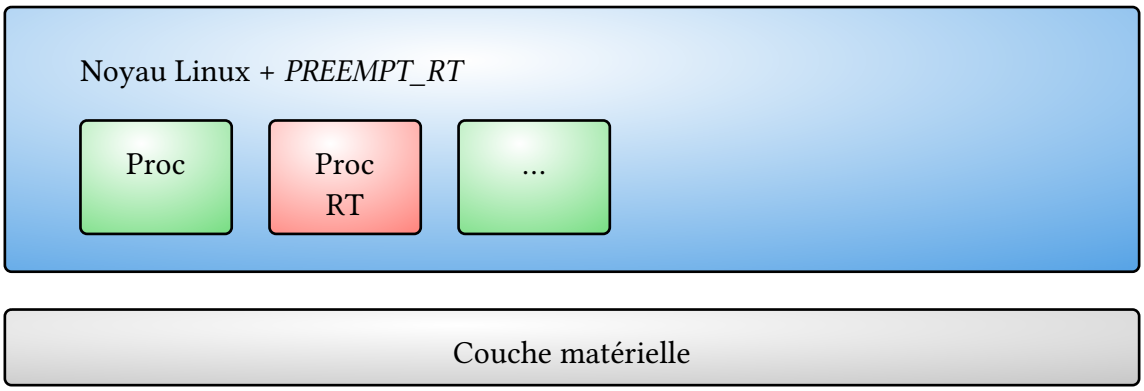

Fig. 2. – Architecture de *Linux* avec *PREEMPT_RT*

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. Pour vérifier que votre noyau en cours d'exécution a été compilé avec ce support, vous pouvez tapez la commande:

```
$ zcat /proc/config.gz | grep PREEMPT_RT
```

Certaines distributions comme *Fedora* ou *Ubuntu* proposent également des noyaux alternatifs avec cette option activée, rendant l'installation de *PREEMPT_RT* plus simple.

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

Les modifications apportées au noyau par le projet *PREEMPT_RT* sont trop complexes et techniques pour en faire ici une révue détaillée. Toutefois, il est intéressant de comprendre certains de leurs aspects afin de cerner les forces et les limites du temps réel dans ce noyau. Plus d'informations sont disponibles dans la documentation officielle [11].

2.3.4.1 Mutex temps réels

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

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

Plus de détails sur ces *mutex* temps réel sont fournis dans la documentation officielle [12], [13].

2.3.4.2 Gestionnaires d'interruption

Lors de l'exécution d'un *ISR (Interrupt Service Routine)*, 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 [14]. 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*.

2.3.4.3 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*.

2.3.4.4 RCU

2.4 KVM

Depuis la version `2.6.20` publiée 2007, *Linux* intègre un hyperviseur baptisé *KVM* (*Kernel-based Virtual Machine*) [15]. Il s'agit d'un hyperviseur de type 1 assisté par le matériel. Il offre également un support pour la paravirtualisation.

2.4.1 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 [16].

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

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

Les principaux contrôleurs sont:

- `cpu`: contrôle l'utilisation du CPU,
- `memory`: contrôle l'utilisation de la mémoire vive et de la mémoire d'échange,
- `io`: contrôle les opérations d'entrée/sortie sur les périphériques de stockage,
- `pids`: limite le nombre de processus et de threads,
- `cpuset`: affecte un groupe de processus à des cœurs CPU spécifiques,
- `hugetlb`: contrôle l'utilisation des *huge pages*.

Plus d'informations sur les *cgroups* sont disponibles dans la documentation officielle [17].

2.4.1.1 Exemple d'utilisation

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

Supposons que nous souhaitions limiter la consommation de mémoire d'un processus à 5 Mio. On commence par créer deux[10] *cgroups* `foo` et `bar`:

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

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

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

int main(void) {
  size_t sz = 0;

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

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

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

```

Liste 1. – `limited.c`

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

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

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

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

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

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

Notez que pour obtenir l'erreur escomptée, il faut prendre garde à deux aspects:

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

- 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`.

2.4.2 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 [18]. D'autre part cet appel n'offre pas le même degré d'isolation que les *namespaces* abordés dans la section 2.4.3.

2.4.3 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.

2.4.3.1 Exemple d'utilisation avec `systemd`

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

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

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

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

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

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

  closedir(dir);
  return EXIT_SUCCESS;
}

```

Liste 2. – `alone.c`

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

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

On obtient alors la sortie suivante:

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

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

2.4.4 *Capabilities*

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

à ces vérifications en se basant sur des identifiants associés au processus[11]. Par exemple la commande suivante:

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

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

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

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

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

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

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

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

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

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

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

2.4.4.1 SetUID

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

2.5 Corruption de la mémoire

Le noyau *Linux* intègre un sous-système nommé *EDAC* (*Error Detection and Correction*) [19] 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 [20] via *sysfs*[12] pour les interfaces de pilotage du scrubbing décrites dans le à l'exception de l'interface *ARS* qui utilise son propre pilote.

2.6 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 mécanismes de *Control-Flow Integrity* (CFI) constituent une famille de défenses contre ces attaques [21]. Le CFI vise 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.

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.

2.7 Écosystème

Linux dispose d'un écosystème riche et mature d'outils de monitoring et d'observabilité [22], [23]. 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* [24]: Moniteurs système interactifs affichant l'utilisation du CPU, de la mémoire et des processus en temps réel.

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

- *netdata* [25]: 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*) [26] : 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* [27]: Permet l'instrumentation dynamique du noyau pour l'analyse approfondie du comportement système.
- *Prometheus/Grafana* [28], [29] : Solutions d'observabilité distribuée largement adoptées pour le monitoring de systèmes critiques.
- *strace/ptrace*: .
- *perf* [30]: 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* [31]: 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* [32]: *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.

2.7.1 Exemple de profilage

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

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

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

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

    return EXIT_SUCCESS;
  }

```

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

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

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

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

        116.46 msec task-clock:u              #    0.991 CPUs utilized
             0      context-switches:u        #    0.000 /sec
             0      cpu-migrations:u          #    0.000 /sec
         1,028      page-faults:u             #    8.827 K/sec
   270,371,076      cycles:u                  #    2.322 GHz
 1,100,144,340      instructions:u            #    4.07  insn per cycle
   100,029,819      branches:u                #  858.891 M/sec
         2,352      branch-misses:u           #    0.00% of all branches
                  TopdownL1                 #  25.1 %  tma_backend_bound
                                            #   1.2 %  tma_bad_speculation
                                            #   0.2 %  tma_frontend_bound
                                            #  73.6 %  tma_retiring

   0.117552543 seconds time elapsed

   0.113162000 seconds user
   0.003969000 seconds sys
```

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

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

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

   1.975546187 seconds time elapsed

   1.968594000 seconds user
   0.001981000 seconds sys
```

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

2.8 Watchdog

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

2.8.1 API bas niveau

Linux offre une *API* unifiée pour interagir avec les *watchdogs* matériels directement dans l'espace utilisateur [33]. Cette communication se fait via un pseudo-périphérique `/dev/watchdog`. À l'ouverture ce périphérique, le *watchdog* s'active et attend d'être réinitialisé dans un certain délai de réponse. Une façon simple de le réinitialiser est d'écrire des données quelconques dans le périphérique `/dev/watchdog`. Quant au délai de réponse, il est configurable via l'appel système `ioctl`. Lorsque le périphérique est fermé, le *watchdog* est désactivé. La Liste 4 contient un exemple simple d'utilisation de *watchdog*.

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

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

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

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

  close(fd);
  return EXIT_SUCCESS;

failed:
  close(fd);
  return EXIT_FAILURE;
}

```

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

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

2.8.2 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*.

2.9 Masquage des interruptions

2.10 Licences

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

2.11 Temps de démarrage

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

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

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

Dans l'article [34], 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 [35], les auteurs comparent et optimisent différents *init systems* à la fois dans un environnement émulé avec *QEMU* et dans une distribution *GNU/Linux* dédiée à l'embarqué. Leur conclusion est qu'une réduction substantielle du temps démarrage de l'*espace utilisateur* est possible via leurs méthodes d'optimisation et que le choix du *init system* est déterminant mais dépendant de l'environnement d'exécution.

(MOVE) Le project Yocto [36] est un projet libre offrant la possibilité de créer sa distribution *Linux* dédiée à l'embarqué.

2.11.1 Profilage de `systemd`

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

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

Par exemple, la commande suivante:

```
$ systemd-analyze time
```

produit une sortie de la forme:

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

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

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

3 MirageOS

MirageOS en bref

- **Type** : LibOS
- **Langage** : OCaml (99%)
- **Architectures** : x86-64, ARM v8, PowerPC[13]
- **Usage principal** : Cloud computing, applications réseau, systèmes embarqués, spatial
- **Points forts** : Sécurité renforcée (surface d'attaque réduite, langage sûr), taille minimale, temps de démarrage rapide, modularité
- **Limitations** : Portabilité limitée sans hyperviseur, débogage complexe, pas d'interface POSIX
- **Licences** : ISC (majoritaire) + LGPLv2 (certaines parties)
- **Projet notable** : SpaceOS (déployé dans l'espace en 2025)

Au tournant des années 2010, l'usage de la virtualisation révolutionne le déploiement des services, permettant de réduire les coûts et d'externaliser une partie de la maintenance via le concept de *cloud computing*. À cette époque, la majorité des *VMs* exécutent un service dans un *GPOS* complet. Cette approche présente l'avantage de circonscrire au système d'exploitation les modifications requises 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.

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)* volumineuse) et les sources de bugs potentiels. Cela est d'autant plus vrai que ces *GPOS* sont souvent écrits dans un langage de programmation[14] 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 [37]. 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* [38] et bon nombre de ces contributeurs ont également contribué au projet *Xen*.

MirageOS adopte une approche de type *LibOS*. Au lieu de fournir un environnement d'exécution pour les services, *MirageOS* se présentent sous la forme d'une collection de bibliothèques modulaires. 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 surface d'attaque à la fois par la réduction de le taille du code source et l'utilisation d'un langage de programmation sûr.
- Une amélioration des performances et notamment du temps de démarrage.

[13] Support limité à l.

[14] La vaste majorité est écrit en langage C, un langage n'offrant pratiquement aucune garantie mémoire et à la sémantique complexe sur les architectures *SMP*.

- Une réduction de la taille des exécutables produits.
- Un profilage simplifié par la suppression d'une couche logicielle.

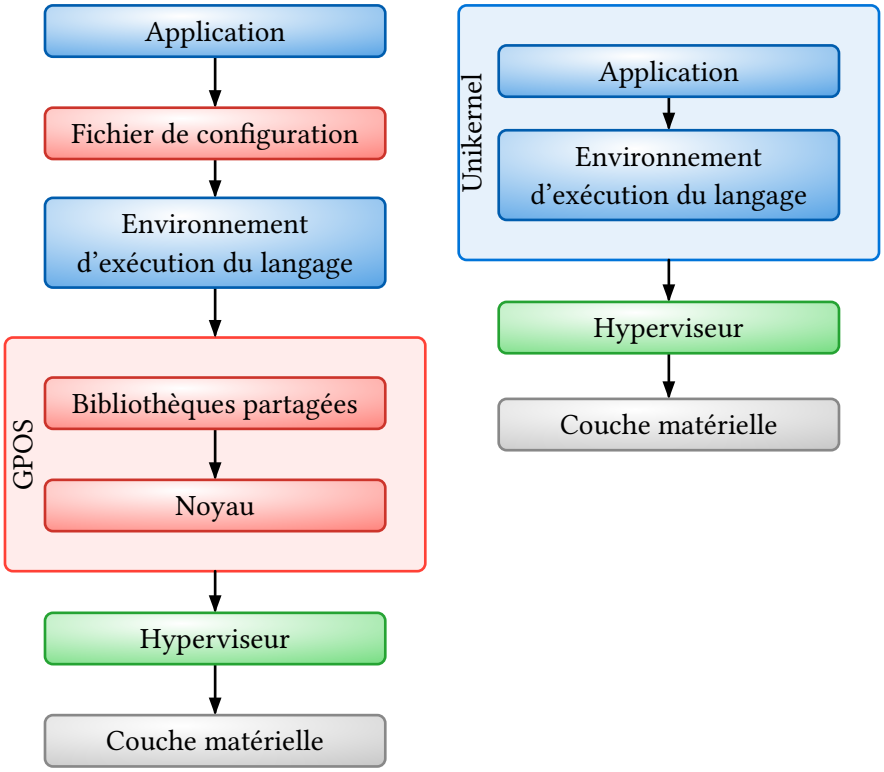

Fig. 3. – Comparaison entre l'approche *GPOS* et l'approche *unikernel*.

3.0.1 SpaceOS

SpaceOS est un système d'exploitation basé sur *MirageOS* développé par *Tarides* pour les applications spatiales et satellitaires [39], [40]. 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:
- 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.

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 que celle-ci soit une cible de compilation du compilateur OCaml. Le compilateur pour OCaml 4 supporte les architectures suivantes: *x86-32*, *x86-64*, *ARM v7*, *ARM v8*, *PowerPC*, *SPARC* et *MIPS*. Toutefois le support[15] des architectures 32-bits a été supprimé à partir d'OCaml 5.

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 l'hyperviseur supporte les architectures citées ci-dessus et que l'environnement d'exécution de *MirageOS* ait été porté dessus. Le projet *solo5* vise à fournir une couche d'abstraction logicielle entre l'environnement d'exécution de *MirageOS* et les différentes *API* d'hyperviseurs et de *GPOS*. Il semble qu'à l'heure actuelle le projet *solo5* n'offre qu'un support pour des systèmes sur *x86-64*, *ARM v8* et *PowerPC*.

Nous considérons donc ces architectures comme étant les seules bénéficiant d'un support officiel par le projet *MirageOS*.

3.3 Support multi-processeur

Les *unikernels* générés avec *MirageOS* étant le plus 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 qu'offre ces processeurs. À ce titre, l'environnement d'exécution d'*OCaml* doit supporter le parallélisme.

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 que certain invariants internes étaient préservés, 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 offerte par le langage *OCaml*. Toutefois un service Web implémenté en *MirageOS* doit pouvoir servir plusieurs utilisateurs simultanément. Lorsque cette application passe la majorité du temps à attendre des entrées/sorties, la programmation asynchrone s'avère un choix judicieux. À cette fin, le projet *MirageOS* utilise une bibliothèque de *threads* coopératifs baptisé *Lwt* [41], [42]. Lorsque le parallélisme est vraiment nécessaire, par exemple si les services doivent effectuer des tâches lourdes en calcul, une solution est d'exécuter plusieurs instances du même *unikernel* et de les synchroniser via

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

les *IPC (Inter-Process Communication)* de l'hyperviseur. Cette solution a été mise en pratique sur l'hyperviseur *Xen*.

Aparté: Bibliothèque *Lwt*

La bibliothèque *Lwt* est une bibiothèque *OCaml* de *threads* coopératifs. Elle simplifie la programmation aynchrone 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.

À partir de la version 5, le *runtime OCaml* permet l'exécution en parallèle de code écrit en *OCaml* via le concept de *domaine* [43]. Un effort est en cours pour porter *MirageOS* sur *OCaml 5* [44], 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.

Aparté: Domaine

Les domaines sont un concept introduit en *OCaml 5* pour permettre l'exécution parallèle de code *OCaml*. Elle permet également l'écriture de programme asynchrone mais sans avoir recours au style monadique comme *Lwt*.

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* [45].
- L'option `unix` permet d'exécuter l'*unikernel* dans *KVM*. L'environnement requière
- 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 [46]. 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 le plus 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*. En particulier, 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* [47] de *MirageOS* pour communiquer entre deux *unikernels*.

3.5.1 Déterminisme

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, il faudra avoir recours à un hyperviseur temps réel et exécuter cette tâche dans une partition distincte.

3.6 Corruption de la mémoire

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

Dans le cas de *Xen* avec un noyau *Linux* dans le domaine *dom0*, il suffira d'utiliser les sous-systèmes décrits dans 2.5 et les *IPC* de *Xen* pour récupérer ces informations dans l'*unikernel*.

3.7 Perte du flux d'exécution

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 2.7. Le manuel *OCaml* contient un guide pour le profilage avec *perf* de programmes *OCaml* [48].

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

- *mirage-monitoring* [49]: Outil de monitoring pour les *unikernels* produits par *MirageOS*. Il supporte le *dashboard Telegraph* de *Grafana*.
- *memtrace* [50]: 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* [51]. Il existe une bibliothèque *MirageOS memtrace-mirage* [52] qui offre un support pour cet outil dans un *unikernel*.
- *memtrace_viewer* [51]: Outil d'exploration de traces produites par *memtrace*.
- *mirage-profile* [53]: 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 [54]. Le projet ne semble plus être maintenu.
- *mirage-trace-viewer* [55]: 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`[16]. Lorsqu'un client établit une connexion, `memtrace` est lancé jusqu'à ce que la connexion soit interrompue. La fonction `alloc` est exécutée de façon concurrentielle afin de produire un grand nombre d'allocations. L'exécution de l'*unikernel* se termine après 100 secondes.

```
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*

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

```
open Lwt.Infix

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

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

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


```

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

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

```
$ make build-memtrace
$ cd 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 > trace
```

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

```
$ make mentrace-view
```

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 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 11 à travers des *bindings* en OCaml. De tels *bindings* existent déjà dans le dossier `tools/ocaml/` du dépôt *Xen*.

3.10 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 [56], 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 [57], 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*.

Dans l'étude récente [58], les auteurs affirment qu'un *unikernel* de *MirageOS* peut démarrer en moins de 3ms en utilisant l'environnement d'exécution *solo5*.

En conclusion, le temps de démarrage de *MirageOS* peut être rendu très faible et négligeable face du démarrage de la partition elle-même.

3.11 Maintenabilité

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

Toutefois la totalité de l'unikernel ne provient pas de la compilation de codes *OCaml*. Il subsiste plusieurs parties en langage C et notamment:

- L'environnement d'exécution du langage OCaml est écrit en C. Cela inclut en particulier son ramasse-miette,
- Quelques bibliothèques en C comme *GMP* au travers de *Zarith*. Leur réécriture en OCaml est théorique possible mais nécessiterait un effort considérable en pratique,
- Les pilotes sont et doivent être écrit dans un langage bas niveau.

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. Cet objectif est atteint en minimisant la surface d'attaque et en utilisant un langage de programmation sûr.

3.13 Licences

Le code de *MirageOS* est publié sous la licence *ISC* avec certaines parties sous licence *LGPLv2*. L'utilisation d'une licence open-source permissive comme *ISC* est nécessaire car l'*unikernel*

produit par *MirageOS* est lié statiquement avec les bibliothèques. Grâce à cette licence, vous n'avez pas les contraintes des licences *GPL* lorsque vous distribuez le binaire de votre *unikernel*.

3.13.1 Traçage

Il existe des `hooks` dans le code de *MirageOS* qui permet un traçage de bout en bout. On peut utiliser un backend spécifique comme `mirageos-trace-viewer`. C'est un atout majeur en comparaison de *strace* qui ne permet que de tracer les appels systèmes.

4 PikeOS

PikeOS en bref

- **Type** : RTOS + Hyperviseur type 1 (inspiré du micronoyau L4)
- **Langage** : C
- **Architectures** : x86-64, ARM v7/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 micronoyau temps réel dédié à l'embarqué critique.

Depuis la fin des années 90, l'entreprise *SYSGO* développait son propre micronoyau baptisé *P4* et inspiré du noyau *L4* de Jochen Liedtke [59]. 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 afin d'être utilisé dans des systèmes 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 concernées 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 publiée en 2005.

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*.

4.1 Architectures supportées

PikeOS supporte les architectures suivantes: *x86-64*, *ARM v7*, *ARM v8*, *PowerPC*, *RISC-V* et *SPARC*.

Le support pour l'architecture *ARM* existe depuis 2006. Quant à son hyperviseur, il propose un support pour la virtualisation matérielle sur les architectures *ARM v7* [60] et *x86-64* [61].

Le support matériel se fait via des *BSP*. En particulier, *PikeOS* supporte les architectures *SPARC LEON3* et *LEON4*.

SYSGO propose également le développement de nouveaux *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 2.2.1). 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 [62].

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 [63].

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 a été conçu pour offrir de solide garantie quant au partitionnement en temps et en espace. Sa conception est inspirée de l'*ARINC 653*, une norme avionique pour les systèmes temps réel. Toutefois *PikeOS* n'est pas conforme *ARINC 653*.

Son hyperviseur permet à la fois la paravirtualisation et la virtualisation de assistée par le matériel.
- *HwVirt* désigne la virtualisation assistée par le matériel.
- *Pv-virt* désigne la paravirtualisation.

4.3.1 Partitionnement en espace

4.3.2 Partitionnement en temps

PikeOS utilise un ordonnanceur hybride breveté baptisé *Adaptive Time-Partitioning Scheduler* [63]. Ce dernier est un ordonnanceur à double niveau. Le premier niveau est chargé de l'isolation temporelle stricte. Il permet de distribuer statiquement des tranches de temps *CPU* entre les partitions de *PikeOS*. Le second niveau est un ordonnanceur par priorité.

4.3.3 Déterminisme

4.3.4 OS invités supportés

L'hyperviseur de *PikeOS* supporte les OS invités[17] suivants:
- *ELinOS* est supporté par *PikeOS*. Il s'agit d'une distribution *Linux* pour l'embarqué temps réel développé par *SYSGO* [64]. 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 [65]. 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* [66].

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

4.4 Corruption de la mémoire

4.5 Outils

Le noyau *PikeOS* étant propriétaire, son écosystème est centré autour de logiciels développés par *SYSGO* ou certains de ses partenaires [67]. Nous avons pu identifier trois logiciels *CODEO*, *RVS* et *TRACE32*.

4.5.1 *CODEO*

La société *SYSGO* propose un *IDE (Integrated Development Environment)* baptisé *CODEO* [68], [69] 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* [70].

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 [71], [72].

4.5.3 *TRACE32*

L'outil *TRACE32* développé par l'entreprise *Lauterbach* comprend un débogueur et un traceur [73]. 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 Support de *watchdog*

4.7 Support de langages de programmation en baremetal

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*.

- 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 [74], [75]. 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 [75].
- Le langage *Rust* est supporté [76].
- Le langage *SCADE* est supporté via un partenariat avec l'entreprise *Ansys* [77].

Nous n'avons pas trouvé d'information cernant *OCaml* sur *PikeOS*.

4.8 Temps de démarrage

4.9 Qualifications et certifications

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

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

Normes:

- Critères communs (quel niveau?)
- SAR

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

4.10 Licences

PikeOS est un logiciel propriétaire aux sources fermées. L'entreprise *SYSGO* ne semble pas communiquer sur ses licences ou sa politique tarifaire. Les modalités et les coûts des licences sont négociés avec *SYSGO* en fonction des besoins du projet.

4.11 Draft

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

5 ProvenVisor

ProvenVisor en bref

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

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

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

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

ProvenCore est un micronoyau qui cherche à la fois à minimiser la taille du code et la surface d'attaque (les deux allant souvent de pair). Il propose des containeurs sécurisé avec la possibilité de communiquer de façon sécuriser entre eux. Il a fait l'objet d'une vérification formelle [78].

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

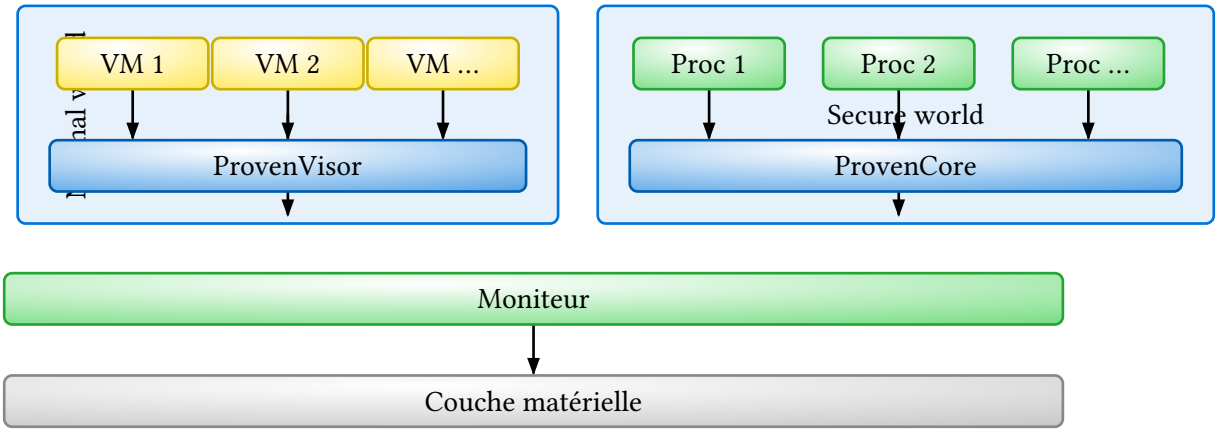

Fig. 4. – Architecture de *ProvenVisor*.

5.1 Architectures supportées

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

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

5.2 Certifications

- Permet la certification critères communs EAL5

5.3 Licences

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 Appications Research Corporaton*) sous contrat de l'*U.S. Army Missile Command*. Cette dernière voulait un système d'exploitation temps-réel basé sur des normes libres et exempt de redevances [79]. À cette époque, le système est destiné à un usage militaire, en particulier dans des missiles[18]. 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'adapteur fonctionne en 3,3V, sans quoi vous détruirez votre *Raspberry*.

Un fichier *Docker* pour générer la chaîne de compilation *RTEMS* pour *Raspberry* est disponible dans le dossier `./rtems/dockers/`. Vous pouvez lancer sa génération avec la commande suivante:

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

ce qui prend environ une demie heure pour terminer. Finalement, notez que les images produites par cette chaîne de compilation nécessite un *bootloader*. Le plus simple est d'utiliser

[18] Le sigle *RTEMS* signifiait alors *Real-Time Executive for Missile Systems*.

le *bootloader* de *Raspberry OS lite* et de remplacer le fichier `/boot/kernel8.img` par l'image produire.

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

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

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 concentrons ici sur les architectures énumérées dans la sous-section 1.5.2. D'après [80], *RTEMS* supporte les familles d'architectures suivantes dans leur version 32bits et 64bits: *x86*, *ARM*, *PowerPC*, *MIPS*, *RISC-V*, *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* [81], 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 ordonnanceurs de tâches dédiés comme *EDF* (*Earliest Deadline First*) qui tient compte d'échéances via le mécanisme de *deadline* (voir la sous-section 6.4.5) et se comporte comme un ordonnanceur à priorité fixe en l'absence de *deadlines*.
- La possibilité de circonscrire une tâche à un sous-ensemble de *CPU* via un mécanisme d'affinité.
- Un support pour la migration de tâches.
- Des mécanismes de synchonisations fins via des protocoles de verrouillage comme *OMIP* et *MrsP*.

Aparté: Activation *SMP*

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

Le support *SMP* a fait l'objet de vérifications et de pré-qualifications comme nous le verrons dans la sous-section 6.9.

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+* [82].

6.4 Partitionnement

6.4.1 Partitionnement spatial

RTEMS n'offre pas beaucoup de garantie quant au partitionnement spatial.

6.4.2 Partitionnement temporel

Il est distribué avec quatre ordonnanceurs différents:

- *Deterministic Priority Scheduler* est un ordonnanceur préemptif basé sur des niveaux de priorités (jusqu'à 256 niveaux). Il offre de très bonnes performances mais il peut requérir trop de mémoire pour les configurations les plus minimaliste. Il existe une version de cet ordonnanceur pour les multi-processeur [83].
- *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 [83].
- *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* [83].
- *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. Il est également possible de rendre une tâche non préemptible.

6.4.3 Protocols de synchonisation

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

- *ICPP* (*Immediate Ceiling Priority Protocol*) pour les architectures monoprocesseur
- *PIP* (*Priority Inheritance Protocol*) pour les architectures monoprocesseur. La prorité d'une tâche est élevée au niveau de la plus haute priorité d'une tâche qui attend un verrou. C'est une approche similaire aux verrous *rt-mutex* de *Linux* vu en sous-section 2.3.4.
- *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*.

6.4.4 Rate Monotonic Manager

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

6.4.5 Déterminisme

En tant que *RTOS*, *RTEMS* a été conçu pour être déterministe. À cette fin, il dispose de plusieurs ordonnanceurs préemptifs couvrant différents cas d'usage. *RTEMS* offrent des ordonnanceurs pour monoprocesseur et pour multiprocesseur de type *SMP*. En *SMP*, tous les ordonnanceurs sont basées sur des priorités.

6.4.6 Draft

(TODO: vérifier)

RTEMS est un *RTOS* ce qui signifie qu'il a été conçu pour avoir un comportement déterministe. Il dispose d'un ordonnanceur préemptif basé sur des priorités. Une tâche préempte immédiatement une éventuelle autre tâche de plus faible priorité. Son ordonnanceur propose deux modes de fonctionnement. Le mode *RMS* (*Rate Monotonic Scheduling*) avec des priorités statiques basée sur des périodes. La tâche avec la période la plus courte est la plus prioritaire. L'autre mode est *EDF* (*Earliest Deadline First*). Cette fois c'est la tâche avec l'échéance la plus proche qui reçoit la plus haute priorité. Les temps de latence sont courts et connus.

Le premier scheduler est plutôt appelé un *Deterministic Priority Scheduler*. D'après la documentation il est plutôt approprié pour les architectures monoprocesseur.

6.5 Corruption de la mémoire

RTEMS ne semble pas fournir d'*API* unifié pour gérer le *scrubbing*. Le support est relatif au *BSP*. Il existe une *API* pour gérer le *scrubbing* dans le fichier `bsps/include/grlib/memscrub.h`. Ce dernier fait parti du *BSP* pour les microprocesseurs *LEON*. L'usage du *scrubbing* étant rendu nécessaire par le rayonnement inhérent aux missions spatiales, il y a fort à parier qu'un tel support est développé dans un *BSP* chaque fois que celui-ci est utilisé dans une telle mission.

6.6 Perte du flux d'exécution

6.7 Écosystème

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

- *profreport* [84]: 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 [85]. Il produit un rapport au format *XML* sur sa sortie standard.
- *rtrace* [86]: 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* [87]: Shell qui comprend de nombreuses commandes affichant des informations sur les tâches en cours d'exécution [88] et permet l'exploration de l'état mémoire [89].
- *CPU Usage Statistics* [90]: *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 *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 6.8.1.

6.8.1 Watchdog matériel avec un *Raspberry PI 4*

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

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

rtems_task Init(rtems_task_argument ignored) {
  raspberrypi_watchdog_init();

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

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

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

```

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

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

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

6.9 Qualifications & certifications

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

6.9.1 Kit de qualifications par l'ESA

L'*ESA* offre un kit de *qualification* pour des applications de *RTEMS* dans le domaine spatial [91]. Il est disponible pour les cartes *Cobham Gaisler GR712RC* (architecture *LEON3* double-cœur) et *GR740* (architecture *LEON4* quadri-cœur). Ce kit est disponible sous licence *Creative Common Attribution-ShareAlike 4.0*.

L'ESA (*European Space Agency*) offre un kit de *qualification* pour des applications de *RTEMS* dans le spatial [91] 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 [92], un model-checker. Edisoft a encore contribué sur cette version.

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

6.9.2 Model checking avec *Promela/SPIN*

La correction fonctionnelle de certaines parties de l'*API* de *RTEMS* ont été vérifié par *model checking* [92] grâce à un financement de l'*ESA*. Cette vérification concerne en particulier les primitives de synchronisation utilisées sur les architectures *SMP*.

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

Plus d'informations sont disponibles dans la documentation officielle [93].

6.10 Temps de démarrage

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

6.11 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*.

6.12 Licences

RTEMS est un logiciel libre distribué sous une multitude de licences libres et open-sources avec pour licence principale *BSD 2-Clause*. Le point commun de ces licences est qu'elles autorisent l'utilisateur a lié son programme avec le code source de *RTEMS* sans devoir redistribuer son propre code source [94].

6.13 Draft

- Il offre un support pour les architectures *SMP* et *AMP*.
- Il permet le cross-développement via d'autres OS: distributions GNU/Linux, Windows, BSD, Solaris, MacOS.
- Il est utilisé dans l'industrie spatiale, notamment chez les acteurs européens.
- ARINC 653 RTEMS
- Il existe un support commercial pour les entreprises européennes ou américaines et la communauté offre bien sûr un support gratuit sans garantie.

6.14 Partitionnement

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

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

RTEMS propose aussi des mécanismes de partitionnement en mémoire.

RTEMS propose un ordonnanceur en *cluster* (*clustered scheduling*). Cet ordonnanceur permet de partitionner l'ensemble des cœurs en des sous-ensembles appelés *cluster*. L'objectif de cette conception est de limiter les migrations de tâches entre cœur pour des raisons de performances[19] tout en préservant un bon contrôle sur la latence dans le pire cas (*worst-case latencies*). *RTEMS* propose également des primitives de synchronisation inter-clusters. En utilisant des clusters et des mécanismes de synchronisation adéquate, il est possible d'avoir des tâches temps réels et des tâches maximisant le *throughput*.

6.15 Profilage

Il y a un support pour profiler les goulots d'étranglement, notamment les verrous et le thread dispatch. Cela produit une sortie XML. [95].

6.16 Watchdog

6.16.1 Time Manager

Il est 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`.

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

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

rtems_id task_id;
rtems_id watchdog_id;

rtems_timer_service_routine dead() {
 }

 rtems_task task(rtems_task_argument ignored) {
  status = rtems_timer_fire_after(
   watchdog_id,
   5 * rtems_clock_get_ticks_per_second(),
   dead,
   NULL
  );
  directive_failed(status, "rtems_timer_fire_after");
 }

 rtems_task Init(rtems_task_argument ignored) {
  rtems_name task_name = rtems_build_name('O', 'C', 'A', 'M');
  puts("Creating task");
  status = rtems_task_create(
   task_name,
   1,
   RTEMS_MINIMUM_STACK_SIZE,
   RTEMS_NO_PREEMPT,
   RTEMS_GLOBAL,
   &task_id
  );
  directive_failed(status, "rtems_task_create");

  puts("Starting task");
  status = rtems_task_start(id, , 0);
  directive_failed(status, "rtems_task_start");

  rtems_name watchdog_name = rtems_build_name('W', 'A', 'T', 'C');
  status = rtems_timer_create(watchdog_name, &watchdog_id);
  directive_failed(status, "rtems_timer_create");

  exit(0);
 }

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


```

Liste 9. – Exemple d'interaction avec un *watchdog* logiciel.

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*[20], aujourd'hui connu sous le nom de Trustworthy Systems. C'est un noyau orienté sécurité dont l'un des premiers objectifs était d'être entièrement vérifié à l'aide de méthodes formelles. Grâce à ces efforts, il peut aujourd'hui être certifié avec le niveau le plus exigent dès Critères communs.

Le noyau *seL4* est un micronoyau de troisième génération. Il inclut un hyperviseur de type 1 et un *RTOS*. Sa conception a débuté en 2006 à l'institut de recherche *NICTA*[21]. 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: *ARM v7*, *ARM v8*, *x86-32*, *x86-64* et *RISC-V*.

[20] Acronyme pour *National Information and Communications Technology Australia.*
[21] Acronyme pour *National Information and Communications Technology Autralia*)

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

Plus d'informations sur le support des différentes plateformes sont disponibles sur leur site [96].

7.3 Support multi-processeur

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

7.4 Support *SMP*

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

Le micronoyau utilise un verrou global *BKL* de type *CLH* comme mécanisme de i synchronisation [97]. 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 [98].

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 [97]. Cette vérification est d'autant plus difficile que les architectures *SMP* modernes ont des modèles de mémoire faible [97], [99]. 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 [99].

7.5 Support *AMP*

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

Il y a également un support pour *OpenAMP*.

Quelques projets qui utilisent *seL4* sur des architectures *AMP*: [100].

7.6 Partitionnement

7.6.1 Partitionnement spatial

7.6.2 Partitionnement temporel

7.6.3 Déterminisme

En plus de disposer d'un ordonnanceur déterministe, *seL4* a fait l'objet d'une analyse de son *WCET* approfondi [101], [102]. Autrement dit, pour un certain nombre de configurations, on

dispose d'une estimation vérifiée du temps d'exécution de toutes les routines du micronoyau. Toutefois cette analyse a été faite sur ARMv6 et ARM ne fournit pas les informations nécessaires pour réitérer cette analyse sur les nouvelles architectures. Il semble qu'il y ait un projet pour une telle analyse sur *RISC-V*.

Le noyau tourne avec les interruptions matérielles désactivées. Ce choix simplifie grandement la conception et la vérification formelle.

Les appels systèmes sont généralement courts. Ceux qui sont trop longs sont préemptible à des points clés ajoutés par les développeurs.

Il permet de faire du temps réel tout en ayant l'isolation spatiale, ce qui n'est pas le cas de nombreux *RTOS* (vérifier pour *RTEMS*).

Il semblerait qu'être préemptible ne soit pas un prérequis pour offrir de faible latences!

7.7 Corruption de la mémoire

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

7.8 Perte du flux d'exécution

7.9 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* [103] 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*.

7.10 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*.

7.11 Langages de programmation en baremetal

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

Il y a une crate Rust [104].

Le *seL4 Microkit* offre également une *API* pour les langages *C* et *Rust* [103].

7.12 Temps de démarrage

7.13 Partitionnement

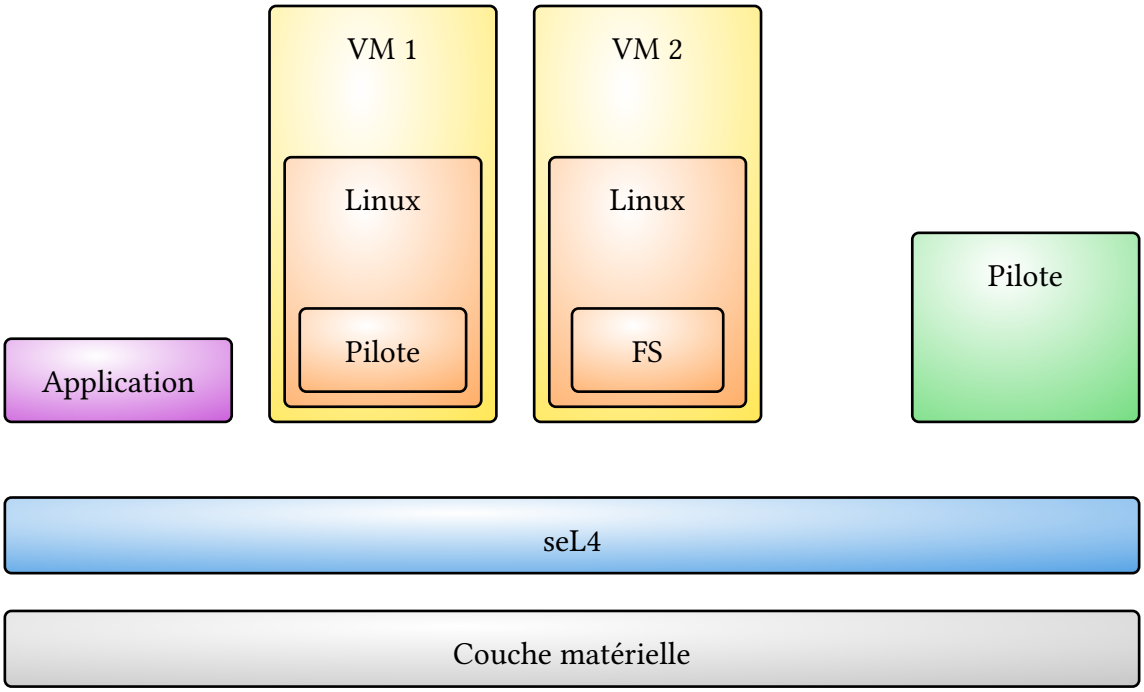

Fig. 5. – Architecture de l'hyperviseur *seL4*.

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

7.13.1 Capabilities

Les *capabilities* de *seL4* sont des jetons donnant à leur possesseur des droits d'accès à une ressource spécifique. Il existe trois types de *capabilities*:

- Les *capabilities* donnant accès à des objets du noyau comme le *thread control block*. Ces *capabilities* sont donnés à la tâche *root* durant l'initialisation du système.
- Les *capabilities* donnant accès à des ressources abstraitres comme *IRQControl*.
- Les *capabilities untyped*.

De façon plus concrète, les *capabilities* se présentent sous la forme de pointeurs constants contenant des informations supplémentaires pour encoder les droits d'accès. Disposer d'un tel pointeur est la seule façon d'accéder à la ressource qu'il pointe.

7.14 Vérifications formelles

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

7.15 Licences & brevets

Le noyau de `seL4` est un logiciel libre distribué principalement sous licence `GNU General Public License version 2 only (GPL-2.0)`. Le code utilisateur et les pilotes peuvent être distribués sous n'importe quelle licence [105].

7.16 draft

Il a l'avantage de supporté les partitions mixtes [106].

seL4 a fait l'objet d'une spécification et d'une vérification formelle à l'aide de l'assistant de preuve *Isabelle/HOL*. La correction[22] de l'implémentation a été démontrée pour plusieurs configurations et il a été également démontré que le code binaire est correct pour les architectures *ARM* et *RISC-V* [107]. Cette vérification formelle implique en particulier que seL4 est dépourvu de certaines erreurs de programmation classiques [108]. Il est notamment dépourvu de débordements de tampon, de déréférencements de pointeurs nuls, de fuites mémoire et de dépassements d'entier.

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

8 Xen

Xen en bref

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

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

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

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

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

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

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

8.1 Tutoriel

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

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

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

Liste 10. – Configuration d'une VM Alpine

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

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

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

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

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

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

Finalement vous pouvez lancer la VM avec la commande suivante:

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

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

8.2 Architectures supportées

Attention:

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

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

L'hyperviseur *Xen* supporte les architectures suivantes: *x86-32* à partir de la version P6[23], *x86-64*, *ARM v7* et *ARM v8*. *Xen* a également supporté l'architecture *IA64* jusqu'à la version 4.2. Il existe des travaux en cours pour supporter les architectures *PowerPC* et *RISC-V*. Un support préliminaire de ces architectures est disponibles depuis *Xen 4.20* [109]. Quant à la virtualisation assistée par le matériel de type *HVM (Hardware Virtual Machine)*, elle nécessite les extensions de virtualisation *Intel VT-X* ou *AMD-V* sur *x86* et les *Virtualization Extensions* sur *ARM* [110].

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

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

8.3 Support multi-processeur

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

8.4 Support *SMP*

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

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

8.5 Support *AMP*

Nous n'avons pas d'informations précises sur l'usage de *Xen* sur des plateformes *AMP*. Toutefois la documentation de *RTEMS* mentionne l'usage de *Xen* sur un *MPSoC Xilinix Zynq UltraScale+* [111].

8.6 Partitionnement

Xen propose trois types de partitions différentes:

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

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

- Les partitions de type *HVM* permettent la virtualisation assistée par le matériel. Elles nécessitent des extensions matérielles (voir la sous-section 8.2) mais aucune modification du système d'exploitation hôte[24]. Les performances sont généralement moindre que pour les partitions de type *PV*.
- Les partitions *PVH* cherchent à offrir le meilleur des deux types de partitions décrits ci-dessus. Certaines parties du systèmes (les entrées/sorties par exemples) sont paravirtualisées et d'autres (comme le *CPU*) reposent sur de la virtualisation assisté par le matériel. Ce type de partition offre souvent de meilleures performances que les partitions *PV* et *HVM* sans avoir besoin de modifiant autant le noyau hôte.

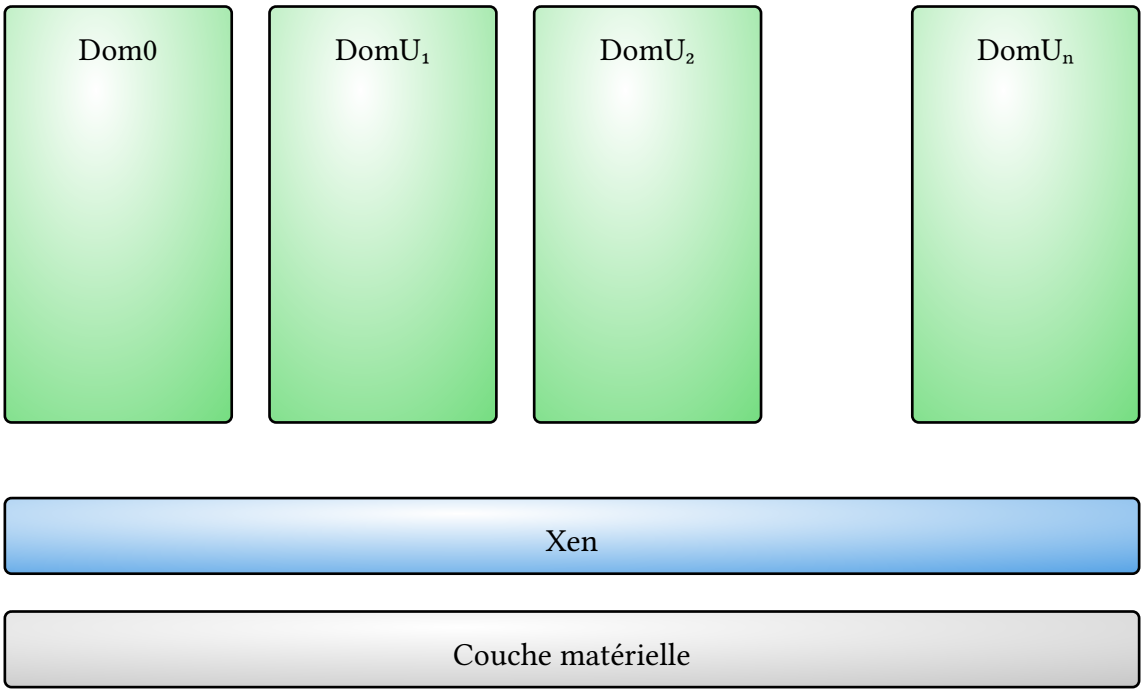

Fig. 6. – Architecture de *Xen*

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

- Le domaine 0 (abrégé *dom0*) désigne un domaine privilégié qui est automatiquement lancé au démarrage de l'hyperviseur. Le système d'exploitation hôte est généralement une distribution *Linux* modifiée (voir la section 8.6.6).
- Les domaines utilisateurs (abrégé *domU*) sont les domaines qui contiennent les OS invités. Il existe deux types de tels domaines. Les domaines de paravirtualisation et les domaines *HVM*.
- *dom0less*.

8.6.1 Stub domains

Aparté: Qu'est-ce qu'un stub domain?

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

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

Un *stub domain* (ou *stubdomain*) est un domaine système spécialisé utilisé pour désagréger le domaine de contrôle (*dom0*) [112], [113]. Il s'agit d'un domaine léger dédié à l'exécution de services ou de pilotes spécifiques, notamment le modèle de périphérique *QEMU* associé à un domaine HVM.

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

La plupart des *stub domains* sont basés sur le système d'exploitation minimaliste *Mini-OS* [45], bien que des travaux aient été menés sur des *stub domains* basés sur *Linux*.

8.6.2 Dom0less

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 (domaine de contrôle) 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 [114], [115]. Cette optimisation répond à un besoin critique dans les systèmes embarqués et temps-réel où le temps de démarrage est déterminant.

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

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

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

8.6.3 Partitionnement spatial

8.6.4 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 [116]. Il permet une répartition juste des processeurs physiques entre les *VM*. Plus précisément chaque *VM* dispose d'une fraction du temps *CPU* total qui est proportionnel à un poids configuré à l'avance. De plus l'ordonnanceur garantit qu'un processeur physique ne restera pas inactif s'il y a une tâche pouvant y être exécutée[25].
- *Credit2 Scheduler* est une évolution de *Credit Scheduler* [117]. 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 8.6.5.

8.6.5 Déterminisme

À ses débuts *Xen* a été conçu pour le *cloud computing*. En particulier, le projet met l'accent sur les garanties que les ressources louées par des clients seront effectivement disponibles lorsque leurs *VMs* les requerront. Les ordonnanceurs que nous avons vus dans la sous-section 8.6.4 cherchent donc à être aussi juste que possibles. Cette garantie est en contradiction avec les besoins du temps réel. En effet une tâche critique peut avoir soudainement besoin de beaucoup de ressources, si ce n'est la totalité des ressources.

Le projet *RT-Xen* visait à dôter *Xen* d'un ordonnanceur temps réel pour ces *vCPU*. À l'origine le projet est développé à partir d'un *fork* de la branche *4.3* de *Xen*. Il a été intégré dans *Xen* à partir de la version *4.5* sous la forme d'un nouvel ordonnanceur baptisé *RTDS* (*Real-Time Deferrable Server*) [118]. L'objectif de *RTDS* est d'assurer que les *VMs* reçoivent un temps *CPU* minimal garanti. Cet ordonnanceur supporte les plateformes *SMP*.

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

8.6.6 OS invités supportés

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

8.7 Corruption de la mémoire

L'hyperviseur *Xen* ne dispose pas d'un système de journalisation des erreurs mémoires. En revanche, il transmet ces erreurs au système d'exploitation exécuté dans le domaine privilégié *Dom0*. Il est alors possible d'utiliser les outils livrés avec ce système pour journaliser ces

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

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 2.5.

8.8 Perte du flux d'exécution

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

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

La nature critique de l'hyperviseur *Xen* en fait une cible privilégiée pour les attaquants. Une compromission du flux d'exécution au niveau de l'hyperviseur peut potentiellement affecter tous les domaines hébergés, d'où l'importance cruciale de ces mécanismes de protection [21].

8.9 Monitoring et profilage

Xen propose plusieurs outils pour le monitoring des performances et de l'état du système [119], [120]:

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

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

8.9.1 L'outil *xl*

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

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

8.9.2 *Xenoprof*

8.9.3 *XenTune*

8.9.4 Le traceur *xentrace*

Le logiciel *xentrace* [121] est un outil distribué dans *Xen*. Il permet de tracer l'activité des CPU virtuels et ainsi de savoir ce que fait une machine virtuelle sur un CPU donné. Ces données sont collectées grâce à des *tracepoints* positionnés à des endroits clés du code de *Xen*. Ils sont activés via *xentrace* lorsqu'il est exécuté dans le domaine *dom0*. Ce dernier produit alors un fichier binaire qui peut ensuite être analysé par *xenanalyze*[26].

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

8.10 Support de langages de programmation en *bare-metal*

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

8.11 Watchdog

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

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

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

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

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

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

  failed:
    xc_interface_close(h);
    return EXIT_FAILURE;
  }

```

Liste 11. – 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* [122]. 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 2.8.1.

8.12 Masquage des interruptions

Les interruptions matérielles sont virtualisées via le concept d'*event channels*. Il est possible de masquer ces évènements via des masques [123].

8.13 Maintenabilité

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

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

Un autre facteur important qui augmente la *TCB* est l'usage d'un noyau *Linux* dans le *dom0*. La compromission de ce système compromettant tout le système, il ne peut en être exclu.

8.14 Licences

Xen est un logiciel libre distribué majoritairement sous licence `GPLv2`. Toutefois certaines parties sont distribuées sous des licences plus permissives afin de pas contraindre les licences des logiciels applicatifs ou des *OS* portés sur *Xen*. Ces exceptions sont spécifiées dans les entêtes des fichiers concernés. Plus d'informations sont disponibles dans le fichier `COPYING` du dépôt git [124].

8.15 Temps de démarrage

9 XtratuM

XtratuM en bref

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

XtratuM est un hyperviseur temps-réel de type 1 qualifié pour un usage dans le spatial. Le projet est initié en 2004 au sein de l'institut *Automática e Informática Industrial* (ai2) de l'*Universidad Politécnica* de Valence en Espagne [125], [126]. Ces travaux universitaires ont abouti à la création de l'entreprise *fentISS* [127] en 2010 avec le soutien du *CNES* et du groupe *Airbus* [126]. 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 [125]. L'entreprise

Le succès de *XtratuM* dans le spatial est remarquable: son hyperviseur temps-réel est désormais déployé dans plus d'un millier de satellites et engins spatiaux [128], [129], en faisant l'un des logiciels système les plus largement adoptés en orbite. Cette présence massive témoigne de la maturité et de la fiabilité du système dans des environnements opérationnels critiques.

9.1 Architectures supportées

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

D'après les dernières brochures [130], *XtratuM* supporte les architectures suivantes: *ARM-v7*, *ARM-v8*, *SPARC*, *RISC-V*. En particulier, il supporte les architectures *SPARCv8 LEON3* et *LEON4* qui sont utilisées dans des missions spatiales. Le support se fait via des *BSP*.

9.2 Support multi-processeur

XtratuM était originellement développé sur des architectures monoprocesseur *x86* et *LEON*. Le support multi-cœur a donc nécessité de profondes modifications. C'est une approche à base de *spinlock* qui fut adopter pour assurer la synchronisation des sections critiques du noyau [131]. Chaque partition peut allouer plusieurs cœurs via une couche d'abstraction sous forme de *CPU* virtuels [131].

9.3 Partitionnement

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

9.3.1 Partitionnement spatial

Les partitions *XtratuM* sont exécutés en mode utilisateur.

La plateforme *LEON2* ne disposait pas nécessairement de *MMU*. Ce n'est plus le cas des architectures *LEON3* et *LEON4* qui incluent un tel dispositif. *XtratuM* intègre donc un support pour ce dispositif. En plus d'améliorer l'isolation spatiale en prévenant les lectures non autorisées, les *MMU* permettent d'implémenter des *IPC* plus rapides [132].

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 [133]. 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* [133], [134]. 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 (Real-time Operating System)* LithOS développé par *fentISS* et conforme *ARINC-653* [133], [135],

- Le *RTOS RTEMS*, notamment sur les plateformes *LEON* [136].
- Le noyau *Linux* [136],
- Le micronoyau temps réel *ORK+* (*Open Ravenscar Kernel*). Il a été porté sur *XtratuM* en 2011 [137]. Il permet le développement d'applications en *Ada* avec le profile *Ravenscar*.

9.3.6 Support de langages de programmation en baremetal

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 [138].

9.4 Corruption de la mémoire

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

9.5 Outils

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

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

9.5.1 *Health monitoring*

XtratuM intègre un service de *health monitoring* conforme à la norme *ARINC-653* [133]. Ce service permet la détection et la gestion des défaillances des partitions.

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

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

9.6 Support de *watchdog*

9.7 Temps de démarrage

9.8 Qualifications et certifications

ECSS-Qualified?

XtratuM est qualifié selon la norme *ECSS* (*European Cooperation for Space Standardization*) catégorie B [139]. Cette qualification en fait un hyperviseur adapté aux missions spatiales critiques.

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

9.9 Licences

À l'origine *XtratuM* était un projet open source sous licence *GPLv2* [140]. Cette version ne semble plus être développée et *fentISS* distribue une réécriture du noyau appelée *XNG* (*XtratuM Next Generation*) sous licence propriétaire. L'entreprise *fentISS* ne semble pas communiquer sur ses licences ou sa politique tarifaire. Les modalités et les coûts des licences sont négociés avec *fentISS* en fonction des besoins du projet.

9.10 Draft

XtratuM virtualises la mémoire, les timers et les interruptions.

XtratuM fait parti du projet *SAFEST* [141]. Il s'agit d'un projet visant à faire collaborer différents acteurs du secteur aérospatial européen afin d'améliorer les performances et de réduire les coûts.

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

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

10 Tableaux comparitifs

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

10.1 Architectures supportées

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

OS	x86-32	x86-64	ARM v7	ARM v8	PowerPC	MIPS	RISC-V	SPARC
Linux [6]	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
KVM [142]	Oui	Oui	Oui	Oui	Oui	Non	Non	Non
MirageOS [143], [144]	Non	Oui	Non	Oui	Non	Non	Non	Non
PikeOS [145]	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui
ProvenVisor [146]	Non	Non	Non	Oui	Non	Non	Non	Non
RTEMS 6.2	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
seL4 [96]	Oui	Oui	Oui	Oui	Non	Non	Oui	Non
Xen	Non	Non	Oui	Non	Oui	Non	Non	Non
XtratuM	Non	Non	Oui	Non	Oui	Non	Non	Non

10.2 SLOC

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

Pour les OS open-sources, nous avons utilisé l'outil `SLOCCount`[147] pour effectuer ces mesures. Cet outil ne compte pas les commentaires.

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

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

11 Glossaire

ABI. Application Binary Interface. interface binare-programme 73

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, 14, 42, 48, 52, 56, 63

API. Applicaton Programming Interface. Interface de programmation applicative. 14, 15, 33, 38, 41, 43, 50, 52, 57, 73

BKL. Big Kernel Lock. todo 14, 42, 56

BSP. Board Support Package. todo 6, 41, 42, 48, 50, 57, 71

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. 55

COTS. Commercial off-the-shelf. todo 12

DRAM. Dynamic Random Access Memory. Type de barette de mémoire utilisée massivement comme mémoire principale sur les serveurs et ordinateurs personnels. 9

ECC. Error correction code. Code utilisé pour permettre la détection et la correction d'erreur dans des données. 9, 10, 67

ECSS. European Cooperation for Space Standardization. Ensembles de normes utilisées dans le spatial. 52

ESA. European Space Agency. todo 47, 51, 52

GNU. GNU is Not Unix. todo 13

GPOS. General-Purpose Operating System. TODO 5, 13, 31, 33, 66

HVM. Hardware Virtual Machine. todo 63

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. 43

IPC. Inter-Process Communication. todo 34, 35, 50, 72

ISR. Interrupt Service Routine. Gestionnaire d'interruption exécutée lorsqu'une interruption matérielle ou logicielle survient. 18

IdO. internet des objets. Réseau d'objets embarqués et connectés au travers d'Internet. 45

MCE. Machine Check Exception. Erreur émise par le matériel qui est généralement fatale. 73

MPSoC. Multiprocessor System on a chip. Système embarqué complet sur un seul circuit intégré comprenant plusieurs processeurs généralement hétérogènes. 7, 14, 42, 48, 56, 63

PMU. Performance Monitoring Unit. Registres matériels intégrés dans les microprocesseurs modernes afin de compter des événements bas niveau dans un but de profilage. 25, 26, 57

RTE. RunTime Environment. todo 12, 43, 73

RTOS. Real-time Operating System. TODO 72, 73

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, 13, 14, 31, 34, 41, 42, 48, 49, 51, 52, 56, 63, 66

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. 31, 45, 70

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. 45

TLB. Translation Lookaside Buffer. Mémoire cache du processeur utilisée dans le but d'accélérer la traduction des adresses virtuelles en adresses physiques. 42

VM. Virtual Machine. todo 12, 31, 61, 66

WCET. Worst Case Execution Time. todo 42, 43, 56

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, 7, 11, 12, 33, 43, 57, 68, 71, 73

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. 28, 29, 47, 48

espace utilisateur. todo 15, 25, 28, 29

hard error. Corruption de la mémoire due à un dysfonctionnement matériel au niveau de la puce mémoire. 9

init system. Premier programme exécuté sur un système d'exploitation de type _UNIX_. 29

monolithique. todo 13

paravirtualisation. todo 61, 63

qualification. todo 51

ramasse-miette. TODO 12

runtime. Environnement d'exécution. Dans le cas d'un langage de programmation managé, cela inclut son garbage collector. 33, 34

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. 9

spinlock. todo 7, 19, 71

sécurité. Mesure prise pour prévenir les attaques d'une entité malveillante. 3

sûreté. Mesure prise pour veiller au bon fonctionnement d'un système et en cas de défaillance pour limiter les dégât occasionné. 3

Bibliographie

[1] A. S. Tanenbaum et H. Bos, *Modern operating systems*. Pearson Education, Inc., 2015.

[2] A. S. Tanenbaum, A. S. Woodhull, et others, *Operating systems: design and implementation*, vol. 68. Prentice Hall Englewood Cliffs, 1997.

[3] A. Silberschatz, P. B. Galvin, et G. Gagne, *Operating system concepts essentials*. Wiley Publishing, 2013.

[4] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, et B. D. de Dinechin, « The shift to multicores in real-time and safety-critical systems », in *2015 International Conference on Hardware/ Software Codesign and System Synthesis (CODES+ ISSS)*, 2015, p. 220-229.

[5] A. Burns et R. I. Davis, « A survey of research into mixed criticality systems », *ACM Computing Surveys (CSUR)*, vol. 50, nº 6, p. 1-37, 2017.

[6] « Linux Documentation – CPU Architectures ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/v6.15/arch/index.html

[7] « What is RCU? ». Consulté le: 18 octobre 2025. [En ligne]. Disponible sur: https://www.kernel.org/doc/html/next/RCU/whatisRCU.html

[8] « 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

[9] « Remote Processor Framework ». Consulté le: 3 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/staging/remoteproc.html

[10] « Remote Processor Messaging ». Consulté le: 3 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/staging/rpmsg.html

[11] « Real-time preemption ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/next/core-api/real-time/index.html

[12] « RT-mutex implementation design ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/locking/rt-mutex-design.html

[13] « RT-mutex subsystem with PI support ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.kernel.org/locking/rt-mutex.html

[14] « Threaded interrupt handler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/threadirq

[15] « Linux KVM website ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://linux-kvm.org/

[16] « Control Groups v1 ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html

[17] « Control Group v2 ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/admin-guide/cgroup-v2.html

[18] « chroot(2) – Linux manuall page ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://man7.org/linux/man-pages/man2/chroot.2.html

[19] « The Linux Kernel Documentation: EDAC subsystem ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/edac/index.html

[20] « The Linux Kernel Documentation: Scrubbing ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/edac/scrub.html

[21] R. J. Walls *et al.*, « Survey of Control-Flow Integrity Techniques for Embedded and Real-Time Embedded Systems », *ACM Transactions on Embedded Computing Systems*, vol. 21, nº 4, 2022, Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://dl.acm.org/doi/full/10.1145/3538275

[22] B. Gregg, « Linux Performance ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.brendangregg.com/linuxperf.html

[23] « 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/

[24] « Htop ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://htop.dev/

[25] « Netdata ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://www.netdata.cloud/

[26] « eBPF ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://ebpf.io/

[27] « SystemTrap ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://en.wikipedia.org/wiki/SystemTap

[28] « Prometheus ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://prometheus.io/

[29] « Grafana ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://grafana.com/

[30] « xenwatchdog man page ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://perfwiki.github.io/main/

[31] « OProfile ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://oprofile.sourceforge.io/about/

[32] « 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

[33] « The Linux Watchdog driver API ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://docs.kernel.org/watchdog/watchdog-api.html

[34] 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.

[35] A. Al Abduallah et A. Hedberg, « Decreasing boot time in an embedded Linux environment », Bachelor Thesis, Örebro, Sweden, 2023.

[36] « Yocto Project ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://www.yoctoproject.org/

[37] A. Madhavapeddy, « Unikernels ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://anil.recoil.org/projects/unikernels

[38] « Xen Project: MirageOS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://xenproject.org/projects/mirage-os/

[39] 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/

[40] 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/

[41] J. Vouillon, « Lwt: a cooperative thread library », in *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, 2008, p. 3-12.

[42] « Lwt manual ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: https://ocsigen.org/lwt/latest/manual/manual

[43] K. Sivaramakrishnan *et al.*, « Retrofitting parallelism onto OCaml », *Proc. ACM Program. Lang.*, vol. 4, nº ICFP, août 2020, doi: 10.1145/3408995.

[44] « MirageOS on OCaml 5 ». Consulté le: 24 septembre 2025. [En ligne]. Disponible sur: https://tarides.com/blog/2025-02-06-mirageos-on-ocaml-5/

[45] « Mini-OS ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Mini-OS

[46] « MirageeOS on Unikraft ». Consulté le: 22 septembre 2025. [En ligne]. Disponible sur: https://discuss.ocaml.org/t/mirageos-on-unikraft/16975

[47] « GitHub ocaml-vchan ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/mirage/ocaml-vchan

[48] « OCaml Profiling ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://ocaml.org/manual/5.4/profil.html

[49] « Git mirage-monitoring ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://git.robur.coop/robur/mirage-monitoring

[50] « Memtrace ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: https://github.com/janestreet/memtrace

[51] « Memtrace Viewer ». Consulté le: 23 septembre 2025. [En ligne]. Disponible sur: https://github.com/janestreet/memtrace_viewer

[52] « Github memtrace-mirage ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/robur-coop/memtrace-mirage

[53] « GitHub mirage-profile ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/mirage/mirage-profile

[54] 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/

[55] « GitHub mirage-trace-viewer ». Consulté le: 18 novembre 2025. [En ligne]. Disponible sur: https://github.com/talex5/mirage-trace-viewer

[56] A. Madhavapeddy et D. J. Scott, « Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? », *Queue*, vol. 11, nº 11, p. 30-44, 2013.

[57] 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.

[58] S. Kuenzer *et al.*, « Unikraft: fast, specialized unikernels the easy way », in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, p. 376-394.

[59] R. Kaiser et S. Wagner, « Evolution of the PikeOS microkernel », in *First international workshop on microkernels for embedded systems*, 2007.

[60] « 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

[61] « 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

[62] « 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

[63] O. M. Motzkus Andreas, « PikeOS Safe Real-Time Scheduling ». 2016.

[64] « ELinOS - Embedded Linux Distribution ». Consulté le: 20 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/elinos

[65] « 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

[66] « Windows as PikeOS Guest OS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/windows

[67] « Technology Partner Alliances ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/technology-alliances

[68] « Codeo ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/codeo

[69] « Codeo for PikeOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/popups/codeo-for-pikeos

[70] « Codeo for ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/popups/codeo-for-elinos

[71] « 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

[72] « Rapita Systems ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/technology-alliances/rapita

[73] « OS-and Hypervisor-aware Debugging ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.lauterbach.com/features/os-awareness

[74] « 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

[75] « 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

[76] « Rust for PikeOS & ELinOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.sysgo.com/rust

[77] « 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

[78] S. Lescuyer, « ProvenCore: Towards a Verified Isolation Micro-Kernel. », in *MILS@ HiPEAC*, 2015.

[79] « About OAR and RTEMS ». Consulté le: 16 octobre 2025. [En ligne]. Disponible sur: https://www.rtems.com/aboutoarandrtems

[80] « RTEMS: Architectures ». Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: https://www.rtems.org/architectures/

[81] « RTEMS Symmetric Multiprocessing ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/symmetric_multiprocessing_services.html

[82] « RTEMS Xilinx Zyna UltraScale+ BSP ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/bsps/arm/xilinx-zynqmp.html

[83] « SMP Schedulers ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/scheduling-concepts/smp-schedulers.html

[84] « 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

[85] « BSP Build System ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/bld/index.html

[86] « Tracing ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/user/tracing/index.html

[87] « RTEMS Shell Guide ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/index.html

[88] « RTEMS Specific Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/rtems_specific_commands.html

[89] « Memory Commands ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/shell/memory_commands.html

[90] « CPU Usage Statistics ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/c-user/cpu_usage_statistics.html

[91] « RTEMS SMP QDP ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://rtems-qual.io.esa.int/

[92] 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.

[93] « Formal Verification Overview ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/eng/fv/overview.html

[94] « 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

[95] « RTEMS Test Suites ». Consulté le: 1 octobre 2025. [En ligne]. Disponible sur: https://docs.rtems.org/docs/main/eng/test-suites.html

[96] « seL4 Supported platforms ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/Hardware/

[97] 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

[98] 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

[99] 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

[100] « Solox AMP Rust ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://github.com/jonlamb-gh/solox-amp-rust

[101] 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.

[102] 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.

[103] « Microkit User Manual ». Consulté le: 14 novembre 2025. [En ligne]. Disponible sur: https://docs.sel4.systems/projects/microkit/manual/latest/

[104] « rust-sel4 ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://github.com/seL4/rust-sel4

[105] « seL4 Licensing ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://sel4.systems/Legal/license.html

[106] S. H. VanderLeest, « The open source, formally-proven seL4 microkernel: considerations for use in avionics », in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, p. 1-9.

[107] « seL4 FAQ Verification ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://sel4.systems/About/FAQ.html#verification

[108] « seL4 What the Proof Implies ». Consulté le: 16 juillet 2025. [En ligne]. Disponible sur: https://sel4.systems/Verification/implications.html

[109] « Xen 4.20 release notes ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: https://xenproject.org/blog/xen-project-4-20-oss-virtualization/

[110] « Xen ARM with Virtualization Extensions ». Consulté le: 5 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions

[111] R. VanVossen, « XEN ON THE ZYNQ ULTRASCALE+ MPSOC ».

[112] « Xen: Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/StubDom

[113] « Xen: Device Model Stub Domains ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains

[114] « Xen: dom0less ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/unstable/features/dom0less.html

[115] « 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/

[116] « Credit Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit_Scheduler

[117] « Credit2 Scheduler ». Consulté le: 25 octobre 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Credit2_Scheduler

[118] 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.

[119] « 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

[120] 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

[121] « xentrace man page ». Consulté le: 20 août 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/4.20-testing/man/xentrace.8.html

[122] « xenwatchdog man page ». Consulté le: 19 août 2025. [En ligne]. Disponible sur: https://xenbits.xen.org/docs/4.20-testing/man/xenwatchdogd.8.html

[123] « Xen: Event Channel Internals ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://wiki.xenproject.org/wiki/Event_Channel_Internals

[124] « 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

[125] 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.

[126] « 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

[127] « fentISS ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://www.fentiss.com/

[128] « 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

[129] « Space Missions ». Consulté le: 3 novembre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/missions/

[130] « Safety Runs on XtratuM/NG ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://storage.googleapis.com/b2match-as-1/QFAPVNvf2nGRmR9JyyQhuYuZ

[131] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, et A. Crespo, « XtratuM hypervisor redesign for LEON4 multicore processor », *SIGBED Rev.*, vol. 11, nº 2, p. 27-31, sept. 2014, doi: 10.1145/2668138.2668142.

[132] 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.

[133] 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

[134] 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

[135] « LithOS ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/lithos/

[136] « XtratuM homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.fentiss.com/xtratum/

[137] 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.

[138] « xng-rs ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://github.com/aeronautical-informatics/xng-rs

[139] fentISS, « XtratuM Hypervisor ». Consulté le: 17 octobre 2025. [En ligne]. Disponible sur: https://www.fentiss.com/xtratum/

[140] « XtratuM code source ». Consulté le: 29 octobre 2025. [En ligne]. Disponible sur: https://github.com/lfd/XtratuM

[141] « SAFEST Project ». Consulté le: 28 août 2025. [En ligne]. Disponible sur: https://safest-project.eu/

[142] « seL4: Interrupts tutorial ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: https://www.linux-kvm.org/page/Processor_support

[143] « MirageOS on ARM64 ». Consulté le: 16 juin 2025. [En ligne]. Disponible sur: https://mirage.io/docs/arm64

[144] « MirageOS Installation ». Consulté le: 16 juin 2025. [En ligne]. Disponible sur: https://mirage.io/docs/install

[145] « PikeOS homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://www.sysgo.com/pikeos

[146] « ProvenRun homepage ». Consulté le: 26 juin 2025. [En ligne]. Disponible sur: https://provenrun.com/provenvisor/

[147] « SLOCCount website ». Consulté le: 3 juillet 2025. [En ligne]. Disponible sur: https://dwheeler.com/sloccount/