Le co-design d'un noyau de système d'exploitation et de sa preuve formelle d'isolation

Narjes Jomaa

Université de Lille

cnrs

École Doctorale régionale Sciences pour l'Ingénieur Lille Nord-de-France

Mémoire pour l'obtention du titre de

Docteur de l'Université de Lille

Discipline : Informatique et applications

présentée et soutenue publiquement par

Narjes JOMAA

le 20 Décembre 2018

Le co-design d'un noyau de système d'exploitation et de sa preuve formelle d'isolation

Jury

M. Timothy BOURKE	Chargé de Recherche, École normale supérieure & Inria	Examinateur
M. Gilles GRIMAUD	Professeur des Universités, Université de Lille	Directeur de thèse
M. David NOWAK	Chargé de Recherche, CNRS & Université de Lille	Encadrant
M. David PICHARDIE	Professeur, École normale supérieure de Rennes	Rapporteur
M. Etienne RIVIÈRE	Professeur, École Polytechnique de Louvain	Rapporteur
Mme Sophie TISON	Professeur des Universités, Université de Lille	Examinatrice

THESE

Remerciements

J'aimerais tout d'abord remercier mes encadrants de thèse, Gilles Grimaud et David Nowak de m'avoir encadré tout au long de cette *aventure*. Je les remercie profondément pour leurs disponibilités, leurs vifs encouragements ainsi que leurs motivations pendant mes moments de doute. Je me souviens toujours de toutes les longues discussions que nous avons eues pour orienter mon travail et clarifier mes propos. Une chose est sure, je n'aurais rien pu faire sans vous.

Je remercie Messieurs David Pichardie et Etienne Rivière qui ont accepté d'être rapporteurs de cette thèse. Je suis honorée de votre présence dans ce jury, ainsi que pour l'intérêt que vous avez porté à ce travail. Je remercie vivement Madame Sophie Tison et Monsieur Timothy Bourke d'avoir accepté de faire partie du jury de cette thèse en qualité d'examinateur.

J'adresse toute ma gratitude à Samuel Hym d'avoir suivi avec intérêt ma thèse. Sa rigueur et sa capacité d'analyse des problèmes m'ont permis d'avancer et de mener à bien ces travaux. Je te remercie pour ton aide précieuse.

Je voudrais également remercier tous les membres de l'équipe 2XS pour l'ambiance amicale tout au long de ces années de thèse.

A titre plus personnel, je remercie chaleureusement mon mari, Mohamed, pour son soutien et ses encouragements.

Une pensée particulière à ma famille qui a supporté (et continue à supporter) mon absence pour que je puisse aller jusqu'au bout de mes rêves.

Je tiens aussi à remercier mes amies en Tunisie Nadia, Rabeb et Abir pour leurs soutiens malgré les rares moments que nous avons pu passer ensemble ces derniers temps. C'est vrai que nos chemins ont divergé mais vous resterez toujours mes sœurs adorées.

Enfin, je remercie mes amies en France Yosr, Rabab, Amina et Ouardia pour toutes ces années magnifiques.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction Générale

Sommaire

1.1 Positionnement scientifique

L'existence d'erreurs et de failles dans les systèmes informatiques utilisés dans des domaines critiques tels que le domaine de l'aérospatial, du transport, du médical, de la finance, etc, est un risque qui peut avoir des conséquences désastreuses. Par exemple, en 2001 [IAE01] une erreur de calcul a été détectée (tardivement) dans une machine de radiothérapie qui devait détruire des cellules cancéreuses des patients. Cet appareil avait provoqué le décès d'au moins cinq personnes à cause d'une erreur dans le calcul de la dose de radiation. Plus récemment, prenons l'exemple de l'exploitation d'une faille dans le système informatique des voitures de la marque Jeep [Tut17]. Celle-ci a été exploitée par des pirates pour la prise de contrôle de ces voitures à plus de 15km de distance. La faille se trouvait dans le programme qui assure la connexion entre un smartphone et le tableau de bord de la voiture. Bien que cette fonctionnalité ne soit pas critique dans ce système informatique, la faille avait engendré des risques importants de sécurité. En effet, le système informatique qui gère cette fonctionnalité gère également d'autres fonctionnalités critiques tels que l'accélérateur ou le volant. Une défaillance au niveau de la gestion de la séparation entre ces deux fonctionnalités a rendu possible la prise de contrôle de tout le système de commande de la voiture. La survenue de ce genre d'événement exige donc une attention particulière lors de développement de ces systèmes critiques.

La sécurité des systèmes informatiques est un défi qui a été identifié depuis leur première apparition. En effet, l'omniprésence des failles logicielles rend les applications qui gèrent nos données critiques fortement vulnérables aux attaques de sécurité. Encore aujourd'hui l'objectif est le même : «*Mieux assurer la protection des données et du code face aux accès/modifications non légitimes ou simplement inattendus*». Pour être capable de lutter contre ces défaillances, il est important d'identifier tout d'abord les entités qui sont responsables de la sécurité dans un système informatique. En effet, ce système consiste en un ensemble de plusieurs équipements physiques gérés directement par le système d'exploitation. Ce dernier correspond à l'interface entre les applications et ces composants matériels et permet de rendre opérationnel un système informatique. D'une part, il se charge de superviser les accès à ces ressources matérielles en autorisant uniquement les accès qui sont légitimes. D'autre part, il permet aux applications de faciliter la manipulation de leurs données à travers les services qu'il expose. Ainsi, le système d'exploitation offre une vue abstraite sur le matériel permettant de faciliter l'exécution des applications. De nos jours, ces systèmes critiques sont de plus en plus vastes et complexes. Ainsi, cette complexité les fragilise fortement vis-à-vis de la sécurité et des services qu'ils doivent assurer.

Pour démontrer qu'un système garantit la sécurité, dans un premier temps, il est important pour ses développeurs d'être capable d'identifier et spécifier clairement ses propriétés. Il est à noter que la difficulté ici réside dans le fait que la sécurité n'est pas une propriété fonctionnelle. Elle ne peut pas être acquise simplement en ajoutant une fonction supplémentaire qui transforme un système non sécurisé en un système sécurisé. De plus, il n'existe pas une définition claire et unique de la sécurité dans un système informatique. Cette propriété critique dépend de plusieurs facteurs : les services fournis par le système d'exploitation, les équipements matériels qu'il gère ; et dans certains cas la sécurité dépend également de l'environnement d'exécution des applications. En pratique, il est clairement impossible d'énumérer toutes les failles qui menacent la protection des données critiques.

Plusieurs travaux ont été effectués pour mieux définir les propriétés de sécurité. La propriété la plus fondamentale qui a été identifiée est l'isolation mémoire. Elle consiste à la fois à assurer la protection des données vis-à-vis des modifications malveillantes et à garantir que seuls les acteurs qui sont autorisés peuvent y accéder. Dans le contexte d'un système d'exploitation, où plusieurs applications potentiellement critiques s'exécutent en parallèle et partagent les mêmes ressources matérielles telles que la mémoire physique, cette propriété doit être assurée.

Démontrer que cette propriété est préservée est une étape primordiale dans le cycle de développement d'un système d'exploitation. Bien que tester le bon fonctionnement d'un programme soit une étape nécessaire, cela n'est pas suffisant pour garantir la sécurité dans des environnements critiques.

De ce fait, des techniques de vérification formelles, permettant de raisonner finement sur le comportement des programmes, sont de plus en plus utilisées pour mieux répondre à cette problématique. Il est à noter qu'elles sont connues pour leur coût important (en temps et en effort humain) et la difficulté de leur intégration dans le processus de conception et de développement des systèmes informatiques. Toutefois, investir plus de temps dans la vérification des système pourra sauver la vie de certaines personnes et aussi nous faire gagner des années de travail. Ces techniques ont montré leur efficacité dans la détection des erreurs d'implémentation et de conception. Elles permettent d'étudier profondément l'implémentation de ces systèmes dans le but de définir une spécification claire de leurs comportements. Cette spécification peut être utilisée par la suite pour vérifier certaines propriétés sur les services fournis par le système d'exploitation.

Les techniques de vérification formelle sont très variées. Il y a celles qui sont automatisées telles que l'analyse statique ou le *model checking*. La première approche consiste à analyser le code du programme sans l'exécuter et essayer d'identifier des erreurs au niveau du code. La deuxième approche consiste à définir un modèle abstrait du programme concerné par la vérification puis montrer qu'il est correct par rapport à sa spécification. La vérification des propriétés attendues par ce programme se fait à travers l'énumération de tous les états possibles produits par ce dernier. Une propriété n'est pas considérée vraie si un contre-exemple est détecté. Ce contre exemple correspond à un état parmi les états énumérés qui ne garantit pas cette propriété. Dans ce cas une correction est nécessaire pour continuer la vérification de toutes les autres propriétés. Pour certains logiciels assez complexes cette approche atteint rapidement ses limites car l'énumération de tous les cas devient impossible. Il existe aussi des méthodes de vérification à l'aide d'un assistant de preuve. Cette méthode demande plus d'intervention humaine dans la construction de la preuve des propriétés du programme à vérifier. Elle est plus coûteuse mais apporte plus de garanties de correction d'anomalies dans un système critique. Dans cette thèse je m'intéresse à l'utilisation de l'assistant de preuve Coq [coq] dans la formalisation et la vérification de la propriété d'isolation d'un noyau de système d'exploitation.

1.2 Problématique

Dans cette section je mets en avant les différentes problématiques liées à la vérification formelle des propriétés de sécurité d'un noyau de système d'exploitation. Avant de commencer ce travail il est primordial d'étudier les aspects fondamentaux liés à la vérification tels que l'identification des propriétés à vérifier, la technique de vérification à utiliser ainsi que la nature du système à vérifier. Dans la suite j'examine tous ces aspects.

Quelles sont les propriétés à vérifier? Pour un système d'exploitation nous identifions deux catégories de propriétés :

Propriétés fonctionnelles : Ce sont les propriétés permettant de spécifier le bon fonctionnement du système. Une propriété fonctionnelle est locale. Elle est spécifique à un comportement ou à un service en particulier. Elle peut être valide quand bien même les autres fonctionnalités du système ne se comportent pas correctement. La vérification de sa validité ne dépend pas de la vérification du bon fonctionnement d'autres services.

Propriétés non fonctionnelles : Cette deuxième catégorie inclut les propriétés qui doivent être assurées par tous les services fournis par le système d'exploitation. Contrairement à une propriété fonctionnelle, une propriété non fonctionnelle est globale. Elle ne peut être garantie que lorsqu'elle est assurée par tous les services du système. Une propriété de sécurité est une propriété non fonctionnelle. Sa vérification nécessite une preuve permettant de démontrer que tous les services exposés par le système l'assurent.

Que faut-il vérifier en priorité? Un système qui fonctionne correctement ne garantit pas obligatoirement la sécurité. Dans le contexte de certain systèmes critiques tels qu'un système d'exploitation, il est moins critique d'avoir des comportements incorrects que d'avoir une faille de sécurité. En effet, les conséquences d'une faille de sécurité sont souvent plus fatales que celles du bon fonctionnement. Dans cette thèse je m'intéresse en particulier à la vérification des propriétés de sécurité d'un noyau de système d'exploitation.

Quelle propriété de sécurité exactement? Dans cette thèse je m'intéresse principalement à la vérification de la propriété d'isolation mémoire. Cette propriété assure l'absence des accès non légitimes des applications aux données et au code des autres applications qui s'exécutent en parallèle ainsi que à ceux du noyau. La propriété d'isolation telle que formalisée par Rushby [Rus81] est une propriété fondamentale dans la sécurité des systèmes d'exploitation. Dans cette thèse j'ai réalisé des travaux sur la vérification de la propriété d'isolation et j'ai montré qu'il est possible de vérifier d'autres propriétés de sécurité telles que la non interférence en utilisant principalement les spécifications des propriétés d'isolation.

Est-il possible de vérifier la propriété d'isolation sans passer par le bon fonctionnement? Il est important de noter qu'une erreur dans le code semble parfois non critique mais il arrive que ses conséquences aient un impact important sur la sécurité. Définir et vérifier l'intégralité des propriétés fonctionnelles et non fonctionnelles d'un système paraît peu raisonnable. De ce fait, il était important de commencer par l'identification de l'ensemble des propriétés primordiales à vérifier afin de réduire le coût de la production de la preuve. Cette expérience de vérification a montré que pour vérifier ces propriétés d'isolation, il était indispensable de vérifier un ensemble de propriétés du bon fonctionnement appelées les propriétés de cohérence et les propriétés internes. Cet ensemble est identifié progressivement durant la vérification des propriétés d'isolation. Il comprend uniquement les propriétés qui sont requises pour l'isolation. Elles concernent principalement la cohérence des données sauvegardées par le noyau en fonction du comportement des services qu'il gère. Cette stratégie de vérification m'a permis d'identifier facilement le plus petit ensemble des propriétés du bon fonctionnement nécessaires pour assurer les propriétés d'isolation mémoire.

La vérification est-elle effectuée sur un modèle abstrait?

« *All models are wrong, but some are useful* » [BD86]

Un modèle d'un système informatique n'est qu'une représentation abstraite (simplifiée) du système réel. De ce fait, la vérification des propriétés de sécurité d'un modèle simplifié n'est pas obligatoirement suffisante pour que ces propriétés soient assurées par le système concret. Toutefois, il est toujours utile d'effectuer des expérimentations sur un modèle abstrait avant de mener une expérience sur un système plus complexe. Dans cette thèse, j'ai défini, dans un premier temps, un modèle abstrait d'un noyau de système d'exploitation à partir duquel j'ai étudié la faisabilité de l'approche de vérification. Cette étude consiste à définir et vérifier la propriété d'isolation mémoire relative à ce modèle de système d'exploitation. Puis j'ai utilisé la même méthodologie pour vérifier les propriétés d'isolation d'un système concret.

Technique de vérification automatique? Utiliser les techniques de vérification automatiques est une approche intéressante lorsqu'il s'agit de vérifier des propriétés fonctionnelles. Les propriétés de sécurité sont souvent difficiles à exprimer dans des langages peu expressifs. Dans ce cas les propriétés de sécurité peuvent être codées mais le risque de ne pas les définir correctement est important [GRRV90].

La preuve formelle est une technique compliquée, est il possible de l'appliquer sur un système aussi complexe ? Plusieurs travaux ont montré qu'il est possible de vérifier formellement des systèmes complexes tels que les noyaux des systèmes d'exploitation. Toutefois, le coût de la production de la preuve est souvent important [KAE+14]. Les travaux réalisés dans cette thèse consistent à intégrer une méthodologie de vérification dans le processus de conception et de développement d'un noyau de système d'exploitation. Le but de cette intégration est le *co-design* de noyau de système d'exploitation et de sa preuve, autrement dit, développer un nouveau système adapté à la vérification. Au sein de l'équipe 2XS, cette approche nous a permis de faire des choix durant les phases de conception et de développement pour réduire le coût de l'établissement de la preuve. Ces choix concernent principalement la réduction de la taille du noyau en minimisant le nombre de services qu'il expose. Ceci est une manière intéressante pour réduire la base de confiance sur laquelle les preuves sont effectuées. De plus, ces travaux de vérification sont fondés sur la séparation des préoccupations. Il s'agit d'une approche incrémentale de vérification. Cette approche consiste à vérifier en priorité les propriétés les plus fondamentales du noyau telles que les propriétés d'isolation mémoire. Puis, à partir desquelles il est possible de vérifier d'autres propriétés complémentaires.

Est-il possible de réduire la taille de la base de confiance ? La complexité d'un système d'exploitation est causée principalement par le nombre important des modules ainsi que par l'interdépendance entre eux. Pour des raisons d'efficacité tous ces modules sont exécutés en mode noyau, c'est-à-dire sans aucune restriction pour modifier l'état de la machine. Ceci élargit la surface d'attaque et donc rend le système plus vulnérable. En revanche, lorsque ce système est utilisé dans un environnement critique où la vérification de ses propriétés est indispensable, il est nécessaire de réduire le nombre de modules qui sont exécutés en mode noyau et ne garder que ceux qui sont nécessaires pour la sécurité. De nombreux travaux [Lie95, EKO], d'abord autour des micro-noyaux, mais aussi autour des exo-noyaux puis de la virtualisation ont montré que c'est tout à fait réalisable. Toutefois il est nécessaire de choisir minutieusement les modules à exporter pour assurer à la fois la sécurité, l'utilisabilité du système et la faisabilité de la preuve. Dans cette thèse, la réduction de la taille de la base de confiance n'était pas un choix individuel. Ceci a été effectué suite à plusieurs échanges avec les spécialistes en système du projet. Les discussions ont essentiellement porté sur l'identification des structures et des services permettant de garantir un niveau acceptable d'efficacité et de réduire drastiquement le coût de la preuve.

En quel langage le noyau est-il implémenté ? Habituellement, un noyau de système d'exploitation est implémenté dans un langage de bas niveau tel que le langage C. Ceci permet une configuration plus aisée du matériel qu'il doit gérer. Sachant que dans ces travaux j'ai utilisé l'assistant de preuve Coq pour établir la preuve de la propriété d'isolation mémoire du noyau. Il est donc plus intéressant d'écrire le code du noyau en Coq puis, en utilisant un outil qui s'appelle *digger* [HO17] développé également au sein de l'équipe 2XS, le traduire automatiquement vers C. Il est important de noter qu'il est possible de vérifier en Coq un programme écrit en C, mais ceci nécessite de gérer en même temps la représentation de la syntaxe et de la sémantique de C en Coq, et la vérification des propriétés de sécurité du programme concerné. C'est pourquoi j'ai choisi d'écrire le code du noyau en Coq. Pour ce faire, puisque Gallina, le langage de spécification de Coq, est un langage fonctionnel, j'ai utilisé une monade d'état [Wad90, Wad95] pour définir et gérer l'état de la machine. Cette technique permet d'intégrer les traits fondamentaux de la programmation impérative (e.g. le changement de l'état de la machine, les accès au matériel, les traitements séquentiels) dans la programmation fonctionnelle.

Comment vérifier des propriétés en Coq ? La majorité des travaux autour de la vérification des systèmes complexes sont fondés sur l'approche de raffinement. Cette méthodologie consiste à développer et vérifier formellement des systèmes de manière incrémentale. Il s'agit de définir, dans un premier temps, un modèle abstrait du système désiré, puis à partir de ce dernier de raffiner graduellement son

implémentation en introduisant à chaque niveau plus de détails sur les objets manipulés par ce système tels que les structures de données et les variables. La preuve des propriétés les plus fondamentales se fait uniquement au niveau du modèle abstrait. Chaque niveau inférieur est considéré comme un raffinement des niveaux précédents et une preuve est nécessaire pour prouver la validité de chaque niveau de raffinement. Cette approche est intéressante car elle permet de réduire le coût de la preuve. En revanche, lorsqu'il s'agit de vérifier des propriétés de sécurité cette approche risque de ne pas assurer la propagation de la propriété de sécurité, vérifiée uniquement sur le modèle abstrait, jusqu'à l'implémentation. Cette problématique est connue sous le nom « *refinement paradox* » [Jac89, Jac88]. Bien qu'il existe des travaux qui ont démontré qu'il est possible de contourner ce problème grâce à une définition plus riche du modèle abstrait [Mor09, Mor12], j'ai suivi une autre stratégie de vérification qui est l'établissement de la preuve directement sur le code pour éviter tous problème de modèle erroné. De plus le code du noyau est écrit en Gallina, le même langage utilisé pour établir la preuve, donc il est tout à fait naturel de suivre cette approche.

1.3 Contexte technologique

Les travaux menés dans cette thèse sont effectués au sein de l'équipe 2XS du laboratoire CRIStAL. Ils sont réalisés grâce au financement du projet européen CELTIC+/ODSI (On Demand Secure Isolation) [On 18]. L'objectif de ce projet consiste à concevoir et développer une nouvelle plate-forme où la propriété de sécurité, l'isolation mémoire, doit être vérifiée par le biais d'une preuve formelle. Cette plate-forme est destinée au contexte où des applications à criticité mixte doivent coexister dans le même environnement. La vérification de cette propriété critique est effectuée sur une base de confiance assez réduite qui correspond à un noyau minimal de système d'exploitation développé également dans le cadre du projet CELTIC+/ODSI. Ceci permet d'assurer une garantie forte pour toutes les applications nécessitant un mécanisme d'isolation de contexte telles que la communication entre machines, l'internet des objets, le partage d'infrastructure réseau, etc. Le niveau d'assurance global est obtenu par la combinaison d'applications certifiées au dessus d'une base de confiance certifiée. La vérification de ces applications concerne principalement leur bon fonctionnement où les propriétés de sécurité sont acquises grâce à la certification des couches inférieures.

Plusieurs partenaires industriels européens sont engagés dans ce projet comme Orange SA (France), Prove and Run (France), Internet of Trust (France), Nextel S.A. (Espagne), Resonate MP4 (Roumanie), etc. Au sein de l'équipe 2XS nous travaillons sur le développement et la vérification formelle d'un noyau minimal appelé Pip. Ce noyau vérifié fournit les fonctions fondamentales pour assurer la sécurité de la plateforme ODSI. Le tableau 1.1 illustre brièvement la répartition des différentes tâches du projet ODSI entre les différents partenaires impliqués dans le projet.

Réf. de la tâche	**Tâche**	**Affectation**
WP1	Conception de l'interface entre le matériel et les applications ODSI et vérification formelle de la propriété d'isolation	Université de Lille
WP2	Conception de la politique de Communication	NEXTEL
WP3	Définition du système de gestion ODSI	Orange
WP4	Certification CC des applications ODSI	INTERNET of TRUST
WP5	Définition des cas d'utilisation de ODSI	City Passenger

TABLEAU 1.1 – La répartition des tâches du projet ODSI.

1.4 Présentation du manuscrit

Le deuxième chapitre présente l'état de l'art et les différentes connaissances utiles à la bonne compréhension de mes travaux de thèse. Le troisième chapitre détaille la première contribution. Cette dernière est une étude préliminaire de faisabilité qui porte essentiellement sur l'approche de la vérification. La deuxième contribution est détaillée dans les chapitres 4, 5 et 6. Le quatrième chapitre introduit les concepts fondamentaux du proto-noyau Pip et définit informellement les propriétés visées par la vérification. Le cinquième chapitre contient les détails de l'implémentation en Coq du proto-noyau. Ensuite, le sixième chapitre présente la formalisation en Coq des propriétés de sécurité et détaille le mécanisme de preuve en expliquant les étapes de la vérification, les difficultés rencontrées et les bugs remontés. La conclusion propose un bilan du travail effectué durant la thèse et un ensemble de perspectives.

Bibliographie

[BD86] George E P Box et Norman R Draper : *Empirical Model-building and Response Surface*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[coq] *The Coq Proof Assistant*. Available at https://coq.inria.fr/.

[EKO] D. R. Engler, M. F. Kaashoek et J. O'Toole, Jr. : Exokernel : An operating system architecture for application-level resource management. *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266.

[GRRV90] S. Graf, J.-L. Richier, C. Rodríguez et J. Voiron : What are the limits of model checking methods for the verification of real life protocols? *In* Joseph Sifakis, éditeur : *Automatic Verification Methods for Finite State Systems*, pages 275–285. Springer Berlin Heidelberg, 1990.

[HO17] S. Hym et V Oudjail : Digger, 2017. https://github.com/2xs/digger.

[IAE01] IAEA : *Investigation of an accidental exposure of radiotherapy patients in Panama*. International Atomic Energy Agency, 2001.

[Jac88] Jeremy Jacob : Security specifications. *In Security and Privacy, 1988. Proceedings., 1988 IEEE Symposium on*, pages 14–23. IEEE, 1988.

[Jac89] Jeremy Jacob : On the derivation of secure components. *In Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 242–247. IEEE, 1989.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski et Gernot Heiser : Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 2014.

[Lie95] J. Liedtke : On micro-kernel construction. *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, 1995.

[Mor09] Carroll Morgan : How to brew-up a refinement ordering. *Electronic Notes in Theoretical Computer Science*, 259:123–141, 2009.

[Mor12] Carroll Morgan : Compositional noninterference from first principles. *Formal Aspects of Computing*, 24(1):3–26, 2012.

[On 18] On Demand Secure Isolation : Available at https://www.celticplus.eu/project-odsi/, 2016-2018.

[Rus81] John M Rushby : *Design and verification of secure systems*. ACM, 1981.

[Tut17] Hilary TUTTLE : Hacking cars : when considering the risks of cutting-edge automotive technology, the first thing that usually comes to mind is autonomous vehicles. but focusing too much on self-driving technology risks ignoring a critical reality : Today's cars and trucks are already connected to the internet, and like any other internet-connected device, they can be hacked. *Risk Management*, pages 20–26, 2017.

[Wad90] Philip WADLER : Comprehending monads. *In LISP and Functional Programming*, pages 61–78, 1990.

[Wad95] Philip WADLER : Monads for functional programming. *In International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

Chapitre 2

État de l'art

Sommaire

Dans ce chapitre nous présentons les différents travaux effectués autours de la mise en place des approches permettant d'améliorer la sécurité dans les systèmes d'exploitation. En particulier, nous nous intéressons à ceux qui se sont concentrés sur la conception et l'implémentation de ces systèmes ou encore ceux qui visent la vérification de leurs propriétés fondamentales.

2.1 Sécurité dans les noyaux des systèmes d'exploitation

2.1.1 Architecture monolithique

La vulnérabilité des systèmes d'exploitation est causée principalement par la taille et la complexité de leur code. Par exemple, actuellement le noyau Linux comprend plus de 20 millions de lignes de code [CKH16], ce qui rend ce dernier difficile à comprendre et à maintenir. Le noyau Linux appartient à la famille des noyaux monolithiques. Il gère tous les services de base nécessaires pour faire fonctionner des applications. Ces services sont gérés sous la forme de nombreux modules interdépendants tels que la répartition de la mémoire entre les applications, la gestion des fichiers, la gestion des périphériques, la communication entre les processus, l'ordonnancement et d'autres modules qui s'exécutent en mode privilégié. En effet, autoriser ces modules à s'exécuter en mode privilégié (aussi appelé mode noyau) consiste à leurs permettre d'accéder à toutes les ressources matérielles sans aucune restriction. Par conséquent ceci rend la surface d'attaque plus large. Par exemple, une manipulation incorrecte de pointeurs dans l'implémentation du système de fichiers qui s'exécute en mode noyau peut causer des vulnérabilités critiques au niveau du noyau. Par conséquent des applications malveillantes peuvent exploiter ces failles et détourner les fonctions du noyau. Également, la faute de l'un des modules noyau peut provoquer la faute de tous les logiciels qui s'exécutent au-dessus.

Il est important de noter que l'avantage principal d'une architecture monolithique est de maximiser les performances du système. En effet, ces derniers dépendent fortement du temps nécessaire pour changer de contexte d'exécution d'un module à un autre et pour gérer la communication entre eux. Dans une architecture monolithique, réduire ce coût important consiste principalement à exécuter la majorité de ces modules en mode noyau. Ceci est tout à fait cohérent puisque dans ce cas n'importe quel module a un accès direct au matériel et peut être évoqué directement par le noyau sans passer par un mécanisme de contrôle assuré par le matériel ou mis en place par le noyau lui-même. En outre, améliorer, à la fois, les performances et la sécurité du système est un défi difficile à réussir notamment avec la présence d'un code aussi complexe et non modulaire. La modularité nécessite une architecture basée sur un ensemble de modules (ou des services) clairement définis et isolés où l'interdépendance est gérée explicitement par le noyau. Plusieurs travaux ont été effectués afin d'améliorer la sécurité des systèmes monolithiques, citant l'exemple de la NSA [Nat] qui a développé SELinux [PL01] en intégrant un nouveau module de sécurité au noyau Linux. Malgré l'importance de ces travaux, il sont insuffisants pour certaines applications qui manipulent des données extrêmement critiques.

La base de confiance dans un noyau monolithique inclut tous les modules s'exécutant en mode privilégié. Autrement dit, c'est l'ensemble de services auxquels nous faisons confiance. Il a été démontré qu'il est tout à fait possible de réduire la taille de cette base de confiance en diminuant le nombre de modules qui s'exécute en mode noyau. Ceci est effectué en plaçant les modules qui ne sont pas critiques pour la sécurité à l'extérieur du noyau et les exécutant en mode non privilégié (aussi appelé mode utilisateur) comme n'importe quelle application. Il s'agit de la minimisation du TCB (*trusted computing base*) [Lat85].

2.1.2 Réduction de la taille de la base de confiance

L'objectif principal de la minimisation du TCB est de réduire la surface d'attaque grâce à la réduction du nombre et de la taille des services critiques fournis par le noyau.

Micro-noyaux

Le concept de la minimisation du TCB n'est pas nouveau. Cette approche a été implémentée par les micro-noyaux dans les années 1980 dans le but de rendre les noyaux plus simples à maintenir et avoir une idée plus claire sur chaque composant fourni par le noyau. Pour ce qui est la taille de code, les micro-noyaux font généralement moins de 10 000 de lignes de code.

La figure 2.1 illustre la minimisation du code noyau défendu par le concept des micro-noyaux. Contrairement aux noyaux monolithiques, la figure à gauche, dans le contexte d'un micro-noyau uniquement quatre modules s'exécutent en mode privilégié. Ils gèrent la répartition de la mémoire physique entre les applications, gère les interruptions, fournit un ordonnanceur pour gérer le temps d'exécution entre les processus, et assure la communication entre eux à travers les messages (IPC).

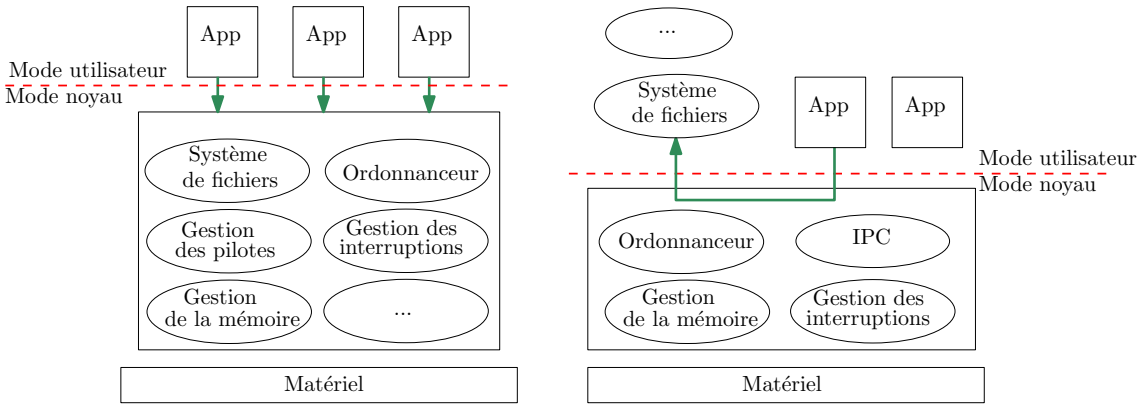


FIGURE 2.1 – Externalisation des modules du système d'exploitation

Plusieurs travaux ont visé l'implémentation de ce type de noyau modulaire tel que Chorus [RM87], L3 [Lie93], Amoeba [TVRVS+90] ou aussi Mach [GDFR90]. Ce dernier a été commercialisé et représente un bon exemple de la mise en place d'un système de fichiers en mode utilisateur au lieu de l'implémenter directement dans le noyau.

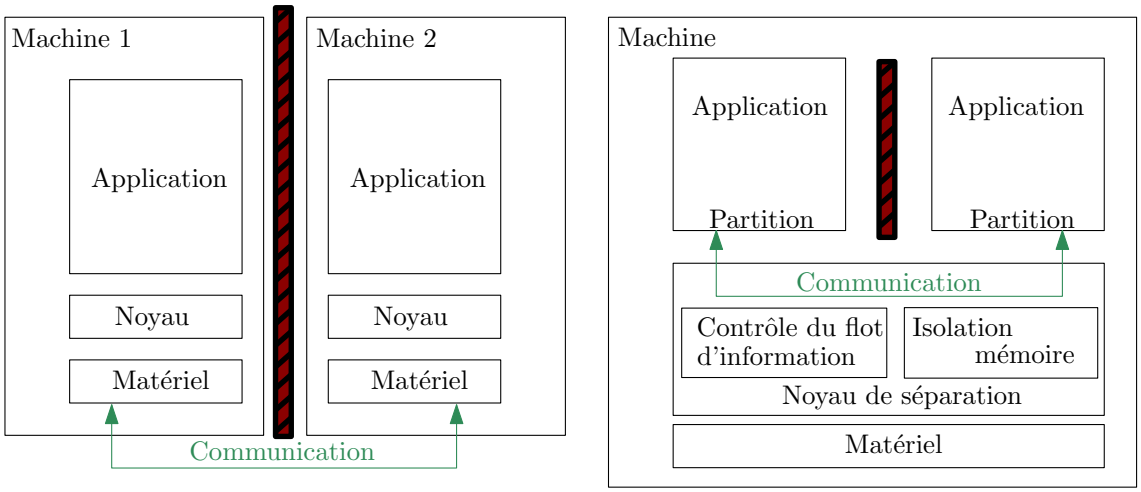
Noyaux de séparation

Les noyaux de séparation sont considérés comme une spécialisation des micro-noyaux. Ils ont été introduits par Rushby [Rus81]. Cette catégorie de noyau consiste à simplifier la spécification, le design et l'analyse des mécanismes de sécurité en se basant sur l'approche de séparation (ou de *partitionnement*). L'objectif de cette architecture est de fournir un environnement sécurisé permettant principalement d'isoler les applications entre elles à travers des mécanismes de virtualisation de telle sorte qu'elles ne peuvent pas distinguer cet environnement partagé (les ressources matérielles telles que la mémoire physique) d'un environnement physiquement distribué. La communication entre ces applications est autorisée uniquement à travers des canaux de communication explicites et gérés par le noyau (voir figure 2.2).

Les noyaux de séparation sont aussi l'origine des architectures MILS (Multiple Independent Levels of Security) [AFOTH06]. Ces architectures adressent deux objectifs principaux : renforcer la sécurité et gérer correctement les ressources partagées. En 2007 la NSA a publié une spécification formelle définissant l'ensemble des objectifs et des exigences de sécurité d'un noyau de séparation connue sous le nom SKPP (Separation Kernel Protection Profile) [Inf07].

2.1.3 Un mécanisme matériel pour la sécurité logicielle

Pour assurer la sécurité et en particulier la propriété d'isolation mémoire, les applications ne peuvent pas accéder directement à la mémoire physique. Néanmoins, le noyau fourni un mécanisme d'accès par



(a) Isolation physique (b) Isolation par le noyau de séparation

FIGURE 2.2 – Gestion de la communication dans un environnement partagé

adressage virtuel afin de contrôler l'exécution de chaque application et éviter tout accès non autorisé. Ce mécanisme s'appuie sur une coopération entre le noyau et un composant matériel appelé MMU (*Memory Management Unit*). En effet, le noyau doit configurer correctement les tables de MMU, appelées *tables de pages*, de telle sorte qu'il n'attribue à chaque application que les pages mémoire auxquelles elle a le droit d'accéder. Puis, au cours de l'exécution d'une application le MMU reçoit des demandes d'accès à la mémoire sous la forme d'adresses virtuelles. Selon la configuration des *tables de pages*, si la traduction de l'adresse virtuelle vers une adresse physique est disponible, le MMU autorise l'accès et fournit l'adresse physique correspondante. Ce mécanisme assure que chaque application n'accédera qu'à sa propre mémoire déjà attribuée par le noyau.

Description formelle du mécanisme de traduction d'adresse par le MMU La figure 2.3 illustre le mécanisme de traduction d'adresse établi par un MMU avec trois niveaux d'indirection. En effet, il existe plusieurs types de ce composant matériel. Le principe de traduction d'adresse est toujours le même, cependant certaines caractéristiques peuvent varier telles que le nombre de niveaux de traduction.

À la création de chaque application, le noyau définit une nouvelle configuration de MMU et l'associe à cette application afin de gérer ses accès à la mémoire. Chaque espace d'adressage est structuré sous la forme d'une structure arborescente où chaque nœud dans l'arbre correspond à une table de pages. Pour identifier cette structure, il suffit de référencer le pointeur vers la racine de cet arbre. Au cours de l'exécution d'une application, le MMU est configuré de telle sorte qu'il référence la racine de la structure associée à cette application. Sur la figure 2.3 le pointeur vers la racine est nommé *ptp* (*page table pointer*).

Dans la suite du document nous considérons que la mémoire physique est un ensemble de pages de taille fixe. Une table de pages a exactement la taille d'une page physique [1]. En effet, si la taille de la page mémoire est égale à *n unités* (une *unité* peut correspondre à plusieurs octets) alors une table de pages est composée de *n* entrées. Une entrée de table est appelée PTE (*page table entry*).

Afin de définir la structure arborescente, chaque entrée dans la table peut contenir (ou pas) un pointeur vers une table de pages du niveau inférieur. La profondeur de cet arbre est fixée selon le MMU utilisé. Le dernier niveau de MMU (les feuilles) contient les numéros de pages physiques attribuées à l'application. Chaque numéro de page est stocké dans un PTE qui contient des informations supplé-

1. Il est à noter que dans ce document nous considérons uniquement les architectures dans lesquelles la taille d'une adresse virtuelle est égale à celle d'une adresse physique. En revanche, il est possible d'avoir d'autres architectures dans lesquelles la taille d'une adresse virtuelle peut différer.

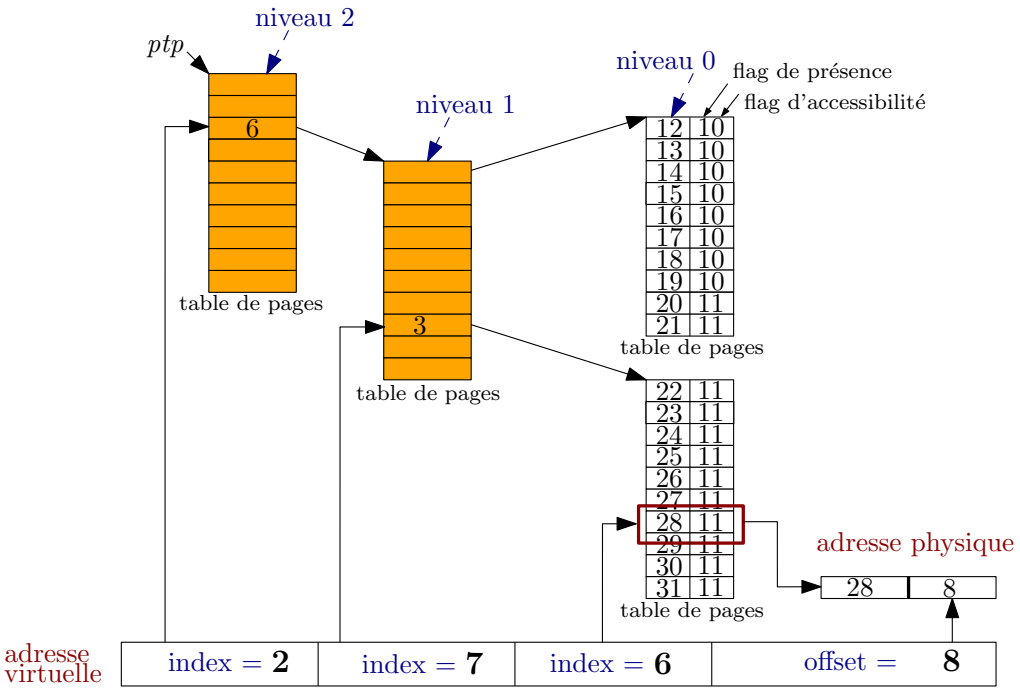


FIGURE 2.3 – Les tables de configuration d'un MMU de trois niveaux d'indirection

mentaires sur cette page telles que les bits de contrôle. Ces derniers sont utilisés par le MMU afin de vérifier si la page associée est accessible par l'application ou pas.

Sachant qu'une adresse virtuelle définit une branche dans l'arbre, pour un MMU de trois niveaux, une adresse virtuelle doit être composée de trois indexes plus un offset. Chaque index correspond à une position dans la table du niveau correspondant. Si la traduction est disponible, la première partie de l'adresse physique correspond au numéro de page récupéré à partir du dernier niveau (la page numéro 28 sur la figure 2.3) et la deuxième partie correspond à l'offset de l'adresse virtuelle. Cet offset est utilisé comme une position dans la page récupérée du dernier niveau. Sur la figure 2.3, l'offset 8 est une position dans la page 28. Cette page peut être utilisée par l'application associée à cette configuration pour stocker son code et ses données.

Il faut noter que pour garantir certaines propriétés de sécurité telles que l'isolation mémoire, les pages de configuration de MMU ne doivent en aucun cas être accessibles par des acteurs autre que le noyau. Sinon, des modifications pourraient compromettre la sécurité de tout le système. C'est pourquoi, tous les travaux qui visent la minimisation du TCB conservent le gestionnaire de la mémoire à l'intérieur du noyau.

2.2 Vérification formelle pour la sécurité des noyaux

Plusieurs travaux ont visé le développement des outils et des méthodes de la vérification formelle afin d'établir une preuve mathématique sur le bon fonctionnement des programmes. Cette preuve pourra vérifier qu'un programme est correct par rapport à une spécification, qu'il ne contient pas de bugs ou alors, plus généralement, qu'il garantit certaines propriétés. Le processus de l'établissement d'une preuve est une activité fastidieuse en matière de temps et d'effort humain. Ce qui rend cette méthode difficile à appliquer sur des programmes assez riches et complexes. Par conséquent, investir dans ces approches n'est pertinent que dans les contextes qui exigent des propriétés extrêmement critiques. Dans ce contexte, des travaux autour de la vérification des noyaux de systèmes d'exploitation ont été élaborés afin de prouver certaines propriétés assez critiques autour de la sécurité et du bon fonctionnement.

2.2.1 Preuve formelle : l'assistant de preuve Coq

La preuve formelle est une technique permettant de raisonner rigoureusement sur un système. Elle consiste à utiliser un outil informatique, appelé *assistant de preuve*, pour vérifier automatiquement une démonstration mathématique. D'une part, comme une preuve mathématique sur papier, le développement de cette démonstration est effectué manuellement. D'autre part, contrairement à une preuve sur papier où la validation de la démonstration est assuré uniquement par une intuition fondée sur des connaissances mathématiques et logiques, cette validation est effectuée automatiquement par le noyau de l'assistant de preuve. Celui-ci est capable de répondre *oui* ou *non* en fonction de la validité de la démonstration (voir figure 2.4). Il s'agit de développer un raisonnement logique sous la forme d'une suite de déductions vérifiables par une machine. Ceci est extrêmement utile lorsqu'il s'agit d'une démonstration longue et complexe où il est tout à fait possible de se tromper ou d'oublier des détails importants. De nombreux systèmes de preuve formelle tels que l'assistant de preuve Coq [coq], ont été développés pour rendre possible l'automatisation de la vérification des démonstrations.

Pour résumer, un assistant de preuve est composé d'un langage formel permettant d'écrire la spécification d'un programme ainsi que ses propriétés dans un formalisme logique. Il fournit également une librairie riche pour aider à l'écriture des démonstrations.

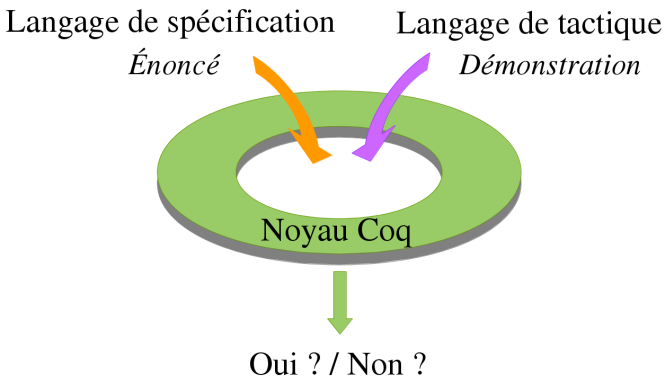


FIGURE 2.4 – Mécanisme de démonstration de l'assistant de preuve Coq.

Le langage de spécification Coq est fondé sur un langage fonctionnel appelé *Gallina*. L'expressivité dans ce langage assure la possibilité de construire des types de données structurés, des fonctions récursives qui terminent obligatoirement, des relations et des formules du calcul des prédicats d'ordre supérieur. Toutes ces caractéristiques permettent de définir des programmes sous la forme d'objets mathématiques et raisonner sur ces objets à travers la spécification de leurs propriétés et la construction des preuves de celles-ci. Il comprend également un ensemble riche de règles et des propositions logiques permettent de formaliser des théories telles que « $\rightarrow$ » pour exprimer une implication, « $\wedge$ » pour exprimer une conjonction, « $\vee$ » pour exprimer une disjonction, « $\forall$ » pour une quantification universelle et « $\exists$ » pour exprimer une quantification existentielle.
Bien que Gallina soit un langage fonctionnel pur (c'est-à-dire qu'il ne permet d'exprimer aucune sorte d'effet de bord) il est possible de l'enrichir avec une monade d'état [Wad95] pour être capable de l'utiliser dans le développement de programmes impératifs tels que les services d'un noyau de système d'exploitation. Pour ce faire, les structures et les composants modifiables par le noyau doivent être modélisés par l'état global manipulé par la monade. Cet état peut être ainsi consulté et mis à jour par les services du noyau.

Le langage de tactique L'assistant de preuve Coq fournit également un langage de tactique appelé L_{tac}. Ce dernier permet d'interagir facilement avec cet outil de démonstration. Ce langage propose un

ensemble de commandes appelées *tactiques* permettant d'établir progressivement le raisonnement. À partir d'un énoncé écrit sous la forme d'une formule logique, les tactiques peuvent être appliquées pour décomposer cet énoncé en un ensemble de formules moins compliquées. Ceci peut être effectué incrémentalement jusqu'à arriver à un ensemble de règles qui peuvent être admises par tout système logique (i.e. les axiomes). Cette étape marque donc la fin du raisonnement. L'utilisation de chaque tactique est contrôlée par les règles d'application de la tactique elle même. Si ses règles ne sont pas satisfaites alors une erreur se produit. Le raisonnement établi est exprimé sous la forme d'un terme de preuve. Ce terme est vérifié postérieurement par le noyau à la fin de la démonstration.

Le noyau Le noyau de l'assistant de preuve Coq constitue sa base de confiance. Celui-ci n'est chargé que de la vérification de la validité de la suite des règles logiques utilisées pour construire la démonstration. Cette décomposition permet de garder le noyau aussi simple que possible et donc réduire le risque d'introduire des erreurs dans l'implémentation de ce programme critique. En effet seule cette partie de cet outil correspond à la base de confiance de la fiabilité de la démonstration qui doit être validée.

2.2.2 Des noyaux de systèmes d'exploitation formellement vérifiés

Dans cette section nous présentons quelques travaux qui ont mis en place des approches de vérification formelle pour garantir un ensemble des propriétés sur les noyaux étudiés. Il ne s'agit pas ici de dresser une liste exhaustive de tous les travaux de vérification, mais simplement de cerner ceux qui sont considérés les plus marquants.

seL4 [KEH+09] est aujourd'hui considéré comme étant le micro-noyau qui a les résultats les plus satisfaisants du point de vue de sa vérification. Son implémentation est fondée sur la famille des micro-noyaux L4 [Lie95]. Il comprend environ 10 000 lignes de C et 600 lignes de code assembleur. Il gère principalement la mémoire virtuelle, l'ordonnancement et la communication entre les processus. Tous les autres modules sont exécutés en mode non privilégié. La sécurité fournie par le noyau est assurée par un mécanisme de capacités [DVH66] où chaque objet noyau tel qu'une page mémoire, un processus ou encore un bus de communication, possède une référence et un ensemble de droits d'accès à cet objet. Il s'agit d'une abstraction qui permet de gérer l'attribution des ressources matérielles aux différents programmes en cours d'exécution.

Le développement et la vérification du micro-noyau seL4 sont fondés sur l'approche de raffinement [DREB98, AL91] en plusieurs couches. La première est la spécification abstraite définissant l'interface du noyau et la modélisation du comportement des différents appels système exposés par le noyau. Ce modèle abstrait a été défini au-dessus de la spécification exécutable du noyau. Cette dernière est écrite en Isabelle/HOL [NPW02] et a été initialement définie sous la forme d'un prototype écrit en Haskell qui sert principalement à faciliter les échanges entre les équipes travaillant sur les aspects systèmes et formels. Le passage entre le prototype Haskell et la spécification exécutable se fait à travers une conversion automatique. Le code C a été implémenté manuellement à partir du prototype Haskell.

L'objectif primordial de ces travaux est d'établir une preuve formelle complète du bon fonctionnement du noyau en appliquant l'approche par raffinement à partir de la spécification abstraite et jusqu'au code binaire. Il est à noter que durant le processus de vérification, des mises à jour au niveau de l'implémentation sont nécessaires pour corriger des *bugs* ou adapter la spécification par rapport au comportement concret du noyau. Dans le cas de seL4, ceci nécessite souvent des modifications dans tous les niveaux de spécification et de preuve. La figure 2.5 présente les différents composants du schéma de développement et de vérification de seL4 ainsi que les mises à jour requises pour corriger un éventuel *bug* au niveau de l'implémentation.

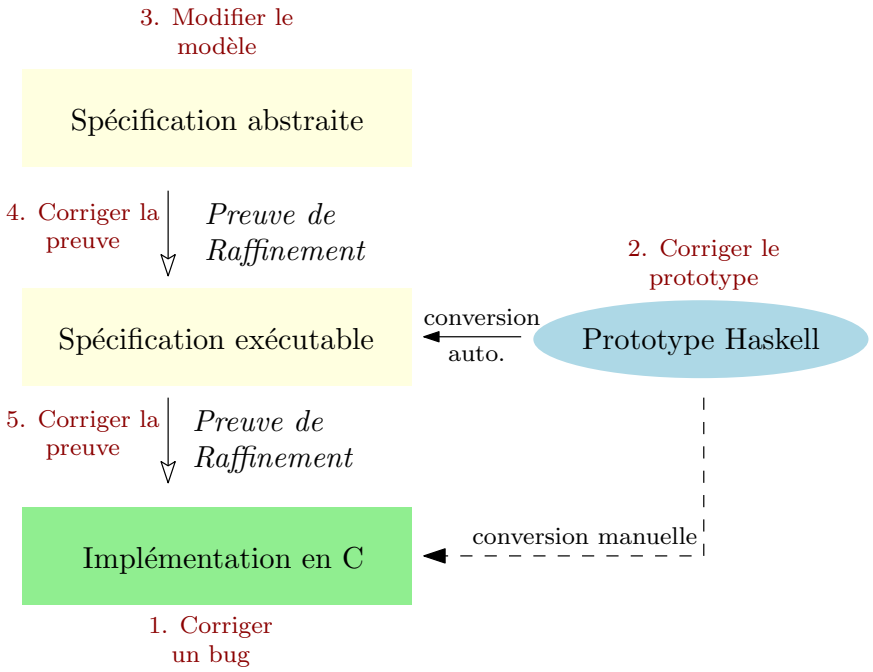


FIGURE 2.5 – Le schéma de développement et de vérification de seL4.

Verisoft est un autre projet de recherche élaboré par le centre aérospatial allemand qui visait dans un premier temps, dans le cadre du projet VerisoftI [AHL+08], l'étude de faisabilité de la vérification formelle de toute la chaîne de développement d'un système informatique y compris le matériel, le compilateur, le micro-noyau et les applications qui sont exécutées par-dessus. Dans un deuxième temps, le projet verisoftXT [BM10], successeur de VerisoftI a visé la vérification de plusieurs noyaux de système d'exploitation tels que PikeOS [BBBB10]. Il s'agit d'un micro-noyau de la famille L4. Il comprend environ 6 000 lignes de code C et assembleur et est fondé sur un modèle de partitionnement des ressources matérielles qui nécessite une configuration statique à l'état initial du système. Un mécanisme de communication entre les partitions et un modèle de mémoire virtuelle ont été vérifiés en s'appuyant sur l'approche de la preuve par raffinement entre un modèle abstrait et son implémentation. VerisoftXT s'intéressait également au développement et à la vérification de l'hyperviseur Hyper-V [LS09]. Ce dernier comprend environ 100 000 de lignes de code en C et 5 000 lignes de code en assembleur. Une partie importante de la preuve a été établie sur son modèle minimaliste Baby Hyper-V [AHPP10]. Ce modèle comprend principalement l'initialisation du modèle de partitionnement ainsi qu'une version simplifiée du gestionnaire de la mémoire virtuelle des partitions. Le processus de vérification de cet ensemble de noyaux a permis de développer des outils de vérification tels que VCC [CDH+09]. Ce dernier prend en paramètre un code C enrichi avec un ensemble d'annotations sous la forme de pré-conditions et d'assertions, puis le convertit en un ensemble des formules logiques qui peuvent être validées automatiquement [dMB08].

CertiKOS [Cer] est un environnement de développement de noyaux vérifiés. Il a été élaboré par des chercheurs à l'université de Yale et fait partie également du projet *DeepSpec* [Dee]. Ainsi, ce projet ne vise pas la vérification d'une implémentation particulière d'un micro-noyau mais plutôt fournit des outils et des approches pour développer d'une manière incrémentale un noyau de système d'exploitation tout en facilitant sa certification. En particulier, l'objectif ici est de réduire le coût de la preuve durant le processus de vérification ainsi qu'après une éventuelle extension du noyau concerné.

- Cette approche a été expérimentée dans un premier temps durant la vérification d'un modèle de gestion de la mémoire des processus BabyVMM [VS12] qui s'exécute sur un modèle simplifié du

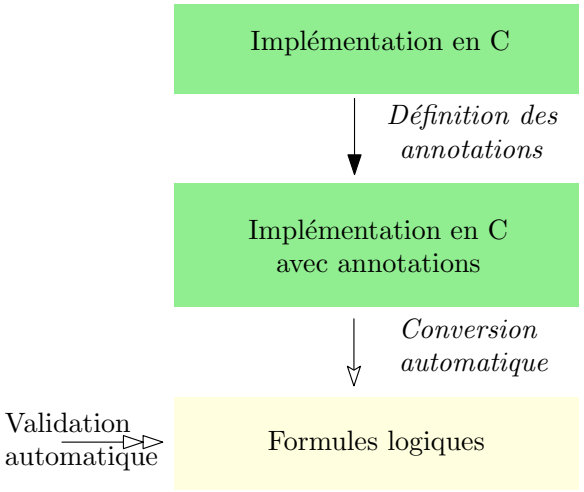


FIGURE 2.6 – Vérification avec VCC.

matériel. L'approche de vérification par raffinement a été mise en place. Le principe est de formaliser un modèle assez abstrait de BabyVMM, puis, à partir de ce modèle, de générer progressivement son implémentation en passant par plusieurs couches de spécification intermédiaires. Une preuve de raffinement est nécessaire afin de vérifier la conformité de chaque nouvelle couche de spécification par rapport à la spécification au-dessus (plus abstraite) jusqu'à arriver à la dernière couche, l'implémentation.

- Cet environnement a été également utilisé dans la vérification du bon fonctionnement des trois variantes *mCertiKOS-base*, *mCertiKOS-rz* et *mCertiKOS-emb* de l'hyperviseur *mCertiKOS* [GKR+15]. La formalisation en Coq de ce dernier est fondée sur 40 couches de spécification abstraites développées au-dessus de l'implémentation en C de cet hyperviseur. Son implémentation comprend environ 5 500 lignes de C et d'assembleur. Ce code est ensuite transformé en code machine en utilisant une version modifiée du compilateur certifié *CompCert* [Ler].

- Une version modifiée de *CertiKOS* est l'hyperviseur mC2 [GSC+16]. Il s'agit d'une extension du noyau *mCertiKOS* où le mécanisme de gestion de concurrence a été développé et vérifié. Il comprend environ 6 500 lignes de code de C et d'assembleur. Les propriétés visées par cette vérification concernent principalement la garantie du bon fonctionnement du noyau.

Prosper [DGK+13] est un noyau de séparation développé au-dessus d'un modèle simplifié de l'architecture matérielle ARMv7 où la propriété de communication entre deux entités séparées a été l'objectif principal de la vérification. Ce mécanisme s'appuie sur la propriété d'isolation entre les entités et un ordonnanceur permettant de partager le temps d'exécution entre elles. L'approche de vérification est fondée sur le modèle formel HOL4 de ARM [FM10]. Les preuves sont fondées sur la vérification du code machine en utilisant la logique de Hoare ainsi que le model-checking pour calculer et vérifier toutes les traces d'exécution possibles.

Xenon [FM11] est un hyperviseur fondé sur l'implémentation de l'hyperviseur Xen[BDF+03]. Ces travaux visent principalement l'étude de la faisabilité de la vérification d'un système existant assez complexe (environ 70 000 lignes de code). La formalisation de Xenon consiste à définir la spécification abstraite de l'interface des appels système de Xen et la propriété de sécurité permettant de garantir la séparation entre les machines virtuelles gérées par l'hyperviseur [MML+12]. L'utilisation de la preuve formelle dans ces travaux a rendu possible une analyse rigoureuse du comportement du noyau. En par-

ticulier, la gestion de flux d'information entre les machines virtuelles a été profondément révisée dans le but d'améliorer la sécurité garantie par l'hyperviseur.

KeyKOS [Har85], **Eros** [SSF99] et **Coyotos** [SDN+04] sont des micro-noyaux dont la politique de sécurité est fondée sur les capacités. En effet ce mécanisme a été largement étudié durant ces travaux dans le but de mettre en place une gestion de contrôle d'accès uniforme à toutes les ressources matérielles gérées par l'hyperviseur. La formalisation formelle du modèle de sécurité de *Eros* a été détaillé dans [SW00]. Ces travaux ont permis également le développement d'un langage dédié à l'implémentation des noyaux prouvables nommé *bitC* [SD08]. Ce dernier est un langage typé permettant de raisonner sur du code bas niveau.

Kit [Bev89] L'objectif principal de ce projet est de garantir et vérifier l'isolation entre des tâches où la communication entre elles est effectuée à travers le passage de messages. En particulier, un ordonnanceur permettant de gérer les temps d'exécution et un mécanisme de communication entre les tâches isolées ont été implémentés et vérifiés. Il a été démontré également que tous les appels système exposés par le noyau terminent, les tâches sont isolées et ces dernières ne peuvent jamais s'exécuter en mode privilégié. Les preuves sont effectuées au niveau du code machine en s'appuyant sur la logique de *Boyer-Moore* [BM88]. Ce travail est l'un des premiers projets de vérification complète d'un système exploitation. Toutefois, il s'agit d'un noyau simplifié : il n'y a pas de mécanisme de création de tâches et l'attribution de la mémoire est effectuée statiquement au démarrage du système.

Les travaux autours de la vérification formelle des noyaux des systèmes d'exploitation ont débuté dans les années 1970-1980 avec les noyaux **UCLA** [WKP80] et **PSOS** [FN79]. Leurs modèles de contrôle d'accès sont gérés par des capacités. Ces travaux représentent une tentative importante dans la vérification des systèmes critiques tels que les noyaux des systèmes d'exploitation. Plusieurs aspects tels que la complexité de la spécification des systèmes complexes et la mise en œuvre de leurs preuves ont été étudiés.

2.3 Contributions

Dans cette thèse j'ai étudié la possibilité du « co-design » d'un noyau et de sa preuve dans le but de réduire la complexité de la vérification de ses propriétés d'isolation. Mes travaux sont divisés en deux parties. Dans cette section j'explique brièvement les objectifs de chaque contribution. Tous les résultats seront détaillés progressivement dans les chapitres qui suivent.

2.3.1 Contribution 1 : MIMIC

La première partie a comme objectif principal de définir une approche permettant de vérifier formellement la propriété d'isolation mémoire d'un noyau de système d'exploitation. Pour ce faire, j'ai d'abord développé un modèle abstrait de micro-noyau appelé MIMIC (MIcrokernel Model In Coq). Ensuite j'ai formalisé et vérifié formellement sa propriété d'isolation mémoire. Ce modèle expose plusieurs primitives, notamment, la gestion de la mémoire physique du système et un ordonnanceur préemptif basé sur les interruptions. Plus précisément mes contributions consistent en :

- Un modèle formel de l'architecture matérielle nécessaire pour assurer l'isolation mémoire.

  Il contient principalement un modèle abstrait de MMU d'un seul niveau d'indirection et de CPU pour gérer les interruptions ;

- Un modèle formel, assez complet, de gestionnaire de mémoire permettant de configurer correctement la mémoire virtuelle des processus ;

- Un modèle formel d'un ordonnanceur préemptif fondé sur les interruptions ;
- Une description explicite des propriétés de cohérence nécessaires pour prouver les propriétés d'isolation mémoire assurées par le micro-noyau ;
- Une preuve formelle de la propriété d'isolation mémoire.

Le projet MIMIC est composé de 2 400 lignes de spécification réparties entre un modèle formel du matériel ainsi que celui du micro-noyau et 9 700 lignes de preuve.

Ces travaux ont été initialement dans les deux conférences suivantes :

- [JNGIC15] JFLA2015 (Journées Francophones des Langages Applicatifs)
- [JNGH16] TASE2016 (The 10th International Symposium on Theoretical Aspects of Software Engineering)

Puis, ils sont développés dans le journal suivant :

- [JNGH17] SCP (Science of Computer Programming).

Les sources et les preuves sont disponibles sur le site `https://github.com/jomaa/MIMIC`.

2.3.2 Contribution 2 : Preuve des propriétés d'isolation de Pip

Cette deuxième partie consiste à développer (lors de mes travaux avec mes encadrants) et vérifier formellement un noyau concret de système d'exploitation que nous avons appelé le proto-noyau Pip en s'appuyant sur la même approche de vérification développée par la première contribution. Il s'agit d'un noyau de système d'exploitation minimal où la minimisation de la taille est principalement motivée par la réduction du coût de la preuve mais aussi de la surface d'attaque. Lors de mes travaux avec mes encadrants ceci nous a amené à définir une nouvelle stratégie de « co-design » du noyau et de sa preuve. Ainsi, nous avons intégré cette approche de vérification dans le processus de sa conception et de son développement afin de guider le développeur dans le choix des services exposés par le noyau ainsi que dans la stratégie de son implémentation. Tous les services fournis par le noyau sont développés en Gallina. Cela représente un premier défi car un noyau est généralement implémenté dans un langage de bas niveau tel que C, contrairement à Coq qui utilise le langage fonctionnel Gallina. La vérification des propriétés d'isolation assurées par le noyau consiste à démontrer que tous les services exposés par le noyau garantissent les propriétés d'isolation mémoire. Cette stratégie de vérification m'a aidé à identifier clairement l'ensemble minimal des propriétés fonctionnelles permettant de garantir l'isolation mémoire et de les vérifier.

Ces travaux ont été publiés dans les conférences suivantes :

- [JTN+18] AVoCS2018 (18th International Workshop on Automated Verification of Critical Systems)
- [JHN18] COMPAS2018 (La conférence d'informatique en Parallélisme, Architecture et Système)
- [TNJC18] VSTTE2018 (10th Working Conference on Verified Software : Theories, Tools, and Experiments)

Les sources et les preuves sont disponibles sur le site `https://github.com/2xs/pipcore`.

Bibliographie

[AFOTH06] Jim ALVES-FOSS, Paul W OMAN, Carol TAYLOR et W Scott HARRISON : The MILS architecture for high-assurance embedded systems. *International journal of embedded systems*, pages 239–247, 2006.

[AHL+08] Eyad ALKASSAR, Mark A HILLEBRAND, Dirk LEINENBACH, Norbert W SCHIRMER et Artem STAROSTIN : The Verisoft approach to systems verification. *In Working Conference on Verified Software : Theories, Tools, and Experiments*, pages 209–224, 2008.

[AHPP10] Eyad ALKASSAR, Mark A HILLEBRAND, Wolfgang PAUL et Elena PETROVA : Automated verification of a small hypervisor. *In International Conference on Verified Software : Theories, Tools, and Experiments*, pages 40–54. Springer, 2010.

[AL91] Martín ABADI et Leslie LAMPORT : The existence of refinement mappings. *Theoretical Computer Science*, pages 253–284, 1991.

[BBBB10] Christoph BAUMANN, Bernhard BECKERT, Holger BLASUM et Thorsten BORMER : Ingredients of operating system correctness, lessons learned in the formal verification of pikeos. *Emb. World Conf., Nuremberg, Germany*, 2010.

[BDF+03] Paul BARHAM, Boris DRAGOVIC, Keir FRASER, Steven HAND, Tim HARRIS, Alex HO, Rolf NEUGEBAUER, Ian PRATT et Andrew WARFIELD : Xen and the art of virtualization. *In ACM SIGOPS operating systems review*, pages 164–177, 2003.

[Bev89] W.R. BEVIER : Kit : a study in operating system verification. *Software Engineering, IEEE Transactions on*, pages 1382–1396, 1989.

[BM88] Robert S BOYER et J Strother MOORE : A computational logic handbook. 1988.

[BM10] Bernhard BECKERT et Michał MOSKAL : Deductive verification of system software in the Verisoft XT project. *KI-Künstliche Intelligenz*, 24(1):57–61, 2010.

[CDH+09] Ernie COHEN, Markus DAHLWEID, Mark HILLEBRAND, Dirk LEINENBACH, Michał MOSKAL, Thomas SANTEN, Wolfram SCHULTE et Stephan TOBIES : VCC : A practical system for verifying concurrent C. *In International Conference on Theorem Proving in Higher Order Logics*, pages 23–42, 2009.

[Cer] CERTIFIED KIT OPERATING SYSTEM : Available at http://flint.cs.yale.e.

[CKH16] Jonathan CORBET et Greg KROAH-HARTMAN : Linux kernel development report, 2016. Available at https://www2.thelinuxfoundation.org/linux-kernel-development-report-2016.

[coq] *The Coq Proof Assistant*. Available at https://coq.inria.fr/.

[Dee] DEEPSPEC : Available at https://deepspec.org/.

[DGK+13] Mads DAM, Roberto GUANCIALE, Narges KHAKPOUR, Hamed NEMATI et Oliver SCHWARZ : Formal verification of information flow security for a simple ARM-based separation kernel. *In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 223–234. ACM, 2013.

[dMB08] Leonardo de MOURA et Nikolaj BJØRNER : Z3 : An efficient SMT solver. *In Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[DREB98] Willem-Paul DE ROEVER, Kai ENGELHARDT et Karl-Heinz BUTH : *Data refinement : model-oriented proof methods and their comparison*, volume 47. Cambridge University Press, 1998.

[DVH66] Jack B DENNIS et Earl C VAN HORN : Programming semantics for multiprogrammed computations. *Communications of the ACM*, pages 143–155, 1966.

[FM10] Anthony FOX et Magnus O MYREEN : A trustworthy monadic formalization of the ARMv7 instruction set architecture. *In International Conference on Interactive Theorem Proving*, pages 243–258, 2010.

[FM11] Leo FREITAS et John MCDERMOTT : Formal methods for security in the Xenon hypervisor. *International Journal on Software Tools for Technology Transfer*, pages 463–489, 2011.

[FN79] Richard J FEIERTAG et Peter G NEUMANN : The foundations of a provably secure operating system PSOS. *In Proceedings of the National Computer Conference*, pages 329–334, 1979.

[GDFR90] David B GOLUB, Randall W DEAN, Alessandro FORIN et Richard F RASHID : UNIX as an application program. *In UsENIX summer*, pages 87–95, 1990.

[GKR+15] Ronghui GU, Jérémie KOENIG, Tahina RAMANANANDRO, Zhong SHAO, Xiongnan Newman WU, Shu-Chun WENG, Haozhong ZHANG et Yu GUO : Deep specifications and certified abstraction layers. pages 595–608, 2015.

[GSC+16] Ronghui GU, Zhong SHAO, Hao CHEN, Xiongnan (Newman) WU, Jieung KIM, Vilhelm SJÖBERG et David COSTANZO : CertiKOS : An extensible architecture for building certified concurrent OS kernels. *In OSDI*, pages 653–669, 2016.

[Har85] Norman HARDY : Keykos architecture. *SIGOPS Oper. Syst. Rev.*, pages 8–25, 1985.

[Inf07] INFORMATION ASSURANCE DIRECTORATE : Protection profile for separation kernels in environments requiring high robustness. *US Government*, 2007.

[JHN18] Narjes JOMAA, Samuel HYM et David NOWAK : La conception d'un noyau orientée par sa preuve d'isolation mémoire. *In Compas 2018*, 2018.

[JNGH16] Narjes JOMAA, David NOWAK, Gilles GRIMAUD et Samuel HYM : Formal proof of dynamic memory isolation based on MMU. *In 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*, pages 73–80, 2016.

[JNGH17] Narjes JOMAA, David NOWAK, Gilles GRIMAUD et Samuel HYM : Formal proof of dynamic memory isolation based on MMU. *Science of Computer Programming*, 2017.

[JNGIC15] Narjes JOMAA, David NOWAK, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY : Preuve formelle d'isolation mémoire dynamique à base de MMU. *In Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, pages 297–300, 2015.

[JTN+18] Narjes JOMAA, Paolo TORRINI, David NOWAK, Gilles GRIMAUD et Samuel HYM : Proof-oriented design of a separation kernel with minimal trusted computing base. *In 18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*, 2018.

[KEH+09] Gerwin KLEIN, Kevin ELPHINSTONE, Gernot HEISER, June ANDRONICK, David COCK, Philip DERRIN, Dhammika ELKADUWE, Kai ENGELHARDT, Rafal KOLANSKI, Michael NORRISH, Thomas SEWELL, Harvey TUCH et Simon WINWOOD : seL4 : Formal verification of an OS kernel. *In Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[Lat85] Donald C LATHAM : Trusted computer system evaluation criteria (orange book). *Department of Denfense*, 1985.

[Ler] X. LEROY : The CompCert verified compiler. Available at http://compcert.inria.fr/.

[Lie93] Jochen LIEDTKE : A persistent system in real use-experiences of the first 13 years. *In Object Orientation in Operating Systems, 1993., Proceedings of the Third International Workshop on*. IEEE, 1993.

[Lie95] J. LIEDTKE : On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, pages 237–250, 1995.

[LS09] Dirk LEINENBACH et Thomas SANTEN : Verifying the microsoft hyper-V hypervisor with VCC. pages 806–809, 2009.

[MML+12] J MCDERMOTT, B MONTROSE, Margery LI, James KIRBY et M KANG : The Xenon separation VMM : Secure virtualization infrastructure for military clouds. *In MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[Nat] NATIONAL SECURITY AGENCY : Available at https://www.nsa.gov/.

[NPW02] Tobias NIPKOW, Lawrence C PAULSON et Markus WENZEL : *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[PL01] NSA PETER LOSCOCCO : Integrating flexible support for security policies into the linux operating system. *In Proceedings of the FREENIX track : USENIX Annual Technical Conference*, 2001.

[RM87] Marc ROZIER et José Legatheaux MARTINS : The chorus distributed operating system : some design issues. *In Distributed Operating Systems*. Springer, 1987.

[Rus81] John M RUSHBY : *Design and verification of secure systems*. ACM, 1981.

[SD08] Jonathan SHAPIRO et Swaroop Sridhar M Scott DOERRIE : The origins of the bitc programming language. Rapport technique, Citeseer, 2008.

[SDN+04] Jonathan SHAPIRO, Michael S. DOERRIE, Eric NORTHUP, Swaroop SRIDHAR et Mark MILLER : Towards a verified, general-purpose operating system kernel. Rapport technique, 2004.

[SSF99] Jonathan S SHAPIRO, Jonathan M SMITH et David J FARBER : *EROS : a fast capability system*, volume 33. ACM, 1999.

[SW00] Jonathan S SHAPIRO et Sam WEBER : Verifying the EROS confinement mechanism. *In Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 166–176. IEEE, 2000.

[TNJC18] Paolo TORRINI, David NOWAK, Narjes JOMAA et Mohamed Sami CHERIF : Formalising executable specifications of low-level systems. *In 10th Working Conference on Verified Software : Theories, Tools, and Experiments (VSTTE 2018)*, 2018.

[TVRVS+90] Andrew S TANENBAUM, Robbert VAN RENESSE, Hans VAN STAVEREN, Gregory J SHARP et Sape J MULLENDER : Experiences with the amoeba distributed operating system. *Communications of the ACM*, 1990.

[VS12] Alexander VAYNBERG et Zhong SHAO : Compositional verification of a baby virtual memory manager. *In Certified Programs and Proofs*, pages 143–159. 2012.

[Wad95] Philip WADLER : Monads for functional programming. *In International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[WKP80] Bruce J. WALKER, Richard A. KEMMERER et Gerald J. POPEK : Specification and verification of the ucla unix security kernel. *Commun. ACM*, pages 118–131, 1980.

Chapitre 3

Étude préliminaire : MIMIC

Sommaire

Ce chapitre présente les détails d'une étude préliminaire sur la faisabilité de l'établissement de la preuve sur le code. Cette étude est effectuée sur un modèle simplifié d'un micro-noyau écrit en Coq que nous avons appelé MIMIC [Jom17]. Pour avoir une vue globale sur l'évolution dynamique de l'état de la machine gérée par ce noyau, nous avons également défini un modèle formel de l'architecture matérielle nécessaire pour assurer l'isolation mémoire. La vérification formelle de la propriété d'isolation a été effectuée à travers les invariants en utilisant la logique de Hoare définie au-dessus d'une monade d'état.

3.1 Modèle formel d'un micro-noyau

Dans cette section nous détaillons le modèle formel spécifiant l'évolution dynamique de l'état du système orientée par un mécanisme de base des interruptions.

3.1.1 La spécification de l'état du système

Dans les implémentations réelles des systèmes exploitation, l'état du système est très complexe et inclut principalement l'état de chaque composant matériel ainsi que toutes les structures internes gérées par le noyau. Dans notre modèle de micro-noyau, l'état s est un enregistrement contenant plusieurs composants que nous avons répartis en deux catégories : l'état matériel et l'état logiciel. Dans chaque catégorie nous définissons seulement les composants qui sont indispensables pour prouver les propriétés qui nous intéressent.

3.1.1.1 Les composants matériels

Les entités suivantes définissent l'état du CPU et de la mémoire physique :

currentptp(s) : le numéro de la page physique unique (puisque dans ce modèle préliminaire nous considérons un MMU avec un seul niveau d'indirection) contenant la table de pages du processus en cours d'exécution ;

kernelMode(s) : une valeur booléenne permettant de définir le mode d'exécution (le mode utilisateur et le mode noyau) ;

currentpc(s) : l'adresse de la prochaine instruction à exécuter ;

interrupts(s) : un modèle simple définissant le flot d'interruptions matérielles et logicielles. Il s'agit d'une liste infinie contenant un élément pour chaque tic de l'horloge. Chaque élément indique si une interruption matérielle ou logicielle a été déclenchée en précisant son numéro si elle existe. Avant l'exécution d'une nouvelle instruction le système vérifie la présence d'une nouvelle interruption, levée pendant l'exécution de l'instruction précédente, en consommant simplement le premier élément dans cette liste ;

memory(s) : c'est la mémoire physique modélisée sous la forme d'une liste. Elle contient principalement les structures de données nécessaires pour gérer la mémoire. La première structure est celle utilisée par le MMU pour traduire les adresses virtuelles vers des adresses physiques et la deuxième contient les références vers les pages physiques libres.

3.1.1.2 Les composants logiciels

Durant l'exécution, notre système sauvegarde d'autres données critiques en mémoire physique qui sont accessibles uniquement en mode noyau. Concrètement, la plupart des implémentations sauvegardent ces données dans des zones mémoires réservées au noyau et donc inaccessibles par les proces-

sus utilisateur. Nous modélisons ces données indépendamment de la mémoire physique afin de simplifier la preuve.

processes(*s*) : la liste contenant les processus en cours d'exécution. Un processus est modélisé par un enregistrement que nous avons appelé *process*. Il contient la référence vers sa table de pages (i.e. *ptp*) et l'adresse de sa prochaine instruction à exécuter (i.e. *pc*). Quand le noyau rend la main à un processus P, il modifie la valeur de *currentptp*(s) de façon à ce qu'il pointe vers la table de pages de *P* et remplace la valeur de *currentpc*(s) par *pc*(P) ;

code(*s*) : la liste contenant l'ensemble d'instructions à exécuter par le noyau ainsi que par chaque processus dans *processes*(*s*). Le but principal de notre preuve est de vérifier l'isolation mémoire des données ; la preuve de l'isolation mémoire du code serait similaire ;

intr_table(*s*) : correspond à la table de descripteur des interruptions. C'est une liste qui associe à chaque numéro d'interruption son propre code : chaque position dans la liste correspond à un numéro d'interruption tel que la valeur associée représente l'adresse de la première instruction à exécuter par l'interruption dans *code*(*s*) ;

stack(*s*) : c'est la pile du noyau nécessaire pour sauvegarder le contexte du processus en cours d'exécution lorsqu'une interruption est levée ;

first_free_page(*s*) : c'est la référence vers la première page libre. Une page est dite libre si elle n'est pas allouée à un processus. Dans notre modèle, nous utilisons les pages libres afin de déterminer la liste des pages libres. Il s'agit de définir une liste chaînée de pages telles que la première information de chaque page contient la référence vers la page libre suivante. Cet encodage est illustré par la figure 3.1. Ainsi, le noyau doit uniquement garder la référence vers la première page libre afin de gérer le reste de la liste. A l'initialisation, toutes les pages non allouées sont initialisées dans la liste des pages libres.

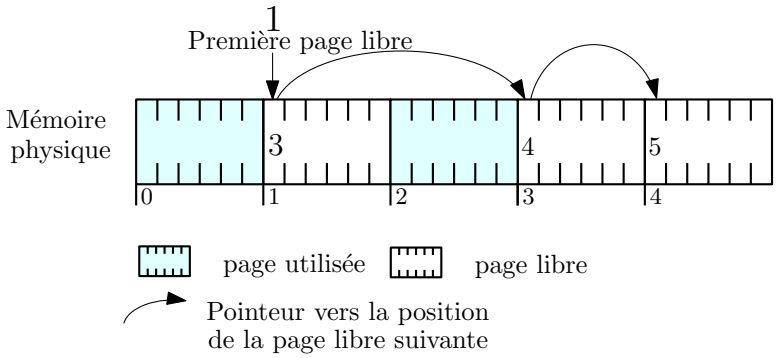


FIGURE 3.1 – Gestion des pages libres en mémoire.

3.1.2 L'évolution dynamique du système

Durant le cycle de vie d'un système, plusieurs événements peuvent se déclencher, tels que la création des processus ainsi que leurs exécutions, entraînant plusieurs changements de l'état des différents composants matériels et logiciels maintenus par le noyau. De ce fait, le micro-noyau doit fournir des mécanismes assurant des transformations correctes dans le but de remettre le système dans un état cohérent. Dans cette section, nous détaillons la formalisation de l'évolution dynamique de l'état dans un langage fonctionnel en s'appuyant principalement sur une monade d'état.

3.1.2.1 La monade H

Gallina, le langage de spécification de Coq, est un langage purement fonctionnel qui ne permet pas des interactions avec l'environnement externe. Par exemple, effectuer un changement d'état ou gérer des exceptions. Cependant, afin de bénéficier à la fois des avantages de ce langage fonctionnel et de l'efficacité de la programmation impérative, nous modélisons notre micro-noyau en Gallina et nous utilisons une monade d'état, que nous avons appelée la monade H (*Hardware monad*). Une monade est une notion mathématique issue de la théorie des catégories [Mog91] et utilisée dans les langages fonctionnels pour enrober les effets de bords dans un type abstrait approprié. Elle permet ainsi de représenter la séquentialité des instructions (des langages impératifs) dans le langage fonctionnel de l'assistant de preuve Coq et offre un moyen pour définir un état modifiable par les fonctions d'un système à état tel qu'un système d'exploitation. La monade H consiste concrètement en un constructeur de type `H(A)` et deux opérations `ret` et `bind`.

- `H(A)` : est le type d'une opération monadique. Celle-ci retourne un résultat de type monadique :

  ```
  Definition H (A :Type) : Type := state → result (A * state).
  ```

  où `state` est le type de l'état du système et `result(X)` est le resultat retourné par une opération monadique. `result` est un type inductif défini avec trois constructeurs : `val` pour retourner une valeur de type `A` et un nouvel état de type `state` , ainsi que `hlt` et `undef` pour définir, respectivement, l'arrêt du système et les comportements non définis (dans ces cas-là il n'y a pas de nouvel état) ;

  ```
  Inductive result (A : Type) : Type :=
  | val : A → result A
  | hlt : result A
  | undef : result A.
  ```

- `ret(A)` : est une opération monadique permettant de retourner une valeur de type `A` ;

  ```
  Definition ret {A : Type}(a : A) : H A := fun s ⇒ val (a, s).
  ```

- `bind` permet la mise en séquence d'un ensemble d'opérations monadiques. Il est défini de la manière suivante :

  ```
  Definition bind {A B : Type} (m : H A)(f : A → H B) : H B :=
    fun s ⇒ match m s with
      | val (a, s') ⇒ f a s'
      | hlt ⇒ hlt
      | undef⇒ undef
      end.
  ```

  la première opération monadique à exécuter est `m` qui peut retourner une valeur `a` de type `A` et un nouvel état `s'` puis `f` sera calculée en fonction de `s'` et `a`. Cette dernière est, également, une fonction monadique qui retourne un nouvel état et une valeur de type `B`.

  Pour simplifier l'utilisation de la fonction `bind` nous avons utilisé le système de notations fourni par Coq. La notation `perform x := m in e` consiste en la mise en séquence par `bind` de `m` et `e` qui peut dépendre de la valeur `x` retournée par `m`. La notation `m1 ;; m2` consiste en la mise en séquence par `bind` de `m1` et `m2` quand `m2` ne dépend pas de la valeur retournée par `m1`. Le point intéressant ici est qu'en utilisant l'opération `bind` pour la mise en séquence, le passage de l'état est implicite, caché sous un type abstrait.

Dans la suite nous utilisons `s` pour nommer un état de type `state`. Ce dernier consiste en un enregistrement composé de l'ensemble de tous les composants logiciels et matériels détaillés plus haut. Il est à noter que dans ce modèle nous distinguons trois catégories d'opérations monadiques :

un composant matériel : modélisant le comportement d'un composant matériel. Cette catégorie d'opérations est nécessaire pour définir, par exemple, la fonction de traduction établie par le MMU qui est indispensable pour la modélisation du gestionnaire de mémoire ;

une instruction : correspond exactement à une seule instruction CPU ;

une routine : est une séquence non-interruptible d'un ensemble d'instructions exécutées par le noyau.

3.1.2.2 Les appels système et interruptions

Notre modèle de CPU, particulièrement la partie qui gère les appels système et les interruptions, est conçue dans un contexte assez générique. La figure 3.2 adaptée de [Gho11] résume le modèle de l'évolution dynamique du système avec et sans interruptions. La spécification de cette transformation est illustrée par la figure 3.3.

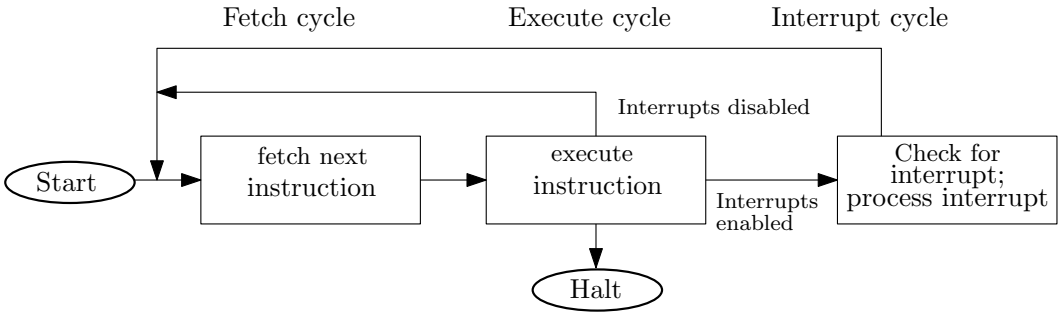


FIGURE 3.2 – L'évolution dynamique du système.

Composant matériel step : H(unit)
Action:
Vérifier la présence d'une interruption;
if *une interruption n a été levée (fech_interrupt)* **then**
  interrupt(*n*) ;
else
  i ← *fetch_instruction* ;
  incrémenter *currentpc* ;
  execute(*i*);
end

FIGURE 3.3 – La spécification de l'évolution dynamique de l'état du système.

Nous identifions trois étapes principales dans le cycle de vie d'un système :

1. **Fetch instruction** : permet de lire ou extraire l'instruction à partir de la mémoire physique. Le composant matériel *fetch_instruction* correspond à cette étape ;

   Il est à noter que dans le cas où la condition n'est pas vraie et il n'y a pas de bloc *sinon*, nous retournons soit une valeur de type *unit* soit une valeur par défaut ou bien nous exécutons l'instruction *halt* pour arrêter le système.

Composant matériel fetch_instruction : H(instr)
Action:
if *currentpc*(*s*) *est une position valide dans code*(*s*) **then**
  retourner l'instruction ;
end

FIGURE 3.4 – Extraction d'instruction.

2. **Execute instruction** : permet d'exécuter une instruction CPU, par exemple, cette instruction pourra demander au CPU de déterminer une certaine adresse physique afin de la charger ou sauvegarder des données dans la mémoire (par exemple un processus qui sauvegarde une valeur dans la mémoire physique en utilisant la routine *write* (voir figure 3.5). En effet avant de sauvegarder une donnée dans la mémoire le MMU doit déterminer l'adresse physique qui correspond à l'adresse virtuelle fournie par le processus, s'il existe une configuration pour cette adresse virtuelle ;

Instruction write : integer → integer → H(unit)
Input : *val* : la valeur à sauvegarder
vaddr : l'adresse virtuelle vers laquelle la valeur sera sauvegardée
Action:
virtual address *vaddr*;
paddr ← *translate*(*vaddr*) ;
if *paddr n'est pas une exception* **then**
  write_phy(*val*, *paddr*) ; *(*écrire val à l'adresse physique paddr*)*
end

FIGURE 3.5 – Écrire une valeur dans la mémoire.

Composant matériel translate : integer → H(integer + exception)
Input : *vaddr* : une adresse virtuelle
Output: une adresse physique ou une exception
Action :
if l'adresse virtuelle *vaddr* est valide **then**
  calculer l'adresse de l'entrée dans la table de pages *pte* et la position dans cette table qui correspond à *vaddr*;
  if il existe une page physique mappée dans *pte* et (*kernel_mode*(*s*) = *true* ou bien *kernel_only*(*pte*) = 1) **then**
    calculer l'adresse physique ; retourner l'adresse physique ;
  end
end

FIGURE 3.6 – Traduire une adresse virtuelle vers une adresse physique.

3. **Check for interrupt** : permet de vérifier s'il y a une interruption à gérer avant de passer à l'exécution de l'instruction suivante. On modélise cette action par le composant matériel `fech_interrupt` (voir figure 3.7). En effet, ce composant doit d'abord vérifier l'existence d'une interruption en consultant le premier élément de la liste infinie qui modélise le flux des interruptions. Si une interruption est levée, le composant matériel *interrupt*, illustré par la figure 3.8, sauvegarde le contexte

courant sur la pile du noyau (i.e. *stack*(*s*)), puis le CPU passe en mode noyau et branche vers le code du gestionnaire de cette interruption. Dans ce cas, le matériel identifie le gestionnaire approprié en utilisant le numéro d'interruption comme position dans `intr_table`. Afin de reprendre correctement l'exécution du processus courant après une interruption, son contexte sera restauré à partir de la pile du noyau en exécutant l'instruction `return_from_interrupt` (voir figure 3.9).

Composant matériel fetch_interrupt : H(option integer)
Action:
retirer le premier élément de la liste *interrupts*(*s*);
retourner le premier élément de la liste *interrupts*(*s*);

FIGURE 3.7 – Vérifier l'existence d'une interruption.

Composant matériel *interrupt* : integer → H(unit)
Input : *n* :le numéro d'interruption à exécuter
Action:
Empiler *currentpc* et le mode d'exécution sur la pile du noyau;
Passer en mode privilégié;
Récupérer l'adresse du gestionnaire d'interruption en fonction de *n*;
Affecter à *currentpc* l'adresse du gestionnaire d'interruption;

FIGURE 3.8 – Gérer une interruption.

Composant matériel *return_from_interrupt* : H(unit)
Action:
Retirer le premier élément dans la pile;
restaurer le mode d'exécution du programme courant;
restaurer *currentpc*;

FIGURE 3.9 – Reprendre l'exécution après une interruption.

Les processus invoquent les appels système afin d'interagir avec le micro-noyau en levant une interruption logicielle. Dans notre modèle on appelle cette interruption *trap* (voir figure 3.10). Le traitement de cette interruption est similaire à celui des interruptions matérielles.

Instruction *trap* : integer → H(unit)
Input : *n* : le numéro d'interruption à exécuter
Action:
incrémenter *currentpc*;
interrupt(*n*);

FIGURE 3.10 – Lever une interruption logicielle.

3.1.3 La formalisation de la propriété d'isolation

Afin de respecter le modèle du MMU, le noyau doit associer à chaque processus sa propre configuration de mémoire virtuelle. Dans ce modèle de micro-noyau nous nous intéressons à un MMU avec un seul niveau d'indirection. Le modèle des entrées de table de pages (PTE) suit exactement la description fournie par la section 2.1.3. Ainsi, chaque PTE correspond à une adresse virtuelle et peut contenir le numéro de la page physique et quelques bits de contrôle d'accès tels que le bit *present* et le bit *kernel_only*. Le premier vaut 1 s'il existe une page mappée dans cette entrée et le deuxième vaut 0 si cette page physique est accessible uniquement par le noyau. Malgré le rôle important du MMU pour contrôler l'accès à la mémoire physique, ce composant matériel n'est pas capable d'assurer la séparation entre les processus sans coopération avec le noyau. En effet, si les tables de pages ne sont pas correctement configurées alors la fonction de traduction traduira l'adresse virtuelle vers une adresse physique qui est déjà associée à un autre processus.

Prenons l'exemple de l'opération *write*, illustrée par la figure 3.5. Cette instruction permet au processus courant d'écrire une valeur `val` dans l'adresse virtuelle *vaddr*. Pour réaliser cette action, le micro-noyau délègue le contrôle d'accès à la fonction de traduction du MMU (voir figure 3.6) afin de calculer l'adresse physique *paddr* qui correspond à l'adresse virtuelle *vaddr* en s'appuyant sur la table de pages du processus concerné. Selon le mécanisme de traduction, le MMU traduira cette adresse virtuelle vers l'adresse physique sans lever aucune exception même dans le cas où la même page physique est mappée dans la table de pages d'un autre processus. Par conséquent, le processus courant peut modifier le contenu d'une adresse physique appartenant à un autre processus. Donc, pour assurer l'isolation, le micro-noyau doit garantir que toutes les tables de page sont correctement configurées de telle sorte qu'un processus ne puisse pas accéder à la mémoire d'un autre. C'est exactement ce qui est défini par la propriété d'isolation mémoire (voir Definition 1). Intuitivement, le but de la preuve de cette propriété est de montrer que pour chaque état `s`, il n'y a pas d'interférence entre deux processus exécutables : autrement dit, si `P1` et `P2` sont deux processus exécutables alors quelle que soit la page attribuée à `P1` elle est différente de toutes les pages attribuées à `P2`.

Definition 1 (Isolation mémoire)**.** *Un état s vérifie la propriété d'*isolation *si et seulement si quels que soit $\mathrm{P}_1, \mathrm{P}_2 \in$ processes(s) où $\mathrm{P}_1 \neq \mathrm{P}_2$ (i.e. ptp$(\mathrm{P}_1) \neq$ ptp(P_2)) et quel que soit $p \in$ UsedPages(P_1), alors $p \notin$ UsedPages(P_2), tel que*

processes(s) est l'ensemble des processus exécutables défini par l'état s ;

ptp(P_i) est le numéro de la page physique représentant la table de pages du processus P_i ;

UsedPages(P_i) est la liste de toutes les pages physiques attribuées au processus P_i. Elle contient toutes les pages référencées par la table de pages *ptp*(P_i) ainsi que la page *ptp*(P_i).

3.1.4 La formalisation des propriétés de cohérence

Nous avons mentionné dans le chapitre précédent que nos travaux se concentrent principalement sur la vérification des propriétés d'isolation mémoire assurées par le noyau d'un système d'exploitation. Nos expérimentations sur ce modèle abstrait ont montré que ceci nécessite également la vérification d'autre propriétés que nous avons appelées les propriétés de cohérence. Ces propriétés sont primordiales pour prouver la propriété d'isolation. Elles capturent précisément plusieurs propriétés sur le bon fonctionnement du noyau, la cohérence de l'état du matériel ainsi que la cohérence des données sauvegardées par le noyau. Le but de nos travaux ne consiste pas à établir la preuve de toutes les propriétés de cohérence possibles. Nous nous intéressons uniquement à celles qui sont nécessaires pour prouver que la propriété d'isolation est préservée. Cet ensemble de propriétés de cohérence permet de mettre en avant toutes les données et composants critiques et indispensables pour la sécurité. Dans cette section,

nous expliquons et justifions les différentes propriétés de cohérence. Nous définissons deux catégories principales : cohérence logicielle et cohérence matérielle.

Les définitions de plusieurs propriétés de cohérence sont liées à la liste des pages libres, nous définissons donc la notation suivante :

Notation. *Étant donné un état s, FreePageList(s) est la liste chaînée définissant l'ensemble de toutes les pages libres.*

Cette liste est encodée en mémoire et représente un composant logiciel important dans la sécurité du noyau.

3.1.4.1 cohérence logicielle

Toutes les pages référencées par la liste des pages libres sont réellement libres

Cette propriété de cohérence REALLY_FREE assure que toutes les pages libres ne sont pas mappées dans l'espace d'adressage d'un processus exécutable.

Definition 2. *Étant donné un état s, la propriété* REALLY_FREE(*s*) *est vérifiée si et seulement si, quel que soit* $p \in FreePageList(s)$, $p \notin AllUsedPages(s)$ *et* $p < nb\_pages$.

où *nb_pages* est le nombre de page de la mémoire physique et *AllUsedPages*(*s*) est la liste de pages associées à tous les processus dans la liste *processes*(*s*).

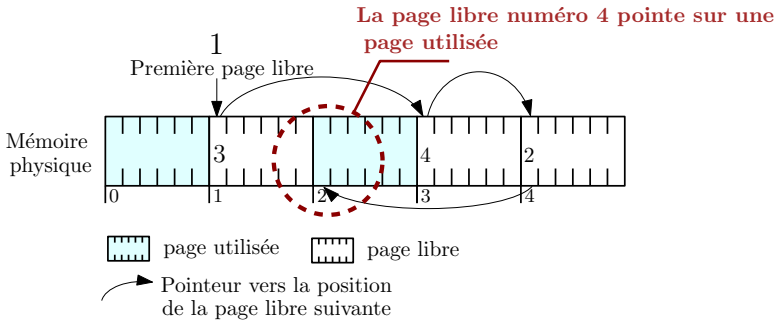


FIGURE 3.11 – Un contre-exemple pour REALLY_FREE.

Ajouter la propriété REALLY_FREE à la liste des propriétés de cohérence permet de vérifier que toutes les pages identifiées dans la liste des pages libres ne sont pas mappées dans l'espace d'adressage des processus exécutables. Par exemple, sans cette propriété nous ne pouvons pas prouver que la routine *add_pte* (qui ajoute une entrée dans la table de pages d'un processus, voir figure 3.12) préserve l'isolation. En effet, cette fonction récupère la première page libre dans la liste *FreePageList*(s) et l'ajoute dans la table de pages d'un processus. Si cette page fait déjà partie de la liste des pages associées à un autre processus (comme illustré par la figure 3.11), l'exécution de *add_pte* pourra engendrer un état dans lequel les tables de pages de deux processus différents référencent la même page physique, donc la propriété d'isolation ne serait plus garantie.

Pas de cycle dans la liste des pages libres

Cette propriété de cohérence NOT_CYCLIC signifie qu'aucune page libre n'apparaît plus qu'une fois dans la liste *FreePageList*(s).

Definition 3. *Étant donné un état s, la propriété* NOT_CYCLIC(*s*) *est préservée si et seulement si quel que soit p,* $nb\_occurrences(p, FreePageList(s)) \leq 1$.

Routine *add_pte* : integer → integer → H(unit)
Input : *permission* : les droits d'accès de la nouvelle page à mapper
index : la position de l'entrée dans la table de pages
Action:
if *permission* and *index* sont valides **then**
  pte ← récupérer l'entrée à la position *index*;
  if il y a une page mappée dans *pte* **then**
    supprimer le contenu de *pte*;
  end
  allouer une nouvelle page physique *p* ;
  ajouter un nouveau mapping dans *pte* qui correspond à *p* et *permission*;
end

FIGURE 3.12 – Ajouter une entrée dans une table de pages.

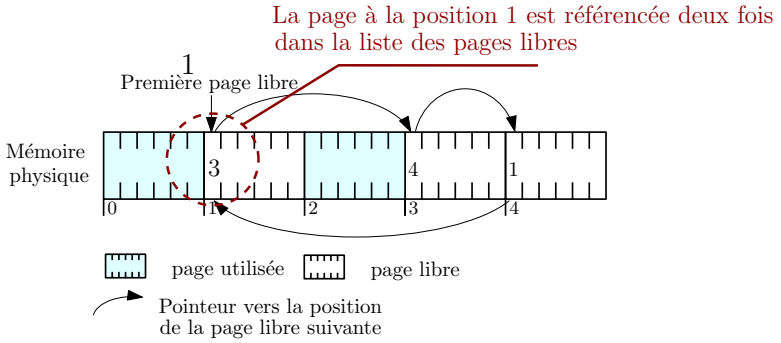


FIGURE 3.13 – Un contre-exemple pour NOT_CYCLIC.

où $nb\_occurrences(p, l)$ est le nombre d'occurrences de p dans la liste l.

Comme détaillé ci-dessus, les pages libres sont référencées dans une liste chaînée de pages libres codée en mémoire physique. Si cette propriété n'est pas préservée par le noyau, la fonction *add_pte* risque d'allouer deux fois la même page physique (pour deux processus différents ou pour le même). La propriété d'isolation ne serait alors plus vraie. La figure 3.13 illustre ce contre-exemple.

Pas de redondance dans la liste des pages utilisées par un processus

La propriété de cohérence suivante est NODUPLIC_PROCESSPAGES.

Definition 4. *La propriété de cohérence* NODUPLIC_PROCESSPAGES *est vérifiée par un état s si et seulement si quel que soit* $P \in processes(s)$*, il n'y a pas de redondance dans UsedPages*(P).

Cette propriété est nécessaire pour prouver que la routine *remove_pte* (voir figure 3.14) préserve la propriété d'isolation. En effet, elle supprime le contenu d'une entrée de table de pages et libère la page physique sauvegardée dans cette entrée en l'ajoutant dans la liste de pages libres. Après l'exécution de *remove_pte* la page physique p doit être réellement libre. Par contre si une autre entrée dans la table de page du même processus référence cette même page physique, après l'exécution de *remove_pte* cette page sera à la fois libre et associée à ce processus. De ce fait, un autre processus pourra allouer la même page p et la propriété d'isolation ne sera plus vraie. La figure 3.15 montre un état inconsistant produit après l'exécution de *remove_pte* tel que une page physique est mappée deux fois par le même processus.

Routine *remove_pte* : integer → H(unit)
Input : *vaddr* : l'adresse virtuelle à libérer
Action:
index ← la position de l'entrée de pages de *vaddr*;
if *index* est valide et il y une page mappée à la position *index* **then**
    Supprimer le contenu de l'entrée et retourner la page *p* mappée dans cette entrée;
    Dans *p* écrire la valeur de la première page libre;
    Remplacer la première page libre de *s* par la valeur *p* et retourner *p*;
end

FIGURE 3.14 – libérer une adresse virtuelle.

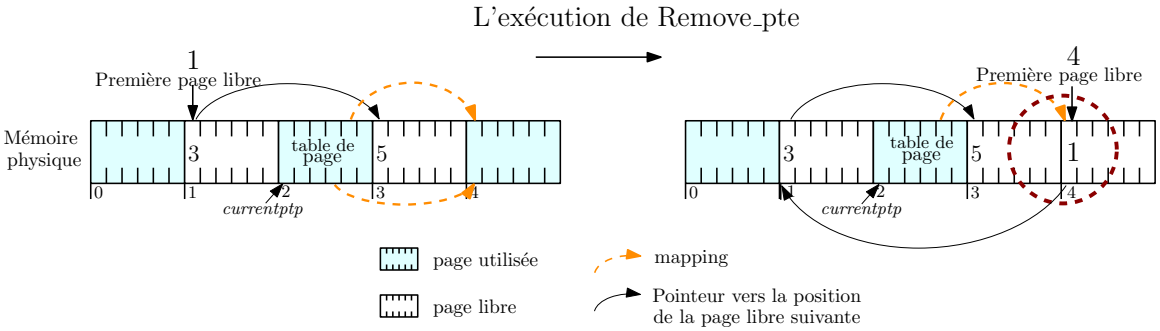


FIGURE 3.15 – Un contre-exemple pour NODUPLIC_PROCESSPAGES.

La table de pages courante appartient à un processus

Cette propriété de cohérence que nous avons appelée CURRPROCESS_INPROCESSLIST (voir définition 5) assure que le numéro de la page sauvegardé dans le composant matériel *currentptp*(*s*) correspond à la référence de la table de pages d'un processus (le processus courant) dans la liste des processus.

Definition 5. *Étant donné un état s, la propriété* CURRPROCESS_INPROCESSLIST(*s*) *est vraie si ptp*(P) = *currentptp*(*s*).

Cette propriété est essentielle pour prouver que les routines qui dépendent de cette partie de l'état (i.e. le processus courant) préservent l'isolation. C'est le cas par exemple de l'ordonnanceur qui change le contexte d'exécution en activant un autre processus dans *processes*(s). Cela peut se faire via la routine *switch_process* qui exécute séquentiellement les routines *save_process* (voir figure 3.16) et *restore_process* (voir figure 3.17). La première fonction supprime le premier processus (le processus courant) de la liste des processus et l'ajoute à la fin de cette liste avec la référence vers sa table de pages et la valeur du *pc* courant. La deuxième routine choisit le processus suivant dans la liste *processes*(s) comme nouveau processus courant. Elle met à jour principalement la valeur de *currentptp*(s) de façon à ce qu'il pointe vers la table des pages du processus choisi et passe à sa prochaine instruction à exécuter. La propriété d'isolation mémoire nécessite que les pages utilisées par deux processus différents soient différentes. Donc quand la routine *save_process* ajoute un processus dans la liste de processus, toutes les pages physiques mappées dans l'espace d'adressage de ce processus doivent être différentes de toutes les pages physiques mappées dans les espaces d'adressage des autres processus. Ainsi, pour prouver l'isolation, il suffit que le processus courant soit dans la liste des processus. Dans notre modèle, c'est le premier de la liste *processes*(s). Il faut noter que cette propriété implique que la liste de processus ne peut pas être vide même à l'état initial.

Routine *save_process* : H(unit)
Action:
Supprimer le premier processus de la liste des processus;
Ajouter le processus courant à la fin de la liste des processus;

FIGURE 3.16 – Sauvegarder le contexte du processus courant dans *processes*(s)

Routine *restore_process* : H(unit)
Action:
$p \leftarrow$ le premier processus dans la liste *processes*(s);
Affecter le processus p à *currentptp*(s);
Empiler l'instruction à exécuter par p et son mode d'exécution sur la pile du noyau;

FIGURE 3.17 – Restaurer l'état d'un processus.

3.1.4.2 cohérence matérielle

La page 0 n'est jamais utilisée

La mémoire physique peut contenir plusieurs types de pages physiques comme des pages libres ou associées à des processus ou encore réservées uniquement au noyau (jamais disponibles pour l'allocation). La dernière catégorie est utile dès que l'on a besoin d'isoler une partie de la mémoire de tout accès durant l'exécution des processus. Cette partie de mémoire pourrait être utile pour sauvegarder le code du noyau et ses données. Cette information doit donc figurer dans l'ensemble des propriétés de cohérence. Pour cela, il faut prouver que ces pages ne sont jamais associées aux processus ni disponibles pour l'allocation. Dans notre modèle, nous avons choisi la page 0 comme un exemple simple de ces emplacements mémoire. Certes, il est possible de généraliser afin couvrir n'importe quelle adresse physique pour ce même objectif. Afin d'assurer cette propriété nous avons défini les deux propriétés de cohérences suivantes : FREE_NOTZERO (voir définition 7) et USED_NOTZERO (voir définition 6). La première propriété permet d'assurer que la page 0 n'est jamais disponible pour l'allocation et la deuxième propriété assure qu'elle n'est associée à aucun processus. La généralisation de ces propriétés dans le but de couvrir d'autres zones mémoire nécessite de remplacer la valeur 0 par l'ensemble des pages physiques réservées uniquement pour le noyau.

Definition 6. *Étant donné un état s,* USED_NOTZERO(s) *est vraie si et seulement si quel que soit le processus* $P \in$ *processes*(s)*, et quel que soit* $p \in$ *UsedPages*(P) *alors* $0 < p <$ *nb_pages.*

Definition 7. *Étant donné un état s,* FREE_NOTZERO(s) *est vraie si et seulement si quel que soit* $p \in$ *FreePageList*(s)*,* $p \neq 0$.

La mémoire physique contient suffisamment de pages

Cette dernière propriété de cohérence assure que la mémoire physique est suffisamment grande.

Definition 8. *Étant donné un état s,* MEMORY_LENGTH(s) *est vraie si et seulement si*

$$nb\_pages \times page\_size \leq length(data(s))$$

tel que length($data$(s)) *correspond à la taille totale de la mémoire physique (en mots machine) et page_size correspond à la taille d'une page physique.*

En résumé, les propriétés de cohérence permettent de capturer les spécifications les plus pertinentes de bon fonctionnement du système afin d'assurer la propriété de sécurité. Le but de ces travaux ne consiste pas à lister et prouver toutes les propriétés de cohérences possibles liées au modèle mais d'identifier uniquement celles qui sont nécessaires pour prouver que la propriété d'isolation est préservée.

3.2 La vérification des propriétés du micro-noyau

3.2.1 Logique de Hoare au-dessus d'une monade d'état

Afin de raisonner sur les différentes opérations exposées par le micro-noyau, nous avons défini une variante de la logique de Hoare [Hoa69] au-dessus de la monade H. Une approche similaire a été utilisée dans [Swi09]. Les propriétés de chaque opération ont été spécifiées à travers des triplets de Hoare $\{\mathrm{P}\}\, c\, \{\mathrm{Q}\}$ où :

- P est une pré-condition, i.e. un prédicat unaire sur l'état avant l'exécution de c ;
- c est une opération monadique qui retourne un résultat de type A, i.e. l'opération c est de type H(A) ;
- Q est une post-condition , i.e. est un prédicat binaire sur la valeur retournée et l'état après l'exécution de c.

Par définition, un triplet $\{\mathrm{P}\}\, c\, \{\mathrm{Q}\}$ est vrai si et seulement si, quel que soit un état s, si s satisfait la propriété P alors soit la commande $c(s)$ arrête l'exécution du système soit elle retourne une paire (a, s') telle que a est la valeur retournée et s' est le nouvel état après l'exécution de c et telle que la post-condition Q est vraie pour a et s'. Si $c(s)$ génère un comportement non défini le triplet $\{\mathrm{P}\}\, c\, \{\mathrm{Q}\}$ n'est pas prouvable.

La plus faible pré-condition d'une opération c et d'une post-condition Q est un prédicat unaire sur l'état initial permettant de spécifier le comportement exact de c. Elle est définie par $wp(\mathrm{Q}, c)$ tel que :

- le triplet $\{wp(\mathrm{Q}, c)\}\, c\, \{\mathrm{Q}\}$ est vrai ;
- pour toute pré-condition P telle que $\{\mathrm{P}\}\, c\, \{\mathrm{Q}\}$ est vrai et pour chaque état s, $\mathrm{P}(s)$ implique $wp(\mathrm{Q}, c)(s)$.

3.2.2 La préservation des propriétés d'isolation et de cohérence

Nous avons formellement prouvé que toutes les instructions, routines et composants matériels modélisés préservent les propriétés d'isolation et de cohérence. Sachant que l'implémentation de chacune de ces opérations consiste en la mise en séquence de plusieurs opérations monadiques plus élémentaires, nous avons donc, afin de vérifier ces séquences, défini et prouvé le triplet de Hoare qui correspond à la plus faible pré-condition de chaque opération élémentaire. Puis nous avons utilisé ces triplets pour prouver que les propriétés d'isolation et de cohérence sont préservées par les opérations plus complexes sous la forme d'invariants. Dans la suite de cette section nous allons détailler la preuve de quelques fonctions. Nous commençons par une instruction assez simple *trap* puis nous passons à un exemple plus enrichi *write* afin de détailler notre approche de vérification.

3.2.2.1 Exemple détaillé : l'instruction *trap*

Comme détaillé dans le chapitre 2, les processus s'exécutent dans un niveau de privilège moins élevé que celui du noyau. Dans notre modèle, afin d'exécuter un ensemble d'opérations qui nécessitent plus de privilèges, un processus peut solliciter le noyau à travers des appels système en passant

par les interruptions logicielles (autrement dit, exécuter l'opération *trap* (voir figure 3.10)). Cette opération incrémente la position de *currentpc* puis exécute l'instruction *interrupt* (voir figure 3.8). L'instruction *interrupt* à son tour sauvegarde le contexte du programme en cours d'exécution dans le but de permettre au noyau de reprendre correctement l'exécution du processus courant après cette interruption. Ainsi, cela nécessite d'ajouter la position de l'instruction suivante à exécuter par le processus courant et son mode d'exécution sur la pile du noyau. A ce moment-là, l'instruction *interrupt* active le mode noyau et le gestionnaire d'interruption qui correspond à cette interruption. Ce dernier est identifié par le numéro d'interruption indiqué par l'argument de cette instruction.

La vérification de *trap* consiste à prouver que les propriétés d'isolation et de cohérences sont préservées après son exécution. Pour cela, il suffit de prouver principalement deux triplets : le premier est celui de sa plus faible pré-condition. Celui-ci sera utilisé pour prouver le deuxième triplet qui correspond à l'invariant de cette instruction. Comme aucune de ces propriétés ne dépend des composants logiciels ou matériels modifiés par cette instruction tel que la pile du noyau, le mode d'exécution ou encore la prochaine instruction à exécuter, la preuve de l'invariant de cette instruction était assez facile. Il est à noter que la preuve en Coq est disponible sur https://github.com/jomaa/MIMIC/blob/master/Instructions_invariants.v.

3.2.2.2 Exemple détaillé : l'instruction *write*

L'instruction *write* (voir figure 3.5) prend deux arguments. Elle permet de sauvegarder une valeur *val* dans une adresse virtuelle *vaddr* dans l'espace d'adressage du processus courant. Dans un premier temps, le MMU vérifie si l'accès à cette adresse virtuelle est possible en utilisant la fonction de traduction *translate*. S'il y a une page physique qui est associée à cette adresse virtuelle, *translate* retourne l'adresse physique *paddr* sinon elle retourne une exception. Dans le premier cas, l'opération *write* exécute l'instruction *write_phy* (voir figure 3.18) qui permet de sauvegarder la valeur *val* dans l'adresse physique *paddr* accessible par le processus courant.

composant matériel *write_phy* : integer → integer → H(unit)
Input : *val* : la valeur à sauvegarder
paddr : l'adresse physique
Action :
$p \leftarrow$ la page de *paddr*;
$i \leftarrow$ la position de *paddr* dans p;
update_memory(val, i, p);

FIGURE 3.18 – Écrire une valeur dans une adresse physique.

Comme pour n'importe quelle opération exposée par le système, le but de notre vérification est de prouver que cette opération *write* préserve l'isolation et la cohérence. Donc, nous devons prouver que le triple `write_invariant` est valide.

Proposition (write_invariant)**.** *Si la propriété d'isolation* I *et la propriété de cohérence* C *sont valides avant l'exécution de write, alors* I *et* C *sont valides pour l'état produit par cette instruction. Formellement nous écrivons :*

$$\{\mathrm{I} \wedge \mathrm{C}\}\ write(v, vaddr)\ \{\mathrm{I} \wedge \mathrm{C}\}$$

Le composant matériel *translate* est la première opération exécutée par *write*. Dans le cas où elle retourne une exception, l'exécution de *write* s'arrête sans aucun changement d'état. Ainsi, l'isolation et la cohérence sont trivialement préservées.

Maintenant nous allons analyser le cas où l'adresse virtuelle *vaddr* est accessible. Afin de prouver que *write* préserve l'isolation et la cohérence il est nécessaire de propager les propriétés I et C jusqu'à la fin

de l'opération. Étant donné que *translate* est la première instruction à exécuter, alors sa pré-condition doit être la même que celle de l'invariant principal de *write*. De même, *write_phy* est la dernière instruction à exécuter donc sa post-condition doit être la même que celle de l'invariant principal. Puisque l'opération *write* est la séquence de ces deux instructions alors la post-condition de la première instruction et la pré-condition de la seconde instruction doivent être identiques. Il est important de noter que l'instruction *write_phy* modifie la mémoire en fonction de la valeur *paddr* retournée par *translate*. Donc pour prouver que *write_phy* préserve nos propriétés nous avons besoin d'une propriété intermédiaire que nous appellerons *R* qui dépend de *paddr* et de l'état retourné par *translate*. En conséquence, l'étape suivante consiste à déterminer la propriété *R* nécessaire pour définir et prouver les invariants *translate_invariant* (voir lemme 1) et *write_phy_invariant* (voir lemme 2).

Lemme 1 (translate_invariant)**.** *Si la propriété d'isolation* I *et la propriété de cohérence* C *sont valides avant l'exécution de translate, alors* I, C *et R sont valides à l'état produit par cette instruction. Formellement nous écrivons :*

$$\{\mathrm{I} \wedge \mathrm{C}\}\ translate(vaddr)\ \{\mathrm{I} \wedge \mathrm{C} \wedge \mathrm{R}\}$$

Lemme 2 (write_phy_invariant)**.** *Si les propriétés* I, C *et R sont valides avant l'exécution de write_phy, donc* I *et* C *sont valides à l'état produit par cette instruction. Formellement nous écrivons :*

$$\{\mathrm{I} \wedge \mathrm{C} \wedge \mathrm{R}\}\ write\_phy(v, paddr)\ \{\mathrm{I} \wedge \mathrm{C}\}$$

Déterminer la propriété *R* Avant de sauvegarder une valeur dans la mémoire physique, *write_phy* doit déterminer le numéro de page physique p et la position i dans p qui, ensemble, correspondent à l'adresse physique *paddr*. Afin de prouver l'isolation et la cohérence, la page p doit être mappée et accessible dans l'espace d'adressage du processus courant et i doit être une position dans cette page. D'où la définition de R :

Étant donné une adresse physique *paddr*, R est prouvable en fonction de l'état s et de l'adresse physique *paddr* si et seulement si il existe une page physique p et une position i telle que :

- $paddr = p \times page\_size + i$;
- $p \in MappedPages(ptp(s))$ et
- $i < page\_size$.

où $MappedPages(ptp(s))$ est la liste de toutes les pages référencées dans la table de pages $ptp(s)$.

D'une part, l'exécution de *translate* préserve l'état précédent donc la preuve de l'isolation et des propriétés de cohérence après l'exécution de *translate* est facile. D'autre part, il faut prouver que la nouvelle propriété $\mathrm{R}(paddr, s)$ est valide après l'exécution de cette première instruction (avec *paddr* l'adresse physique retournée par *translate*). La preuve de cette propriété est établie en utilisant la plus faible pré-condition de *translate*. Sachant que la formalisation de la fonction de traduction est assez complexe (elle consiste en la mise en séquence de plusieurs instructions plus élémentaires) nous ne donnerons pas des détails ni sur la définition de sa plus faible pré-condition ni sur la preuve de la validité de ce triplet. En revanche, d'une manière générale, ceci correspond à l'une des difficultés rencontrées pendant l'établissement de la preuve surtout quand il s'agit d'une opération qui change plusieurs fois l'état du système. Dans ce cas, il devient assez compliqué de capturer l'ordre des changements de l'état ainsi que l'état final produit par cette opération.

Contrairement à *translate*, la seconde instruction *write_phy* modifie l'état courant, plus précisément la mémoire physique, et ne retourne pas de valeur. Dans ce cas il faut prouver que si I, C et R sont valides avant l'exécution de *write_phy* alors I et C sont toujours valide après la modification de la mémoire. De même, nous utilisons la plus faible pré-condition pour définir la spécification de cette instruction et prouver son invariant.

Cette preuve implique huit cas : un premier cas pour prouver l'isolation et un cas par propriété de cohérence. Dans la suite nous détaillons la preuve de chaque propriété. Il est à noter que la preuve en Coq est disponible sur https://github.com/jomaa/MIMIC/blob/master/Access_invariant.v.

ISOLATION(S) : Cette propriété exige que si le processus courant écrit une valeur dans la mémoire physique, il ne doit pas altérer la configuration du MMU de n'importe quel processus, y compris lui-même, en essayant par exemple de s'approprier une nouvelle page physique. Cette preuve consiste à vérifier que la référence de la page p (voir figure 3.18) est différente de toutes les références des tables de page des processus et qu'elle soit l'une des pages mappées dans son propre espace d'adressage.

Prenons donc deux processus différents P_1 et P_2 dans la liste des processus *processes*(s). La propriété de cohérence CURRPROCESS_INPROCESSLIST(s) (voir définition 5) assure que la table de pages du processus courant est associée à l'un des processus dans la liste *processes*(s). Par conséquent, nous aurons trois cas (dont deux sont symétriques).

Un premier cas consiste à considérer que *currentptp*(s) ne correspond ni à *ptp*(P_1) ni à *ptp*(P_2). Dans ce cas, pour montrer que la propriété d'isolation est préservée il suffit de montrer que les tables de page de *ptp*(P_1) et *ptp*(P_2) ne changent pas de contenu. Ceci est vrai grâce à deux propriétés : La première est la propriété R. Elle permet de garantir que la page p est mappée dans l'espace d'adressage d'un processus différent (i.e. *currentptp*(s)). La deuxième est la propriété d'isolation sur l'état précédent. Elle permet de garantir que les pages mappées dans un processus sont différentes de toutes les pages associées à n'importe quel autre processus y compris la référence vers sa table de pages.

Dans les deux autres cas nous considérons, respectivement, que *currentptp*(s) correspond à *ptp*(P_1) puis à *ptp*(P_2). Pour ces deux cas symétriques nous avons, également, besoin de la propriété R pour deux raisons différentes : la première parce qu'il faut prouver que le processus courant ne modifie pas sa propre table de pages. Cela est vrai grâce à R et à la propriété de cohérence NODUPLIC_PROCESSPAGES. Cette dernière assure que la référence vers la table de pages d'un processus est différente de toutes les références des pages mappées. Enfin, la deuxième raison est qu'il est nécessaire de prouver que le processus courant ne modifie pas la configuration des autres processus. Cela est vérifié en s'appuyant sur R et la propriété d'isolation sur l'état précédent (tous les autres processus garderont le même contenu de leurs tables de page après l'exécution de cette instruction).

REALLY_FREE(S) : Cette propriété dépend de la liste des pages libres *first_free_page*(s) ainsi que la table des processus *processes*(s). Elle nécessite de vérifier que si un processus écrit une valeur dans la mémoire physique, il ne peut pas modifier ces structures de données qui sont réservées uniquement au noyau. La propriété de cohérence REALLY_FREE préservée par l'état précédent garantit que toutes les pages mappées dans l'espace d'adressage d'un processus ne sont pas référencées par la liste des pages libres. En utilisant REALLY_FREE et la propriété R nous prouvons que la liste des pages libres n'est pas modifiée par cette instruction. De plus, nous utilisons la propriété de cohérence NODUPLIC_PROCESSPAGES pour assurer que le processus ne modifie pas sa propre table de pages et la propriété d'isolation pour prouver qu'il ne modifie pas les tables de pages d'un autre processus ;

NOT_CYCLIC(S) : Afin de prouver que l'instruction n'a pas créé de cycle dans la liste des pages libres il suffit de suivre le même raisonnement précédent pour prouver que le processus n'a pas modifié le contenu de cette liste. Ceci possible grâce aux propriétés R et REALLY_FREE ;

NODUPLIC_PROCESSPAGES(S) : Pour garantir cette propriété il suffit de prouver qu'aucune des tables de page n'est altérée par cette instruction. Pour cela nous utilisons la propriété R et la propriété NO-

DUPLIC_PROCESSPAGES sur l'état précédent pour prouver que le processus n'a pas modifié sa propre table de pages. Nous utilisons aussi la propriété d'isolation et la propriété *R* pour prouver que les autres tables de pages garderont le même contenu après l'exécution de cette instruction ;

FREE_NOTZERO(s) : Pour prouver que toutes les pages libres demeurent différentes de la page 0 (réservée uniquement au noyau) après l'exécution de cette instruction il suffit de prouver que la configuration de cette liste n'a pas changé. D'où exactement le même raisonnement que celui de NOT_CYCLIC ;

USED_NOTZERO(s) : Pour prouver que la page 0 n'est pas utilisée par un processus dans la liste de processus après l'exécution de *write_phy*, il suffit de prouver que toutes les tables de pages garderont le même contenu. Pour cela nous pouvons appliquer le même raisonnement que celui utilisé dans la preuve de NODUPLIC_PROCESSPAGES(s) ;

MEMORY_LENGTH(s) : La vérification de cette propriété nécessite de prouver que la mémoire garde toujours la même taille après l'exécution de *write_phy* ;

CURRPROCESS_INPROCESSLIST(s) : La preuve de cette propriété est triviale puisqu'elle ne dépend pas du composant matériel *memory*(*s*).

3.2.2.3 Autre exemple : Ajouter une nouvelle entrée de table de pages

La preuve de l'invariant *write_invariant* détaillée plus haut a permis d'explorer la plupart des aspects importants et communs à toutes les opérations exposées par le modèle de micro-noyau, afin de comprendre notre approche de vérification. En outre, il arrive que certaines opérations soulèvent des problématiques particulières tel que le cas de la routine *add_pte* qui est une opération assez compliquée. Dans ce paragraphe nous allons discuter brièvement cet aspect.

Le comportement souhaité de cette opération (voir figure 3.12) est de configurer une entrée de page dans la table de pages du processus courant. En effet, s'il n'y a pas de page auparavant dans cette entrée alors elle alloue une nouvelle page à partir de la liste des pages libres en exécutant l'instruction *alloc_page*. Puis elle mappe cette page avec les permissions requises. En suivant la séquence d'instructions de cette routine, le premier test effectué est nécessaire pour garantir qu'il n'y avait pas de configuration dans cette entrée (ce test est spécifié par une propriété que nous appellerons H_i). Le problème est que, entre ce test et la configuration de la nouvelle entrée, la routine *add_pte* exécute *alloc_page* qui change l'état de la mémoire. Dans ce cas il faut prouver que *alloc_page* préserve la propriété H_i. Dans notre modèle, l'instruction *alloc_page* est utilisée dans d'autres routines telles que la création de processus. Donc nous devons prouver plusieurs invariants pour la même instruction *alloc_page* qui prouvent que l'isolation et la cohérence sont préservées et d'autres propriétés qui doivent être propagées. Il est à noter que la preuve en Coq est disponible sur https://github.com/jomaa/MIMIC/blob/master/Addpte_invariant.v.

3.2.3 L'état initial du système et la création des processus

Dans notre approche nous avons prouvé que les propriétés I et C sont invariantes par toutes les opérations exposées par le noyau. Donc, quand le système démarre, la première tâche à faire est de mettre le système dans un état cohérent et vérifiant l'isolation. Autrement dit, il doit respecter toutes les propriétés d'isolation et de cohérence que nous avons définies. C'est le but de la routine *boot*.

Un exemple d'un état initial pourrait correspondre à l'initialisation de la liste des processus avec un unique processus P qui sera considéré comme processus courant. Donc la valeur de *currentptp*(*s*) sera initialisée à la valeur de *ptp*(P) (i.e. la table de pages du processus P). De même, la mémoire physique doit être initialisée. Ce qui implique l'initialisation de la liste des pages libres ainsi que de la valeur de la

première page libre (i.e. *first_free_page*(*s*)). Finalement, la liste *intr_table*(*s*) doit contenir les références vers tous les gestionnaires d'interruption qui seront configurés dans *code*(*s*). Un exemple de telle liste est de choisir les interruptions numéro 0 et numéro 1, respectivement, comme références vers les routines *switch_process* et *create_process*. Afin de vérifier cet état initial, nous avons prouvé le triplet de Hoare défini par *boot_invariant* (Lemme 3).

Lemme 3 (boot_invariant)**.** *La propriété d'isolation* I *et la propriété de cohérence* C *sont valides après l'exécution de l'opération boot*

$$\{True\}\ boot\ \{\mathrm{I} \wedge \mathrm{C}\}$$

Après le démarrage du système, n'importe quel processus peut créer de nouveaux processus en utilisant la routine *create_process* (voir figure 3.19) qui alloue une nouvelle page pour la définir comme table de pages de ce processus, initialise cette table avec la valeur par défaut et ajoute ce processus dans la liste des processus.

Routine *create_process* : integer → H(unit)
Input : *pc* : La position de la première instruction à exécuter par le processus
Action :
table ← Allouer une nouvelle page physique;
Initialiser *table* avec la valeur par défaut 0;
Définir *table* comme table de pages du nouveau processus;
Définir *pc* comme pointeur vers la première instruction à exécuter par le processus;
Ajouter le processus à la liste *processes*(*s*).

FIGURE 3.19 – Créer un nouveau processus

En outre, durant le démarrage, le système peut configurer l'horloge afin de mettre en place un ordonnancement préemptif. Dans notre modèle ce mécanisme est mis en place par la routine *boot* de la manière suivante :

- Associer un numéro d'interruption à l'ordonnanceur dans *intr_table*;
- Le composant matériel *interrupts* est configuré de telle sorte qu'il déclenchera périodiquement le changement de contexte.

3.3 Synthèse de formalisation

Dans cette section nous décrivons la structure de notre développement en Coq. Le code source est disponible sur `https://github.com/jomaa/MIMIC`. Il est possible de le compiler avec la version `8.5pl2` de Coq. Il consiste en 2 400 lignes de spécification et 9 700 lignes de preuve. Une synthèse pour chaque fichier est disponible dans la table 3.1.

- La définition de notre monade H qui gère l'état du système est répartie en deux fichiers StateMonad.v et HMonad.v;
- Notre modèle d'architecture matérielle est réparti en plusieurs fichiers :
  - Le composant matériel *translate* du MMU est modélisé dans MMU.v;
  - L'allocation de la mémoire est implémentée dans MemoryManager.v par la routine *alloc_page*;

Catégorie	Nom de fichier	lignes de spec	lignes de preuve
monade H	StateMonad.v	101	70
	HMonad.v	145	70
	Sous-total	**246**	**140**
Architecture matérielle	MMU.v	84	170
	MemoryManager.v	93	76
	Access.v	107	42
	Instructions.v	168	117
	Step.v	32	0
	Sous-total	**484**	**405**
micro-noyau	PageTableManager.v	207	76
	Scheduler.v	115	25
	ProcessManager.v	115	6
	Sous-total	**437**	**107**
Définition et preuves d'isolation et de cohérence	Properties.v	58	0
	MMU invariant.v	18	273
	Access_invariants.v	117	1487
	Scheduler_invariant.v	140	347
	Instructions_invariants.v	55	108
	ProcessManager_invariant.v	123	596
	Step_invariant.v	8	37
	Alloc_invariants.v	37	102
	Addpte_invariant.v	122	1749
	Removepte_invariant.v	323	3763
	Sous-total	**1001**	**8462**
Autres fichiers	Lib.v	182	525
	LibOs.v	29	69
	Example.v	83	89
	Sous-total	**294**	**683**
Total		**2462**	**9797**

TABLEAU 3.1 – L'organisation du modèle

  - La gestion d'accès à la mémoire physique est modélisée dans Access.v. Dans ce fichier nous définissons les instructions *write* et *read* ;
  - L'évolution dynamique du système (i.e. *step*), comportant la gestion des interruptions (*interrupt*, *fetch_interrupt*, *fetch_instruction* et *return_from_interrupt*), est modélisée dans Instructions.v et Step.v.

- La définition de notre modèle du micro-noyau est répartie en plusieurs fichiers :

  - Les routines *add_pte* et *remove_pte* pour modifier les tables de pages des processus sont définies dans PageTableManager.v ;
  - Les routines *save_process*, *restore_process* et *switch_process* pour gérer le changement de contexte et l'ordonnancement sont définies dans Scheduler.v ;
  - La création des processus modélisée par la routine *create_process* est définie dans ProcessManager.v.

- Le fichier Properties.v contient les définitions des propriétés d'isolation et de cohérence ;
- La preuve de la préservation d'isolation et de cohérence est répartie dans les fichiers suivants : MMU_invariant.v, Access_invariants.v, Scheduler_invariant.v, Instructions_invariants.v, Step_invariant.v, Alloc_invariants.v, Addpte_invariant.v, Removepte_invariant.v, ProcessManager_invariant.v.

3.4 Conclusion

Dans ce chapitre nous avons présenté un modèle abstrait d'un micro-noyau implémenté en Gallina en utilisant un style monadique permettant de spécifier toute sorte d'effets de bord tel que le changement d'état et la gestion des exceptions. Ce modèle abstrait de micro-noyau comprend principalement un gestionnaire de mémoire permettant de gérer dynamiquement l'allocation de la mémoire physique ainsi que la mémoire virtuelle des processus et un mécanisme d'ordonnancement préemptif fondé sur les interruptions. Durant ces travaux, nous avons défini une approche assez simple de vérification permettant de prouver une propriété primordiale de sécurité qui est l'isolation mémoire. À cette fin, nous avons formalisé la plupart des comportements fondamentaux de MMU et CPU permettant de gérer correctement la mémoire physique afin de garantir l'isolation. Cela nous amène à conclure qu'il existe de nombreuses détails sur l'architecture matérielle et les structures maintenues par le micro-noyau qui doivent être pris en compte durant le processus de vérification de la propriété d'isolation.

Nous avons montré également que lorsque nous nous intéressons à la vérification des propriétés de sécurité nous n'avons pas besoin de vérifier toutes les propriétés de bon fonctionnement. En revanche, nous pouvons nous intéresser uniquement à celles qui sont nécessaires pour vérifier la sécurité. Ces propriétés fonctionnelles, que nous avons appelées les propriétés de cohérence, ont été intégrées au fur et à mesure pour pouvoir établir la preuve d'isolation mémoire. Ceci nous évite de définir un modèle abstrait spécifiant le comportement du noyau et d'établir une preuve de raffinement pour montrer qu'il est conforme à son implémentation concrète.

L'approche de vérification présentée dans ce chapitre permet de définir dès le départ ce qui doit être respecté comme propriétés de sécurité puis établir le raisonnement progressivement en introduisant à chaque fois le strict minimum des propriétés de bon fonctionnement nécessaire pour vérifier la sécurité. Ceci nous permet de gagner en matière de coût de preuve puisque nous vérifions uniquement le strict minimum des propriétés.

Il est important de noter que ce travail est une abstraction de ce qu'est un micro-noyau. Les expériences menées sur ce modèle générique nous a permis de mettre en place une méthodologie de vérification des propriétés de sécurité du code d'un noyau formalisé en Coq. Cette méthodologie sera validée dans les chapitres suivants en s'appuyant sur une implémentation concrète d'un proto-noyau que nous avons appelé Pip.

Bibliographie

[Gho11] Subrata GHOSHAL : *Computer Architecture and Organization : From 8085 to Core2Duo and beyond*. Dorling Kindersley, 2011.

[Hoa69] Charles Antony Richard HOARE : An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Jom17] Narjes JOMAA : MIcrokernel Model In Coq, 2017. Available at https://github.com/jomaa/MIMIC.

[Mog91] Eugenio MOGGI : Notions of computation and monads. *Information and Computation*, 93(1): 55–92, 1991.

[Swi09] Wouter SWIERSTRA : A Hoare logic for the state monad. *In International Conference on Theorem Proving in Higher Order Logics*, pages 440–451. Springer, 2009.

Chapitre 4

Les principes du design de Pip

Sommaire

Une méthodologie usuelle de développement d'un noyau et sa vérification consiste à suivre d'une manière séquentielle les étapes suivantes : Conception, implémentation, puis preuve. En revanche, le *co-design* de noyau avec sa preuve nous amène à adapter cette approche en autorisant des *feedbacks* entre ces étapes (voir figure 4.1). Autrement dit, influencer le développement d'une étape par une autre pour réduire le coût de la preuve (e.g. le temps passé à établir la preuve, sa taille, sa complexité, etc) tout en garantissant l'utilisabilité du système.

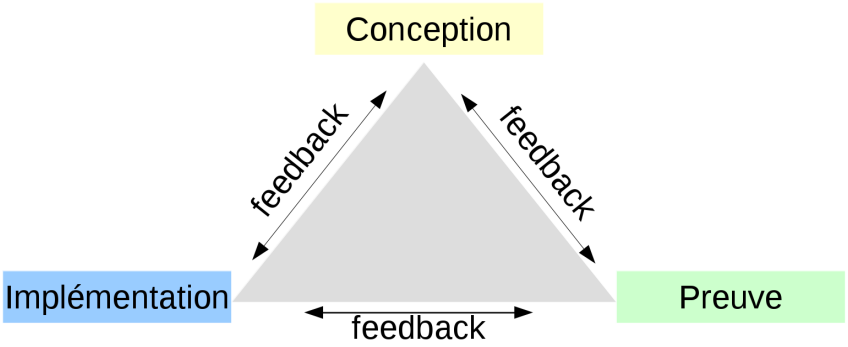


FIGURE 4.1 – Cycle de développement et de vérification fondé sur les *feedback*.

En effet la conception d'un noyau nécessite une attention particulière dès les premières phases de conception. Le choix des fonctionnalités, la stratégie d'implémentation, le modèle du matériel à gérer, les propriétés à vérifier et l'approche de vérification sont tous à prendre en compte pendant le développement du noyau. Ce chapitre discute de la stratégie de *co-design* d'un nouveau noyau de système d'exploitation, que nous avons appelé le proto-noyau Pip, et de sa preuve. L'objectif visé est d'adapter le design du noyau pour réduire la complexité de la production de sa preuve tout en préservant son utilisabilité. La propriété étudiée dans ces travaux est une propriété de sécurité, exprimée en terme d'isolation mémoire.

4.1 Concepts de base de Pip

4.1.1 Minimisation de la base de confiance

Pip est un proto-noyau. Il a été conçu dans le but de minimiser le TCB et apporter plus de garanties de sécurité sur l'ensemble du système. L'objectif principal de cette minimisation est double : réduire la surface d'attaque ainsi que l'effort de l'établissement de la preuve.

Cependant, la réduction de la taille du TCB n'est pas un concept inconnu. Elle a été mise en place par la famille des micro-noyaux [Lie95]. Il a été montré qu'il est possible de réduire le code exécuté en mode noyau à la gestion de la mémoire virtuelle, l'ordonnancement, la communication entre les processus, le multiplexage et le changement de contexte tel qu'il est illustré par la figure 4.2. Le reste des modules tels que la gestion des fichiers et celle des périphériques peuvent s'exécuter au même niveau que les applications utilisateur sans avoir d'impact sur la sécurité.

Les exo-noyaux [EKO95] ont été également conçus en suivant le même principe. En effet, ces travaux ont montré qu'il est possible de réduire encore plus le nombre de fonctionnalités exécutées en mode noyau. Leur modèle de sécurité exige d'implémenter uniquement la gestion de la mémoire virtuelle, le changement de contexte et le multiplexage en mode privilégié tel qu'il est illustré par la figure 4.3.

De façon similaire, le *co-design* du proto-noyau Pip avec sa preuve nous a mené à réduire le TCB au strict minimum en exportant tous les modules qui ne sont pas nécessaires pour la sécurité en mode utilisateur, y compris le multiplexage. La réduction de cette base de confiance dans le contexte de Pip a pour objectif d'inclure uniquement les mécanismes qui sont liés à l'isolation. Ainsi, les fonctionnalités telles que le multiplexage (qui assure le partage équitable des ressources, et qui ne peut donc pas être vu

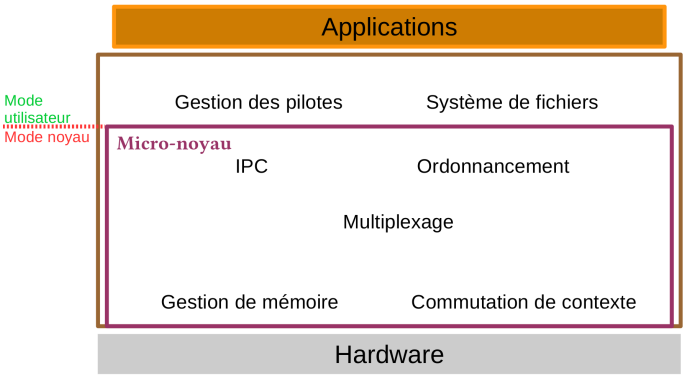


FIGURE 4.2 – Architecture logicielle des micro-noyaux.

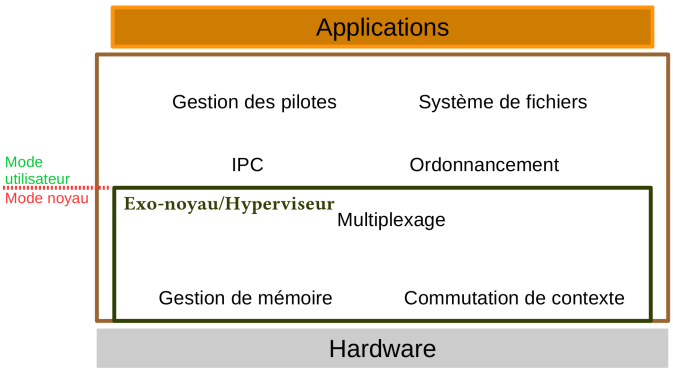


FIGURE 4.3 – Architecture logicielle des exonoyaux.

comme un élément de sécurité) peut être externalisé, c'est à dire sorti de la base de confiance nécessaire à garantir l'isolation de la mémoire. Nous illustrons l'architecture logicielle de Pip par la figure 4.4.

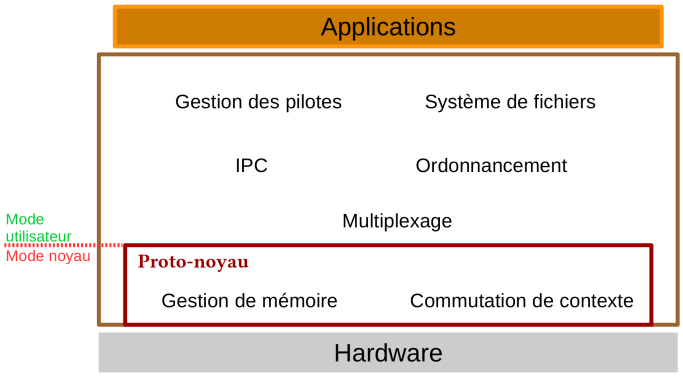


FIGURE 4.4 – Architecture logicielle du proto-noyau Pip

Le choix des services fournis par le noyau est motivé principalement par la faisabilité de la preuve et l'utilisabilité du système en matière de performance et de fonctionnalités de base. Il n'offre aucune abstraction de haut niveau telle que les threads, les processus, le système de fichier ou encore la pile réseau. En revanche, il fournit un environnement d'exécution sous la forme de machines virtuelles que nous avons appelées *partitions*. Les partitions sont autorisées à configurer l'unité de gestion de la mémoire virtuelle (MMU) ainsi que gérer le flot d'exécution à travers les services fournis par Pip. Cet ensemble de services constitue l'API du proto-noyau. Il s'agit de dix appels système choisis minutieusement pendant la phase de conception afin d'assurer à la fois la faisabilité de la preuve et l'utilisabilité du système. En utilisant uniquement ces services de configuration, il est possible de porter le noyau Linux aussi bien

que le noyau pour les systèmes embarqués FreeRTOS ou encore un hyperviseur tel que Xen [BDF+03]. Cela nécessite principalement de porter ces systèmes au dessus de Pip sous la forme de partition en s'appuyant sur l'approche de paravirtualisation. Il est à noter que d'autres travaux plus centrés sur le domaine des architectures de systèmes que sur le domaine de la vérification a été l'objet d'autres études [YGG+18, BGIC18b, BGIC18a, YGIC18] menées conjointement avec les travaux présentés dans cette thèse.

Pip n'est pas un hyperviseur puisqu'il ne fournit pas un mécanisme de multiplexage des ressources matérielles. Exécuter le mécanisme de multiplexage en mode utilisateur nous permet de réduire considérablement la taille du TCB sans aucun impact sur la sécurité du système.

4.1.2 Le modèle de partitionnement

Au démarrage, une seule partition est créée, c'est la partition racine. Toute la mémoire physique, sauf celle qui est réservée à Pip, est utilisable par cette partition. Le temps de CPU est également géré par la partition racine sauf celui qui est utilisé par Pip pour retransmettre les interruptions. La gestion de la mémoire assurée par Pip est fondée sur un modèle de partitionnement hiérarchique tel qu'il est illustré par la figure 4.5 où chaque nœud dans l'arbre correspond à une partition et chaque lien entre deux nœuds est la relation entre une partition parent et un enfant. Ce modèle permet à toute partition de créer ses propres sous-partitions et partager une partie de sa mémoire ainsi que son temps d'exécution entre cet ensemble de sous-partitions. Cela permet de construire un arbre où la première partition créée à l'état initial est la partition racine.

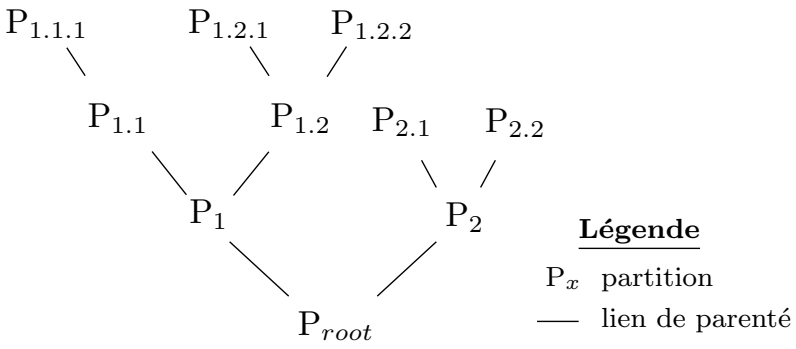


FIGURE 4.5 – Un exemple d'arbre de partition.

4.1.2.1 Relation de parenté entre les partitions

Nous proposons dans ce paragraphe les détails sur les concepts fondamentaux de l'arbre de partition géré par Pip. Nous présentons les relations de parenté entre les partitions ce qui facilite la compréhension de l'ensemble de propriétés de sécurité visées par la vérification.

- Parent : Il correspond simplement, à un nœud dans l'arbre de partition et il peut avoir une ou plusieurs partitions enfants. Sur la figure 4.5, P_1 est la partition parent de $P_{1.1}$ et $P_{1.1}$ est une partition enfant de P_1.

  Un parent peut avoir également plusieurs descendants. Sur la figure 4.5, les partitions $P_{1.1}$, $P_{1.1.1}$, $P_{1.2}$, $P_{1.2.1}$ et $P_{1.2.2}$ sont les descendants de P_1.

- Ancêtre : Un ancêtre d'une partition correspond à l'une des partitions qui se trouve sur la branche entre la partition racine et le parent y compris la partition parent et la racine. Sur la figure 4.5 les partitions $P_{1.1}$, P_1 et P_{root} sont les ancêtres de $P_{1.1.1}$.

- Sœurs : Deux partitions sont considérées comme sœurs si elles ont la même partition parent. Sur la figure 4.5 les partitions $P_{1.1}$ et $P_{1.2}$ sont des partitions sœurs.

- Incomparables : Dans un arbre de partitions géré par Pip nous considérons deux partitions comme incomparables si l'une des deux ne correspond pas à l'ancêtre de l'autre et inversement. Sur la figure 4.5 $P_{1.1.1}$ n'est pas un ancêtre de $P_{2.2}$ et $P_{2.2}$ n'est pas un ancêtre de $P_{1.1.1}$ donc elles sont incomparables. Deux partitions sœurs sont également incomparables.

4.1.2.2 Principes de fonctionnement

La gestion de cette structure hiérarchique est effectuée grâce à un schéma récursif où la sécurité d'une partition repose sur l'environnement d'exécution géré par son parent ainsi que par le noyau. Cela attribue naturellement un caractère hiérarchique au TCB associé à chaque partition. Il pourrait être exprimé en termes d'une simple relation de transitivité : la sécurité de chaque partition est fondée sur ses propres ancêtres ainsi que sur le noyau. Le but principal de ce design est d'établir une preuve garantissant la validité de cette politique de TCB hiérarchique. Cette preuve est fondée sur la vérification des propriétés du modèle de partitionnement.

Du point de vue de Pip, l'arbre de partitions n'est qu'une structure hiérarchique de la mémoire physique. Ainsi, le code exécuté par ces partitions et leurs données sont tout à fait abstraits par rapport aux propriétés de sécurité assurées par le proto-noyau. Seules la relation entre les partitions et la configuration de leurs structures de données sont indispensables pour vérifier que le noyau garantit les propriétés d'isolation ainsi que la cohérence des structures de données. Ceci réduit considérablement les préoccupations du proto-noyau aussi bien que la complexité de la preuve.

Pip ne s'occupe ni du multiplexage des interruptions ni du partage du temps CPU entre les différentes partitions. Il gère uniquement les interruptions qui consistent à exécuter les services de son API et redirige les interruptions logicielles vers la partition parent de l'appelant et les interruptions matérielles vers la partition racine. Dans ce sens, Pip délègue le multiplexage à la partition racine. Cette dernière peut implémenter n'importe quelle politique de multiplexage permettant ainsi, récursivement, à chaque partition de partager son temps d'exécution entre ses enfants.

Ceci est illustré par la figure 4.6 où un multiplexeur s'exécute en tant que partition racine permettant de gérer les ressources matérielles entre un système d'exploitation linux comme première sous-partition et un système temps réel (FreeRTOS) comme deuxième sous-partition. Grâce au modèle de partitionnement mis en place par Pip, malgré la coexistence de ces deux systèmes d'exploitation sur la même machine physique, l'intégrité du noyau linux ne sera pas dépendante du bon fonctionnement de freeRTOS et inversement. De plus, les tâches de freeRTOS sont désormais isolées les unes des autres grâce à la propriété d'isolation assurée par Pip. Pour créer une nouvelle tâche, FreeRTOS doit solliciter le proto-noyau à travers une interruption logicielle. Une partie de la mémoire de FreeRTOS sera utilisée pour créer la partition associée à cette nouvelle tâche. Les appels système provenant des tâches $Tâche_1$, $Tâche_2$ ou $Tâche_3$ autres que ceux exposés par Pip seront redirigés directement vers FreeRTOS. Ce dernier doit s'occuper du traitement de ces interruptions logicielles. Cette redirection concerne également les fautes et les exceptions. Concernant les interruptions matérielles, le proto-noyau s'occupe uniquement de les rediriger vers le multiplexeur qui se charge de les traiter puisque cette partition racine possède tous les droits sur les ressources matérielles.

Finalement, il est important de noter que seul le proto-noyau Pip s'exécute en mode privilégié. Ainsi, l'ensemble des programmes dans l'arbre de partition, y compris les noyaux portés au-dessus de Pip, peuvent s'exécuter uniquement en mode utilisateur.

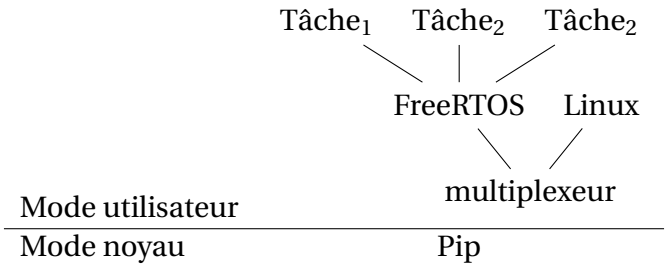


FIGURE 4.6 – Un exemple de cas d'utilisation de Pip.

4.1.3 Pip du point de vue utilisateur

À l'état initial, la mémoire est partagée entre Pip et la partition racine. Plus précisément, pour gérer les données du proto-noyau ainsi que la configuration de cette première partition, des zones mémoire seront accessibles uniquement en mode privilégié. Le reste de la mémoire disponible est intégralement attribué et accessible par la partition racine. Cette configuration initiale est mise en place par le « boot ». Ce dernier configure le MMU pour effectuer cette répartition entre la partition racine et le noyau.

L'unité pour gérer la mémoire des partitions est la *page*. C'est une zone de mémoire contiguë et de taille fixe qui dépend de l'architecture matérielle. Une page peut être **attribuée** à une partition. Dans ce cas nous identifions trois statuts différents de cette page :

mappée et accessible : elle est référencée par les tables de pages du MMU et marquée comme accessible. Elle est utilisée par la partition pour stocker ses données ainsi que son code.

mappée et non accessible : elles est référencée par le MMU mais accessible uniquement en mode noyau. C'est par exemple le cas des pages initialement mappées dans la partition et utilisées ultérieurement pour gérer les structures de données de ses sous-partitions. C'est également le cas des pages qui contiennent le code du noyau puisqu'il est mappé dans chaque partition à des adresses fixes. Toutes ces pages qui contiennent des données critiques ne doivent jamais être accessibles par une partition.

non mappée (donc non accessible) : elle est utilisée comme page de configuration de cette partition.

Cependant, une page **non attribuée** à une partition ne peut pas être accessible et ne correspond à aucune de ses pages de configuration.

L'API de Pip est constitué de dix appels système permettant de gérer la mémoire physique ainsi que le flot d'exécution. La définition de cet ensemble d'appels système a été étudié pendant la phase de conception afin d'assurer à la fois la faisabilité de la preuve et l'utilisabilité du système. Dans la suite de cette section nous présentons l'API du proto-noyau de point de vue utilisateur. Ceci permet d'avoir une vision claire sur l'utilisation des appels système de Pip.

4.1.3.1 Création des partitions

Lorsqu'une partition crée une partition enfant, elle délègue une partie de sa mémoire disponible au noyau pour initialiser les structures de données de la partition nouvellement créée. Cette opération est effectuée en utilisant l'appel système `createPartition` dont le prototype est le suivant :

```
Definition createPartition (descChild pdChild shadow1Child shadow2Child
                                    linkedlistchild: vaddr): LLI bool
```

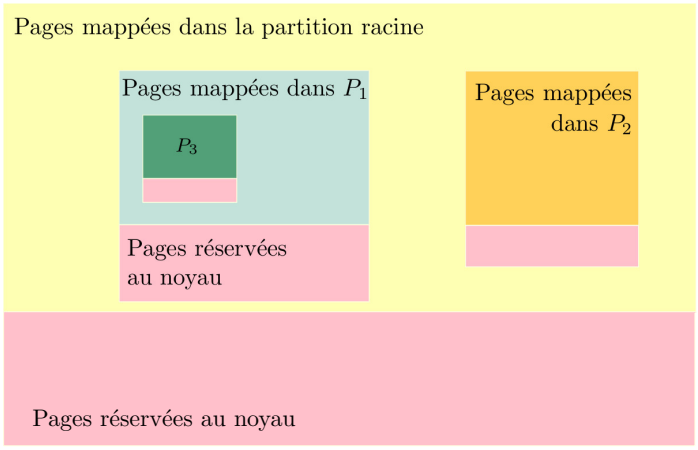


FIGURE 4.7 – Représentation physique d'un arbre de partition.

En effet, la partition doit fournir au noyau cinq adresses virtuelles. Les pages associées à ces adresses doivent être accessibles et non partagées avec d'autre partitions enfant. Si ce n'est pas le cas, l'exécution de cet appel échoue. À la fin de cette opération ces pages restent toujours mappées dans l'espace d'adressage du parent ainsi que de ses ancêtres mais elle ne seront plus accessibles. Ceci est primordial pour éviter tout accès non légitime à toute la mémoire y compris celle du noyau. En particulier, si une partition peut modifier les données du noyau stockées dans ces pages, elle peut alors modifier la configuration de la mémoire virtuelle de son enfant sans passer par l'API de Pip. Ce faisant, elle pourrait donner des droits quelconques à son enfant, comme celui d'accéder à une page de mémoire physique à laquelle le père lui-même n'aurait pas accès. Ainsi, l'isolation ne serait plus garantie.

4.1.3.2 Partage de mémoire

Pour étendre l'espace d'adressage de ses enfants, une partition peut partager une partie de sa mémoire disponible avec eux en gardant le droit d'y accéder. Ceci est important pour assurer principalement un mécanisme de communication entre un père et son fils ainsi qu'avec ses ancêtres. Cette opération est effectuée à travers l'appel système `addVaddr`.

```
Definition addVAddr (vaInCurrentPartition descChild vaChild: vaddr)
                                                (r w e: bool): LLI bool
```

Cette fonction mappe une page physique dans l'espace d'adressage d'un enfant de la partition qui est en cours d'exécution. Elle prend en paramètre trois adresses virtuelles ainsi que les droits d'accès de la partition enfant sur la page physique à partager. La première adresse virtuelle est associée à la page physique à partager dans l'espace d'adressage de la partition en exécution. Elle doit être accessible par cette partition et n'être attribuée à aucun de ses enfants. Si ce n'est pas le cas, l'exécution de cet appel échoue. La deuxième doit être associée à l'identifiant d'un enfant (Nous expliquons dans le chapitre suivant comment il est possible d'identifier les partitions). La dernière sera associée à la page physique partagée dans l'espace d'adressage de l'enfant. Il est à noter également que cette adresse virtuelle ne doit être associée à aucune page physique. L'appel système échoue si cette propriété n'est pas valide. À la fin de cette opération la page physique est encore mappée dans l'espace d'adressage du père (partition courante) ainsi que dans tous ses ancêtres.

Toutefois, le partage des pages avec les enfants nécessite également de déléguer une partie de mémoire au noyau pour effectuer les configurations nécessaires des structures internes des partitions. La figure 4.7 illustre l'organisation d'un arbre de partition géré par Pip. Chaque partition a un ensemble de pages qui sont à la fois accessibles et partagées avec des descendants. La configuration de chaque descendant fait partie de ses pages mappées mais uniquement accessible par le noyau. Cette configuration

est effectuée à travers les appels système `count` et `prepare`.

```
Definition count (descChild vaChild: vaddr): LLI count
```

L'appel système `count` permet de compter le nombre de pages à fournir au noyau pour qu'il puisse effectuer la configuration. La première adresse virtuelle doit être l'identifiant d'un enfant. La seconde est celle qui sera associée à la page physique à partager (i.e. le même paramètre `vaChild` de `addVaddr`).

```
Definition prepare (descChild vaChild fstVA: vaddr) (b: bool): LLI bool
```

L'appel système `prepare` effectue la configuration associée à l'adresse virtuelle en question (i.e. le même paramètre `vaChild` de `addVaddr` ainsi que celui de `count`). Le rôle de la partition ici est d'indiquer au noyau les pages à utiliser pour la configuration sous la forme d'une liste chaînée d'adresses virtuelles dont la première est définie par le paramètre `fstVA`.

4.1.3.3 Récupération des pages

La partition peut également demander au noyau de récupérer l'accès à un certain ensemble de pages attribuées à un enfant. Ceci est effectué à travers les appels système `removeVaddr` et `collect`.

D'une part, parmi les aspects fondamentaux de ce modèle hiérarchique est que la partition parent a toujours le droit de retirer des pages à ses enfants tant qu'elle a le droit d'y accéder. Ceci peut être effectué en utilisant l'appel système `removeVaddr`. C'est l'opération inverse de `addVaddr`. Elle rend à la partition parent le droit de partager cette page avec ses enfants.

```
Definition removeVaddr (descChild vaChild: vaddr): LLI bool
```

D'autre part, la partition parent peut également récupérer une page qui est accessible uniquement en mode noyau. Ceci est possible si et seulement si, à un moment donnée, cette page a été utilisée comme page de configuration dans les structures de données de l'enfant et qui, maintenant, ne contient plus de données utiles. Ceci arrive suite à la suppression de toutes les associations des adresses virtuelles à des adresses physiques mises en place en utilisant cette page.

```
Definition collect (descChild vaChild: vaddr): LLI bool
```

Finalement, à la suppression d'un enfant, la partition parent récupère toutes les pages attribuées à cet enfant y compris celles qui ont été utilisées par le noyau pour configurer la structure de la partition enfant. Cette opération est effectuée à travers l'appel système `deletePartition`.

```
Definition deletePartition (descChild: vaddr): LLI bool
```

4.1.3.4 Informations sur les pages

Une partition peut utiliser l'appel système `mappedInChild` pour savoir si une adresse virtuelle (dans son espace d'adressage) est associée à une page physique partagée avec l'un de ses enfants et si c'est le cas, avec qui elle a été partagée. Cette information est retournée sous la forme d'une adresse virtuelle qui correspond au `PDI` de l'enfant.

```
Definition mappedInChild (va: vaddr): LLI vaddr
```

4.1.3.5 Commutation de contexte

La gestion des flots de contrôle est effectuée à travers les deux appels système `dispatch` et `resume`. Le premier permet de notifier une partition de l'arrivée d'une interruption, sauvegarder son contexte et activer la partition notifiée. Une partition peut notifier soit son père soit l'un de ses fils. Le deuxième permet de reprendre l'exécution d'une partition déjà interrompue en restaurant son contexte sauvegardé. La formalisation en Coq de ces deux appels système est un travail en cours de réalisation.

4.2 Propriétés du modèle de partitionnement

Le choix des propriétés à vérifier dépend fortement du contexte dans lequel le système sera utilisé. Par exemple, dans le contexte des systèmes temps réel il est indispensable de vérifier le respect d'échéance sur les opérations effectuées par le système. Dans d'autres cas il est primordial de prouver que les programmes se comportent correctement. Autrement dit établir une preuve mathématique sur le comportement fonctionnel du système. En revanche, un système qui fonctionne correctement est susceptible de contenir des bugs de sécurité ce qui rend les programmes facilement attaquables. Dans le contexte des logiciels critiques tels que les noyaux des systèmes d'exploitation, il est prioritaire de prouver des propriétés de sécurité plutôt que de se concentrer sur le bon fonctionnement. En effet, le choix des propriétés fonctionnelles dépend fortement des détails d'implémentation du système. Si le système est complexe il est nécessaire de définir et vérifier de nombreuses propriétés pour prendre en compte tous les comportements possibles prévus par ce système, ce qui risque de faire apparaître un nombre important de propriétés. En revanche, la sécurité n'est pas une propriété fonctionnelle et donc elle n'est pas liée à une fonctionnalité en particulier. Ainsi, sa preuve consiste à prouver qu'elle est préservée par chaque appel système.
La sécurité, également, est un terme assez général. Dans le contexte d'un système d'exploitation la sécurité, telle qu'elle a été formalisée par Rushby [Rus84, Rus81], consiste principalement à assurer la séparation entre les entités ainsi qu' à contrôler les communications entre elles. Une première étape dans ce processus de vérification nécessite donc de définir clairement les propriétés de sécurité visées par la preuve et d'identifier les différents composants matériels ou logiciels sur lesquels reposent ces propriétés. Dans ce contexte nous nous intéressons à la propriété d'isolation mémoire. Cette propriété permet d'assurer qu'un programme ne peut pas accéder à la mémoire d'un autre programme. Nous considérons que c'est la propriété de sécurité la plus fondamentale car elle permet de prouver d'autres propriétés telles que celles qui consistent à assurer la sécurité de la communication entre les processus.

Dans cette section nous détaillons informellement l'ensemble des propriétés visées par notre vérification.

4.2.1 Isolation du noyau

Cette première propriété de sécurité permet au noyau d'assurer sa propre protection. Autrement dit, isoler ses données des accès non légitimes qui viennent principalement de l'espace utilisateur. En effet, les données du noyau ainsi que son code permettant la gestion de toutes les configurations des partitions sont stockées par le noyau dans des pages physique protégées. Cela nécessite de les protéger en s'appuyant sur le mécanisme matériel sur lequel repose la sécurité du système. Cette protection est assurée principalement par le MMU.

Comme déjà expliqué dans le chapitre 2 ce composant matériel assure le contrôle de tous les accès provenant des applications. Il traduit les adresses virtuelles vers des adresses physiques et vérifie à chaque fois les droits d'accès à l'adresse physique concernée. De ce fait il est primordial de d'assurer que les pages réservées au noyau ne sont pas accessibles par les applications selon les tables de configuration du MMU.

Puisque cette configuration est effectuée par les appels système exposés par le noyau, la preuve ici consiste à vérifier que ces services ne donnent jamais accès à une page réservée au noyau. Plus précisément l'ensemble des pages utilisées pour définir les structures de données des partitions ne doivent jamais être accessibles en mode utilisateur. Autrement, une partition pourrait accéder à toute la mémoire physique. Comme nous l'avons expliqué dans la section 4.1.3.1, si une partition a accès à une page de configuration elle a la possibilité d'alter l'intégralité des données sauvegardées dans la mémoire physique à travers la modification du contenu des pages de configuration. Ainsi, pour la vérification des propriétés d'isolation il est nécessaire de démontrer que les partitions n'ont jamais accès à ces pages à

travers la propriété d'isolation du noyau. De plus, il est important de noter que cette propriété est également utilisée *implicitement* pour assurer la validité du raisonnement sur les structures de données. En particulier, durant le processus de vérification nous considérons que les propriétés sur les données sauvegardées dans les structures comme des prémisses pour raisonner sur le comportement des appels système.

4.2.2 Modèle d'accès

4.2.2.1 Isolation horizontale

Cette propriété de sécurité est indispensable pour assurer la protection de la mémoire réservée à chaque partition des accès non légitimes. Prouver l'isolation mémoire dans le contexte de Pip signifie essentiellement prouver que la configuration de l'espace d'adressage d'une partition ne lui permet pas d'accéder à la mémoire physique d'une autre partition incomparable. Par exemple, grâce à l'isolation horizontale la partition $P_{1.2}$ sur la figure 4.8 ne peut pas accéder à la mémoire de la partition $P_{1.1}$. Et puisque $P_{1.1.1}$ et $P_{1.1.2}$ sont des descendants de $P_{1.1}$, leurs configurations ne les autorisent pas à accéder à la mémoire de $P_{1.2}$.

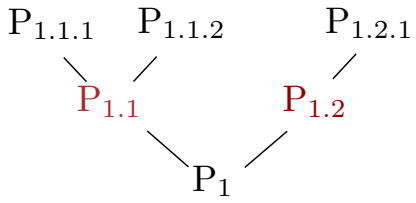


FIGURE 4.8 – Isolation horizontale dans un arbre de partitions.

4.2.2.2 Partage vertical

La structure arborescente mise en place par le proto-noyau Pip assure qu'une partition a des privilèges (ou pouvoirs) sur ses descendants. Ainsi, une des propriétés primordiales de notre modèle de partitionnement est le partage vertical. Cette propriété garantit que toutes les pages attribuées à une partition (à l'exception de la partition racine) sont également attribuées à sa partition parent ainsi qu'à ses ancêtres (par relation de transitivité). De ce fait, le partage vertical définit un modèle hiérarchique du TCB où la sécurité de chaque partition est fondée sur le noyau ainsi qu'à ses ancêtres.

Le concept de TCB hiérarchique ne s'applique en aucun cas sur deux partitions qui sont incomparables. Ceci est important car la propriété d'isolation ne consiste pas uniquement à empêcher les accès non légitimes entre les partitions mais aussi à empêcher la propagation de défaillances des applications dans l'ensemble qui constitue l'arbre de partition. L'exécution d'une partition ne peut jamais affecter ou être affectée par une partition isolée.

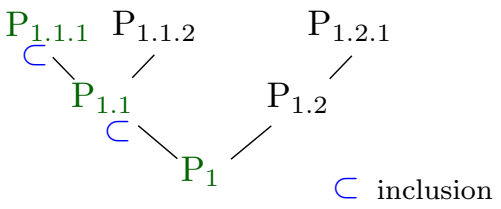


FIGURE 4.9 – Partage vertical sur une branche dans un arbre de partitions.

4.2.2.3 Communication entre partitions

Un résultat attendu du principe du TCB hiérarchique est la communication verticale entre les partitions. Grâce au partage vertical une partition parent peut communiquer avec ses descendants ainsi que partager ses ressources avec eux, et réciproquement. Ceci est indispensable pour porter des systèmes d'exploitation tels que Linux ou FreeRTOS au-dessus de Pip.

Pip fournit également un moyen pour échanger les données entre deux partitions isolées en utilisant uniquement les deux appels système `removeVaddr` et `addVaddr`. Cela se fait en déplaçant une page d'un espace d'adressage vers un autre. Cette tâche nécessite l'intervention du parent pour décider de la politique de transfert des données entre ces deux partitions. Pour ce faire, la partition parent doit implémenter son propre modèle de communication qui est spécifique à la politique de partage de ressources entre ses enfants.

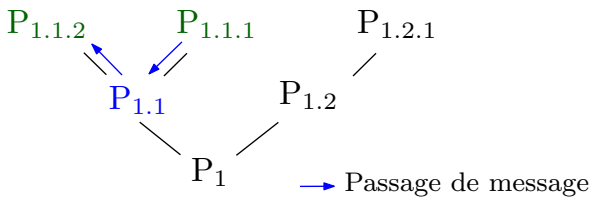


FIGURE 4.10 – Communication entre deux partitions isolées contrôlée par la partition parent.

4.3 Conclusion

Dans ce chapitre nous avons présenté d'une manière générale les concepts de base du proto-noyau Pip. En particulier, nous avons discuté la minimisation de la base de confiance qui consiste essentiellement à réduire le nombre des modules exécutés en mode privilégié. Nous avons également détaillé le modèle hiérarchique mis en place par Pip pour gérer la répartition de la mémoire entre les partitions. Cette structure hiérarchique est configurée uniquement à travers les appels système exposés par le noyau. Dans cette thèse nous avons choisi de développer un nouveau noyau de système d'exploitation en vue de pouvoir adapter le design de ce dernier à sa preuve. Le choix de l'architecture de Pip ainsi que le nombre restreint de ses services est motivé essentiellement par la réduction du coût de la preuve et la surface d'attaque. Dans la section 4.2 nous avons présenté informellement l'ensemble des propriétés de sécurité du modèle de partitionnement visé par notre vérification. Dans le chapitre suivant nous présentons tous les détails du développement de la spécification exécutable du noyau ainsi que la formalisation en Coq de l'ensemble de l'API du proto-noyau Pip.

Bibliographie

[BDF+03] Paul BARHAM, Boris DRAGOVIC, Keir FRASER, Steven HAND, Tim HARRIS, Alex HO, Rolf NEUGEBAUER, Ian PRATT et Andrew WARFIELD : Xen and the art of virtualization. *In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[BGIC18a] Quentin BERGOUGNOUX, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY : Evolution du modèle du proto-noyau pip vers les architectures smp. *In Compas 2018*, 2018.

[BGIC18b] Quentin BERGOUGNOUX, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY : Porting the pip proto-kernel's model to multi-core environments. *In 16th IEEE Intl Conf on Dependable, Autonomic and Secure Computing (IEEE-DASC'2018).*, 2018.

[EKO95] D. R. ENGLER, M. F. KAASHOEK et J. O'TOOLE, Jr. : Exokernel : An operating system architecture for application-level resource management. *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[Lie95] J. LIEDTKE : On micro-kernel construction. *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[Rus81] John RUSHBY : The design and verification of secure systems. *In Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[Rus84] John RUSHBY : A trusted computing base for embedded systems. *In Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 294–311, 1984.

[YGG+18] Mahieddine YAKER, Chrystel GABER, Gilles GRIMAUD, Jean-Philippe WARY, Julien CARTIGNY, Xiao HAN et Vicente SANCHEZ-LEIGHTON : Ensuring IoT security with an architecture based on a separation kernel. *In 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 120–127. IEEE, 2018.

[YGIC18] Mahieddine YAKER, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY : Utilisation d'un proto-noyau d'isolation mémoire pour maintenir les propriétés temps-réel d'un système à criticité mixte. *In Compas 2018*, 2018.

Chapitre 5

Le développement de la spécification exécutable

Sommaire

En premier lieu dans ce chapitre, nous présentons les détails de l'organisation en mémoire d'une partition et comment l'arbre de partition est maintenu grâce aux données sauvegardées dans les structures de données des partitions. Nous détaillons également comment gérer dynamiquement cet arbre à travers l'ensemble des appels système exposés par Pip. En particulier, nous étudions l'impact de chaque appel sur l'état du système. Ensuite, nous présentons la stratégie de développement de l'API de Pip. Celle-ci est fondée sur une architecture en deux couches où la première correspond à la spécification exécutable de Pip et la seconde est destinée à gérer l'accès au matériel. Finalement, nous décrivons la formalisation en Gallina de l'ensemble de l'API.

5.1 Gestion de l'arbre des partitions

5.1.1 Structures de données des partitions

La figure 5.1 est une illustration détaillée de la configuration d'un arbre de partition dans la mémoire physique. Au moment de la création d'une partition enfant, Pip lui attribue une première page qui contient son descripteur `PD` (*Partition Descriptor*). Pour maintenir un lien hiérarchique entre les partitions, le descripteur de chaque partition doit être une page déjà attribuée à la partition parent mais accessible uniquement en mode noyau. Le noyau utilise l'adresse physique de cette page pour identifier une partition dans l'arbre et nous l'avons appelé `PDI` (*Partition Descriptor Identifier*) [1]. La configuration d'une partition est définie essentiellement par quatre entités : La structure arborescente des tables du MMU, deux autres structures isomorphes, `shadow1` et `shadow2` et une dernière qui prend la forme d'une liste chaînée de pages appelée `linkedList`. Comme leurs noms l'indiquent, `shadow1` et `shadow2` reprennent exactement la même hiérarchie que celle des tables du MMU. Le dernier niveau de MMU contient les pointeurs vers les pages mappées de la partition et le dernier niveau de chacune des deux autres structures contient une information complémentaire sur la page mappée. Ces informations sont atteignables par le noyau en utilisant la même adresse virtuelle.

shadow1 : Elle est nécessaire pour contrôler l'attribution des pages aux partitions enfants et donc assurer l'isolation mémoire. En effet, lorsqu'une partition demande de partager une page avec son fils le noyau s'appuie sur les informations préalablement stockées (par lui-même) dans cette structure pour vérifier que la page en question n'est pas attribuée à un enfant. Si c'est le cas il marque la page comme partagée et effectue l'opération sinon il annule l'opération.

shadow2 : Elle est utile pour faciliter la récupération des pages partagées entre un père et son fils. Dans ce cas, au moment de l'attribution d'une page le noyau sauvegarde l'adresse virtuelle source (c'est-à-dire l'adresse virtuelle dans l'espace d'adressage du père) dans la structure `shadow2` de l'enfant. Ainsi, au moment de la récupération de la page il suffit de parcourir cette structure avec la même adresse virtuelle (celle qui est associée à la page physique mappée dans l'espace d'adressage de l'enfant) pour déterminer l'adresse virtuelle qui mappe la même page physique dans l'espace d'adressage du père.

En outre, **linkedList** contient des informations supplémentaires sur les pages de configuration de la partition. Le besoin de cette dernière structure s'explique par le fait que les pages de configuration attribuées à une partition ne sont pas mappées dans son propre espace d'adressage. Donc aucune de ces

1. Puisque le noyau a un accès direct à la mémoire physique, il peut identifier une partition par l'adresse physique de son descripteur. Ainsi, uniquement dans ce cas un `PDI` est considéré comme une adresse physique. En outre, une partition peut identifier ses enfants uniquement avec des adresses virtuelles dans son espace d'adressage. Donc dans ce cas un `PDI` d'une partition enfant est considéré comme une adresse virtuelle dans l'espace d'adressage de cette partition parent. Enfin, Le noyau peut, également, considérer un `PDI` comme une adresse virtuelle lorsqu'il nécessite d'identifier une partition par l'adresse virtuelle dans l'espace d'adressage de sa partition parent.

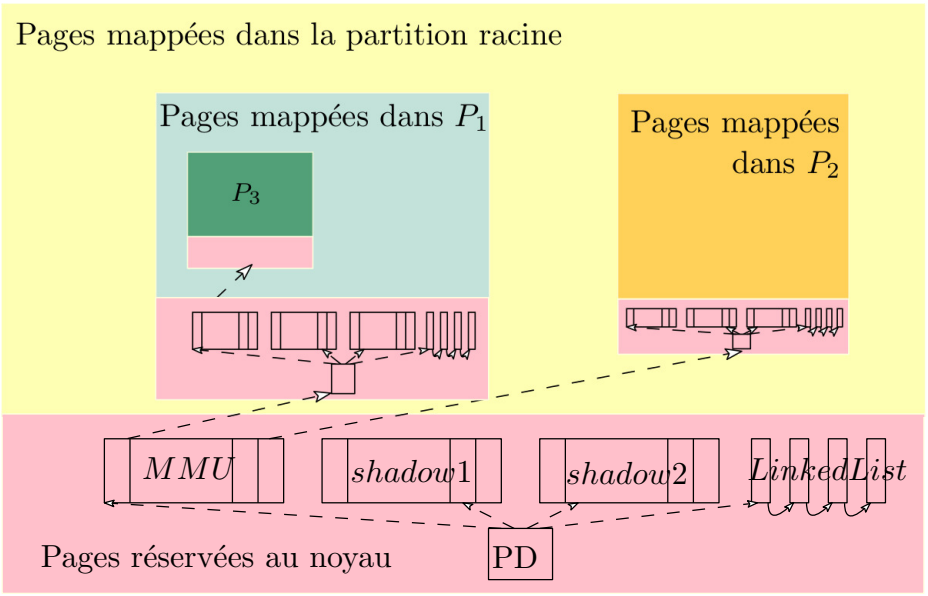


FIGURE 5.1 – La configuration d'un arbre de partition.

pages n'est atteignable par une adresse virtuelle et nous ne pouvons utiliser aucune des deux structures précédentes. Ainsi, nous avons défini une dernière structure assez simple pour les identifier et les récupérer efficacement. Le choix de ces structures est fortement motivé par la réduction du coût de la preuve et l'amélioration de l'utilisabilité du système en terme d'efficacité.

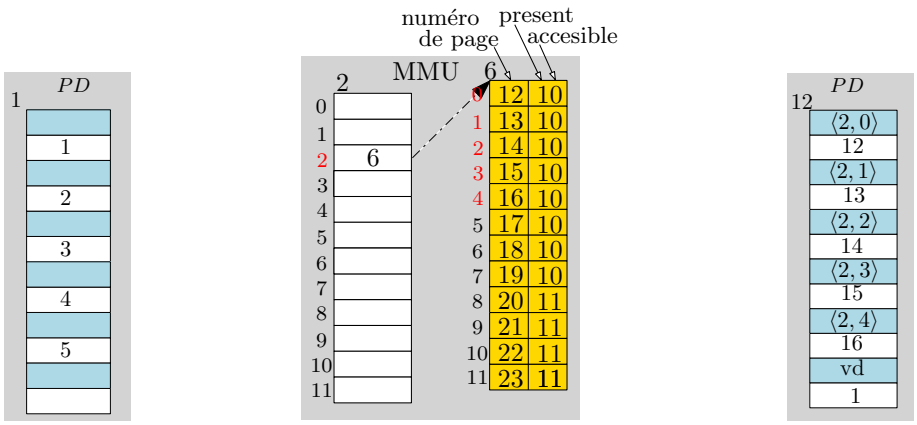
La figure 5.2 illustre les détails de la configuration des deux partitions P_2 et P_5 dans la mémoire physique tel que P_2 est le père de P_5. Dans la suite de ce paragraphe, nous allons décomposer cette figure en plusieurs parties pour faciliter sa compréhension.

Afin d'explorer les structures arborescentes des partitions, Pip s'appuie sur le même principe de traduction du MMU. Supporter n niveaux de table de traduction de MMU nécessite des adresses virtuelles de n index où chaque index correspond à une position dans le niveau correspondant de table de MMU. Dans notre exemple de configuration, nous considérons un MMU avec deux niveaux d'indirections. Donc notre adresse virtuelle est définie de la manière suivante : $\langle i, j \rangle$ où i est la position d'une entrée dans la table du premier niveau (la racine) et j est la position d'une entrée dans la table du dernier niveau. Par exemple, la figure 5.3b indique que la page 12 est atteignable par le noyau à travers l'adresse virtuelle $\langle 2, 0 \rangle$ dans l'espace d'adressage de P_2. Elle est mappée dans le père et correspond au `PDI` de son fils. Ainsi, P_2 est identifiée par la page 1 (voir figure 5.3a) et P_5 est identifiée par la page 12. Le descripteur de chaque partition contient les quatre références vers les structures de données. Par exemple, les pages 13, 14, 15 et 16 sont mappées dans le père (voir figure 5.3b) et sont également attribuées à son fils (voir figure 5.3c). Elles correspondent respectivement aux pointeurs racine du MMU, `shadow1`, `shadow2` et `linkedList` sauvegardés dans le descripteur de la partition P_5. Dans un descripteur de partition nous sauvegardons également les adresses virtuelles associées à chacune de ces pages pour faciliter leur récupération.

De plus, puisqu'elles sont utilisées comme pages de configuration de P_5, elles sont marquées comme présentes mais accessibles uniquement en mode noyau dans l'espace d'adressage de P_2. Cela est déterminé à travers les bits de contrôle (`present` et `accessible`) associés à chaque page (voir figure 5.3b).

Le noyau explore le `shadow1` en utilisant également une adresse virtuelle pour vérifier si la page associée à cette adresse est attribuée à un enfant. Cette information est stockée sous la forme d'une adresse virtuelle dans le dernier niveau d'indirection de cette structure. Si la valeur sauvegardée ne correspond pas à la valeur par défaut `vd`, utilisée pour initialiser les entrées de ce niveau d'indirection, la page associée est considérée comme attribuée et la valeur sauvegardée dans cette deuxième structure est l'adresse virtuelle de l'enfant avec qui la page a été partagée. Le noyau peut également vérifier si la même page

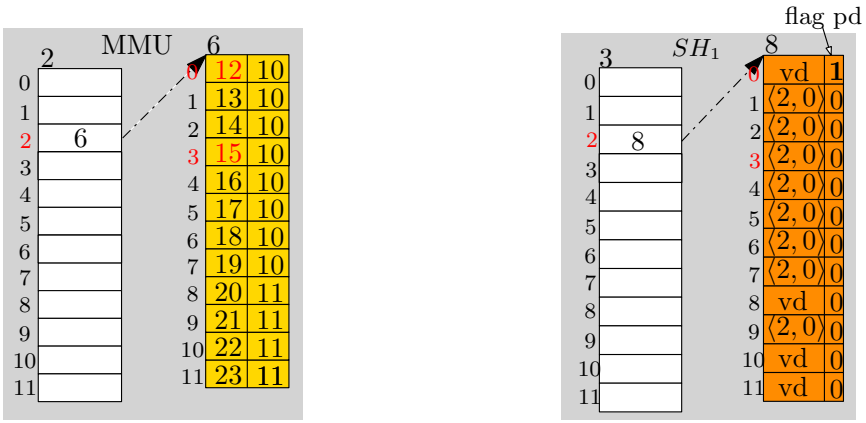
FIGURE 5.2 – Représentation détaillée de la configuration des partitions P_2 et P_5.



(a) Configuration du PD de P_2. (b) Configuration MMU de P_2. (c) Configuration du PD de P_5.

FIGURE 5.3 – Le rôle du descripteur de partition pour accéder aux différentes structures de données d'une partition.

physique correspond à un PDI en consultant la valeur du flag pd stocké dans la même entrée. La figure 5.4 montre que l'adresse virtuelle $\langle 2,0\rangle$ associée à la page 12 correspond à un descripteur de partition et l'adresse virtuelle $\langle 2,3\rangle$ associée à la page 15 est attribuée à l'enfant $\langle 2,0\rangle$.

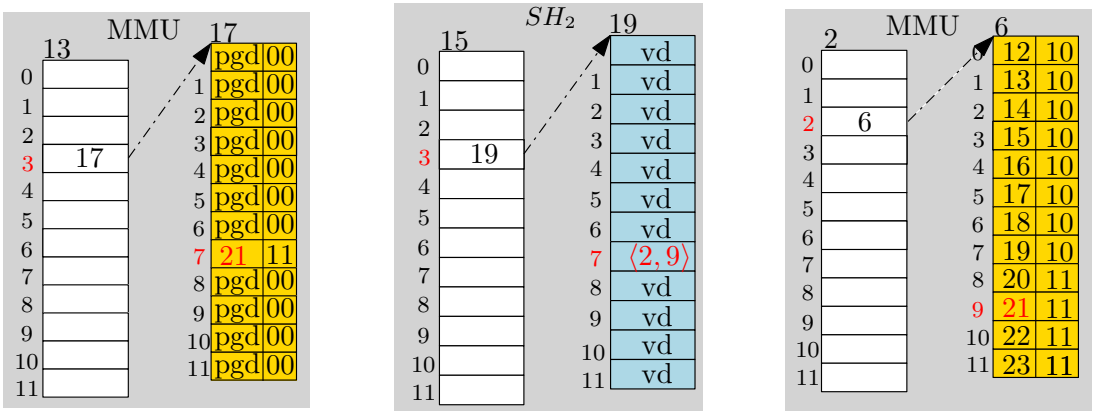


(a) La configuration du MMU de P_2. (b) La configuration du shadow2 de P_2.

FIGURE 5.4 – Le rôle de la structure shadow1 dans le contrôle de l'attribution des pages.

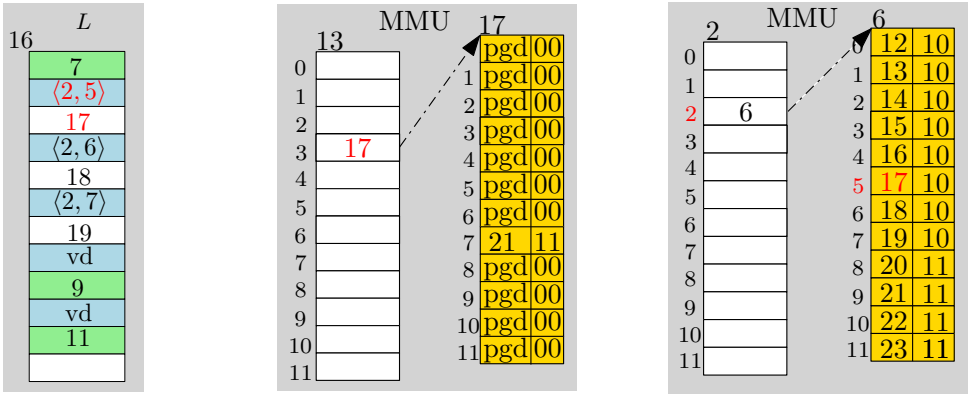
Pour retirer une page mappée, le noyau explore la structure shadow2 de l'enfant en utilisant pareillement l'adresse virtuelle associée à cette page dans l'espace d'adressage de l'enfant. Ceci permet au noyau de déterminer facilement l'adresse virtuelle qui mappe la même page physique dans l'espace d'adressage du père et évite de parcourir ce dernier dans le but de retrouver cette adresse virtuelle. La configuration de la structure shadow2 de la partition P_5 (voir figure 5.5b) indique que la page physique 21 associée à l'adresse virtuelle $\langle 3,7\rangle$ (voir figure 5.5a) est également associée à l'adresse virtuelle $\langle 2,9\rangle$ dans l'espace d'adressage de P_2 (voir figure 5.5c).

Pour retirer une page de configuration, le noyau doit parcourir la dernière structure linkedList. Chaque élément de la liste est une page physique où sa dernière entrée pointe sur la page suivante. Une page est composée d'un ensemble de pair d'entrées. Chaque paire associe une page de configuration à son adresse virtuelle dans l'espace d'adressage de la partition parent. Sur la figure 5.2, et selon les informations stockées dans la structure linkedList (voir figure 5.6a) la page 17 qui correspond à la dernière table d'indirection de MMU de la partition P_5 (voir figure 5.6b) est mappée à l'adresse virtuelle $\langle 2,5\rangle$ dans l'espace d'adressage de P_2 comme indiqué par la figure 5.6c.



(a) Configuration du `MMU` de P_5 (b) Configuration du `shadow2` de P_5 (c) Configuration du `MMU` de P_2

FIGURE 5.5 – Le rôle de la structure `shadow2` dans la révocation des pages mappées.



(a) Configuration de `linkedList` de P_5 (b) Configuration du `MMU` de P_5 (c) Configuration du MMU de P_2

FIGURE 5.6 – Le rôle de la structure `linkedList` dans la récupération des pages de configuration d'une partition.

5.1.2 API minimale de Pip

Dans la section 4.1.3 nous avons présenté brièvement ce que l'utilisateur a besoin de savoir pour pouvoir utiliser l'API correctement. Dans cette section nous détaillons le fonctionnement de chaque appel système et nous mettons l'accent sur l'ensemble des structures concernées par la modification.

createPartition crée une partition enfant en mettant en place son descripteur ainsi que les racines de ses structures de données : MMU, `shadow1`, `shadow2` et `linkedList`. Cet appel système prend en paramètre cinq adresses virtuelles. La page physique associée à chaque adresse virtuelle sera utilisée comme page de configuration de la nouvelle partition. Le noyau doit vérifier un ensemble de propriétés avant d'effectuer l'attribution des pages. Principalement, ces pages doivent être accessibles dans l'espace d'adressage de la partition courante et non attribuées à l'un de ses enfants. Ensuite, avant de configurer la nouvelle partition, le noyau rend ces pages accessibles uniquement en mode privilégié dans la partition courante ainsi que dans tous ses ancêtres y compris la partition racine et les marque comme attribuées dans la structure `shadow1` de la partition courante. Puisque ces pages étaient accessibles par la partition il est important de les initialiser avec des valeurs par défaut pour éviter une configuration erronée.

addVaddr cet appel système modifie exactement le contenu de trois tables. Reprenons la figure 5.2. Supposons que nous souhaitons mapper la page 21 dans l'espace d'adressage de P_5 à l'adresse virtuelle $\langle 2, 7\rangle$. Si les tables d'indirection du MMU (i.e. 13 et 17) ne sont pas mises en place au préalable par le noyau (en utilisant l'appel système `prepare`) cet appel système échoue. Si la page est accessible par la partition courante et n'est attribuée à aucun de ses enfants, le noyau sauvegarde la référence de la page à partager dans la dernière indirection du MMU, met à jour la dernière indirection de la structure `shadow2` de l'enfant et marque la page comme partagée dans la structure `shadow1` de la partition courante.

removeVaddr cet appel système doit supprimer les informations qui avaient été mises en place par `addVaddr`. La page concernée par la révocation doit être accessible et non partagée par l'enfant avec d'autres descendants.

mappedInChild retourne à la partition le `PDI` de son enfant (sous la forme d'une adresse virtuelle) auquel une page a été attribuée. Cet appel système effectue un accès à la structure `shadow1` de la partition courante à l'adresse virtuelle associée à la page physique concernée. Si cette page n'est associé à aucun enfant, une valeur par défaut est retournée.

prepare cet appel système attribue de nouvelles tables d'indirections à un enfant pour associer une nouvelle page physique à une adresse virtuelle. En effet, la partition courante doit fournir au noyau les pages nécessaires, sous la forme d'adresse virtuelles, pour effectuer la configuration. Ces pages doivent être accessibles et n'être attribuées à aucun enfant. Ce service prend quatre paramètres. Le premier est l'ad-resse virtuelle de l'enfant. Le second est l'adresse virtuelle à configurer dans l'enfant. Le troisième est l'adresse virtuelle de la première page à attribuer à l'enfant en tant que table de configuration. Le dernier paramètre est un booléen précisant au noyau si la partition nécessite une page supplémentaire pour configurer la dernière structure `linkedList`.

Il est important de noter que le nombre de pages à fournir par la partition varie selon sa configuration. C'est pour cela que l'ensemble des pages est fournie au noyau sous la forme d'une liste chaînée. Cette liste doit être encodée par la partition afin que l'adresse virtuelle de chaque page soit un élément de la liste. Il suffit de fournir le pointeur vers la première adresse virtuelle puis le noyau récupère les adresses des autres pages au moment de la configuration ;

count permet à la partition de calculer le nombre de pages à fournir au noyau pour mapper une nouvelle page physique. Ceci est effectué en parcourant les tables de MMU associées à l'adresse virtuelle concernée ainsi que la structure `linkedList` pour vérifier si une page supplémentaire est nécessaire.

collect permet à un père de récupérer un ensemble de pages utilisées comme tables de configuration. Ces tables doivent être vides. En effet, le père doit fournir une adresse virtuelle et le noyau s'occupe de vérifier si les tables de traduction associées à cette adresse contiennent des données utiles. À la fin de l'exécution de ce service l'ensemble des pages inutiles sont de nouveau accessibles par le père ainsi que par ses ancêtres. L'adresse virtuelle de chaque page physique (utilisée comme page de configuration) est identifiée facilement grâce à la structure `linkedList`.

deletePartition la suppression d'une partition consiste à récupérer toutes les pages attribuées à une partition enfant. La récupération des pages mappés est effectuée grâce à leurs adresses virtuelles sauvegardées dans la structure `shadow2` de l'enfant, et les pages de configuration sont récupérées grâce aux adresses virtuelles sauvegardées dans la structure `linkedList` de l'enfant.

En plus de la gestion de la mémoire nous avons défini deux autres services pour gérer le flot de contrôle des partitions.

dispatch l'activation d'une partition nécessite plusieurs vérifications par le noyau pour éviter l'exécution d'une entité qui ne correspond pas à une partition dans l'arbre de partition. Ceci est effectué grâce au données sauvegardées par le noyau dans les structures de données des partitions. Il utilise les configurations du MMU et du `shadow1` pour vérifier que c'est bien une partition enfant et le `PDI` de la partition pour activer le père de la partition en exécution.

resume la restauration du contexte est effectuée grâce à une structure de données, que nous avons appelée `vidt` (*virtual interrupt description table*), sauvegardée dans l'espace d'adressage de la partition. Cette structure contient toutes les informations sur le contexte de la partition pour assurer la reprise de son exécution.

Comme nous avons vu précédemment, ces dix appels système permettent uniquement la gestion des structures internes des partitions et le flot de contrôle. En revanche, tous les mécanismes qui sont liés à la communication entre ces entités et le multiplexage des ressources ne font pas partie du noyau. Cela est tout à fait raisonnable puisque ces mécanismes peuvent être implémentés à l'extérieur du noyau en utilisant uniquement les services décrits ci-dessus. L'avantage de cette approche est essentiellement de minimiser la taille du TCB et donc faciliter la preuve. De plus, cela laisse plus de liberté à la partition de mettre en place son propre modèle de communication ou d'ordonnancement. La vérification de cette fonctionnalité peut être établie en s'appuyant sur les propriétés d'isolation sans aucune interférence avec les preuves existantes.

5.2 Formalisation en Coq de l'API

Notre preuve consiste à établir la preuve des propriétés de sécurité directement sur le code sans passer par des modèles abstraits. Le développement du proto-noyau Pip est fondé sur une structure en couche comme présentée sur la figure 5.7. La première couche (à gauche) est la formalisation de l'API en Coq. Elle correspond à la spécification exécutable de l'ensemble des appels système du noyau allant de la gestion de l'arbre de partition jusqu'à la gestion du flot de contrôle. Elle est complètement indépendante de l'architecture matérielle. Cet API est formalisé au-dessus d'une couche d'abstraction du matériel que nous avons appelé HAL (*Hardware Abstraction Layer*). Cette dernière correspond à l'unique interface entre le noyau et l'architecture matérielle. Elle consiste en un ensemble de primitives écrites

en C et en assembleur et permettant au noyau d'accéder au matériel. Nous avons modélisé chacune de ces primitives en Gallina pour pouvoir raisonner sur le comportement des appels système.

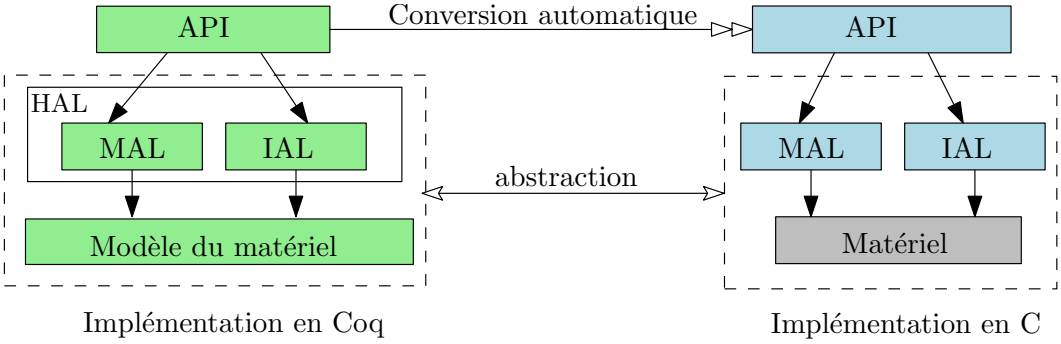


FIGURE 5.7 – Le développement du proto-noyau Pip.

5.2.1 Abstraction du matériel

Nous avons vu dans la section précédente que la configuration des partitions nécessite plusieurs structures qui contiennent des valeurs de types différents. Puisqu'il s'agit des données manipulées par le noyau, il est donc possible de prévoir tous les types de données à ranger ou à récupérer de la mémoire. Dans la suite, nous introduisons la formalisation en Gallina de tous les types de données sauvegardées dans le modèle de la mémoire physique. Ceci permet principalement d'assurer la cohérence du calcul effectué par le noyau et de vérifier facilement que les valeurs utilisées sont bien typées. Nous utilisons les types dépendants de la librairie standard de Coq pour définir les types des données manipulées par le noyau. Par analogie à la formalisation en Coq de la monade `H` (voir section 3.1.2) du micro-noyau MIMIC, nous utilisons le constructeur `LLI` pour définir la monade d'état de Pip.

```
Definition LLI (A :Type) : Type := state → result (A * state).
Inductive result (A : Type) : Type :=
| val : A → result A
| undef : nat → state → result A.
```

5.2.1.1 Types abstraits

Le type `state` correspond à la formalisation en Gallina de l'état de la machine. Il comprend les différents composants de l'état permettant de définir la spécification du comportement du noyau.

```
Record state: Type:= { currentPartition: page;
                       memory: list (paddr * value)}.
```

Mise à part la référence vers l'identifiant du descripteur (`PDI`) de la partition en cours d'exécution définie par le premier élément de l'état `currentPartition`, toutes les structures de données gérées par le noyau sont modélisées dans la mémoire physique. Cette dernière correspond au deuxième élément `memory` du modèle de l'état du système.

Au niveau du HAL, la mémoire physique est modélisée par une liste d'association qui mappe chaque adresse physique, réservée au noyau, à sa valeur. Ce modèle est complètement abstrait par rapport aux données des applications. Ceci est tout à fait raisonnable car pour vérifier les propriétés du modèle de partitionnement et d'isolation en général, nous avons besoin uniquement des valeurs stockées par le noyau. Ces données servent à définir l'état du matériel et les structures internes des partitions.

La taille de la mémoire physique est déterminée en utilisant deux paramètres liés à l'architecture matérielle `pageSize` et `nbPage`. Le premier paramètre est la taille d'une page. Le deuxième est le nombre de pages dans la mémoire physique.

En effet, chaque page est identifiée par un numéro de type `page`. Ce numéro correspond au pointeur vers la première valeur dans la page concernée. `page` est un type dépendant, il s'agit d'un entier naturel (i.e. `nat` en Gallina) strictement inférieure à `nbPage`.

```
Record page: Type := {p :> nat;  Hp : p < nbPage}.
```

Ce type est utilisé pour stocker les références des pages dans les tables de configuration. Il est à noter qu'une table de configuration a exactement la taille d'une page.

À partir de ce type nous avons défini le type d'une adresse physique dans la mémoire que nous avons appelé `paddr`. Il s'agit d'une référence vers une page et une position dans cette page.

```
Definition paddr: Type := (page * index).
```

où `index` est également un type dépendant exprimant une position dans une page mémoire. Sa valeur doit être strictement inférieure à la taille d'une page.

```
Record index: Type := {i :> nat ; Hi : i < pageSize}.
```

L'accès direct aux données du noyau est modélisé dans le HAL par une fonction générique que nous avons appelé `lookup`. À travers une page et une position dans cette page le noyau peut récupérer une valeur de type `value` stockée dans la liste d'association qui correspond au modèle de la mémoire physique.

```
Inductive value: Type :=
 |PE: Pentry → value
 |VE: Ventry → value
 |PP: page → value
 |VA: vaddr  → value
 |I:  index  → value.
```

`value` est un type inductif permettant d'exprimer plus de détails sur cette valeur. Chaque constructeur `PE` , `VE`, `PP`, `VA` et `I` est associé à un type différent. En effet, le noyau peut sauvegarder une valeur de type `page` sous le label `PP` et une valeur de type `index` sous le label `I`.

De plus, suivant la configuration des structures de données des partitions, nous avons besoins d'autres types plus complexes. Pour cela nous avons défini le type `Pentry` pour sauvegarder les données d'une entrée de table de page de MMU. Dans cette entrée le noyau stocke la référence vers une page ainsi que les informations sur les droits d'accès de cette page. Les données de type `Pentry` sont sauvegardées sous le label `PE`.

```
Record Pentry: Type:=
{read: bool; write: bool; exec: bool; present: bool; user: bool; pa: page}.
```

Nous avons également défini le type `Ventry` pour sauvegarder des données dans les entrées des dernières indirections de la structure `shadow1`.

```
Record Ventry: Type:=
{pd: bool; va: vaddr}.
```

Le dernier type est `vaddr`. Il est nécessaire pour définir les adresses virtuelles et les sauvegarder dans la mémoire physique. Il consiste en une liste de positions de type `index`. La taille de cette liste dépend du nombre des niveaux d'indirection que nous avons appelé `nbLevel`.

```
Record vaddr: Type:=
{va:> list index ; Hva: length va = nbLevel + 1}.
```

En utilisant les propriétés de chaque valeur il est possible de vérifier, en tant que propriétés de cohérence, que les structures internes des partitions sont bien typées par rapport à ces valeurs et prouver qu'elles sont préservées par tous les appels système du noyau.

Notons que la modélisation de la mémoire physique par une liste d'association présuppose que la mémoire a un caractère non volatile. En effet, comme détaillé auparavant une partie importante de notre vérification consiste à prouver que les données stockées par le noyau dans les structures des partitions sont cohérentes et que cette propriété est préservée. Pour que ceci soit toujours vrai, il est primordial que ces informations soient conservées tant que le noyau ne les a pas modifiées. Dans le cas contraire nous n'aurions aucune garantie sur les données sauvegardées dans les structures de données des partitions et donc il ne serait pas possible de raisonner sur les appels système.

5.2.1.2 Primitives du HAL

Dans cette section nous détaillons la modélisation en Coq de quelques primitives HAL. Ces opérations nécessite un accès à l'état de la machine. Nous définissons ainsi la fonction `get` pour récupérer un état de type `state` et une fonction `put` pour modifier l'état. Nous définissons aussi la fonction `modify` qui applique une fonction sur l'état :

```
Definition modify (f : state → state) : LLI unit :=
  perform s := get in put (f s).
```

Ces fonctions sont uniquement utilisées par la couche d'abstraction du matériel afin de modéliser les primitives d'accès au matériel. Elles ne sont bien sûr pas utilisées dans le code source de Pip. Au niveau du HAL nous distinguons quatre catégories de primitives :

- La première regroupe l'ensemble des primitives nécessaires pour sauvegarder les données du noyau dans la mémoire physique. Pour pouvoir effectuer ces opérations nous avons défini, pour chaque type de valeur présenté ci-dessus, une primitive permettant de ranger une valeur de ce type sous le label associé. Pour résumer, nous avons modélisé les primitives suivantes : `writeVirtual`, `writePhysical`, `writeVirEntry`, `writePhyEntry`, `writeAccessible`, `writePresent`, `writePDflag`, et `writeIndex`.

  Prenons l'exemple de la primitive `writeVirtual`. Elle est modélisée de la manière suivante :

  ```
  Definition writeVirtual (p: page) (idx: index) (va: vaddr): LLI unit:=
   modify (fun s ⇒ {| currentPartition := s.(currentPartition);
     memory := add paddr idx (VA va) s.(memory) beqPage beqIndex|}).
  ```

  C'est une fonction de type monadique. Elle permet de sauvegarder une adresse virtuelle `va` dans une adresse physique donnée. L'adresse physique est identifiée par une page `p` et une position `idx` dans cette page. L'opération `add` est utilisée pour ajouter un nouvel élément dans la liste d'association.

- La catégorie suivante concerne les primitives permettant de récupérer des données du noyau de la mémoire. Ce sont : `readVirtual`, `readPhysical`, `readVirEntry`, `readPhyEntry`, `readAccessible`, `readPresent`, `readPDflag`, et `readIndex`.

  Par exemple, la primitive `readVirtual` permet de récupérer la valeur d'une adresse virtuelle préalablement sauvegardée par le noyau à partir d'une adresse physique donnée.

  ```
  Definition readVirtual (p: page) (idx: index): LLI vaddr:=
  perform s := get in
  match lookup p idx s.(memory) beqPage beqIndex with
    |Some (VA v) ⇒ ret v
    |Some _ ⇒ undefined 3
    |None ⇒ undefined 2
  end.
  ```

Pour que cet accès soit valide cette adresse mémoire doit nécessairement contenir une adresse virtuelle sinon il est considéré comme un comportement non défini.

- La troisième catégorie consiste en un ensemble d'opérations permettant d'effectuer des calculs sur les valeurs manipulées par le noyau. Prenons la primitive `succ`. Elle est nécessaire pour calculer le successeur d'une position dans un niveau d'indirection du MMU.

```
Program Definition succ (n : index) : LLI index :=
let isucc := n+1 in
if (lt_dec isucc tableSize)
then
  ret (Build_index isucc _)
else  undefined 28.
```

Une particularité de cette primitive est que si la valeur calculée du successeur dépasse la taille d'une page ceci est considéré comme un comportement indéfini. Cette modélisation permet d'assurer que le noyau n'écrit jamais à l'extérieur de l'espace réservé à une page et n'écrase ainsi des données réservées à d'autres structures.

- La dernière catégorie assure la récupération de certaines informations liées à l'architecture matérielle ou à la configuration des structures des partitions. Par exemple, l'opération `getSh1idx` permet de récupérer la position de la référence de la racine de la structure `shadow1` dans un `PDI`.

```
Definition getSh1idx : LLI index:= ret sh1idx.
```

où `sh1idx` est la position dans le `PDI`.

Pour résumer, une primitive HAL consiste simplement en une opération élémentaire, du point de vue du modèle, permettant d'accéder à la mémoire, effectuer un calcul logique ou alors récupérer la valeur d'un paramètre lié à l'architecture. Il est important de noter que plusieurs primitives du HAL appartenant à la même catégorie peuvent être associées à une unique fonction en C. Ceci est tout à fait raisonnable puisque différents types définis en Gallina pourront être également associés à un unique type en C.

5.2.2 API de Pip en Gallina

Tous les appels système exposés par le noyau sont initialement écrits en Gallina dans un style impératif grâce à une monade d'état qui permet d'introduire et gérer facilement toute sorte d'effet de bord tel que la modification de l'état du système. Ces appels système sont automatiquement traduits de Gallina vers C en utilisant *Digger* [HO17] (un outil écrit en Haskell). Ce dernier est un traducteur qui a été développé pour traduire un sous-ensemble limité du langage gallina en C, précisément le sous ensemble utilisé pour développer les appels système de Pip.

L'exemple suivant correspond à une fonction interne de Pip (`getFstShadow`), écrite en Gallina et sa traduction *mot à mot* en C. Cette fonction permet de récupérer une référence vers une structure de configuration d'une partition. Elle consiste en une séquence d'appels à trois primitives : `getSh1idx`, `succ` et `readPhysical` et retourne la référence en question.

Le code Gallina (à gauche) sera ainsi automatiquement traduit dans le code C (à droite) :

```
Definition getFstShadow (part: page):
                          LLI page :=
  perform idx := getSh1idx in
  perform idxSucc := Index.succ idx in
  readPhysical part idxSsucc.
```

```
uintptr_t getFstShadow (uintptr_t part) {

  const uint32_t idx = getSh1idx();
  const uint32_t idxSsucc = succ(idx);
  return readPhysical(part, idxSsucc); }
```

Dans un premier prototype nous avions utilisé l'outil d'extraction fourni par l'assistant de preuve Coq pour convertir l'API de Pip de Gallina vers Haskell. Ceci nous a permis de tester les premières versions de Pip dans un langage fonctionnel avec un environnement d'exécution. Toutefois, comme la plupart des langages fonctionnels, cet environnement d'exécution repose sur un ramasse-miette qui introduisait une latence importante à l'exécution du noyau à cause de la gestion automatique de la mémoire physique. Cela n'est pas acceptable dans le contexte d'un noyau de système d'exploitation. D'une part, ce mécanisme dégrade fortement les performances du système et d'autre part, nous n'avons aucune confiance sur l'effet de bord de celui-ci sur la mémoire. Donc comme seconde étape nous avons travaillé sur la traduction automatique de la spécification exécutable de l'API de Gallina vers C.

En effet, le code monadique que nous avons développé pour la spécification exécutable des appels système est bien similaire à un code impératif. Nous avons utilisé, par exemple, uniquement des récursions terminales. Ces récursions se terminent toujours et il est possible de les programmer à l'aide de l'instruction itérative *for* dans un langage impératif. De plus, les structures de données gérées par Pip telles que les listes et les arbres ne sont pas formalisés par les types inductifs fournis par la librairie standard de Coq. Mais elles sont encodées directement dans la mémoire avec des pointeurs. Ceci est important pour faciliter la traduction automatique de Gallina vers C mais nécessite une preuve supplémentaire pour s'assurer que les structures sont encodées correctement dans la mémoire.

Concrètement, ce code monadique correspond à un *shallow embedding* permettant d'exprimer le comportement des programmes similairement à leur développement dans le langage ciblé par la traduction.

Parmi les difficultés rencontrées pendant le développement de l'API en Gallina est le fait que ce langage fonctionnel exige que toutes les fonctions récursives se terminent. Cette terminaison est détectée par Coq uniquement lorsque l'argument de récursion est plus petit dans l'appel récursif suivant un ordre bien fondé. Mais il n'est pas évident de rendre explicite que le parcours des structures de données gérées par Pip se termine puisqu'elles sont encodées dans la mémoire avec des pointeurs. Usuellement, nous utilisons un argument de récursion de type inductif tel que `list`. Cependant, ces types n'ont pas d'équivalent en C. Notre solution pour ce problème est fondée sur le fait que toutes les fonctions récursives utilisées pour définir les appels système gérés par Pip sont bornées. Le nombre d'appels récursifs est calculé en fonction de certains paramètres de l'architecture matérielle tels que la taille de la mémoire. Dans ce cas, il est nécessaire de vérifier, pour chaque fonction, que la borne est assez grande pour arriver à la fin de la récursion.

5.2.3 Confiance en HAL

Concrètement, le comportement du matériel est plus compliqué que notre modèle abstrait du HAL en Gallina. En plus de la mémoire physique, il peut être nécessaire de gérer d'autres composants internes tels que les caches ou le TLB. Nous considérons que le HAL représente une abstraction du comportement du matériel géré par le noyau. C'est une manière de formaliser l'idée que le développeur se fait du comportement du matériel tout en gardant une formalisation indépendante d'une architecture en particulier.

Nous partons du principe que la vérification des propriétés de sécurité nécessite dans un premier temps d'identifier clairement le rôle du noyau et celui du matériel dans le contexte de la sécurité. Cela nous permet de découpler l'implémentation du noyau des hypothèses sur lesquelles reposent les propriétés de sécurité. Le but de nos travaux est de vérifier les propriétés du noyau, donc la manière dont le matériel effectue sa tâche devient un problème orthogonal et nécessite une preuve complémentaire [BBCL12, SK17, KGG+18, LSG+18].

Par exemple, Pip contrôle l'accès des processus à la mémoire physique à travers le MMU. Donc nous prouvons que la propriété de sécurité est préservée à travers une configuration correcte des tables de MMU. Cette configuration est effectuée par le noyau en s'appuyant sur l'hypothèse que le MMU traduit

correctement les adresses virtuelles en adresses physiques.

De même, la configuration des structures internes des partitions nécessite des opérations d'écriture dans la mémoire physique. Le rôle du matériel ici est d'effectuer la modification. En revanche c'est le noyau qui doit fournir les valeurs. Par conséquent, la vérification porte essentiellement sur les valeurs fournies par le noyau en fonction desquelles le matériel effectue l'opération. L'exemple suivant permet de comparer l'implémentation en C d'une primitive HAL avec son modèle en Gallina.

La primitive `readPhysical` en C :

```
uint32_t readPhysical(uint32_t table, uint32_t idx) {
    disable_paging();  /* En mode noyau : nous pouvons désactiver le MMU */
    uint32_t dest = table | (idx * sizeof(uint32_t));
    uint32_t val = *(uint32_t *) dest;              /* Récupère l'entrée */
    enable_paging();                                /* Réactive le MMU */
    return val & 0xFFFFF000;                        /* Retourne la valeur */
}
```

Son modèle en Gallina :

```
Definition readPhysical (paddr : page) (idx : index) : LLI page :=
 (* Récupère l'état *)
  perform s := get in
(* Récupère l'entrée *)
  let e := lookup paddr idx s.(memory) beqPage beqIndex in
  match e with
(* Retourne la valeur si elle est de type Page *)
   | Some (PP a) ⇒ ret a
(* Sinon signale un comportement indéfini *)
   | _           ⇒ undefined 5
  end.
```

Cette primitive, `readPhysical`, retourne l'adresse physique sauvegardée à l'adresse passée en paramètre. Dans le contexte de `getFstShadow`, la valeur retournée par cette primitive correspond à une valeur dans une page de configuration d'une partition. En C, il s'agit simplement de récupérer la valeur sauvegardée dans `table` à la position `idx`. Par contre, son modèle en Coq, qui est une fonction monadique, met en place un test supplémentaire permettant de vérifier que le noyau avait déjà sauvegardé une valeur de type *page* (i.e. PP) à cette adresse physique sinon c'est considéré comme un comportement indéfini. Dans ce cas, pour prouver que cette primitive retourne bien un numéro d'une page physique (et donc éviter un comportement indéfini), il suffit de prouver que les arguments fournis par le noyau ont les propriétés nécessaires pour satisfaire le test défini dans le modèle en Coq et n'engendrent pas un comportement indéfini. Donc concrètement, nous supposons que l'implémentation de `readPhysical` en C est correcte mais nous prouvons que les valeurs `table` et `idx` fournies par le noyau ne produisent pas un comportement indéfini.

5.3 Conclusion

Ce chapitre était consacré à la présentation de la spécification exécutable du proto-noyau Pip. Nous avons expliqué d'une manière détaillée les caractéristiques de chaque structure mise en place par Pip pour gérer efficacement l'arbre de partition. Ces structures sont indispensables pour stocker des données permettant au noyau d'assurer à la fois l'isolation mémoire et l'efficacité durant l'exécution de ses appels système. Dans ce chapitre nous avons également détaillé le schéma général de développement de l'API de Pip en Coq. Le développement de ces appels système a été effectué sous la forme de deux

couches. La première consiste en un ensemble de primitives non-calculatoires qui montent des instructions assembleur élémentaires au niveau Gallina permettant d'interagir avec le matériel. Au-dessus de cette couche nous avons développé l'ensemble des appels système de Pip. En utilisant l'outil de conversion *digger* cette couche supérieure est convertie automatiquement de Gallina vers C. L'intérêt de cette stratégie de développement est de rendre possible l'exécution du code du noyau (initialement écrit dans un langage fonctionnel) dans un langage de bas niveau et aussi d'être capable d'établir la preuve de propriété d'isolation directement sur le code.

Bibliographie

[BBCL12] Gilles BARTHE, Gustavo BETARTE, Juan Diego CAMPO et Carlos LUNA : Cache-leakage resilient OS isolation in an idealized model of virtualization. *In Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 186–197. IEEE, 2012.

[HO17] S. HYM et V OUDJAIL : Digger, 2017. https://github.com/2xs/digger.

[KGG+18] Paul KOCHER, Daniel GENKIN, Daniel GRUSS, Werner HAAS, Mike HAMBURG, Moritz LIPP, Stefan MANGARD, Thomas PRESCHER, Michael SCHWARZ et Yuval YAROM : Spectre attacks : Exploiting speculative execution. *arXiv preprint arXiv :1801.01203*, 2018.

[LSG+18] Moritz LIPP, Michael SCHWARZ, Daniel GRUSS, Thomas PRESCHER, Werner HAAS, Stefan MANGARD, Paul KOCHER, Daniel GENKIN, Yuval YAROM et Mike HAMBURG : Meltdown. *arXiv preprint arXiv :1801.01207*, 2018.

[SK17] Hira SYEDA et Gerwin KLEIN : Reasoning about translation lookaside buffers. *In LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 490–508, 2017.

Chapitre 6

Vérification du modèle de partitionnement

Sommaire

Dans ce chapitre nous abordons plusieurs aspects autour de la vérification des propriétés d'isolation du proto-noyau Pip. Dans un premier temps nous présentons la formalisation en Coq des différentes propriétés détaillées dans la section 4.2. Ensuite, en s'appuyant sur la preuve de l'appel système `addVaddr` nous discutons les différentes étapes de notre démarche de vérification, nous détaillons la spécification de quelques propriétés de cohérence et nous mettons également en lumière les difficultés rencontrées durant l'établissement de la preuve. À la fin de ce chapitre nous décrivons comment nous avons bénéficié des propriétés d'isolation et de cohérence pour vérifier des propriétés de haut niveau telles que la non-interférence.

6.1 Formalisation des propriétés en Coq

6.1.1 Isolation du noyau

Cette propriété est nécessaire pour assurer l'isolation du noyau des accès non légitimes provenant des applications. Sa formalisation en Coq considère deux partitions quelconques `partition1` et `partition2` de l'arbre de partition. Elle consiste à assurer qu'il n'y a pas d'intersection entre l'ensemble des pages accessibles attribuées à la première partition et les pages de configuration de la deuxième partition. Étant données que les deux partitions ne sont pas obligatoirement différentes cette propriété permet également d'assurer que les pages de configuration de la partition ne sont pas accessible par la partition elle-même.

```
Definition KI (s: state): Prop := ∀ partition1 partition2: page,
  partition1 ∈ (partitionTree s) →  partition2 ∈ (partitionTree s) →
  (accessiblePages partition1 s) ∩  (kernelPages partition2 s) = ∅ .
```

où

- `partitionTree s` : C'est la liste des `PDI` de toutes les partitions. Cette liste est construite par un parcours récursif de l'arbre de partition à partir de la partition racine où le nombre des appels récursifs est borné par le nombre de pages dans la mémoire physique. En effet, à partir de la configuration de chaque partition il est possible de récupérer l'ensemble de ses partitions enfant. Ainsi récursivement sur la liste des partitions enfant de chaque partition nous obtiendrons l'ensemble des partitions.

- `accessiblePages partition s` : C'est la liste de toutes les pages marquées comme accessibles par la configuration du MMU d'une partition à un état donné. Cette liste est construite à travers une énumération exhaustive de toutes les adresses virtuelles de l'espace d'adressage d'une partition en appliquant un filtre permettant de garder uniquement les pages physiques qui sont marquées comme accessibles.

- `kernelPages partition s` : C'est la liste de toutes les pages utilisées pour configurer les structures d'une partition à un état donné. Cette liste est construite à partir d'un parcours en largeur de chacune de ses structures arborescentes (MMU, `shadow1`, `shadow2`) ainsi que le parcours de la liste chaînée `linkedList`. Elle contient également le `PDI` de la partition.

6.1.2 Isolation horizontale

Cette propriété permet d'assurer l'isolation mémoire entre deux partitions incomparables. La formalisation de cette propriété considère trois partitions quelconques : une partition parent `parent` dans l'arbre de partition et deux enfants différents `child1` et `child2` de cette partition où il n'y a pas d'intersection entre les pages attribuées à `child1` et les pages attribuées à `child2`. Il est à noter que deux partitions sont différentes si et seulement si leurs `PDI` sont différents.

```
Definition HI (s: state): Prop := ∀ parent child1 child2 : page,
  parent ∈ (partitionTree s) → child1 ∈ (children parent s) →
  child2 ∈ (children parent s) → child1 ≠ child2 →
  (attributedPages child1 s) ∩  (attributedPages child2 s) = ∅ .
```

où

- `children partition s` : C'est la liste des `PDI` de toutes les partitions enfants d'une partition à un état donné. Elle est construite, également, à travers une énumération de toutes les adresses virtuelles en appliquant un filtre permettant de tester si une page physique mappée par les tables de configuration de MMU correspond à un descripteur de partition. Ceci est identifié grâce à la structure `shadow1`.

- `attributedPages partition s` : C'est la liste de toutes les pages mappées dans l'espace d'adressage d'une partition à un état donné ainsi que les pages réservées par le noyau pour configurer les structures de données de cette partition.

6.1.3 Partage vertical

La formalisation de la dernière propriété de sécurité du modèle de partitionnement de Pip considère deux partitions : une partition parent `parent` et une partition enfant `child` vérifiant que l'ensemble des pages attribuées à `child` consiste en un sous ensemble des pages mappées dans l'espace d'adressage de `parent`.

```
Definition VS (s: state): Prop := ∀ parent child : page,
  parent ∈ (partitionTree s) → child ∈ (children parent s) →
  (attributedPages child s) ⊂  (mappedPages parent s).
```

6.2 Vérification de addVaddr

Nos travaux de vérification consistent à prouver que l'ensemble des propriétés formalisées dans la section précédente sont préservées par chaque appel système exposé par le noyau. Pour ce faire, nous utilisons le triplet de Hoare pour formaliser les invariants. Un triplet de Hoare est défini en Coq de la manière suivante :

```
Definition hoareTriple {A: Type} (P: state → Prop) (c: LLI A)
(Q: A → state → Prop): Prop :=
  ∀ s, P s → match c s with
    | val (a, s') ⇒ Q a s'
    | undef⇒ False
    end.
```

Pour simplifier l'utilisation de cette fonction nous utilisons la notation `{{P}}c{{Q}}` pour définir le triplet `hoareTriple P c Q` où `P` est la précondition de `c` et `Q` est sa postcondition. Dans cette section nous détaillons la preuve de l'appel système `addVaddr` et nous montrons que, similairement au modèle abstrait présenté dans le chapitre 3, des propriétés de cohérences (notées `C`) sont nécessaires pour vérifier que les propriétés d'isolation sont préservées. Nous vérifions également que les appels système ne génèrent pas de comportements indéfinis. Les détails de cette preuve nous permettent également de mettre en avant les difficultés rencontrées durant le processus de vérification ainsi que l'ensemble des solutions adoptées pour réduire la complexité de cette preuve.

Le triplet suivant correspond à l'invariant de `addVaddr`.

```
Lemma addVaddrInvariant (src dst child: vaddr):
{{fun (s: state) ⇒ HI s ∧ KI s ∧ VS s ∧ C s}} addVaddr src dst child
{{fun _ (s: state) ⇒ HI s ∧ KI s ∧ VS s ∧ C s }}.
```

Cet appel système est appelé par la partition en cours d'exécution pour partager l'une de ses pages accessibles (que nous appellerons par la suite *srcp*) avec un de ses enfants. Elle prend trois paramètres : l'adresse virtuelle *src* (dans l'espace d'adressage de la partition courante) associée à la page physique à partager. Une deuxième adresse virtuelle *child* (également dans l'espace d'adressage de la partition courante) doit être associée au `PDI` de l'enfant avec qui la page sera partagée. À la fin de l'exécution de `addVaddr`, *srcp* sera associée à l'adresse virtuelle *dst* dans l'espace d'adressage de l'enfant.

D'une manière générale, un appel système exposé par Pip est une séquence de primitives HAL structurée en deux parties. La première consiste à vérifier un ensemble de propriétés sur les paramètres en effectuant les accès nécessaires à l'état du système. Ensuite, si toutes ces propriétés sont satisfaites l'état sera modifié. Ceci est très important pour éviter toute incohérence dans les structures de données due à une modification partielle de l'état. La figure 6.1 illustre une structure simplifiée de l'implémentation de `addVaddr` :

sysCall addVaddr : vaddr → vaddr → vaddr → LLI(bool)
Input : *src* : l'adresse virtuelle source
dst : l'adresse virtuelle destination
child : l'adresse virtuelle de l'enfant
Action:
c1 ← vérifier la validité de *src*;
c2 ← vérifier la validité de *dest*;
c3 ← vérifier la validité de *child*;
if *c1 c2 c3* **then**
  update_state(*src*, *dst*, *child*);
else
  retourner(*false*)
end

FIGURE 6.1 – Structure simplifiée de l'appel système `addVaddr`

6.2.1 Validation des paramètres de l'appel système

Le triplet `addVaddrInvariantProgress` définit l'invariant (simplifié) de `addVaddr` après la propagation de toutes les propriétés liées la vérification de la validité de ses paramètres. Avant de détailler ce triplet, il est important de noter qu'en plus de la propagation des propriétés de sécurité et de cohérence à travers les instructions, le processus de vérification consiste également à propager un ensemble de propriétés spécifiques à l'appel système. Ces propriétés sont appelées les **propriétés internes**. Elles sont propagées à travers la séquence des instructions qui suivent jusqu'à ce qu'elles ne soient plus vraies ou bien plus utiles. Cette propagation permet de garder toutes les informations sur les paramètres, les variables locales ainsi que l'état de la machine, nécessaires pour prouver que l'appel système préserve toutes les propriétés d'isolation et de cohérence.

```
Lemma addVaddrInvariantProgress :
{{fun s ⇒ HI s ∧ KI s ∧ VS s ∧ C s ∧ isChild child s ∧ isPresent srcp s ∧
 isAccessible srcp s ∧ notShared srcp s ∧ isEmpty dst s}}
writeVirtual shadow2TableDst idx src;;
writeVirEntry shodow1TableSrc idx child;;
writePEntry MMUtableDst idx srcp.
```

```
{{fun _ s ⇒ HI s ∧ KI s ∧ VS s ∧ C s}}.
```

Ainsi, avant de partager la page physique avec la partition enfant, il est nécessaire de vérifier qu'elle est présente et accessible dans l'espace d'adressage de la partition parent en s'appuyant sur les bits de contrôle `present` et `user` sauvegardés par la configuration du MMU. Ces propriétés sont spécifiées, respectivement, par les prédicats `isPresent` et `isAccessible`. La validation de ces deux propriétés est importante car une partition est autorisée uniquement à partager les pages qui lui appartiennent. De plus si la page était déjà réservée au noyau, l'enfant aurait la possibilité d'accéder à toute la mémoire physique.

Il faut vérifier également que la page n'est pas partagée avec une autre partition en s'appuyant sur l'information sauvegardée dans `shadow1` de la partition courante. Cette propriété est spécifiée par le prédicat `notShared`.

Finalement, et avant de passer à la phase de modification il est important de vérifier que l'adresse virtuelle *dst* n'est pas associée à une page physique dans l'espace d'adressage de l'enfant ce qui évite d'écraser des données sauvegardées par le noyau et mettre les structures dans un état incohérent. Cette dernière propriété est spécifiée par le prédicat `isEmpty`.

6.2.2 Mise à jour de l'état

D'une manière générale, la mise à jour de l'état par une instruction classe les propriétés des invariants dans l'un des deux cas suivants où chacun peut être également décomposé en plusieurs sous cas. Cette classification est illustrée par la figure 6.2 où dans certains des cas une révision des scripts de preuve, en cours d'établissement, est nécessaire.

A. La propriété à propager est prouvable après l'exécution de l'instruction.

- **a.** Elle n'est pas exprimée en fonction de la structure modifiée par cette instruction.
  D'une manière générale, il est facile d'établir la preuve permettant la propagation de cette propriété grâce aux propriétés liées au typage qui sont stockées dans le modèle de la mémoire physique.
- **b.** Elle est exprimée en fonction de la structure modifiée par cette instruction.
  Dans ce cas, il faut montrer que la modification est cohérente avec la propriété à propager. Autrement dit, il faut être capable de donner des explications (sous la forme d'une preuve en utilisant les tactiques de coq) qui justifient que cette modification est validée par la propriété en question.

B. La propriété à propager n'est pas prouvable après l'exécution de l'instruction.

- **a.** Elle n'est plus nécessaire pour raisonner sur les instructions restantes.
  Dans ce cas il suffit de mettre à jour la post-condition de cette instruction en supprimant cette propriété de l'ensemble des propriétés à propager.
- **b.** Elle est nécessaire pour raisonner sur les instructions restantes.
  Dans ce cas il existe encore deux possibilités pour résoudre ce problème.
  - **i.** Si cette propriété ne correspond pas à l'une de nos propriétés d'isolation ou de cohérence, il est possible de la remplacer localement par une déduction de celle-ci et supprimer l'ancienne.
    Il est à noter qu'au moment de leur spécification, les propriétés internes à propager à travers les instructions d'un appel système sont des conjectures. En effet, ces propriétés sont des prédicats a priori dont l'utilité est confirmée uniquement lorsqu'elles sont utilisées dans la preuve des prochaines instructions qui en dépendent. Ainsi, il est tout

à fait possible de ne pas définir le bon prédicat permettant de spécifier la propriété requise. Modifier la propriété à l'endroit dans la preuve où elle a été définie nous amène à modifier les preuves liées à sa propagation. Ainsi, il est recommandé de définir localement le prédicat adéquat qui pourrait être propagé et permettant de vérifier le reste des instructions. Dans la plupart des cas, cette transformation n'est pas compliquée à établir puisque ce prédicat est une déduction de celui qui a été initialement propagé.

ii. Si cette propriété est une propriété d'isolation ou de cohérence, nous avons identifié quatre possibilités pour la propager.

1. Propager une nouvelle propriété interne liée à une instruction exécutée précédemment par l'appel système.
Ceci nous permet de compléter la précondition de l'instruction par la propriété nécessaire pour vérifier la propriété en question. Dans ce cas, il faut identifier l'instruction concernée puis spécifier et propager la nouvelle propriété à travers les instructions qui suivent.

2. Modifier le code de l'appel système en déplaçant l'instruction.
La modification du code entraîne généralement une mise à jour assez importante au niveau de la preuve existante. En revanche, ceci dépend fortement de la nature de la modification.
D'une part, si l'instruction en question doit être exécutée plus tard, il suffit de la déplacer dans la bonne position. Dans ce cas il est important d'identifier cela avant de commencer l'établissement de la preuve liée à cette instruction car le raisonnement dépend fortement du contexte dans lequel l'instruction est exécutée. D'autre part, s'il s'agit d'exécuter l'instruction plus tôt dans le code ceci entraîne la mise à jour de tous les invariants des instructions qui seront exécutées après cette instruction.
Ainsi, pour limiter ce genre de modification progressive du code il est important de commencer par une lecture initiale de l'appel système avant de commencer le processus de vérification pour essayer de réordonner les instructions en fonction des propriétés.

3. Corriger un bug.
L'impact de la correction d'un bug sur la preuve dépend fortement de la nature de la modification. S'il s'agit simplement d'ajouter une instruction ou un test manquant, dans la plus part des cas il n'est pas compliqué de réviser la preuve pour tenir compte du changement. En revanche, lorsque le bug est lié à un détail de la conception, les corrections nécessaires risquent d'invalider une bonne partie de la preuve.

4. Ajouter une nouvelle propriété de cohérence.
À ce niveau une propriété de cohérence rend possible la propagation de la propriété en question car elle apportera une information supplémentaire sur les données sauvegardées dans les structures de données des partitions. Cette mise à jour est considérée comme étant le processus le plus compliqué à gérer puisqu'il est nécessaire, dans ce cas, de mettre à jour une partie importante de la preuve établie. Nous détaillerons dans le reste de ce chapitre les techniques utilisées pour réduire le coût de cette modification.

La mise à jour de l'état par `addVaddr` consiste à modifier le contenu de trois adresses physiques grâce à la mise en séquence de trois primitives HAL. Une étape importante du raisonnement consiste à identifier dans quel cas (dans la liste détaillée précédemment) s'inscrit chaque propriété à propager à travers l'instruction en question et à changer l'ordre d'exécution si nécessaire.

L'exécution de `writeVirtual`

La première instruction de mise à jour est `writeVirtual`. Elle sauvegarde l'adresse virtuelle *src* dans la structure `shadow2` de l'enfant à l'adresse virtuelle *dst*. Comme déjà détaillé dans le chapitre précédent ceci permettra au parent de récupérer rapidement la page physique associée.

Ainsi, cette instruction modifie l'état de la mémoire. Il est donc important de prouver que toutes les propriétés d'isolation et de cohérence sont préservées par cette instruction ainsi que toutes les autres propriétés à propager. La majorité de ces propriétés s'inscrivent dans le cas détaillé dans **A.a.**.

Pour cette instruction nous détaillons uniquement la preuve de la propagation de la propriété de cohérence `accessibleChildPageIsAccessibleIntoParent` qui s'inscrit dans le cas **A.b.**. En effet, cette propriété garantit que toute page accessible par une partition enfant doit être accessible également par son père où l'adresse virtuelle (dans l'espace d'adressage virtuel du parent) associée à cette page est sauvegardée dans la structure `shadow2` de l'enfant.

Au moment de l'exécution de `writeVirtual` (pour sauvegarder *src* dans la structure `shadow2` de l'enfant) la page physique n'est pas encore attribuée à l'enfant donc elle n'est pas accessible non plus. Ainsi, dans ce cas `writeVirtual` produit un état dans lequel cette propriété de cohérence est vraie.

L'exécution de `writeVEntry`

L'instruction de mise à jour suivante est `writeVEntry`. Elle partage partiellement l'adresse virtuelle *src* avec l'enfant. Cette instruction indique que *src* est attribuée à cette partition en sauvegardant l'adresse virtuelle *child* dans la structure `shadow1` du parent (la partition en cours d'exécution) à l'adresse virtuelle *src*. À ce stade de l'exécution de `addVaddr`, la page physique *srcp* est marquée comme partagée. En revanche, elle n'est toujours pas affectée à l'espace d'adressage de l'enfant. Ainsi, la preuve de la propagation de la majorité des propriétés à travers cette instruction est similaire à celle de l'instruction précédente où la vérification de la cohérence de la structure `shadow1` du parent a été la plus marquante.

Il est important de noter qu'après l'exécution de cette instruction la propriété interne `notShared` , indiquant que `srcp` n'est pas marquée comme étant partagée (selon les données sauvegardées dans la structure `shadow1` du parent) n'est plus vraie. Toutefois, cette propriété est encore nécessaire pour vérifier l'instruction suivante. Ainsi, cette propriété s'inscrit dans le cas **B.b.i.** où une propriété *similaire* à `notShared` doit être spécifiée et propagée. Nous appellerons cette nouvelle propriété *H*. En effet, au lieu d'exprimer le fait que `srcp` n'est pas partagée en fonction des données sauvegardées dans la structure de données du parent, nous l'exprimerons en fonction de l'ensemble de pages attribuées aux enfants de cette partition. En effet, *H* indiquera que cette page n'appartient à aucun espace d'adressage de ses descendants. Contrairement à `notshared` cette propriété est vraie après l'exécution de `writeVEntry`.

L'exécution de `writePEntry`

La dernière instruction à exécuter par cet appel système est `writePEntry`. Cette instruction met à jour la configuration du MMU de la partition enfant en associant la page physique `srcp` à l'adresse virtuelle `dst`. L'invariant de cette instruction est illustré par le triplet suivant :

```
{{fun s ⇒ HI s∧ KI s ∧ VS s ∧ C s ∧ isChild child s ∧ isPresent srcp s
∧ isAccessible srcp s ∧ H srcp s ∧ isEmpty dst s}}
writePEntry MMUtableDst idx srcp {{fun _ s ⇒ HI s ∧ KI s ∧ VS s ∧ C s}}.
```

La vérification de cet invariant consiste à prouver que les propriétés d'isolation et de cohérence sont préservées en utilisant les propriétés propagées par les instructions précédentes. Dans ce paragraphe nous détaillons uniquement la preuve associée à la vérification de la propriété d'isolation horizontale `HI`. Cette propriété exige de prouver que la liste de toutes les pages attribuées à un enfant *child1* ne contient aucune des pages physiques attribuées à une partition incomparable *child2*. Au début de cette preuve nous identifions trois cas :

1. L'enfant *childp* n'est ni *child1* ni *child2*.

2. Le deuxième et le troisième cas sont symétriques : *childp* peut être soit *child1* soit *child2*

Pour prouver que `HI` est préservée dans ces trois cas, il est important de prouver les propositions suivantes où *s* est l'état du système avant l'exécution de `writePEntry` et *s'* est l'état final :

- ✓ Aucune partition n'est ajoutée :
  $\forall\ part,\ part \in \texttt{partitionTree}\ (s') \rightarrow part \in \text{partitionTree}(s)$.

- ✓ Aucun enfant n'est ajouté :
  $\forall\ parent\ child,\ parent \in \text{partitionTree}(s) \rightarrow child \in \text{children}(parent,\ s') \rightarrow child \in \text{children}(parent,\ s)$.

- ✓ Aucune page de configuration n'a été ajoutée à n'importe quelle partition dans l'arbre de partition :
  $\forall\ part,\ part \in \text{partitionTree}(s) \rightarrow \text{kernelPages}(part,\ s) = \text{kernelPages}(part,\ s')$.

- ✓ La liste des pages attribuées à une partition qui ne correspond pas à *childp* reste inchangée :
  $\forall\ part,\ part \in \text{partitionTree}(s) \rightarrow part \neq childp \rightarrow \text{attributedPages}(part,\ s) = \text{attributedPages}(part,\ s')$.

Ces propositions sont necessaires pour une double raison : d'une part parce que `HI` est exprimée en fonction de chacune de ces fonctions (c'est-à-dire `partitionTree`, `children`, `attributedPages`, `kernelPages`) et d'autre part parce que ces fonctions sont aussi exprimées en fonction de la structure modifiée par cette instruction (i.e. la configuration du MMU d'une partition qui est dans ce cas *childp*).

Le premier cas est résolu en utilisant ces quatre propositions ainsi que la propriété `HI` sur l'état précédent.

La preuve des deux cas symétriques nécessite les trois premières propositions (vu que la dernière n'est pas vraie pour la partition *childp*), la propriété `HI` sur l'état précédant ainsi que la propriété interne `H` qui garantit que `srcp` n'est attribuée à aucun enfant de la partition courante à l'état *s*.

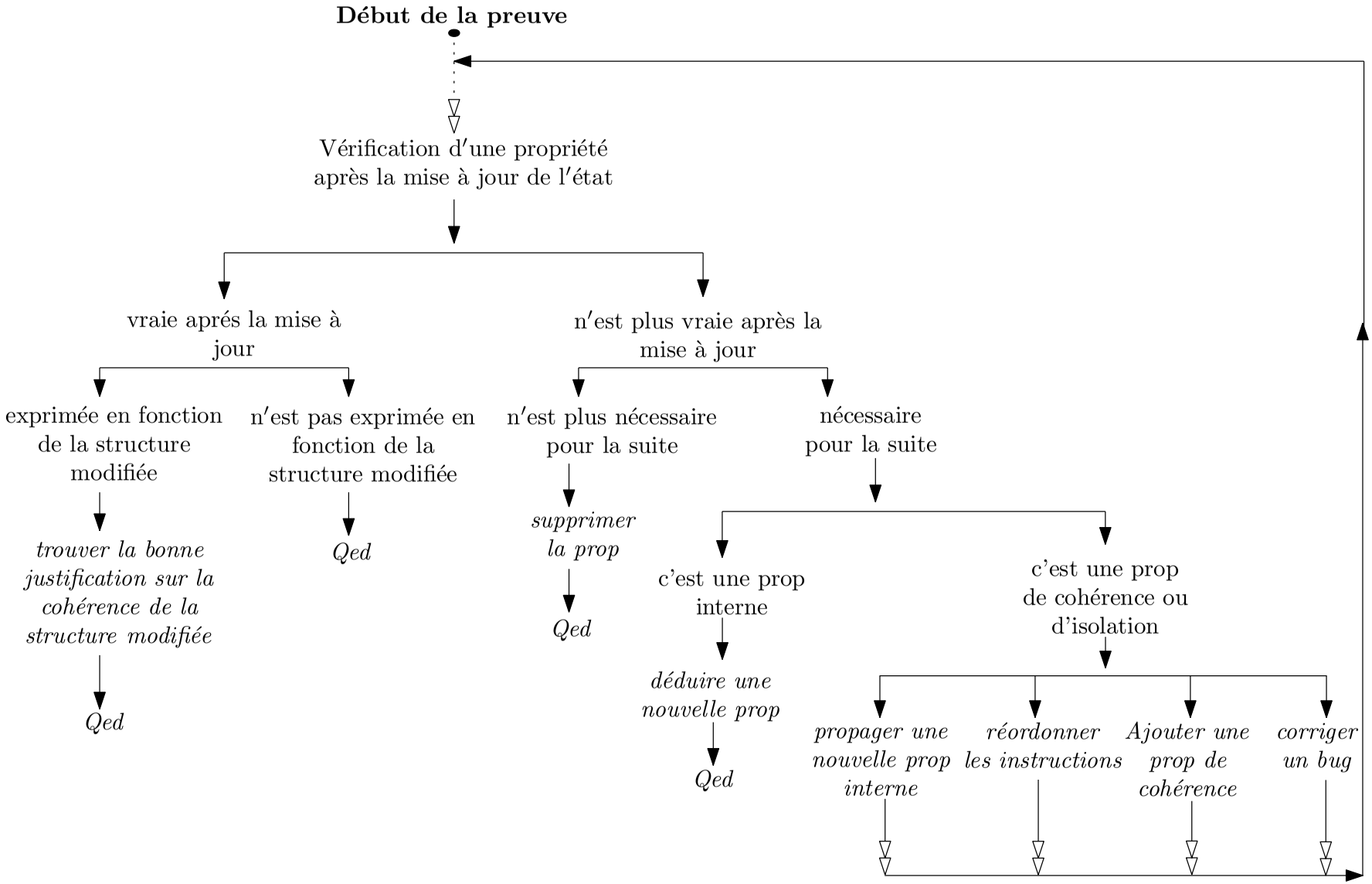


FIGURE 6.2 – Classification des propriétés

6.3 Les propriétés de cohérence de Pip

6.3.1 Propager une nouvelle propriété de cohérence

Pour que cette étape soit facile à gérer, il faut être capable d'agir uniquement sur l'endroit ou il faut ajouter le script de preuve qui correspond à la vérification de cette nouvelle propriété, sans être obligé de repasser sur les scripts existants.

Nous avons identifié deux techniques assez simples et complémentaires permettant de répondre à cette problématique. En effet, le contexte de la vérification de chaque instruction, tel qu'il est représenté par la formule ci-dessous :

```
∀ I C1 P: (state → Prop), ∀ s s': state, I s ∧ C1 s ∧ P s → I s' ∧ C1 s' ∧ P s'.
```

consiste en une hypothèse sous la forme d'une conjonction de plusieurs propriétés sur l'état précédent telles que l'isolation, la cohérence et l'ensemble de propriétés internes à propager à travers cette instruction, et une conclusion, également, sous la forme d'une conjonction des même propriétés sur le nouvel état. La première technique consiste à utiliser le système de bullet pour décomposer proprement la conclusion en un ensemble de plusieurs buts. Ceci semblait triviale mais d'après notre expérience nous avons constaté que cette technique est la meilleure pour gérer un nombre considérable de propriétés. Chaque but doit correspondre à une seule propriété et identifié par un bullet tel qu'il est défini par le script de preuve ci-dessous :

```
Proof.
(* Propriété d'isolation *)
+ preuve de I s'.

(* Première propriété de cohérence *)
+ preuve de C1 s'.

(* Propriétés propagées *)
+ preuve de P s'.
Qed.
```

Cette technique évite de se perdre dans les différents buts, plus particulièrement dans le cas où une erreur inattendue apparaît avant celle qui consiste à prouver la nouvelle propriété. L'apparition de ce genre d'erreur est généralement causée par le changement de l'identificateur de l'hypothèse utilisée pour prouver le but en question (i.e. celui qui a causé l'erreur inattendue) suite à l'introduction de la nouvelle propriété de cohérence. Cela nous amène à déduire qu'il faut éviter de déstructurer l'hypothèse initiale en fixant manuellement un identificateur pour chaque propriété.

Revenant à la formule précédente, naturellement, un utilisateur Coq pense que pour bien structurer sa preuve il est conseillé d'appliquer la tactique `intros` de la manière suivante : `intros I C1 P s s' [Hi [Hc1 Hp]]` afin de fixer pour chaque propriété l'identificateur qui lui correspond et se référer à cet identificateur à chaque fois qu'il a besoin de l'hypothèse qui lui correspond. Dans notre cas, la deuxième technique consiste à éviter ce genre de nommage car dans le cas où il faut introduire une nouvelle propriété de cohérence l'application de la tactique `intros` sera différente (i.e. elle doit être remplacée par `intros I C1 C2 P s s'[Hi [Hc1 [Hc2 Hp]]]`) et donc une erreur sera produite à ce niveau dans la preuve. Pour la même raison, il faut éviter toute sorte de décomposition manuelle de conclusion en utilisant des tactiques telles que `split`.
Pour résoudre ce problème nous proposons d'appliquer une tactique, telle que `intuition`, permettant de déstructurer automatiquement l'hypothèse principale en associant à chaque nouvelle propriété un identificateur aléatoire, de décomposer le but principal en plusieurs sous-but et résoudre tous les sous-but triviaux. Ensuite, lorsque nous avons besoin d'une hypothèse en particulier (par exemple celle qui

identifie la propriété d'isolation) nous appliquons la tactique suivante : `assert (Hi: I s) by trivial .` Cette dernière permet de recréer l'hypothèse `Hi` en évitant toute dépendance avec sa position dans l'hypothèse principale.

Il est important de noter que la tactique `intuition` est coûteuse en mémoire et en temps surtout quand il s'agit de destructurer une conjonction d'une centaine de propriétés. Dans ce cas il est nécessaire d'appliquer cette tactique au plus une fois par instruction, qui est généralement le cas dans notre processus de vérification. Comme résultat, le script (simplifié) de la preuve aura la forme suivante :

```
Proof.
intuition.

(* Propriété d'isolation *)
+ assert (Hi: I s) by trivial.
  assert (Hc1: C1 s) by trivial.
  preuve de I s'.

(* Première propriété de cohérence *)
+ assert (Hc1: C1 s) by trivial.
  preuve de C1 s'.

(* Deuxième propriété de cohérence *)
+ assert (Hc2: C2 s) by trivial.
  preuve de C2 s'.

(* Propriétés propagées *)
+ preuve de P s'.
Qed.
```

6.3.2 Exemples de propriétés de cohérence

Dans ce paragraphe nous détaillons quelques propriétés de cohérence qui ont été spécifiées et vérifiées durant le processus de vérification des appels système de Pip. En effet, ces propriétés sont beaucoup plus complexes et nombreuses en comparaison de celles du modèle abstrait détaillé dans le chapitre 3. Ceci est tout à fait logique puisqu'ici il ne s'agit plus de la vérification d'un modèle abstrait mais plutôt d'un noyau concret. En revanche, nous avons remarqué que dans certains cas certaines propriétés *se ressemblent*. Vu le nombre important des propriétés nous avons classifié celles-ci en cinq catégories et nous détaillons uniquement une propriété par catégorie.

La cohérence de la structure de l'arbre de partition Les propriétés classées dans cette catégorie sont nécessaires pour assurer la cohérence de la structure de l'arbre de partition géré par Pip. En effet, comme précisé antérieurement, toutes les structures des partitions sont encodées dans la mémoire physique à l'aide de pointeurs. Dans ce cas la forme réelle de la structure n'est pas explicite. Ceci nécessite un ensemble de propriétés permettant de préciser la topologie de cette structure. Ces propriétés sont indispensables pour vérifier les propriétés de sécurité.

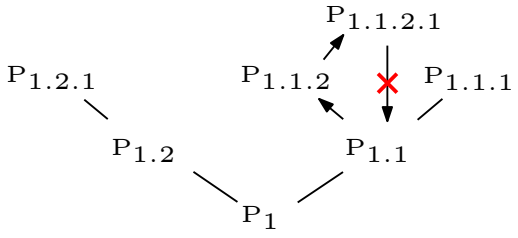


FIGURE 6.3 – Illustration de la propriété de cohérence `noCycleInPartitionTree`

Un exemple de ces propriétés est `noCycleInPartitionTree` illustré par la figure 6.3. Elle garantit qu'il n'y a jamais de cycle dans l'arbre de partition : Une partition dans l'arbre ne peut jamais être un descendant de ses propres descendants. Cette propriété est formalisée de la manière suivante :

```
Definition noCycleInPartitionTree s := ∀ ancestor partition: page,
partition ∈ partitionTree s → ancestor ∈ ancestors partition s →
ancestor ≠ partition.
```

où `ancestors` est la liste de tous les ancêtres d'une partition. Cette liste est construite à partir des informations sauvegardées dans les descripteurs des partitions. En effet, à partir de chaque `PD` il est possible d'identifier le `PDI` du père. Ainsi une récursion jusqu'à la racine permettant de récupérer tous les ancêtres d'une partition.

La sémantique des flags Cette catégorie de propriétés consiste à garantir la cohérence des valeurs des *flags* sauvegardés dans les tables des structures de données des partitions. Nous prenons comme exemple la propriété `isPresentNotDefaultIff` illustrée par la figure 6.4. Cette propriété consiste

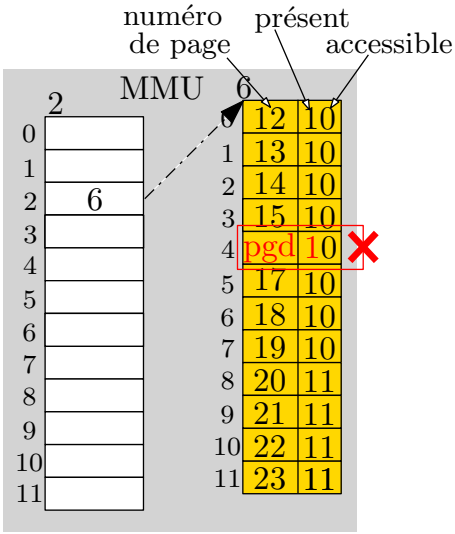


FIGURE 6.4 – Illustration de la propriété de cohérence `isPresentNotDefaultIff`

à garantir que les valeurs stockées dans un `PEntry` sont cohérentes. Ainsi, si le *flag* `present` est mis à *un* il faut que l'identifiant de la page physique associée à ce *flag* soit valide. En effet, pour l'initialisation d'une table de configuration de MMU nous utilisons la valeur par défaut *pgd* (*page default*) choisie à l'initialisation du système. Au moment de l'association d'une page physique à une adresse virtuelle ce *flag* est mis à *un* et l'identifiant de cette page ne doit pas être égal à *pgd*. Cette propriété est formalisée par le prédicat suivant :

```
Definition isPresentNotDefaultIff (s: state) :=
∀ table: page, ∀ idx: index,
readPresent table idx (memory s) = Some false ⟷
readPhyEntry table idx (memory s) = Some pgd.
```

Les propriétés sur les pages attribuées Cette catégorie permet de spécifier certaines propriétés sur les pages mappées ainsi que celles qui sont réservées au noyau pour configurer les structures des partitions. Par exemple, la propriété `noDupMappedPagesList` garantit qu'il n'y a pas de redondance dans l'ensemble des pages mappées dans une partition. Ceci évite de partager la même page entre deux partitions qui doivent être isolées. `noDupMappedPagesList` est illustrée par la figure 6.5.

Nous formalisons cette propriété comme suit :

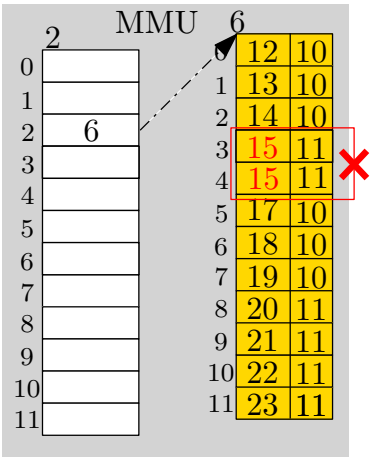


FIGURE 6.5 – Illustration de la propriété de cohérence `noDupMappedPagesList`

```
Definition noDupMappedPagesList (s: state) := ∀ partition,
partition ∈ partitionTree s → NoDup (mappedPages partition s).
```

Cette définition utilise le prédicat `noDup` qui spécifie qu'une liste est sans doublon. La liste concernée ici est `mappedPages`, elle contient toutes les pages mappées dans l'espace d'adressage d'une partition.

Relation entre structures de données Cette catégorie regroupe toutes les propriétés permettant de spécifier des relations entre les structures de données d'une partition. Un exemple intéressant ici est celui de la propriété `physicalPageNotShared`. En effet, comme nous l'avons détaillé dans la section 5.1.1 pour vérifier si une page physique, mappée dans une partition, est partagée avec un de ses enfants nous avons besoin de la configuration du MMU où la page est mappée, et la structure `shadow1` où le `PDI` de la partition enfant pourrait être sauvegardé. Cette propriété assure que si la page physique est marquée comme non partagée dans la structure `shadow1` de la partition parent alors elle n'est mappée dans aucune de ses partitions enfant.

```
Definition physicalPageNotDerived (s: state):=
∀ parent pageParent: page, ∀ va: vaddr, parent ∈ (partitionTree s) →
isShared parent va s = False → getMappedPage parent s va = p →
∀ child, child ∈ (getChildren parent s) → p ∉ mappedPages child s.
```

La sûreté du typage Cette dernière catégorie concerne les propriétés sur le type des valeurs sauvegardées par le noyau dans les structures de données des partitions. Il est important que notre modèle de mémoire garde des informations sur chaque valeur sauvegardée pour assurer que les structures ne contiennent jamais des données non cohérentes. Par exemple, lorsque le noyau sauvegarde une adresse virtuelle dans la mémoire physique il est primordial de la considérer comme adresse virtuelle, tant qu'elle n'est pas modifiée par le noyau. Sinon l'accès à la mémoire sera considéré comme comportement non défini. `dataStructurePdSh1Sh2asRoot` exprime ce genre de propriété en spécifiant les type des valeurs sauvegardées dans chaque table des structures MMU, `shadow1` et `shadow2`. Par exemple, elle spécifie que le dernier niveau d'indirection de la structure `shadow2` d'une partition contienne des valeurs de type `vaddr` tel qu'il est illustré par la figure 6.6.

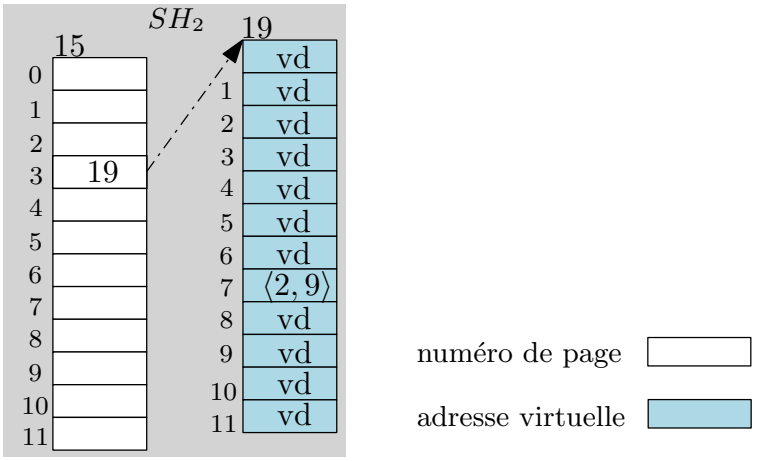


FIGURE 6.6 – Illustration partielle de la propriété de cohérence `dataStructurePdSh1Sh2asRoot`

6.4 Propagation des propriétés : ingénierie de la preuve

Dans la section précédente nous avons discuté l'importance des propriétés propagées dans la vérification des propriétés d'isolation. Il est à noter que le nombre de ces propriétés peut devenir considérable après avoir vérifié une bonne partie du code de l'appel système.
Par exemple, le triplet `addVaddrInvariantProgress` détaillé dans la section précédente n'est qu'une représentation simplifiée de l'invariant réellement vérifié en Coq. Dans cette section nous nous intéressons à la complexité liée à la propagation des propriétés durant le processus de la vérification d'un appel système. Nous exposons ainsi des cas particuliers où il est indispensable de gérer des redondances des scripts de preuves. Celles-ci découlent principalement de la propagation d'un ensemble de propriétés à travers la même instruction dans plusieurs contextes similaires. Dans ce document, le contexte d'exécution d'une instruction signifie la spécification de sa pré-condition et sa post-condition en fonction de l'état courant du système.

Nous utilisons un extrait de l'implémentation de l'appel système `createPartition` pour expliquer la démarche de la spécification des propriétés internes et leur propagation à travers les instructions. Le triplet suivant correspond à l'invariant de `createPartition` ainsi que l'extrait de son code :

```
Lemma createPartitionInvariant (v1 v2 v3 v4 v5: vaddr):
{{HI & VS & KI & C}}

(* Récupérer le nombre de niveau du MMU *)
perform nbL := getnbLevel in

(* Récupérer le PDI de la partition courante *)
perform curPart := getCurPartition in

(* Récupérer la racine de la configuration du MMU *)
perform curPd := getPd currentPart in
  ...

(* Récupérer la dernière table d'indirection de v1 *)
perform ptv1 := getTableAddr curPd v1 nbL in
perform ptv2 := getTableAddr curPd v2 nbL in
perform ptv3 := getTableAddr curPd v3 nbL in
  ...

(* Récupérer la position de l'entrée dans ptv1 *)
perform idxv1 := getIndexOfAddr v1 lzero in
perform idxv2 := getIndexOfAddr v2 lzero in
```

```
perform idxv3 := getIndexOfAddr v3 lzero in
  ...
(* Récupérer la valeur du flag user associer à v1 *)
perform accessv1 := readAccessible ptv1 idxv1 in
perform accessv2 := readAccessible ptv2 idxv2 in
perform accessv3 := readAccessible ptv3 idxv3 in
(* Assurer que les pages sont accessibiles*)
if (negb accessv1 & negb accessv2 & negb accessv3 & ..)

then ret false
else
 writeAccessible ptv1 idxv1 false ;;
 writeAccessible ptv2 idxv2 false ;;
 writeAccessible ptv3 idxv3 false ;;
 writeAccessible ..
 writeAccessible ..
    ...
    ...
 writePDflag ptsh1v1 idxv1 false ;;
{{HI & VS & KI & C}}
```

Comme détaillé dans le chapitre précédent cet appel système prend en paramètre cinq adresses virtuelles. Si toutes les pages physiques associées aux adresses virtuelles satisfont toutes les contraintes, elles seront utilisées comme pages de configuration de la nouvelle partition créée à la fin de l'exécution de `createPartition`.

6.4.1 Définition des propriétés internes

L'extrait du code de `createPartition` montre qu'une instruction correspond soit à une primitive HAL soit à une fonction interne de Pip. Lorsqu'il s'agit d'une primitive HAL son comportement est exprimé en fonction de sa plus faible précondition.

Prenons l'exemple de la primitive `writeAccessible`.

```
Definition writeAccessible (paddr: page) (idx: index) (flag: bool): LLI unit:=
perform s := get in
let entry := lookup paddr idx s.(memory) beqPage beqIndex in
match entry with
| Some (PE a) ⇒ modify (fun s ⇒ {| currentPartition := currentPartition s;
                                   memory := add table idx
                                             (PE {| read := entry.(read);
                                                    write :=entry.(write);
                                                    exec := entry.(exec);
                                                    present := entry.(present);
                                                    user := flag;
                                                    pa := entry.(pa)|})
                                             (memory s) beqPage beqIndex |})
| Some _ ⇒ undefined 14
| None ⇒ undefined 13
end.
```

Cette primitive ne retourne pas de valeur (d'où l'utilisation de `tt`) et change l'état de système. Elle met à jour la valeur du *flag* `user` d'une entrée de type `PEntry` à l'adresse physique `(ptv1, idxv1)` (dans

l'appel système `createPartition`). Elle peut également générer un comportement indéfini si la valeur sauvegardée, auparavant, à l'adresse physique donnée n'est pas de type `PEntry`. Ainsi, sa plus faible précondition est définie de la manière suivante.

```
Lemma writeAccessible (table: page) (idx: index) (flag: bool)
(P: unit → state → Prop):
{{fun  s ⇒ exists entry: PEntry,
lookup table idx s.(memory) beqPage beqIndex = Some (PE entry) ∧
P tt {| currentPartition := currentPartition s;
        memory := add table idx
         (PE {| read := entry.(read);
               write :=entry.(write);
               exec := entry.(exec);
               present := entry.(present);
               user := flag;
               pa := entry.(pa)|})
        (memory s) beqPage beqIndex |}}} writeAccessible table idx flag {{P}}.
```

Cette plus faible précondition sera ensuite utilisée pour vérifier ses invariants qui, à leur tour, permettent la propagation des propriétés.

6.4.2 Propagation à travers les primitives ne changeant pas l'état

Pour chaque primitive HAL qui ne change pas l'état nous avons défini un unique invariant permettant de propager n'importe quel ensemble des propriétés à travers cette primitive. Le triplet suivant correspond à l'invariant de `readAccessible`. Sa vérification est fondée principalement sur sa plus faible pré-condition.

```
Lemma readAccessible (table: page) (idx: index) (P: state → Prop) :
{{ fun s ⇒ P s ∧ isPE table idx s  }} MAL.readAccessible table idx
{{ fun (isaccessible: bool) (s: state) ⇒
            P s ∧ entryUserFlag table idx isaccessible s }}.
```

où `P` est la (ou les propriétés) à propager, `isPE` spécifie le type de la valeur sauvegardée à l'adresse physique `(table, idx)`, et `entryUserFlag` exprime la relation entre les paramètres de la primitive et sa valeur retournée. Pour utiliser ce triplet il suffit d'instancier `P` avec l'ensemble des propriétés à propager et montrer que `isPE` est vrai avant l'exécution de `readAccessible`. De ce fait, cet invariant permet de propager les propriétés internes ainsi que celle d'isolation et de cohérence et introduire le comportement de cette primitive dans le contexte de l'exécution des instructions suivantes.

6.4.3 Propagation à travers les primitives de mise à jour

Ce cas est plus compliqué à gérer puisque la définition d'un invariant paramétré par l'ensemble des propriétés à propager n'est pas possible. En effet, comme nous avons expliqué dans la section 6.2.2 il existe plusieurs possibilités pour vérifier une propriété après la mise à jour de l'état. Donc il est primordial de la définir explicitement dans l'invariant.

Dans ce cas, nous avons besoin d'autant de triplets que le nombre d'occurrences de la même primitive de mise à jour. Prenons l'exemple de `writeAccessible`. Elle est utilisée plusieurs fois dans l'appel système `createPartition` pour rendre non accessible les pages utilisées dans la configuration de la nouvelle partition. Le problème ici est que si nous définissons un invariant par primitive `writeAccessible`, un nombre considérable de propriétés à propager par la même instruction sera le même. Et donc une partie importante de la preuve sera redondante. Ceci est illustré par les invariants suivants, où `P` est l'en-

semble des propriétés à propager. Elle consiste en une centaine de propriétés donc nous ne donnons pas la définition ici.

```
Lemma invariant1 (ptv1: page) (idxv1: index):
{{fun s ⇒ P s ∧ isPE ptv1 idxv1 s}}
MAL.writeAccessible ptv1 idxv1 false
{{fun (s: state) ⇒ P s ∧ entryUserFlag ptv1 idxv1 false s}}.

Lemma invariant2 (ptv1 ptv2: page) (idxv1 idxv2: index):
{{fun (s: state)⇒ P s ∧ isPE ptv2 idxv2 s ∧
                        entryUserFlag ptv1 idxv1 false s}}
MAL.writeAccessible ptv2 idxv2 false
{{fun (s: state)⇒ P s ∧ entryUserFlag ptv1 idxv1 false s ∧
                        entryUserFlag ptv2 idxv2 false s }}.

Lemma invariant3 (ptv1 ptv2 ptv3: page) (idxv1 idxv2 idxv3: index):
{{fun (s: state)⇒ P s ∧ isPE ptv3 idxv3 s ∧
                        entryUserFlag ptv1 idxv1 false s ∧
                        entryUserFlag ptv2 idxv2 false s}}
MAL.writeAccessible ptv3 idxv3 false
{{fun (s: state)⇒ P s ∧ entryUserFlag ptv1 idxv1 false s ∧
                        entryUserFlag ptv2 idxv2 false s ∧
                        entryUserFlag ptv3 idxv3 false s}}.

Lemma invariant4 (ptv1 ptv2 ptv3 ptv4: page) (idxv1 idxv2 idxv3 idxv4: index):
{{fun (s: state)⇒ P s ∧ isPE ptv4 idxv4 s ∧
                        entryUserFlag ptv1 idxv1 false s ∧
                        entryUserFlag ptv2 idxv2 false s ∧
                        entryUserFlag ptv3 idxv3 false s}}
MAL.writeAccessible ptv4 idxv4 false
{{fun (s: state)⇒ P s ∧ entryUserFlag ptv1 idxv1 false s ∧
                        entryUserFlag ptv2 idxv2 false s ∧
                        entryUserFlag ptv3 idxv3 false s ∧
                        entryUserFlag ptv4 idxv4 false s}}.
```

Une solution à cette problématique est d'utiliser les règles de la logique de Hoare pour décomposer chaque triplet en plusieurs sous-buts et vérifier chaque conjonction séparément. Pour résumer nous avons besoin de l'ensemble de triplet suivants :

- Le premier permet d'introduire la spécification du comportement de cette instruction dans le contexte d'exécution des instructions suivantes.

  ```
  {{ fun s ⇒ isPE table idx s}}
  MAL.writeAccessible table idx false
  {{ fun (s: state) ⇒ entryUserFlag table idx false s }}.
  ```

- Le deuxième invariant permet de propager l'ensemble des propriétés

  ```
  {{ fun s ⇒ P s ∧ isPE table idx s ∧ propsToProveP table idx s }}
  MAL.writeAccessible table idx false
  {{ fun (s: state) ⇒ P s }}.
  ```

- Le troisième permet de vérifier la validité d'une mise à jour précédente de l'état après l'exécution de l'instruction courante.

```
{{ fun s ⇒ entryUserFlag table1 idx1 false s ∧
           PropsToProveEntryUserFlag table table1 idx idx1 s }}
MAL.writeAccessible table idx false
{{ fun (s: state) ⇒ entryUserFlag table1 idx1 false s }}
```

6.4.4 Hypothèse : la logique de Hoare

Utiliser la logique de Hoare pour vérifier que certaines propriétés sont préservées par un appel système exige que les interruptions soient désactivées au cours de l'exécution de cet appel. En effet, un appel système consiste, d'une manière générale, en une séquence de plusieurs modifications interdépendantes au niveau de différentes parties de l'état. Notre approche de vérification consiste à suivre le même enchaînement des instructions en exprimant à chaque fois de nouvelles propriétés sur cette séquence pour établir à la fin le raisonnement complet permettant de prouver que l'ensemble des propriétés est préservé après l'exécution de l'appel système en question.

Ainsi, dans le cas où les interruptions seraient activées, nous pourrions perdre les propriétés sur la séquence des instructions. En effet, une interruption pourrait se déclencher à n'importe quel instant et pourrait éventuellement altérer la partie de l'état concernée par l'exécution de cet appel système et donc notre raisonnement ne serait plus valide. De ce fait il est indispensable que les interruptions soient désactivées juste avant l'exécution des services du noyau.

Malgré cette contrainte il existe un moyen pour réduire le nombre des cas où il est nécessaire de désactiver les interruptions. Comme nous l'avons déjà détaillé dans le chapitre précédent, deux partitions isolées ne partagent aucune partie de l'état, donc si l'appel système est exécuté par l'une de ces deux partitions et l'interruption concerne l'autre partition, nous pouvons admettre que dans ce cas il n'y aura pas d'effet de bord sur l'exécution de l'appel système. Toutefois, cela nécessite une preuve formelle sur la validité de cette proposition.

6.4.5 Hypothèse : le boot

Vérifier que chaque appel système préserve les propriétés d'isolation suppose que ces services soient utilisés lorsque l'état du système est stable. En effet, il est indispensable que l'état initial du système satisfasse toutes les propriétés définies par les invariants car sinon nous n'aurions aucune garantie sur les effets de bord de l'exécution de ces appels système à savoir au niveau fonctionnel ou de sécurité.

Pour compléter nos travaux de vérification, et pour avoir plus de garantie sur l'état initial du système il est possible de développer un ensemble de fonctions permettant de tester la validité de cet état par rapport à chaque propriété vérifiée. Par exemple, la propriété sur l'isolation du noyau exige que toutes les pages réservées au noyau ne soient pas marquées comme accessibles par la configuration des tables du MMU. Dans ce cas il suffit de développer une fonction permettant de tester si l'état initial satisfait cette propriété. Concernant les autres propriétés de sécurité il est inutile de définir des fonction permettant de les tester car au démarrage du système seule la partition racine est créée, et donc les propriétés du partage vertical et d'isolation horizontale sont trivialement vraies.

Toutefois, pour avoir plus de confiance sur cette fonctionnalité fondamentale il est également pertinent de vérifier formellement le code du boot et donc éliminer cette hypothèse. Dans ce cas, vérifier le boot consiste à prouver que ses opérations aboutissent aux propriétés d'isolation et de cohérences assumées vraies par la suite.

6.5 Récapitulatif de la vérification

La spécification exécutable du proto-noyau Pip consiste en 1 300 lignes de Gallina. Nous estimons l'effort nécessaire à environ trois mois de conception et neuf mois de développement. Actuellement,

trois appels système sont vérifiés : `createPartition`, `addVaddr` et `mappedInChild`.

Nous avons commencé par la vérification de l'appel système `createPartition`. Celui-ci a introduit la majorité des propriétés de cohérence, ce qui donne environ 60K lignes de preuves, comme indiqué dans le tableau 6.1. Durant la vérification, nous avons constaté que le raisonnement sur `addVaddr` et `mappedInChild` reposait largement sur les lemmes introduits et prouvés durant la vérification de l'appel système `createPartition` ce qui résulte en environ 18K lignes de preuves supplémentaires. Il est à noter que certains lemmes définis pour être utilisés dans la vérification du premier service, n'ont pas été adaptés mais dupliqués (avec un changement mineur) pour être utilisés dans la preuve de `addVaddr` ou celle de `mappedInChild`. Ceci nous évite la mise à jour de la preuve de `createPartition` et aussi la reformulation (essentiellement une généralisation) du lemme principal et les éventuels lemmes qui en dépendent pour tenir compte des nouveaux cas possibles. Un travail de restructuration de la preuve consisterait dans ce cas à factoriser ce genre de duplication de scripts de preuve afin de réduire la taille de la preuve. La vérification de ces appels système m'ont nécessité environ un an de travail réparti entre la formalisation des propriétés et la preuve.

Invariants	**Lignes de preuve**	**durée**
createPartition ($\approx 300 loc$)	$\approx$ 60000	$\approx$ 10 mois
createPartition + addVaddr ($\approx 110 loc$)	$\approx$ 78000	$\approx$ 2 mois
createPartition + addVaddr + mappedInChild ($\approx 40 loc$)	$\approx$ 78300	$\approx$ 4 heures

TABLEAU 6.1 – Récapitulatif de la vérification

6.6 Non-interférence

6.6.1 Formalisation

Dans les sections précédentes nous avons mis en lumière l'importance des propriétés du modèle de partitionnement de Pip pour garantir la protection des données des accès non légitimes. Dans cette section nous montrons comment nous avons utilisé les propriétés d'isolation et de cohérence pour vérifier la non-interférence entre les partitions incomparables. Autrement dit, montrer qu'il n'y a pas de fuite d'informations (ou de données) permettant de mettre en cause la confidentialité des données manipulées par les partitions. Il est à noter que ces travaux ne consistent pas à définir une politique de communication entre les partitions. Mais il s'agit plus particulièrement d'établir un lien entre ce que nous avons vérifié comme propriétés d'isolation et la notion de non interférence qui est considéré primordiale dans la sécurité des systèmes critiques.

Les concepts de non interférence entre les entités qui évoluent et coexistent dans un environnement critique ont été présentés dans [BLP76, FLR77, GM82] et approfondis par Rushby dans [Rus92] où il a pu identifier plusieurs faiblesses et reformuler plus clairement ces propriétés. Pour résumer Rushby a défini cette propriété comme suit :

« Un domaine noté u n'interfère pas avec un autre domaine noté v si et seulement si toutes les actions effectuées par u n'affectent pas les valeurs retournées par les actions effectuées par v. »

Une première lecture de cette affirmation permet d'identifier les acteurs principaux de la propriété de non interférence : les domaines. Ici, un domaine est la portion de l'état observable par les actions de cet acteur. Dans notre contexte un domaine peut être connecté à la notion de partition. Pour une partition donnée, deux états sont considérés équivalents si et seulement si l'observation de cette partition est exactement la même dans les deux états. Ceci est défini comme suit :

$$s_1 \sim^{ps} s_2 := \forall (p : \mathsf{page})\ (i : \mathsf{index}),\ p \in ps \rightarrow \mathsf{select}\ s_1\ p\ i = \mathsf{select}\ s_2\ p\ i$$

où,

- select : est la fonction qui permet de récupérer la valeur stockée à une adresse physique donnée ;
- ps : est la liste des pages observables par une partition.

Selon Rushby pour vérifier la non interférence entre deux domaines (ou partition dans le contexte de Pip) il est nécessaire de vérifier les deux propriétés complémentaires suivantes [Rus92, p. 13] :

step consistent : cette première propriété concerne les actions effectuées par la partition en cours d'exécution. Elle signifie que l'exécution d'une action sur deux états non distinguables par une partition produit deux états non distinguables par cette partition.

local respect : cette propriété concerne les actions effectuées par les partitions incomparables (ou isolées). Elle considère deux états *s* et *s'* où *s'* est l'état produit à partir de *s* à travers une action effectuée par la partition courante. Cette propriété indique qu'une partition incomparable à la partition courante doit garder exactement la même observation sur les deux état *s* et *s'*.

Il est important de noter que la reformulation des propriétés de non interférence par Rushby ne repose pas sur le même niveau de détail que celui de nos propriétés d'isolation. En effet, dans le contexte de non interférence l'étude s'intéresse principalement aux actions effectuées par les domaines. En revanche, la formalisation de nos propriétés de sécurité ne tient pas compte de ce qui se passe dans les partitions. Autrement dit, la vérification de nos propriétés de sécurité est complètement indépendante des actions effectuées par la partition.

Pour établir facilement le lien entre nos propriétés d'isolation et de cohérence avec celles étudiées et présentées par Rushby nous avons modélisé un environnement d'exécution fondé essentiellement sur le modèle de contrôle d'accès assuré par le MMU. Il important de noter que cette hypothèse formulée sur le fonctionnement du matériel, est souvent le sujet des attaques DMA (*Direct Memory Access*) ou encore celui des attaques de type *spectre* [KGG+18] ou *meltdown* [LSG+18] récemment. Toutefois, le travail de preuve présenté ici est circonscrit uniquement aux aspects logiciels et supposant que le matériel se comporte conformément à sa spécification. De ce fait, nous avons défini la fonction `vstep` comme suit :

```
Definition vstep (MMUref: page) (pstep: state → list page → value → state)
(valist: list vaddr) (v: value) (s: state): state :=
pstep s (MMUtranslate valist MMUref s) v.
```

où

- `MMUref` est la référence vers les tables de traduction du MMU de la partition en cours d'exécution ;
- `pstep` est la fonction qui correspond à un pas d'exécution d'une partition. Elle prend en paramètre l'état courant (de type `state`) de la machine, la liste des pages concernées par l'exécution de cette opération et une valeur de type `value`. `pstep` peut être une fonction de lecture ou de mise à jour. Nous formalisons chacun de ces deux comportements de la manière suivante :

  Mise à jour : l'opération `pstepWCond` indique que seul le contenu des pages associées à `pstep` peut changer après l'exécution de cette opération de mise à jour.

  ```
  Definition pstepWCond (k: state → list page → value → state): Type :=
   ∀ (s: state) (listpage: list page) (p1: page) (v: value) (i: index),
   p1 ∉ listpage → select s p1 i = select (pstep s listpage v) p1 i.
  ```

  Lecture : l'opération `pstepRCond` indique que les valeurs lues par `pstep` dépendent uniquement du contenu des pages associées à cette opération de lecture.

```
Definition pstepRCond (k: state → list page → value → state): Type :=
 ∀ (s1 s2: state) (listpage: list page) (v: value) (i: index) (p: page),
 p ∈ listpage → (∀ (i: index), select s1 p i = select s2 p i)→
 select (pstep s1 listpage v) p i = select (pstep s2 listpage v) p i.
```

- `MMUtranslate` est la fonction permettant d'identifier l'ensemble des pages observables par la partition à travers la fonction de traduction du MMU.

Les deux propriétés de non interférence ont été formalisées en Coq à travers les théorèmes suivants :

```
Theorem stepConsistency:
∀ s1 s2  part1, consistency s1 → consistency s2 →
part1 ∈ (partitionTree s1) → part1 ∈ (partitionTree s2) →
∀ MMUref1 MMUref2, getMMUref part1 s1  = Some MMUref1 →
                   getMMUref part1 s2  = Some MMUref2 →
MMUref1 = MMUref2 → observation part1 s1 = observation part1 s2 →
∀ p, p ∈ (observation part1 s1) → (∀ i, select s1 p i = select s2 p i) →
(∀ va, translate MMUref1 va nbL s1 = translate MMUref2 va s2 →
∀ s1' s2' v valist k, pstepWCond k → pstepRCond k →
vstep MMUref1 k valist v s1 = s1' → vstep MMUref2 k valist v s2 = s2' →
∀ i, select s1' p i = select s2' p i.
```

Ce théorème est la formalisation de la première propriété de non interférence *step consistent*. Il consiste à montrer que les observations d'une partition `part1` sont égales sur deux états `s1'` et `s2'` si et seulement si :

- Les observations d'une partition `part1` sont égales sur deux états `s1` et `s2` où `s1'` et `s2'` sont les nouveaux états produits respectivement à partir des états `s1` et `s2` et,
- La transition d'état se fait avec la fonction `vstep` sous un contrôle d'accès effectué par le MMU.

```
Theorem localRespect:
∀ s s' part2, isolatedPartitions (currentPartition s) part2 s→
(currentPartition s) ≠ part2 → HI s → VS s → C s →
part2 ∈ (partitionTree s) → ∀ p k, In p (observation part2 s) →
∀ k valist v, pstepWCond k → vstep MMUref k valist v s = s' →
∀ i, select s p i = select s' p i.
```

Ce théorème est la formalisation de la deuxième propriété de non interférence *local respect*. Il considère deux partitions isolées (la partition en exécution et une autre partition incomparable `part2` dans l'arbre de partition) et montre que l'action effectuée par la partition en exécution à travers la fonction `vstep` n'affecte jamais les valeurs observées par `part2`.

L'établissement de la preuve de ces deux théorèmes a montré que les propriétés d'isolation et de cohérence sont indispensables pour vérifier les propriétés de non interférence. Ces propriétés ont été ajoutées comme hypothèses progressivement (au besoin) durant la vérification. Ce travail résulte environ 1300 lignes de spécification et de preuve.

6.6.2 Accès direct à la mémoire physique : DMA

Dans le paragraphe précédent nous avons montré que pour assurer la non interférence entre les partitions tout accès à l'état (en lecture ou en écriture) doit être impérativement contrôlé par le MMU.

De ce fait, nous supposons que la configuration des structures des partitions n'est modifiée qu'à travers les appels système exposés par le noyau et que les accès en mode utilisateur à la mémoire physique se font uniquement à travers le MMU. Ce composant matériel vérifie à chaque fois si c'est un accès légitime ou non selon les données préalablement sauvegardées dans les indirections par le noyau.

6.7 Bugs remontés

Dans cette section nous proposons une liste non exhaustive des bugs que nous avons corrigés grâce la vérification des propriétés de sécurité.

- Un bug marquant que nous avons identifié grâce à la vérification des appels système est le *bug des ancêtres*. En effet, nous nous sommes rendu compte qu'au moment de la création des partitions le noyau marquait uniquement l'ensemble de pages, qui sont utilisées dans la configuration de la structure interne de la nouvelle partition, comme plus accessibles dans la partition parent mais pas dans tous les ancêtres. Par conséquent il était impossible de prouver qu'à la fin de l'exécution de cet appel système satisfait encore la propriété de sécurité `KI`. Pour corriger ce problème critique il a fallu définir une fonction récursive permettant de remonter l'arbre de partition jusqu'à la racine en mettant à jour la configuration MMU des ancêtres. Cette fonction a été intégrée dans tous les appels système qui délèguent ou récupèrent des pages de configuration du noyau (i.e. `createPartition`, `prepare`, `collect`, `deletePartition`). Cet exemple illustre un retour depuis la preuve vers la conception qui dans ce cas était défaillante.

- Un autre type de corrections qui cette fois consiste dans le retour depuis la preuve vers l'implémentation. Il s'agit d'ajouter un certain ensemble de tests manquants sur les paramètres. Par exemple, vérifier que les adresses virtuelles fournies par la partition ne sont pas les mêmes ou encore elles ne correspondent pas à une valeur par défaut.

- De plus il nous est parfois arrivé d'oublier d'initialiser des tables d'indirections avec la valeur par défaut adéquate avant son attribution à une partition.

Voilà des exemples d'erreur d'implémentation que nous avons réussi à identifier grâce à la vérification des propriétés de sécurité. La nature de ces erreurs montre que la vérification du bon fonctionnement du noyau n'est pas suffisante pour garantir la sécurité. Par exemple dans aucun cas l'exemple *bug des ancêtres* ne peut être lié au bon fonctionnement de `createPartition`. En particulier, la vérification des propriétés fonctionnelles de cet appel système consisterait par exemple de démontrer que la partition en cours d'exécution possède un enfant de plus après l'exécution de cet appel système ou encore s'assurer que le `PDI` de la partition a été correctement initialisé.

6.8 Conclusion

Dans ce dernier chapitre nous avons détaillé les différentes étapes d'établissement de la preuve d'isolation mémoire sur une implémentation concrète de noyau de système d'exploitation. Nous avons commencé par la formalisation en Coq des différentes propriétés qui doivent être assurées par le noyau. Puis en s'appuyant sur la vérification d'un appel système parmi ceux qui étaient vérifiés, nous avons mis en évidence les difficultés rencontrées durant le processus de vérification ainsi que l'ensemble des solutions que nous avons mis en place pour mener à bien cette expérience. À la fin de ce chapitre nous avons également montré qu'il est possible de vérifier d'autres propriétés de sécurité telles que la non interférence en utilisant uniquement la spécification des propriétés d'isolation assurées par le noyau. L'utilisation de la même approche de vérification développée dans le chapitre 3, a montré que les propriétés de cohérence sont également nécessaires pour vérifier l'ensemble des propriété d'isolation d'un

système concret. Certes, le nombre de ces propriétés est plus important et leur spécification est plus complexe mais ceci est tout à fait raisonnable pour un système concret.

Il est important de noter que notre approche de vérification des propriétés directement sur le code permet au *prouveur* de maîtriser tous les détails d'implémentation. Suite à notre expérience, cela s'est avéré important pour faciliter le raisonnement sur certaines séquences d'instructions. Bien que, du point de vue développeur c'est le résultat final de l'exécution d'un service noyau qui compte, l'ordre dans lequel les instructions sont exécutées peut compliquer la preuve. Dans ce contexte nous avons montré qu'il est important d'éviter les incohérences temporaires des structures de données pendant l'exécution d'un service pour être capable de propager les propriétés à travers les instructions. Ainsi, à plusieurs reprises nous avons décidé de changer l'ordre d'exécution de certaines instructions pour faciliter l'établissement de notre preuve

Bibliographie

[BLP76] D Elliott BELL et Leonard J LA PADULA : Secure computer system : Unified exposition and multics interpretation. Rapport technique, MITRE CORP BEDFORD MA, 1976.

[FLR77] Richard J FEIERTAG, Karl N LEVITT et Lawrence ROBINSON : Proving multilevel security of a system design. *In ACM SIGOPS Operating Systems Review*, pages 57–65. ACM, 1977.

[GM82] Joseph A GOGUEN et José MESEGUER : Security policies and security models. *In Security and Privacy, 1982 IEEE Symposium on*, pages 11–11. IEEE, 1982.

[KGG+18] Paul KOCHER, Daniel GENKIN, Daniel GRUSS, Werner HAAS, Mike HAMBURG, Moritz LIPP, Stefan MANGARD, Thomas PRESCHER, Michael SCHWARZ et Yuval YAROM : Spectre attacks : Exploiting speculative execution. *arXiv preprint arXiv :1801.01203*, 2018.

[LSG+18] Moritz LIPP, Michael SCHWARZ, Daniel GRUSS, Thomas PRESCHER, Werner HAAS, Anders FOGH, Jann HORN, Stefan MANGARD, Paul KOCHER, Daniel GENKIN, Yuval YAROM et Mike HAMBURG : Meltdown : Reading kernel memory from user space. *In 27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, 2018. USENIX Association.

[Rus92] John RUSHBY : *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

Chapitre 7

Conclusion et perspectives

7.1 Résumé

Dans ce manuscrit j'ai présenté mes travaux autour de la conception et la vérification formelle d'un noyau de système d'exploitation que nous avons appelé Pip. La raison principale de la conception d'un nouveau noyau est d'étudier la possibilité de la réduction du coût de la vérification en mettant en place une méthodologie de « co-design » du noyau et de sa preuve. Le coût important de la preuve est essentiellement lié à la complexité du système à vérifier et la nécessité de comprendre l'intégralité de ses détails. Repenser l'architecture d'un noyau en passant par les fonctionnalités qu'il doit fournir, les structures de données qu'il manipule ainsi que la stratégie d'implémentation de ses services peut rendre sa preuve plus facile à établir. Ce « co-design » peut aider les développeurs à faire des choix concernant le coût de la preuve tôt dans le cycle de développement pour acquérir certains avantages dès le départ. Ce choix est essentiellement orienté par un point de vue *formel* qui veille à garder une base de confiance minimale tant que le logiciel est encore à l'étude.

Avant d'aborder l'étape de la vérification d'une implémentation concrète d'un noyau, j'ai effectué une première expérimentation de l'approche de vérification sur un modèle abstrait de micro-noyau. Cette expérience était une double épreuve. La première était de définir une approche permettant de raisonner sur le code d'un programme faisant partie des modules d'un système d'exploitation et la deuxième était d'analyser les difficultés et les problématiques rencontrées pour être capable de guider le développeur dans ses choix durant le développement d'un nouveau noyau de système d'exploitation.

Lorsqu'il s'agit de vérifier formellement un programme, identifier l'ensemble des propriétés fondamentales à vérifier est une étape importante. Ces travaux de thèse ont montré qu'il faut dans un premier temps définir les propriétés de sécurité centrales, puis durant la phase de la vérification il est possible d'identifier progressivement l'ensemble minimal de propriétés fonctionnelles nécessaires pour prouver les propriétés de sécurité identifiées. Ceci permet de réduire le coût de la preuve en minimisant le nombre de propriétés à vérifier.

7.2 Hypothèses

D'une manière générale et peu importe le contexte de vérification, il existe toujours un ensemble d'hypothèses sur lesquelles repose le raisonnement. En revanche, il n'est pas courant que cet ensemble soit explicitement défini et cadré. L'identification de ces ces hypothèses permet d'une part d'avoir une idée claire sur la validité du raisonnement par rapport à la réalité et d'autre part cela facilite l'identification des travaux futurs sur lesquels il est possible d'intervenir pour réduire le plus possible leur nombre. Ceci est donc indispensable pour les développeurs du système visé par la vérification ainsi que pour ses utilisateurs. Dans cette section nous allons reprendre la liste des hypothèses, que nous avons discuté

progressivement dans ce document, sur lesquelles se fonde notre raisonnement.

- Confiance en HAL : la preuve repose sur une implémentation correcte de toutes les primitives du HAL ;
- Accès direct à la mémoire physique (DMA) : nous supposons que les accès DMA sont assujettis au même contrôle d'accès que les partitions, ce qui peut certes être mis en œuvre dans la HAL, mais qui reste sans doute une hypothèse simplificatrice forte pour la gestion efficace de certains matériels par des partitions ;
- Validité de l'état initial par rapport aux propriétés vérifiées : nous supposons que la fonction du boot démarre le système dans un état cohérent ;
- Non volatilité de la mémoire : le problème de volatilité de la mémoire est particulièrement à redouter dans deux cas : lorsque l'exécution d'une fonction est interrompue pour exécuter une fonction Pip qui accède aux mêmes données, ou bien lorsque deux cœurs d'une architecture multi-cœur exécutent simultanément un appel système Pip. Nous avons pris pour hypothèse que les interruptions seraient bloquées lorsque les fonctions de Pip sont exécutées, et nous n'avons pas considéré dans la preuve le raisonnement sur les caractéristiques des architectures multi-cœur (cette problématique est traitée séparément dans d'autres travaux [BGIC18]) ;
- Spécification des propriétés d'isolation : la définition des propriétés à vérifier est une étape cruciale dans un processus de vérification. Il est important que cette spécification définisse exactement ce que nous entendons par isolation mémoire dans un noyau de système d'exploitation. Pour assurer l'exactitude de ces propriétés, nous avons fait attention à ce qu'elles soient lisibles et compréhensibles. Nous avons également validé leurs modélisations vis-à-vis de quelques propriétés de sécurité définies par d'autres travaux qui s'intéressent principalement à la définition des propriétés de séparation dans les systèmes d'exploitation [Rus81]. De plus, cette expérience de vérification nous a montré que la moindre ambiguïté dans la définition de l'une de ces propriétés nous empêche de prouver sa validité à la fin de l'exécution de l'appel système.

Ces hypothèses concernent la validité de raisonnement sur les appels systèmes. Toutefois, il existe d'autres hypothèses qui concernent la chaîne de conversion du code écrit en Gallina vers le langage machine. En effet, il est à noter qu'une traduction incorrecte en C par *digger* ou une compilation incorrecte, également, vers le langage machine peut changer le comportement de ces services. Ainsi, si cette conversion était erronée elle rendrait non valide les propriétés prouvées.

De ce fait, pour compléter la chaîne de vérification il est indispensable de vérifier la validité de la conversion de Gallina vers C et d'utiliser un compilateur certifié tel que *CompCert* [Ler].

7.3 Réflexions personnelles

Les métriques sont trompeuses

Malgré les efforts consentis pour réduire le coût de la preuve, il est cependant compliqué d'évaluer à quel point ce défi a été réussi. En effet, pour évaluer les performances d'un programme il suffit de faire des tests comparatifs sur son temps d'exécution. Le résultat de l'analyse de ces expériences permet de déduire l'écart entre l'efficacité de ce programme et celle d'un modèle de référence. En revanche, dans le contexte de la vérification formelle il n'existe pas de métriques efficaces permettant d'évaluer le coût de l'approche de vérification. Il n'existe pas non plus un modèle de référence avec lequel il est possible d'effectuer la comparaison. Les présentations des travaux de vérification des noyaux de système d'exploitation proposent souvent une analyse autour de l'effort de vérification. Cependant ceci ne peut

pas être considéré comme une évaluation véritable du coût de l'approche utilisée pour établir la preuve. D'après mes expériences de vérification effectuées durant cette thèse j'ai constaté que chaque travail de vérification dépend fortement de plusieurs facteurs difficiles à mesurer. Dans la suite de ce paragraphe j'aborderai les métriques souvent utilisées et j'explique pourquoi (à mon avis) elles ne fournissent pas un consensus clair pour effectuer l'évaluation du coût de l'approche de vérification.

Taille du code par rapport à la taille de la preuve

Certaines études proposent d'utiliser la taille du code et la taille de la preuve produite pour vérifier ce code comme métriques d'évaluation. Je trouve que cette combinaison est absurde pour deux raisons : La première est parce que la taille de la preuve ne dépend pas uniquement de la taille du code mais aussi des propriétés à vérifier. Ainsi, ceci peut inverser complètement le résultat de cette évaluation. En effet, il est possible que dans une première expérience le nombre de propriétés à vérifier soit plus important que celui d'une autre expérience de vérification. La taille de la preuve peut varier en fonction du nombre et de la nature des propriétés à vérifier. La deuxième raison est qu'il est *toujours* possible d'écrire une preuve plus courte. Toutefois, essayer d'établir une preuve plus compacte peut prendre beaucoup plus de temps qu'établir la même preuve sans se préoccuper de sa taille. De plus, parfois pour assurer une mise à jour facile de la preuve, comme il a été évoqué dans le chapitre 6, il est préférable d'éviter dans certains cas la factorisation des scripts de preuve. Donc essayer de réduire la taille de sa preuve peut ne pas aider le *prouveur* à effectuer son travail.

La durée et l'effort de vérification

Calculer la durée des travaux de vérification ainsi que le nombre des personnes impliquées dans ce travail est aussi considéré comme un moyen pour évaluer le coût de l'approche. Toutefois, chaque expérience de vérification nous aide à améliorer nos compétences et donc réussir mieux nos travaux futurs de vérification. De ce fait, il est indispensable que l'analyse de ces résultats tienne compte, également, des compétences des personnes impliquées. Cependant la compétence est un paramètre difficilement quantifiable, et l'incidence des années d'expérience en preuve de programme sur le temps de preuve n'a jamais, à ma connaissance été un objet d'étude scientifique. Aussi, l'appréciation du temps de preuve reste essentiellement subjective.

Modèle de référence

Lorsqu'il s'agit de comparer deux approches il est nécessaire d'effectuer les deux expériences sur le même modèle pour être capable d'évaluer les résultats. Supposons que ce travail de vérification soit effectué par la même équipe où deux approches différentes sont utilisées pour vérifier les mêmes propriétés sur le même programme. Bien qu'ici nous essayons de réduire les facteurs qui peuvent agir sur la validité de l'évaluation mais il faut être conscient que c'est souvent la première expérience qui permet comprendre les détails du fonctionnement d'un système et donc la deuxième expérience sera plus facile à établir.

Enfin, j'apprécie l'idée que ces données peuvent nous informer sur l'effort nécessaire pour établir une preuve mais je pense qu'elles ne devraient jamais considérées comme des métriques pour évaluer une approche de vérification formelle. Elles ne permettent donc pas vraiment, à mon avis, de comparer différentes approches entre elles.

7.4 Perspectives

Vérification de la traduction de Gallina vers C

L'implémentation des appels système de Pip dans un style impératif (en utilisant une monade d'état) était également motivée par la limitation de la possibilité d'introduction des bugs dans le traducteur. Nous sommes contraints à utiliser uniquement les *traits* du langage cible ce qui nous a permis de pro-

duire une traduction *mot à mot* de Gallina vers C. Il est à noter qu'avant de commencer la vérification nous avons défini en Coq un environnement de simulation permettant d'effectuer des tests sur l'API de Pip. Bien que nous ayons trouvé quelques incohérences entre les résultats de la simulation et ceux de l'exécution réelle, ceci n'a jamais été lié à un bug de traduction. Cependant, pour compléter ce travail de vérification il est nécessaire de prouver formellement que la traduction du code de Pip en Gallina vers C est correcte. Cette vérification est le sujet d'un travail en cours [TNJC18].

Spécification abstraite de Pip

La vérification des applications qui utilisent les appels système de Pip nécessite dans un premier temps une spécification abstraite de chaque appel système. Ceci est indispensable pour être capable de raisonner sur le comportement de ces applications. En particulier, il s'agit de définir un modèle abstrait de l'arbre de partition et de spécifier l'effet de bord de chaque appel système sur cet arbre sans spécifier les détails qui ne sont pas visibles par un utilisateur (comme la configuration du MMU ou des shadows).

Prenons l'exemple d'un protocole de communication qui pourrait être utilisé par une partition parent pour gérer la communication entre ses enfants. Celui-ci pourrait être implémenté en utilisant les appels système de Pip `removeVaddr` et `addVaddr` pour déplacer les pages entre les espaces d'adressage des partitions enfants concernées. Bien que les aspects liés à la confidentialité des données mis en place par ce protocole de communication soient la responsabilité de la partition parent, il est possible de vérifier formellement des propriétés sur la politique qui gère la circulation des données entre les enfants. Pour vérifier des propriétés de haut niveau (niveau applicatif) il est indispensable de fournir une spécification formelle de ces deux appels système.

Intégrer la logique de séparation

Dans cette thèse nous avons utilisé la logique de Hoare pour vérifier que les propriétés d'isolation et de cohérence sont préservées par chaque appel système. Cependant, il existe d'autres extensions de cette logique, telles que la logique de séparation [Rey02], qui sont souvent utiles pour faciliter l'établissement de preuves d'algorithmes manipulant des pointeurs. Intégrer cette logique dans le processus de vérification des propriétés de Pip et étudier les aspects liés à l'utilité de cette logique dans la réduction du coût de la preuve attise ma curiosité pour de futures recherches. Ceci peut être intéressant pour vérifier par exemple une version multi-cœur du proto-noyau Pip où il est indispensable de montrer que la configuration des partitions qui partagent la même mémoire ne se fait pas simultanément par diffèrent cœurs en exécution. Cette propriété est fondamentale car si ce n'est pas le cas le raisonnement sur la séquence des instructions qui composent un appel système devient invalide. En effet, comme détaillé dans le chapitre 6, chaque instruction dépend du comportement des instructions précédentes, donc il est important de montrer par exemple que deux cœurs travaillent sur deux partitions séparées et donc qu'ils ne peuvent pas modifier simultanément les structures de données d'une même partition.

Bibliographie

[BGIC18] Quentin BERGOUGNOUX, Gilles GRIMAUD et Julien IGUCHI-CARTIGNY : Porting the pip proto-kernel's model to multi-core environments. *In 16th IEEE Intl Conf on Dependable, Autonomic and Secure Computing (IEEE-DASC'2018).*, 2018.

[Ler] X. LEROY : The CompCert verified compiler. Available at http://compcert.inria.fr/.

[Rey02] John C REYNOLDS : Separation logic : A logic for shared mutable data structures. *In Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[Rus81] John M RUSHBY : *Design and verification of secure systems.* ACM, 1981.

[TNJC18] Paolo TORRINI, David NOWAK, Narjes JOMAA et Mohamed Sami CHERIF : Formalising executable specifications of low-level systems. *In 10th Working Conference on Verified Software : Theories, Tools, and Experiments (VSTTE 2018)*, 2018.

Résumé

Dans cette thèse nous proposons un nouveau concept de noyau adapté à la preuve que nous avons appelé « proto-noyau ». Il s'agit d'un noyau de système d'exploitation minimal où la minimisation de sa taille est principalement motivée par la réduction du coût de la preuve mais aussi de la surface d'attaque. Ceci nous amène à définir une nouvelle stratégie de « co-design » du noyau et de sa preuve. Elle est fondée principalement sur les *feedbacks* entre les différentes phases de développement du noyau, allant de la définition des besoins jusqu'à la vérification formelle de ses propriétés. Ainsi, dans ce contexte nous avons conçu et implémenté le proto-noyau Pip. L'ensemble de ses appels système a été choisi minutieusement pendant la phase de conception pour assurer à la fois la faisabilité de la preuve et l'utilisabilité du système. Le code de Pip est écrit en Gallina (le langage de spécification de l'assistant de preuve Coq) puis traduit automatiquement vers le langage C. La propriété principale étudiée dans ces travaux est une propriété de sécurité, exprimée en termes d'isolation mémoire. Cette propriété a été largement étudiée dans la littérature de par son importance. Ainsi, nos travaux consistent plus particulièrement à orienter le développement des concepts de base de ce noyau minimaliste par la vérification formelle de cette propriété. La stratégie de vérification a été expérimentée, dans un premier temps, sur un modèle générique de micro-noyau que nous avons également écrit en Gallina. Par ce modèle simplifié de micro-noyau nous avons pu valider notre approche de vérification avant de l'appliquer sur l'implémentation concrète du proto-noyau Pip.

Mots-clés : sécurité, systèmes d'exploitation, isolation mémoire, proto-noyau, preuve formelle, Coq.

Abstract

In this thesis we propose a new kernel concept adapted to verification that we have called protokernel. It is a minimal operating system kernel where the minimization of its size is motivated by the reduction of the cost of proof and of the attack surface. This leads us to define a new strategy of codesign of the kernel and its proof. It is based mainly on the feedbacks between the various steps of development of the kernel, ranging from the definition of its specification to the formal verification of its properties. Thus, in this context we have designed and implemented the Pip protokernel. All of its system calls were carefully identified during the design step to ensure both the feasibility of proof and the usability of the system. The code of Pip is written in Gallina (the specification language of the Coq proof assistant) and then automatically translated into C code. The main property studied in this work is a security property, expressed in terms of memory isolation. This property has been largely discussed in the literature due to its importance. Thus, our work consists more particularly in guiding the developer to define the fundamental concepts of this minimalistic kernel through the formal verification of its isolation property. The verification strategy was first experimented with a generic microkernel model that we also wrote in Gallina. With this simplified microkernel model we were able to validate our verification approach before applying it to the concrete implementation of the Pip protokernel.

Keywords : security, operating system, memory isolation, protokernel, formal proof, Coq.