Enseigner l'architecture des systèmes d'exploitation

Gilles Grimaud [1]

Cet article offre un retour d'expérience sur l'enseignement de l'architecture des systèmes d'exploitation tel qu'il a été enseigné à l'université de Lille durant les vingt dernières années. Il distingue l'enseignement de l'architecture des systèmes d'exploitaiton (ASE) des autres enseignements relatifs aux systèmes. Puis il précise l'intention pédagogique en la restituant dans les différents aspects de la discipline. Les objectifs pédagogiques, les modalités d'enseignement et les contenus techniques sont ensuite décrits et discutés au regard de nos expériences personnelles. En particulier, ce document explore les méthodes d'enseignement utilisées, comme le développement d'un logiciel sur un matériel nu, et la stratégie pédagogique retenue pour mettre en œuvre ces connaissances. L'article se conclut sur les perspectives d'avenir pour l'enseignement de cette discipline cruciale.

Introduction

L'architecture des systèmes d'exploitation nous apparaît comme l'un des enseignements disciplinaires fondamentaux de l'informatique au niveau master. La maîtrise des principes architecturaux qui guident la conception des systèmes d'exploitations nécessite, d'un côté, la connaissance du lien fort qui existe entre le matériel

1. Professeur des universités, université de Lille.

DOI:10.48556/SIF.1024.22.89 **1024** – Bulletin de la SIF, numéro 22, novembre 2023, p. 89–107

et les logiciels, et de l'autre, la construction d'abstractions consistantes avec les besoins d'ingénierie des logiciels. Dans cet article, je présente une synthèse de mes réflexions et retours d'expérience autour de ces cours [2] dispensés depuis une vingtaine d'années en master informatique à l'université de Lille. Je tiens encore à préciser que ces réflexions font écho à mes nombreuses conversations et réflexions avec les enseignants-chercheurs avec qui j'ai collaboré au fil du temps sur ce sujet. Si je ne les ai pas consultés pour produire ce document, je suis pleinement conscient qu'ils ont été essentiels à l'élaboration de mes réflexions. Je pense principalement à Philippe Marquet et Samuel Hym depuis de longues années, mais aussi, plus récemment Giuseppe Lipari et Clément Ballabriga.

Le document est structuré en trois parties principales. La première partie, aborde les notions d'architecture des systèmes d'exploitation en présentant différents points de vue (utilisateur, développeur, administrateur et électronicien) sur le rôle des systèmes et les principales familles d'architectures. L'objectif de cette section est de souligner la diversité des approches et de mettre en avant les enseignements et compétences associés à ces perspectives. Chacune de ces approches aborde la question des systèmes d'exploitation, mais passe à côté d'une partie importante de ce que leur conception a à enseigner. Aussi, dans la suite de cette première partie, la lumière est mise sur les grandes familles de paradigmes et sur les aspects méthodologiques que les architectures des systèmes d'exploitations ont contribués à élaborer au fil des décennies. La deuxième partie se concentre sur les objectifs pédagogiques et la stratégie d'enseignement adoptée pour transmettre les connaissances relatives à l'architecture des systèmes d'exploitation. Elle souligne l'importance de comprendre le lien entre matériel et logiciel, ainsi que l'approche pédagogique axée sur les travaux pratiques guidés. J'y présente également l'organisation que nous avons retenue pour les cours, les travaux dirigés et les différentes évaluations. Enfin, pour conclure la dernière partie ouvre sur l'évolution récente et les perspectives d'avenir de cet enseignement. Les avancées technologiques telles que le *cloud* posent de nouveaux défis alors que l'évolution des centres d'intérêt et les outils d'intelligence artificielle invitent à des adaptations possibles du contenu pédagogique.

En somme, dans cet article, j'argumente sur le rôle, les enjeux et les défis de l'enseignement de l'architecture des systèmes d'exploitation dans un contexte en constante évolution, mais surtout j'argumente sur l'importance de cet enseignement pour les futurs professionnels de l'informatique.

Notion d'architecture des systèmes d'exploitation

De manière générale un système est défini comme :

2. `https://www.fil.univ-lille.fr/portail/index.php?dipl=MInfo&sem=CC&ue=ASE&label=PrÃ©sentation`.

(1) un ensemble organisé de principes coordonnés de façon à former un tout scientifique ou un corps de doctrine (e.g. système philosophique) ;

(2) un ensemble d'éléments considérés dans leurs relations à l'intérieur d'un tout opérant de manière unitaire pour servir une fonction (e.g. système nerveux).

Ici le système d'exploitation est un ensemble organisé de principes coordonnés de façon à former un tout technologique opérant de façon unitaire pour servir l'exploitation. Mais l'exploitation de quoi ? Et pour qui ?

De l'enseignement des systèmes d'exploitation

Lorsque l'on traite des systèmes d'exploitation il est essentiel de répondre à la question « pour qui ? ». En effet l'objet d'étude est suffisamment complexe pour que la perspective de l'observateur en change la lecture. Pour ce qui concerne les systèmes d'exploitation, il convient de distinguer quatre points de vue associés à quatre catégories d'utilisateurs de ce logiciel, et en conséquence quatre enseignements associés ; chacun d'eux étant dispensés à l'université de Lille.

Pour l'utilisateur, le système d'exploitation sert principalement à lancer et gérer les autres programmes, comme l'illustre par exemple le système Windows de Bill Gates qui permet de lancer des applications `.exe`. Pour les utilisateurs, un système doit *permettre l'exploitation des applications*. Dans cette logique l'université de Lille propose un cours de première année de licence intitulé « Maîtrise des systèmes Unix [3] » pour familiariser les étudiants avec les compétences liées à cette lecture du rôle d'un système d'exploitation. Ils y apprennent à exploiter l'interface utilisateur et à utiliser un terminal, à organiser des fichiers, à exécuter des commandes de base, etc. Parmi les ouvrages à visée pédagogique on peut citer [14, 7].

Les développeurs connaissent une autre facette du système. Ils utilisent des abstractions destinées à *faciliter l'exploitation de l'environnement d'exécution* de leur logiciel. Cela inclut des concepts tels que les flux entre applications, les processus, les signaux, les fichiers, les interfaces utilisateur et bien d'autres abstractions qui facilitent le développement et la coordination des logiciels. Cette représentation du rôle d'un système, comme « principe coordonné » et comme « corps de doctrine » a été abondamment enseigné et documenté par le corps académique [3, 6]. Les compétences associées à cette lecture des systèmes d'exploitation sont dispensées, à Lille, dans un cours intitulé « Programmation des systèmes ». Les étudiants y apprennent à écrire des logiciels qui utilisent un système de fichiers et des processus par exemple.

Pour les administrateurs, un système d'exploitation est un outil permettant d'*organiser l'exploitation des ressources informatiques*, mais aussi d'administrer les droits des utilisateurs. De ce point de vue, notamment défendu par Richard Stallman,

3. `https://www.fil.univ-lille.fr/portail/index.php?dipl=L&sem=S3&ue=Unix&label=PrÃ©sentation`.

le système d'exploitation ne s'arrête pas au noyau. L'université de Lille propose, aux étudiants en première année de master informatique spécialité *Cloud Computing*, un cours intitulé « Déploiement et administrations du *Cloud*[4] » pour enseigner les compétences liées à cette perspective. Les étudiants y apprennent comment utiliser des outils avancés d'administration des systèmes, tels que le déploiement d'un *OpenStack*, la configuration et le déploiement de conteneurs Docker, et l'utilisation de Kubernetes par exemple [15, 13].

Enfin, les concepteurs de support informatique, que nous appelons ici « les électroniciens » mais qui seraient plus justement dénommés « architectes des matériels informatiques », considèrent le système d'exploitation comme un logiciel destiné à *piloter l'exploitation du matériel informatique*. Les échanges entre les concepteurs de matériels et les concepteurs de systèmes d'exploitation peuvent être riches et instructifs, comme l'ont montré les discussions entre Linus Torvalds et NVIDIA il y a quelques années par exemple. L'université de Lille propose un cours de « Conception de systèmes temps réels[5] » pour les étudiants en seconde année de master informatique spécialité *Internet of Things* (IoT). Ils y apprennent comment concevoir et optimiser les logiciels pour piloter des appareils électroniques et des systèmes embarqués, et notamment les contraintes temporelles. Ici, en répondant à la question « l'exploitation pour qui ? », on répond aussi, en partie à la question « l'exploitation de quoi ? » ([4] est un exemple d'ouvrage pédagogique pertinent de ce point de vue).

Ces points de vue sont associés à des enseignements « orientés métier » pour lesquels l'université de Lille, parmi de nombreuses autres, offre différentes unités d'enseignement. Cependant aucun d'elles ne développe la notion de principes coordonnées (système), ni la notion d'organisation possible de ces principes (architecture).

À l'enseignement de la notion de système

Dans tous ces points de vue, ce qui est discuté ce n'est pas la notion de système. Ce qui est enseigné c'est l'utilisation d'un logiciel qui permet « l'exploitation », c'est-à-dire « le bon usage » de la machine. Mais si un traitement de texte est une application pour éditer du texte, un système d'exploitation n'est pas destiné à un usage précis. C'est un cadre de fonctionnement pour l'ensemble des applications et des usages. Par ailleurs, même si un système d'exploitation était résumé à un logiciel comme un autre, nous avons montré dans la section précédente, que l'objet de ce logiciel diffère selon les points de vue. C'est ici que la notion de système prend tout son sens, dans sa définition la plus générale : un ensemble de principes coordonnés qui forme un corps de doctrines. Enseigner la notion de système c'est donc enseigner

4. https://www.fil.univ-lille.fr/portail/index.php?dipl=MInfo&sem=CC&ue=ASE&label=PrÃl'sentation.

5. https://portail.fil.univ-lille.fr/portail/index.php?dipl=MInfo&sem=IOT&ue=RTS/&label=Programme.

un ensemble de paradigmes et de retour d'expériences. Ces enseignements guident l'élaboration du premier logiciel, de celui qui coordonne les autres, qui les organise, qui les administre et qui optimise les ressources utilisées et enfin qui gère le matériel. Dès lors, la question est de savoir lesquels des grands paradigmes guidant l'organisation des systèmes d'exploitation méritent l'attention des étudiants. Car, dans cette jeune science qu'est l'informatique beaucoup se sont opposés et s'opposent encore aujourd'hui dans une coexistence pleine de contradictions. Il n'y a en cela rien de surprenant, puisque c'est ainsi que progresse la science en général [10]. Mais le débat reste ouvert quant à savoir ce qui doit être enseigné. De nombreux clivages ont animé et animent encore la communauté des chercheurs en système. Parmi une liste bien plus longue, je retiens quelques clivages qui me sont apparus, au fil des années, particulièrement formateurs.

Virtualisation et abstraction

Ce sont deux concepts clefs en informatique, qui sont parfois utilisés comme synonymes alors qu'ils ont des objectifs et des usages distincts. Ils ne sont pas nécessairement opposés, mais plutôt complémentaires dans de nombreux cas.

Paradigme de conception orienté virtualisation. La virtualisation concerne la création de versions virtuelles de ressources matérielles ou logicielles, permettant à plusieurs instances de fonctionner simultanément et indépendamment les unes des autres. La virtualisation s'efforce de découpler les ressources de leur mise en œuvre matérielle et de les rendre accessibles de manière plus flexible et dynamique. Les exemples incluent la virtualisation de serveurs, de stockage et de réseaux. Le paradigme de conception orienté virtualisation se concentre sur la manière dont les systèmes et les applications peuvent tirer parti de la virtualisation pour améliorer leur flexibilité, leur évolutivité et leur résilience. Il met l'accent sur la gestion des ressources virtuelles, la répartition dynamique des charges de travail et la séparation des environnements d'exécution.

Paradigme de conception orienté abstraction. L'abstraction est un concept général en informatique qui consiste à simplifier et à généraliser des détails techniques pour en faciliter la compréhension et la manipulation par les développeurs et les utilisateurs. L'abstraction permet de cacher les détails de mise en œuvre et de se concentrer sur les fonctionnalités et les interactions de haut niveau. Le paradigme de conception orienté abstraction se concentre sur la manière dont les systèmes et les applications peuvent être conçus pour cacher la complexité sous-jacente et fournir des interfaces simples et cohérentes. Il met l'accent sur la modularité, la réutilisabilité et la séparation des préoccupations. Les abstractions implantées par les systèmes les plus usuels sont les fichiers et les APIs associées, les APIs liées aux applications, aux interfaces graphiques ou encore à Internet (coté client et coté serveur).

En résumé, la virtualisation se concentre sur le découplage des ressources de leur mise en œuvre matérielle, tandis que l'abstraction se concentre sur la simplification et la généralisation des détails techniques pour simplifier la compréhension et la manipulation. Dans de nombreux cas, la virtualisation et l'abstraction peuvent être utilisées ensemble pour créer des systèmes plus flexibles, évolutifs et maintenables. Par exemple, les architectures de microservices combinent souvent des éléments de virtualisation (par exemple, les conteneurs) et d'abstraction (par exemple, les API RESTful) pour permettre le déploiement et la gestion indépendante de services individuels.

Conception dirigée par les événements et conception dirigée par les flots d'exécution

Ce sont deux approches différentes pour concevoir et structurer des applications, en particulier en ce qui concerne la gestion de la concurrence et la communication entre les composants.

Conception dirigée par les événements. Dans ce paradigme, le flux de l'application est déterminé par la réception et le traitement des événements, généralement déclenchés par des actions de l'utilisateur ou des modifications dans l'environnement du système. Les composants de l'application sont conçus pour réagir aux événements et exécuter des actions spécifiques en réponse. La communication entre les composants se fait principalement par l'échange d'événements ou de messages. Le modèle dirigé par les événements est particulièrement adapté aux applications interactives, telles que les interfaces utilisateur graphiques, les applications de communication en temps réel et les systèmes de traitement de flux de données. Au nombre des avantages de ce paradigme, on compte : une meilleure réactivité, une facilité de mise à l'échelle et une séparation claire des préoccupations entre les composants. Cependant, la conception dirigée par les événements peut également entraîner une complexité accrue et des difficultés de *debugging*, car le flux de contrôle n'est pas linéaire et peut être difficile à suivre.

Conception dirigée par les flots d'exécution. Dans ce paradigme, le flux de l'application est structuré autour de plusieurs séquences d'exécution (les *threads* et les processus). Chaque *thread* peut accéder aux ressources partagées, telles que la mémoire et les périphériques. La communication entre les *threads* se fait généralement par le partage de données et la synchronisation à l'aide de mécanismes tels que les verrous, les sémaphores et les moniteurs. Le modèle dirigé par les *threads* est couramment utilisé dans les applications qui nécessitent une utilisation efficace des ressources matérielles, telles que les serveurs, les systèmes d'exploitation et les applications de traitement intensif de données. Au nombre des avantages de ce paradigme, on compte : une meilleure utilisation des ressources, une meilleure performance dans les systèmes multiprocesseurs et une meilleure lisibilité des programmes. Cependant, la conception dirigée par les flots d'exécution peut entraîner des problèmes

complexes tels que les interblocages, les conditions de concurrence et les problèmes de synchronisation, qui peuvent être difficiles à détecter et à résoudre.

Ces deux paradigmes peuvent également être combinés pour tirer parti de leurs avantages respectifs. Par exemple, une application peut utiliser un modèle dirigé par les événements pour gérer les interactions utilisateur et la communication entre les composants, tout en utilisant des *threads* pour effectuer des tâches de calcul intensif ou des opérations d'entrée-sortie en parallèle. Cependant leur combinaison n'est généralement pas exempte de complications. Lorsque les événements et les *threads* sont utilisés ensemble, le code peut devenir plus complexe et difficile à comprendre, car il doit gérer à la fois les interactions asynchrones basées sur les événements et la synchronisation des *threads*. Cela peut introduire des problèmes de synchronisation, tels que les conditions de concurrence, les interblocages et les problèmes d'accès aux données partagées.

Sécurité des moindre privilèges et sécurité par conception

Il s'agit de deux approches qui reflètent des visions orthogonales de la sécurité. La première pense la sécurité comme un cadre fixé par une autorité alors que la seconde pense la sécurité comme un cadre fixé par ce qui est possible.

Sécurité du moindre privilège. La sécurité guidée par le principe du moindre privilège peut être considérée comme une approche qui repose sur l'interposition d'une autorité qui met en place la sécurité en contrôlant et en restreignant l'accès aux ressources et aux informations. Le pilotage de cette autorité logicielle implémente la politique de sécurité en fonction des besoins spécifiques des utilisateurs, des applications et des processus. Selon ce paradigme la politique de sécurité repose sur le principe que chaque entité doit disposer uniquement des privilèges nécessaires pour accomplir ses tâches, de manière à limiter les possibilités d'exploitation des privilèges excessifs par des attaquants ou des logiciels malveillants. Le contournement de l'autorité est « hors du champ des préoccupations » de la politique de sécurité ainsi définie.

Sécurité par conception. La sécurité par conception, aussi appelée sécurité par design, peut être considérée comme une approche qui met l'accent sur des choix de conception destinés à rendre nécessaire le respect de la sécurité pour permettre le simple fonctionnement du programme. L'idée est ici de remplacer l'autorité de contrôle de la politique de sécurité par une impossibilité opérationnelle. Cette démarche est issue de la cryptographie où l'on s'appuie sur des fonctions irréversibles (en temps polynomial) sans un nombre « clef » pour le chiffrement des données. L'idée est de s'efforcer d'anticiper et de traiter les problèmes potentiels de sécurité dès le début du processus de développement en tirant parti des techniques, des mécanismes et des technologies de sécurité disponibles. La cryptographie plutôt que le contrôle d'accès, les méthodes formelles plutôt que la chasse aux vulnérabilités, etc.

L'objectif est de créer un système intrinsèquement sûr en intégrant des fonctionnalités de sécurité dans sa conception.

Ces deux paradigmes peuvent et doivent être combinés pour créer un système d'exploitation sécurisé. Le moindre privilège constitue une stratégie de sécurité spécifique qui peut être intégrée dans l'approche globale de la sécurité par la conception. En appliquant le principe du moindre privilège tout au long du processus de conception et de développement, il est possible de renforcer la sécurité du système d'exploitation et de réduire la probabilité d'exploitation réussie par des attaquants.

Liaison et interposition

Le concept de liaison est au cœur des systèmes d'exploitation, il relie un symbole à une entité logicielle ou matérielle. La mise en place de ce paradigme fondamental de dénomination des ressources en facilite un second : le paradigme d'interposition.

Liaison. La liaison, telle qu'elle est implémentée dans les systèmes d'exploitation, peut être considérée comme un mécanisme par lequel les ressources matérielles et logicielles sont reliées et rendues accessibles aux applications. Ce processus permet d'abstraire les détails sous-jacents et de fournir une interface standard pour faciliter l'interaction entre les applications et les ressources. On peut par exemple illustrer ce paradigme avec l'implémentation de la liaison des ressources matérielles. Les mécanismes de liaison permettent d'associer un nom symbolique (e.g. un nom de fichier) à une ressource physique (e.g. les périphériques de stockage, les interfaces réseau et les périphériques d'entrée/sortie). Ce nom symbolique peut ensuite être utilisé par les applications pour désigner la ressource. Un nom de fichier est transformé en une série de blocs sur un disque, par l'implémentation d'un mécanisme de liaison au cœur de la notion de système de fichier. La liaison de code dynamique, également connue sous le nom de chargement dynamique, est un processus par lequel les bibliothèques partagées et les modules de code sont chargés en mémoire et liés à l'application lors de l'exécution. Le système d'exploitation est alors responsable de la gestion des bibliothèques partagées et de la résolution des noms de fonctions appelées en adresses de fonctions dans la mémoire utilisée par le microprocesseur. En pratique le système d'exploitation utilise un mécanisme de liaison pour relier diverses ressources logicielles, telles que les fichiers, les *sockets* et les objets de synchronisation, aux applications. Le système d'exploitation fournit des interfaces, telles que les API de système de fichiers et les API réseau, pour faciliter l'accès à ces ressources.

Interposition. L'interposition est un mécanisme par lequel un composant logiciel s'insère entre deux autres composants pour intercepter, inspecter ou modifier les communications ou les interactions entre eux. Cela peut être utilisé pour diverses raisons, telles que la surveillance, le *debugging*, la modification du comportement ou l'amélioration des performances. L'interposition peut être réalisée de différentes manières, telles que par la reconfiguration de composants matériels (e.g. MMU, Intel

VT-x, etc), redirection de l'appel de fonction, ou la modification du code de l'application ou du système d'exploitation.

Dans le contexte des systèmes d'exploitation, l'interposition peut être utilisée pour surveiller ou modifier les appels système, les appels de bibliothèque ou les interactions entre les processus. Par exemple, une bibliothèque d'interposition peut être utilisée pour intercepter et enregistrer les appels système effectués par une application, ou pour modifier le comportement d'une fonction de bibliothèque en la remplaçant par une implémentation personnalisée. Entre les paradigmes de liaison et d'interposition il n'est pas tant question de clivage que de complémentarité et d'adaptabilité. Lorsqu'un système d'exploitation met en place un mécanisme de liaison, il rend l'accès à la ressource facilement interposable. Lorsque ce n'est pas le cas, (généralement pour des raisons de performance), l'interposition peut néanmoins être mise en oeuvre par l'exploitation de mécanismes de virtualisation des matériels.

Dans cette section, j'ai introduit divers paradigmes et j'ai accentué leurs différences pour mettre en lumière des archétypes. Cependant, ce serait une simplification excessive de réduire les complexités de leur intégration uniquement à ces dichotomies. Par exemple, la combinaison répandue des mécanismes de virtualisation, d'abstraction et d'interposition introduit un écart croissant entre ce que les développeurs pensent qu'ils demandent à la machine de faire et ce qu'elle fait réellement. Le développeur appelle une fonction système, mais il ne connaît qu'une abstraction de celle-ci et reste ignorant de sa fonctionnalité précise. La situation est encore compliquée par le fait que le système d'exploitation a été configuré pour interposer un traitement entre cet appel et la mise en œuvre de l'abstraction. De plus, l'exécution de la mise en œuvre de l'abstraction est effectuée non pas sur le matériel réel sur lequel on supposait qu'elle opérait, mais sur une couche matérielle virtuelle mise en place par un mécanisme de virtualisation. Ce scénario, qui est monnaie courante dans nos environnements informatiques, souligne la complexité introduite par la combinaison des paradigmes présentés. Ce décalage a favorisé un sentiment parmi les professionnels de l'informatique, parfaitement résumé dans l'expression de Tim Herd, « *Les systèmes d'exploitation modernes font un million de choses, mais leur tâche fondamentale est de mentir aux programmes* ». Cependant, au cœur de ces différents clivages se trouve un débat plus profond qui a traversé l'histoire de la conception des systèmes d'exploitation. C'est le modèle des architectures monolithiques qui est généralement opposé au modèle des architectures de micro-noyaux. Il me semble que ces paradigmes architecturaux clivants éclairent toute la philosophie de conception des systèmes, acquise avec beaucoup de peine au fil des décennies.

Et à la maîtrise des choix architecturaux

L'architecture est l'art de créer et d'organiser des formes en vue d'une fonction et en présence de contraintes. À gros trait, les systèmes d'exploitation sont conçus avec une architecture logicielle qui permet de séparer et d'organiser les différentes

fonctions qu'ils doivent remplir en intégrant de nombreuses contraintes, comme vu précédemment [9]. Ces composants logiciels traitent une ou plusieurs préoccupations spécifiques, parmi lesquelles on peut distinguer, là encore de manière très générique, trois aspects principaux : la mise en œuvre de la sécurité et de la fiabilité, la mise en œuvre d'abstractions et la gestion et l'optimisation de l'usage des dispositifs physiques.

L'organisation de ces composants logiciels est de façon assez générique une organisation en couches. Cette organisation en couches est illustrée par l'exemple de l'implémentation des systèmes de fichiers. Lorsqu'un programme demande l'ouverture d'un fichier, le système d'exploitation vérifie d'abord les droits (couche de sécurité), puis exécute des algorithmes qui transforment la demande d'ouverture du fichier en identification des blocs associées à ce fichier sur le disque (couche d'abstraction) et enfin pilote le contrôleur du disque physique pour échanger ces données (couche gestion du matériel) [8].

L'ordre dans lequel les trois préoccupations sont traitées, c'est-à-dire la manière dont l'architecture met en œuvre ces trois couches, a été l'objet de 50 ans de discussions scientifiques et d'innovations technologiques autour des architectures de systèmes d'exploitation. L'histoire retient, de l'évolution de ces architectures trois grandes « familles » d'architectures pour les noyaux : les noyaux monolithiques, les micro-noyaux et les exo-noyaux.

Les noyaux monolithiques sont des noyaux où toutes les fonctions du système d'exploitation sont intégrées dans un seul bloc de code. Les différentes couches sont imbriquées les unes dans les autres, sans séparation explicite. Dans ce contexte, la couche qui gère la sécurité « chapeaute » l'ensemble de l'édifice logiciel, et l'ordre rencontré est donc (1) sécurité (2) abstraction puis (3) gestion. Les recherches académiques reconnaissent à cette approche le mérite d'être très performante. Précisément, ce type d'architecture répond principalement à la volonté de tirer le meilleur parti possible du matériel. On pourrait résumer la philosophie de cette architecture par la formule suivante : « un système unique pour tout faire ». Cependant, le noyau de tels systèmes très volumineux soulève des problèmes de maintenance et de fiabilité du code. En plaçant les mécanismes de sécurité et de fiabilité au dessus de l'implémentation des abstractions des matériels, ces derniers ne peuvent pas en bénéficier. La fiabilité mise en œuvre par le système ne profite pas au système lui-même.

Les micro-noyaux cherchent à répondre au problème de fiabilité et de maintenabilité des noyaux monolithiques, en adoptant une approche plus modulaire [1]. Le noyau, minimal, n'intègre plus que la gestion du matériel et des mécanismes de sécurité. Les autres fonctions du système, notamment celles qui implémentent des abstractions (comme la notion de fichier, ou de pile de protocole réseau par exemple) sont traitées par des modules séparés qui s'exécutent au dessus du noyau et communiquent entre eux. Cette approche permet une meilleure séparation des préoccupations et une plus grande flexibilité architecturale. Elle permet surtout aux abstractions

de bénéficier des mécanismes de sécurité et de fiabilité du noyau. Si la gestion des fichiers rencontre une défaillance, cette dernière ne compromet pas le fonctionnement de tout le système. Il suffit de redémarrer le service « système de fichiers ». Ici l'architecture générale retenue est (1) abstraction (2) sécurité puis (3) gestion. La philosophie des micro-noyaux serait résumé par l'adage *small is beautiful* [11]. Les recherches académiques ont reconnu à l'approche en micro-noyau une meilleure lisibilité et maintenabilité du logiciel, au prix d'une moindre performance. Ici, l'architecture répond davantage à une préoccupation de maintenabilité et de fiabilité du noyau. Cependant en séparant très explicitement la gestion du matériel des abstractions, nombre d'optimisations d'usage n'ont plus pu être implémentées. Le logiciel qui gère le matériel ne tient plus compte des spécificités de l'abstraction.

Dans cette lecture architecturale, l'interposition de la fiabilité entre gestion et abstraction est dépréciée. Pour rétablir les opportunités d'optimisation, les exo-noyaux ont proposé une architecture plus radicale encore [5]. Une architecture où le noyau est réduit à son minimum : la sécurité/fiabilité du matériel. L'essentiel des fonctions est délégué à des librairies logicielles externes au système lui-même. Cela permet à ces librairies de gérer de manière optimale le matériel (sécurisé/virtualisé) que leur présente le noyau pour l'abstraction qu'elles en font. Les développeurs d'applications reprennent le contrôle sur les abstractions et les optimisations des matériels. Cette approche offre une grande flexibilité et permet de mieux adapter les abstractions aux besoins spécifiques des applications, mais elle nécessite que les développeurs aient des connaissances approfondies en système pour qu'ils parviennent à en bénéficier. Ici l'architecture retenue, en mitigeant la contrainte de performance et celle de fiabilité a conduit à une solution où l'utilisabilité du système est pénalisée. Cependant la mise en œuvre d'un noyau de système seulement centré sur la présentation sécurisée du matériel a ouvert la porte à une nouvelle famille de logiciels capables de faire fonctionner plusieurs services de gestion et d'abstraction du matériel, et donc plusieurs systèmes d'exploitations du matériel. Ainsi sont nés les premiers hyperviseurs [2]. La philosophie de ces systèmes fait écho à la citation de Raymond Queneau « *avoir un système borne son horizon, n'en avoir pas est impossible, le mieux est d'en posséder plusieurs* ».

Cependant, pour comprendre et arbitrer des choix architecturaux tels que ceux-ci, il faut regarder dans les détails les verrous technologiques qui entravent une décision qui pourrait paraître purement organisationnelle a priori. Enseigner l'architecture des systèmes, c'est enseigner à garder à l'esprit l'objet dans son ensemble tout en mesurant l'incidence de certains détails critiques sur lui. C'est pourquoi l'architecture des systèmes d'exploitation revêt une dimension holistique, comme l'illustre par exemple le couplage entre performance et intégration de la gestion des matériels dans l'abstraction qui en est proposée.

Définition d'un enseignement en architecture des systèmes

L'architecture des systèmes d'exploitation est donc guidée par la volonté de prendre en charge la gestion laborieuse des matériels pour présenter, à l'ingénieur, des abstractions efficientes pour la conception de ses applications.

Enseigner comment abstraire le lien entre les logiciels et les matériels

En première lecture, l'enseignement de l'architecture des systèmes d'exploitation pourrait être défini par l'enseignement du lien entre matériel et logiciel. Or cette compétence est souvent considérée comme d'une grande technicité, mais aussi très spécifique au regard des métiers de l'informatique. Certains étudiants mais aussi des collègues enseignants avancent des arguments tels que la spécificité des systèmes, la complexité des compétences requises, et les débouchés limités, pour justifier de leur désintérêt pour la discipline.

Le professeur Sacha Krakowiak [9] aime cependant à rappeler qu'une telle définition de l'objet d'un système d'exploitation est trop grande (les compilateurs aussi, par exemple, gèrent le lien entre le matériel et le logiciel). Toutefois au-delà de ces questions de périmètre, réduire l'enseignement de l'architecture des systèmes d'exploitation à un domaine technique occulte sa véritable importance. L'enseignement de l'architecture des systèmes implique de dévoiler ce qui se cache derrière les abstractions courantes que les informaticiens apprécient. Rappelons que le terme « abstraction » provient du latin *abstractio*, dérivé du verbe *abstrahere*, signifiant « éloigner, extraire largement ». Ainsi, si l'étymologie de « abstraction » renvoie à l'idée de « rendre plus général » on oublie volontiers que cette généralité est obtenue par le retrait ou l'omission de certaines propriétés ou caractéristiques de la notion ainsi généralisée.

En ce sens, enseigner de quoi sont faites les abstractions que nos apprenants utilisent me paraît fondamental. Car enseigner l'architecture des systèmes c'est dévoiler ce que cachent les abstractions usuelles que les informaticiens utilisent. Bien sûr, l'abstraction permet de simplifier la compréhension et la gestion des notions complexes en en masquant les détails. Mais, ce faisant, elle en retire aussi de nombreux aspects, selon la perspective adoptée. Pour les informaticiens que nous formons, les aspects les plus souvent soustraits par les abstractions qu'ils connaissent sont ceux liés au matériel, et à travers lui, à la performance, à l'énergie consommée, à la latence ou encore à l'empreinte mémoire. Si la complexité algorithmique (que nous serons sans doute tous prêts à ranger dans la catégorie des connaissances fondamentales) enseigne l'art de mesurer les algorithmes, cette mesure n'a de sens que si elle est rapportée à une unité. C'est la connaissance des unités de l'informatique qu'enseigne l'architecture des systèmes, et c'est en ce sens qu'elle nous paraît tout aussi fondamentale que la complexité algorithmique.

Ensuite, enseigner la relation entre matériel et logiciel permet aux étudiants de connecter leurs abstractions logicielles à une réalité physique. Cette connexion leur offre également la possibilité d'explorer des usages du matériel différents de ceux standardisés par les systèmes d'exploitation courants. L'histoire des systèmes d'exploitation montre que même si certaines abstractions sont privilégiées, aucune abstraction du matériel n'est plus universelle que le matériel lui-même. Là encore, il faut comprendre qu'abstraire c'est retirer, en l'occurrence retirer des possibles que les informaticiens ne voient plus. Ainsi, les étudiants peuvent découvrir des opportunités pour repousser les frontières de ce qu'il est possible d'attendre d'un logiciel [12].

Enfin, la maîtrise de la sécurité des logiciels est un argument récent et crucial pour apprendre le lien entre matériel et logiciel. Savoir ce qu'est physiquement un programme ou une donnée permet de mieux les protéger, les isoler et les administrer. Cette connaissance est indispensable pour assurer la sécurité et la confidentialité des données, ainsi que la résilience des systèmes face aux cyber-menaces. Savoir ce qu'est une donnée dans le monde réel, savoir ce qu'est un traitement informatique en train d'opérer, c'est savoir quelle menace il faut redouter.

Mon propos ici n'est pas un plaidoyer contre l'abstraction en tant que synthèse et généralisation des connaissances. Cela serait complètement anachronique et contraire à toute démarche scientifique. Mon propos est de mettre en évidence l'importance de ne pas faire rimer abstraction avec ignorance ; alors que la facilité nous y incite naturellement. En somme, mon argument est que loin d'être une compétence anecdotique, la maîtrise du lien entre matériel et logiciel éclaire les logiciels et leur interaction avec le monde réel. Cela est une connaissance préalable nécessaire pour guider des choix technologiques pertinents et exploiter pleinement les capacités des technologies de l'information disponibles. Mais, au-delà de cet argument technique, enseigner le lien entre matériel et logiciel c'est enseigner à « expérimenter, expliquer et prédire » le comportement des logiciels ; dans notre monde mesurable et quantifiable. Et cela nous paraît être un enseignement scientifique fondamental ce qui n'est pas si fréquent dans les parcours de l'enseignement supérieur en informatique.

Objectifs pédagogiques

L'importance et le périmètre des enjeux pédagogiques ont été largement discutés dans ce document, et je ne reviendrai pas sur ces aspects. Cependant, pour mettre en place une unité d'enseignement efficace, il convient d'exprimer explicitement les objectifs pédagogiques visés.

Un cours d'architecture des systèmes d'exploitation doit viser à transmettre trois savoirs fondamentaux.

Gérer les matériels pour servir les applications. Cet objectif pédagogique doit permettre aux étudiants d'acquérir une compréhension de la matière première utilisé pour concevoir des systèmes informatiques : le matériel informatique. Il doit aussi

leur permettre de mesurer explicitement l'impact d'algorithmes logiciels sur les performances matérielles. Car l'optimisation de l'usage des matériels avec leur mise en sécurité forme le « rasoir d'Occam » qui permet de trancher entre les différentes architectures systèmes possibles.

Penser la définition de cadres de fonctionnement applicatifs. Il s'agit d'amener les étudiants, confrontés à ce qu'est l'exécution d'un logiciel sur un matériel « nu », à définir comment ils peuvent réaliser des bibliothèques, des services, des cadres de travail plus pertinents pour l'ingénierie des applications. Les grands paradigmes de conception qui ont été présentés dans la section « À l'enseignement de la notion de système » sont au cœur de cet enseignement.

Expérimenter et mesurer l'impact des choix. Le but de l'enseignement de l'architecture des systèmes d'exploitation est aussi, comme cela a été largement expliqué, d'amener les étudiants à débattre des possibles et de leur impact sur les applications.

En pratique, à travers ces enseignements, la plupart des étudiants découvrent les liens entre matériel et logiciel. Ils apprennent à argumenter des choix architecturaux et se dotent d'une vision holistique de l'informatique. Parallèlement, les apprenants développent des savoir-faire avancés tels que la programmation des matériels et la contribution aux systèmes d'exploitation (notamment à travers la programmation de modules noyau).

Moyens pédagogiques

Ces enseignements, de niveau master informatique, s'adressent aux étudiants ayant déjà acquis des compétences préalables en architecture matérielle, pratique du langage C, systèmes d'exploitation et algorithmique.

Il est organisé en trois modules de 3 crédits (ECTS) chacun. L'enseignement est réparti sur un total de 12 sessions de 1h30 de cours magistraux et 3 séries de 6 séances de travaux dirigés de 3 heures chacune, appelées TD machines. Seuls 3 crédits sont communs à toutes les spécialités du master d'informatique de Lille. Des spécialités très marquées (réalité virtuelle et augmentée d'une part et *machine learning* d'autre part) sont dispensées des 3 crédits suivants. Enfin les trois derniers crédits sont réservés à des spécialités où le lien entre matériel et le logiciel est essentiel et justifie un approfondissement internet des objets et *cloud computing*.

La stratégie pédagogique retenue par l'équipe pédagogique a consisté, depuis la mise en place de ces enseignements, à mettre en œuvre l'approche « essayer, observer et discuter » sur des projets relativement volumineux. Les étudiants sont amenés à réaliser leur « premier-premier programme », c'est-à-dire le premier programme qui démarre sur une machine sans système d'exploitation préalable. Comme cela a été dit, cette expérience fait prendre conscience à beaucoup de nos apprenants de ce qui est apporté par le système d'exploitation lui-même, et ce que le matériel nu offre vraiment.

Contenu pédagogique

Le cours magistraux se décomposent en quatre grandes parties, chacune abordant des aspects clés de l'architecture des systèmes d'exploitation.

La première partie porte sur la gestion des systèmes de fichiers, incluant l'étude de la pile logicielle (partition, système de fichiers et liaison), la sécurité à travers le contrôle d'accès et le cryptage, et la fiabilité par le biais de la tolérance aux pannes. C'est un bon moyen d'illustrer la notion de pile de logiciels et les architectures en couches. L'autre excellent exemple est l'implémentation de la pile réseau, mais généralement (et notamment à l'université de Lille) le modèle OSI est traité dans le cours d'introduction aux réseaux.

La deuxième partie se concentre sur la gestion du microprocesseur, en abordant la séquence d'amorçage d'un système, le partage du microprocesseur par l'ordonnancement et la synchronisation, ainsi que le chargement dynamique de code machine. Cette partie illustre notamment les problématiques d'asynchronisme. La troisième partie traite de la gestion de la mémoire de travail, en explorant la notion de mémoire virtuelle, l'allocation et la libération de la mémoire, et les techniques de « ramasse-miettes » pour le nettoyage automatique de la mémoire. Cette série de cours nous permet de revenir sur les notions de couche logicielle, mais surtout sur la notion de virtualisation des matériels. Cela permet aussi de discuter de la différence entre virtualisation (via la MMU) et abstraction (mémoire à objet et mécanisme de ramasse-miette). Enfin, la quatrième partie se penche sur l'hypervision, en étudiant la virtualisation des matériels, les différentes approches telles que l'émulation, la para-virtualisation et l'accélération matérielle, et la comparaison entre l'hypervision et la containérisation. Il s'agit ici de discuter de différentes stratégies architecturales et de transmettre une « culture générale » des problématiques système, en balayant « en largeur » une grande diversité de sujets. Il n'est bien sûr pas possible d'étudier en profondeur chacun de ces sujets en travaux dirigés. Aussi nous concentrons les TPs sur trois sujets. Au cours du temps les sujets ont changé, mais il y a toujours trois des quatre sujets suivants.

TP #1 – Système de fichiers. Dans ce TP, les étudiants programment de manière synchrone un contrôleur de disque simplifié afin de mieux comprendre les systèmes de fichiers. Ils accèdent de manière asynchrone au matériel en utilisant des registres, des états et des interruptions, et font le lien entre complexité et performance grâce à l'exemple du formatage. Ils abordent le partitionnement du disque, en implémentant une virtualisation des blocs, et apprennent à traduire les numéros de blocs virtuels-réels ainsi qu'à allouer et libérer des blocs. Enfin, ils mettent en œuvre l'abstraction classique du fichier, en étudiant le coût de l'abstraction et en discutant des éléments cachés par cette abstraction. Ce TP illustre l'architecture en couches de l'OS, du matériel à l'application.

TP #2 – Ordonnancement. Au cours de ce TP, les étudiants manipulent l'état du microprocesseur, en travaillant avec des fonctions telles que `longjump()`, `try()`, `throw()` et `switch_to()`. Ils étudient l'élection d'un contexte d'exécution à l'aide de `sched_yield()` et de coroutines, ainsi que l'interruption « *Timer* » et la notion de préemption et de partage. Ils explorent également les activités concurrentes et les synchronisations, en mettant en place des sections critiques de code et en comprenant la notion de concurrence. Les étudiants mettent en œuvre des sémaphores avec `sem_up()` et `sem_down()`. Ce TP permet d'éprouver les aspects réflexifs et asynchrones des systèmes d'exploitation.

TP #3 – Ordonnancement multi-cœurs. Ce TP (contributeurs P. Marquet puis G. Lipari) propose de mettre l'accent sur l'ordonnancement multi-cœur, en étudiant une abstraction commune (la notion de *thread*) et différentes implémentations de cette dernière. Les étudiants implémentent aussi des *spin-locks* et expérimentent avec un module noyau Linux. Ils travaillent également sur la gestion des cœurs, optimisant les caches grâce à l'entrelacement et à l'affinité des *threads*. De plus, ils abordent la gestion des entrées/sorties asynchrones en intégrant un matériel asynchrone, comme un clavier, et en analysant l'interaction entre l'ordonnancement et les périphériques. Ce TP permet d'expérimenter le sens du mot système dans l'interaction de ses membres.

TP #4 – Mémoire virtuelle. Dans ce TP (contributeur C. Ballabriga), les étudiants apprennent à programmer la MMU Intel en travaillant avec des registres, des états et des interruptions, et à virtualiser une ressource physique, comme la mémoire de travail. Ils sécurisent le système en isolant les accès mémoire, en séparant les pages du noyau et les pages utilisateur. Les étudiants mettent également en œuvre un mécanisme de `syscall()` pour comprendre l'interface utilisateur/noyau. Ensuite, ils gèrent la ressource mémoire en implémentant l'allocation de pages mémoire, en illustrant la fonction `stack_guard()`, et en réalisant un mécanisme de `fork()` ainsi que la politique de *copy-on-write*. Ce TP permet d'explorer les concepts de virtualisation et d'interposition, en comprenant les aspects sous-jacents de la virtualisation et en se familiarisant avec le concept d'interposition. Les étudiants apprennent ainsi à travailler avec des systèmes qui pourraient exister, mais qui ne sont pas encore implémentés, et à explorer l'importance des éléments placés entre différentes parties d'un système.

Chacun de ces travaux pratiques permet aux étudiants d'aborder des aspects spécifiques et détaillés de l'architecture des systèmes d'exploitation. Ils acquièrent ainsi une compréhension approfondie des différentes composantes et des interactions entre ces dernières sur des exemples concrets.

Les travaux pratiques sont conçus de manière à ce que les étudiants puissent exploiter efficacement le langage de programmation C, avec une utilisation minimale de l'assembleur. Pour faciliter l'apprentissage, une base de code minimale intitulée

my-kernel est fournie aux étudiants. Cette base de code génère une image ISO bootable, en utilisant des outils tels que `grub`, `gcc`, `ld` et `make`. Les étudiants peuvent alors tester leur travail sur une clé USB ou en utilisant un émulateur tel que `qemu`, leur offrant ainsi une expérience pratique et réaliste du développement de systèmes d'exploitation. Les sujets des travaux pratiques sont très scriptés et structurés sous forme de tutoriels, permettant aux étudiants de suivre étape par étape les instructions et de progresser à leur rythme tout en acquérant des compétences avancées.

Validation des acquis

Le contrôle des connaissances dans ce cours est effectué à travers deux méthodes d'évaluation qui visent à valider trois facettes de l'enseignement.

La première consiste en une revue de code, où les étudiants sont évalués sur la qualité de leur code et doivent mettre en œuvre une amélioration spécifique en fonction des commentaires reçus. Cela permet d'apprécier la compréhension des étudiants sur les concepts clés et leur capacité à appliquer ces concepts dans la pratique. Ils démontrent ainsi qu'ils ont acquis la capacité à développer des stratégies de gestion d'un matériel pour servir une abstraction.

La seconde méthode d'évaluation est un examen surveillé. Il s'agit d'abord de valider l'acquisition des connaissances sur le pilotage des matériels et la compréhension des paradigmes. Pour évaluer les connaissances factuelles, l'examen a recours à des questions à choix multiples (QCM). Enfin pour évaluer la capacité des étudiants à utiliser ces connaissances pour apprécier différentes approches, l'examen comporte des questions qui appellent des réponses argumentées. Ainsi les étudiants démontrent leur compréhension des choix architecturaux et leur capacité à justifier ces choix de manière réfléchie et structurée.

Évolution des enseignements et perspectives

Au cours des 20 années de mise en œuvre de ce cours, diverses évolutions et adaptations ont été apportées pour répondre aux besoins changeants des étudiants et aux avancées technologiques. Récemment, l'essor du *cloud* a soulevé des questions sur la manière d'enseigner l'architecture des systèmes aux étudiants du *cloud*, où les concepts tels que le langage machine, l'unité de donnée élémentaire, la structure de données élémentaire et l'ordinateur en tant qu'objet ont été modifiés ou ont disparu. L'impact des outils tels que ChatGPT sur notre méthode d'enseignement peut être considéré de deux manières : un second professeur qui apporte une aide supplémentaire et un rappel pour les étudiants de l'importance de maintenir leur esprit critique face à des réponses parfois surprenantes.

Les perspectives d'évolution pour ce cours incluent la possibilité :

— d'abandonner l'approche *from scratch* ;

— d'utiliser un langage de programmation plus moderne comme Rust ;

— d'illustrer la notion de portage avec des architectures comme ARM ou NUMA ;
— et de débattre davantage sur les différentes architectures : comme *user-space* vs *kernel-space* ; *micro-kernel* vs *uni-kernel* ; ou encore la performance vs l'énergie.

Ces adaptations permettront au cours de rester attractif et d'offrir aux nouvelles générations d'étudiants les compétences et les connaissances nécessaires pour réussir dans le monde de l'informatique en constante évolution.

En conclusion, j'ai voulu défendre, dans ce document, l'idée que l'architecture des systèmes d'exploitation est un enseignement fondamental nécessaire à la bonne formation des titulaires d'un BAC+5 en informatique. Il s'agit tant de doter nos étudiants d'un moyen de comprendre la réalité sous-jacente à toute réalisation informatique, que de savoir bénéficier de la longue expérience des nombreux paradigmes de programmation qui y ont été élaborés. Mais surtout l'architecture des systèmes d'exploitation est un excellent moyen de donner une vision holistique de l'informatique à nos étudiants. C'est enseigner à garder à l'esprit l'objet dans son ensemble tout en mesurant, au sens scientifique du terme, l'incidence de certains détails critiques à son endroit. Savoir argumenter sur les choix architecturaux des systèmes d'exploitation et des abstractions qu'ils proposent, m'apparaît comme l'une des pierres angulaires de nos formations. Elle doit permettre aux futurs professionnels que nous formons, de faire des choix organisationnels et technologiques éclairés ; et de savoir implémenter une démarche expérimentale pour valider leurs choix.

Références

[1] M.J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach : a new kernel foundation for unix development. *USENIX Summer Conference*, pages 93–112, July 1986.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5) :164–177, 2003.

[3] Samia Bouzefrane. *Les systèmes d'exploitation : Unix, Linux et Windows XP avec C et Java*. Edition Dunod, 2003.

[4] Francis Cottet and Emmanuel Grolleau. *Systèmes temps réel embarqués - 2e éd. - Spécification, conception, implémentation et validation temporelle*. Dunod, 2014.

[5] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel : An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5) :251–266, 1995.

[6] M. Hailperin. *Operating Systems and Middleware : Supporting Controlled Interaction*. Thomson Course Technology, 2007.

[7] M. Joy, S. Jarvis, and M. Luck. *Introducing UNIX and Linux*. Grassroots. Bloomsbury Publishing, 2017.

[8] Sacha Krakowiak. *Principes des systèmes d'exploitation des ordinateurs*. Dunod, 1991.

[9] Sacha Krakowiak. La discipline « système » état et évolution. https://lig-membres.imag.fr/krakowia/Files/Enseignement/Systemes/systemes.pdf, 02 2003.

[10] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962.

[11] John Mashey. Small is beautiful and other thoughts on programming strategies. 2003.

[12] David Nagle. Synergy between software and hardware. *ACM Computing Surveys (CSUR)*, 28(4es) :32–es, 1996.

[13] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley, and Dan Mackin. *UNIX and Linux System Administration Handbook (5th Edition)*. Addison-Wesley Professional, 5th edition, 2017.

[14] Nicolas Pons. *Linux : Cours et Exercices corrigés - Principes de base de l'utilisation du système (4e édition)*. Edition ENI, 2021.

[15] Sébastien Rohaut. *LINUX : Maîtrisez l'administration du système (6e édition)*. Editions ENI, 2020.