Equité d'accès aux ressources dans les systèmes partagés best-effort

François Goichon

N° d'ordre : 2013-ISAL-0162 Année 2013



Thèse

Équité d'accès aux ressources dans les systèmes partagés best-effort

Présentée devant

L'Institut National des Sciences Appliquées de Lyon

École Doctorale Infomaths

pour l'obtention du

Grade de Docteur

Spécialité Informatique

par

François Goichon

soutenue publiquement le 16 décembre 2013

Devant le jury composé de

Président :	Noël DE PALMA,	Professeur,	Université de Grenoble 1
Rapporteurs :	Gilles GRIMAUD,	Professeur,	Université de Lille 1
	Gaël THOMAS,	Maître de Conférences HDR,	Université de Paris 6
Directeurs :	Stéphane FRÉNOT,	Professeur,	INSA de Lyon
	Guillaume SALAGNAC,	Maître de Conférences,	INSA de Lyon

Thèse effectuée au sein du Centre d'Innovation en Télécommunications et Intégration de Services (CITI) de l'INSA de Lyon et des équipes AMAZONES et SOCRATE de l'INRIA Rhône-Alpes

Remerciements

Je tiens tout d'abord à remercier Stéphane Frénot de m'avoir permis d'effectuer mon master ainsi que cette thèse sous sa direction, ainsi que pour la transmission de son expérience sur le travail de recherche. Je remercie particulièrement Guillaume Salagnac pour son encadrement scientifique ces 3 années. Je suis conscient que me superviser n'est pas toujours reposant, mais ton implication sans faille, ta disponibilité et cet assortiment de confiance et d'esprit critique m'ont permis d'avancer continuellement et d'arriver au résultat qu'est cette thèse.

Je remercie les rapporteurs Gilles Grimaud, Professeur à l'Université des Sciences et Technologies de Lille, ainsi que Gaël Thomas, Maître de Conférences à l'Université Pierre et Marie Curie de Paris, pour leurs retours poussés et critiques. Je remercie également Noël de Palma, Professeur à l'Université Joseph Fourier de Grenoble, d'avoir accepté d'évaluer ce travail de thèse.

Je quitte ainsi le laboratoire CITI de l'INSA de Lyon au sein duquel j'ai eu la chance de travailler pendant plus de 4 ans, et une ambiance de travail chaleureuse dont je n'ai pas profité autant que j'aurais dû. Merci en particulier aux anciens membres d'AMAZONES pour une année très édifiante sur le plan scientifique, ainsi qu'à Gaëlle et Joëlle qui savent rendre simple les choses compliquées. De la même façon, merci à l'équipe du département Télécommunications, et en particulier à François Lesueur et Nicolas Stouls qui m'ont permis de m'amuser dans l'enseignement. Je souhaite également citer Stéphane Coulondre, Julien Ponge et de nouveau François Lesueur, qui ont alimenté, inconsciemment peut-être, l'élaboration de mon projet professionnel.

Cette thèse c'est aussi grâce à ma famille au grand complet. Déjà par les valeurs que représentent pour moi mes grands-parents et qui me permettent de prioriser les choses de la vie. Mais aussi pour les choses simples comme les retrouvailles sporadiques mais pleines d'affection avec mon frère et ma soeur. Et bien sûr grâce à mes parents, directement dans la thèse par le goût de l'informatique, le pragmatisme et la curiosité qui m'ont été transmis, mais surtout un soutien constant, important aux moments où j'ai songé à arrêter.

Epi mesi baw menm. An pa séten an té ké fine tez-la-sa tou sèl. Men an té pé vwè kijan tez aw pa té fasil, menm si sa ka sanm lwen konyéla, é ou pa jen sonjé aréte. Ou sav an motivé enki pou sé biten-la ki ka entérésé mwen, é pou tez-la-sa ou té sous motivasyon an mwen. Bon épi gg, dé doktè sa ni ganm.

J'ai une pensée particulière pour mon grand-père, qui a été pendant cette thèse et demeurera toujours une source de motivation dans les moments difficiles.

Résumé

Au cours de la dernière décennie, l'industrie du service informatique s'est métamorphosée afin de répondre à des besoins client croissants en termes de disponibilité, de performance ou de capacité de stockage des systèmes informatisés. Afin de faire face à ces demandes, les hébergeurs d'infrastructures ont naturellement adopté le partage de systèmes où les charges de travail de différents clients sont exécutées simultanément.

Cette technique, mutualisant les ressources à disposition d'un système entre les utilisateurs, permet aux hébergeurs de réduire le coût de maintenance de leurs infrastructures, mais pose des problèmes d'interférence de performance et d'équité d'accès aux ressources. Nous désignons par le terme *systèmes partagés best-effort* les systèmes dont la gestion de ressources est centrée autour d'une maximisation de l'usage des ressources à disposition, tout en garantissant une répartition équitable entre les différents utilisateurs.

Dans ce travail, nous soulignons la possibilité pour un utilisateur abusif d'attaquer les ressources d'une plateforme partagée afin de réduire de manière significative la qualité de service fournie aux autres utilisateurs concurrents. Le manque de métriques génériques aux différentes ressources, ainsi que le compromis naturel entre équité et optimisation des performances forment les causes principales des problèmes rencontrés dans ces systèmes.

Nous introduisons le temps d'utilisation comme métrique générique de consommation des ressources, métrique s'adaptant aux différentes ressources gérées par les systèmes partagés best-effort. Ceci nous amène à la spécification de couches de contrôle génériques, transparentes et automatisées d'application de politiques d'équité garantissant une utilisation maximisée des ressources régulées.

Notre prototype, implémenté au sein du noyau Linux, nous permet d'évaluer l'apport de notre approche pour la régulation des surcharges d'utilisation mémoire. Nous observons une amélioration significative de la performance d'applications typiques des systèmes partagés best-effort en temps de contention mémoire. De plus, notre technique borne l'impact d'applications abusives sur d'autres applications légitimes concurrentes, puisque l'incertitude sur les durées d'exécution est naturellement amoindrie.

Abstract

Over the last ten years, the IT services industry has gone through major transformations, to comply with customers ever-growing needs in terms of availability, performance or storage capabilities of IT infrastructures. In order to cope with this demand, IT service providers tend to use shared systems, executing mutiple workloads from distinct customers simultaneously on the same system.

This technique allows service providers to reduce the maintenance cost of their infrastructure, by sharing the resources at their disposal and therefore maximizing their utilization. However, this assumes that the system is able to prevent arbitrary workloads from having significant impact on other workloads' performance. In this scenario, the operating system's resource multiplexing layer tries to maximize resource consumption, as well as enforcing a fair distribution among users. We refer to those systems as best-effort shared systems.

In this work, we show that malicious users may attack a shared system's resources, to significantly reduce the quality of service provided to other concurrent users. This issue of resource control layers in shared systems can be linked to the lack of generic accounting metrics, as well as the natural trade-off that such systems have to make between fairness and performance optimization.

We introduce the utilization time as a generic accounting metric, that can be applied to the different resources typically managed by best-effort shared systems. This metric allows us to design a generic, transparent and automated resource control layer, which enables the specification of simple resource management policies centered around fairness and resource consumption maximization.

We applied this approach to the swap subsystem, a traditional operating system bottleneck, and implemented a prototype within the Linux kernel. Our results show significative performance enhancements under high memory pressure, for typical workloads of best-effort shared systems. Moreover, our technique bounds the impact of abusive applications on other legit applications, as it naturally reduces uncertainties over execution duration.

Table des matières

Chapitre 1

Introduction

Contexte

L'évolution de la puissance et de la capacité des systèmes informatisés et des réseaux sur les dernières années a transporté le service informatique vers une toute nouvelle dimension. Les fournisseurs de services maintiennent des infrastructures toujours grandissantes afin d'offrir à leurs clients la possibilité de disposer de ces systèmes à la demande et à distance. Ce modèle de service a initialement été consommé à grande échelle avec l'expansion des serveurs Web mutualisés. La possibilité technologique de supporter la virtualisation de systèmes sur des matériels toujours plus puissants a permis sa généralisation à toutes les tâches de traitement intensives, pour lesquelles les clients ont rarement les ressources informatiques à disposition ou la main d'oeuvre nécessaire.

D'un point de vue technique, ce modèle s'appuie sur le partage de systèmes, par virtualisation et/ou mutualisation, afin de réduire les coûts de maintenance tout en exploitant les ressources des infrastructures au maximum de leur capacité. La répartition optimale des charges de travail clientes est donc un point essentiel pour le profit des fournisseurs de services. D'un autre côté, les clients qui souscrivent à ces services ont besoin de garanties sur la performance générale des applications qu'ils déportent, et donc sur les ressources à leur disposition. Le niveau de service attendu par le client est cristallisé contractuellement dans les *Service Level Agreements* (SLA) qui comprennent diverses mesures de qualité de service, par exemple la bande passante réseau minimale ou la disponibilité de l'infrastructure. Le non-respect des SLA implique des pénalités tarifées dues au client par le fournisseur de service. Comme démontré par le litige entre Amazon et Bitbucket [Bit09], les deux parties sont perdantes en cas de manque aux SLA, que ce soit en termes d'image ou de bénéfices financiers.

Un facteur crucial dans ce cadre est l'impact potentiel de charges de travail égoïstes et potentiellement antagonistes les unes sur les autres. Cette interférence se matérialise notamment par le partage des ressources d'un système. En effet, les couches de gestion des ressources dans de tels systèmes doivent naturellement effectuer un compromis non trivial entre l'équité du service délivré aux

différents utilisateurs, et la maximisation globale de l'usage des ressources à disposition. Les fournisseurs de services répondent généralement à ce problème en partitionnant les ressources a priori, afin de pouvoir résoudre un problème statique d'optimisation de la répartition des charges de travail. Cette solution est cependant coûteuse puisque les ressources sont sous-utilisées lors de charges normales. De plus, de légères sur-réservations sont souvent adoptées afin d'optimiser la distribution et peuvent potentiellement mettre en danger la qualité de service délivrée aux utilisateurs lors de fortes contentions autour d'une ou plusieurs ressources. Cette approche à base de réservation statique et a priori s'oppose à une régulation dynamique de l'accès aux ressources, imposant une équité d'accès tout en essayant de tirer profit au maximum de la capacité des ressources à disposition. Notre travail se concentre ainsi sur la capacité des systèmes à effectuer cette régulation dynamique. Nous regroupons les systèmes d'exploitation ayant cet objectif sous le terme *systèmes partagés best-effort*. Les systèmes classiques utilisés couramment en tant que serveurs partagés, comme Linux ou Windows, ainsi que les environnements de virtualisation basés sur ceux-ci, comme Xen ou Hyper-V, sont des exemples typiques de systèmes partagés best-effort.

Motivations

Les SLA portent généralement sur un nombre restreint de ressources, comme le nombre de processeurs, la quantité de mémoire ou la bande passante de périphériques d'entrées/sorties. Cependant, les systèmes d'exploitation fournissent d'autres ressources, dont la qualité de service peut également être critique pour la performance de certaines applications. Des exemples de ressources supplémentaires incluent les structures de données du système, les nombres aléatoires, l'attribution du processeur ou le mécanisme de swap. Dans la pratique, aussi bien les ressources classiquement liées aux SLA que ces ressources additionnelles peuvent être manipulées afin de mettre en péril l'équité d'accès et donc la qualité de service fournie aux différents utilisateurs.

En premier lieu, la performance d'un certain nombre de ressources dépend d'heuristiques d'optimisation qui peuvent parfois porter préjudice à la qualité de service délivrée. Ainsi, les piles réseau priorisent certains protocoles, dans certains sens de transfert, ou ajustent la fréquence d'envoi des paquets en fonction d'en-têtes de paquets reçus. Nous montrons par exemple que dans le cas de la pile réseau Linux, un serveur TCP en réception obtient toujours une bande passante supérieure aux autres types de trafic. Les ordonnanceurs de requêtes d'entrées/sorties à destination de disques durs essaient quant à eux de minimiser les mouvements de la tête de disque. Dans ce cas, des accès intensifs localisés permettent à un attaquant de significativement différer d'éventuels accès disque concurrents. De la même façon, les algorithmes de remplacement de pages, mis en place par les services de swap, essaient de conserver les pages accédées le plus récemment en RAM. Or, cette heuristique est contre-productive pour le cas de charges de travail effectuant des accès séquentiels sur de larges

portions de mémoire. Un utilisateur abusif reproduisant ce type de charge de travail peut ainsi réduire dramatiquement les performances des accès mémoire du système, en maximisant le nombre de chargements et déchargements de pages mémoire.

En second lieu, certaines structures de données liées à la gestion des ressources ont une capacité maximale de traitement. De ce fait, les utilisateurs effectuant des requêtes intensives mettent naturellement en danger les propriétés d'équité ou d'optimisation recherchées par le système. Par exemple, les files d'attente de l'ordonnanceur d'entrées/sorties de Linux sont à la fois limitées en nombre et en capacité. Une attaque par inondation remplissant ces files peut entraîner une forte congestion, dégradant de manière significative la performance d'entrées/sorties concurrentes. D'autres ressources sont limitées en nombre mais peuvent être consommées en quantité arbitraire par les applications. C'est le cas des objets et structures algorithmiques du noyau, dont l'assèchement induit souvent un déni de service du système, mais également d'autres ressources sporadiques comme les nombres aléatoires.

Enfin, certaines techniques de facturations peuvent être trompées, permettant aux attaquants de reporter leur consommation sur la facture d'autres applications concurrentes. C'est par exemple le cas de l'ordonnancement CPU à base d'échantillonnage périodique, encore employé dans certains systèmes comme Xen.

Ces exemples illustrent ainsi la difficulté rencontrée par les systèmes partagés best-effort à assurer une équité dans l'accès aux ressources, tout en exploitant celles-ci au maximum de leur capacité. Chaque ressource est gérée de manière spécifique, en adoptant des critères d'optimisation et des techniques d'équité dédiées. Ceci complexifie à la fois l'administration globale d'un système, mais aussi l'apport de solutions génériques permettant une régulation équitable et performante de l'accès aux ressources.

Contributions

Dans ce cadre, notre travail est axé autour de la formulation de couches de contrôle génériques, applicables aux différentes ressources typiquement gérées par les systèmes partagés best-effort. Ces couches de contrôle doivent ainsi permettre la mise en place de politiques d'équité simples et efficaces, tout en maximisant la consommation des ressources.

Notre première contribution consiste en l'introduction d'une métrique générique de facturation des ressources : le temps d'utilisation. Cette métrique est employée pour la répartition de l'accès au processeur depuis l'apparition des systèmes à temps partagé, mais n'a pas été appliquée aux autres ressources des systèmes partagés best-effort. Elle permet pourtant une mise en évidence directe du travail imposé à une ressource, et le pourcentage de ce temps d'utilisation ramené aux différents utilisateurs permet de souligner d'éventuelles violations

d'équité. D'autre part, nous définissons une facturation à base d'évènements, dépourvue des écueils de facturations à base d'échantillonnage, applicable de manière générique aux différents types de ressources tout en n'imposant qu'une modification minimale des couches logicielles existantes.

Cette métrique nous permet ensuite de spécifier une architecture de contrôle du partage des ressources, transparente pour les utilisateurs et permettant la mise en place de politiques d'équité simples, dénuées de paramètres arbitraires. Notre couche de contrôle se pose en interception des requêtes, et estime le pourcentage de temps d'utilisation afin de détecter les utilisateurs abusifs et d'éventuellement différer certaines de leurs requêtes pour rétablir une répartition équitable. Les requêtes d'utilisateurs abusifs peuvent tout de même être transmises, au cas où elles ne risquent pas de pénaliser un autre utilisateur plus raisonnable. Ceci nous permet d'éviter l'apparition d'un sous-régime en dehors des périodes de contention. Dans les architectures à micro-noyaux, notre couche de contrôle peut de plus être déployée de manière automatique devant chaque gestionnaire de ressource, et réguler les accès sans configuration ou intervention extérieure.

Enfin, nous instancions notre proposition en développant un prototype de notre couche de contrôle, intégré au sein du noyau Linux et destiné à réguler l'utilisation du mécanisme de swap. Ce prototype nous permet d'évaluer l'impact de notre approche par rapport au système de swap de Linux, dont l'équité est basée sur son ordonnanceur d'entrées/sorties, ainsi que par rapport à une technique d'optimisation à l'état de l'art, le swap token [Jia09]. Nos résultats expérimentaux soulignent une amélioration considérable des performances de charges de travail typiques des systèmes partagés best-effort lorsque mises en concurrence avec des applications abusives. De plus, notre couche de contrôle réduit l'impact de ces applications abusives sur le comportement des charges de travail légitimes, puisque la variabilité des durées d'exécution en situation de contention mémoire est significativement diminuée.

Cadre de la thèse

Cette thèse a été réalisée dans le cadre d'une allocation du Ministère de l'Éducation Nationale, de la Recherche et de la Technologie (MENRT). Les travaux ont été effectués dans l'enceinte du laboratoire CITI de l'INSA de Lyon, au sein des équipes-projet AMAZONES puis SOCRATE de l'INRIA, et sous la direction conjointe du Docteur Guillaume Salagnac et du Professeur Stéphane Frénot.

Plan

Le chapitre 2 introduit le contexte de notre travail, et en particulier la nécessité de partage des systèmes dans les plateformes de services informatiques. Nous décrivons les technologies abritées par la notion de systèmes partagés best-effort, puis nous détaillons comment des attaques simples sur les ressources permettent de réduire de manière significative la qualité de service délivrée aux utilisateurs concurrents dans de tels systèmes.

Dans un deuxième temps, nous effectuons au sein du chapitre 3 un panorama des techniques et architectures de gestion des ressources dans les systèmes d'exploitation. La littérature scientifique a notamment apporté des algorithmes d'ordonnancement équitables génériques, ainsi que différentes techniques d'optimisation de la performance, dédiées à certaines ressources. Nous rapportons également les concepts nécessaires à une facturation efficace et juste de la consommation des ressources, ainsi que les architectures de régulation dynamiques qui ont été appliquées à différents types de systèmes et ressources.

Ceci nous permet d'introduire, dans le chapitre 4, notre approche concernant l'application d'un contrôle des ressources générique, équitable et performant. Nous montrons que le temps d'utilisation est une métrique générique applicable aux différents types de ressources et représentative de la pression imposée par les applications. Cette métrique nous permet de définir diverses politiques d'arbitrage des requêtes, équitables et évitant les sous-régimes, tout en abandonnant l'usage de paramètres arbitraires dédiés à une ressource en particulier. Enfin, nous introduisons une architecture générique de déploiement automatisé de cette couche de contrôle dans les systèmes à micro-noyaux.

Nous détaillons ensuite dans le chapitre 5 les techniques spécifiques à la gestion de la mémoire, en particulier celles concernant les situations de contention où la somme des besoins des applications dépasse la quantité de mémoire physique disponible. Ces techniques englobent notamment les algorithmes de remplacement de pages, les propositions d'optimisation du mécanisme de swap, ou encore l'élasticité mémoire exploitée dans les environnements virtualisés afin d'adapter dynamiquement la quantité de mémoire allouée aux systèmes invités.

Enfin, le chapitre 6 détaille l'application de notre approche définie dans le chapitre 4 à la régulation de l'utilisation du mécanisme de swap de Linux. Nos résultats expérimentaux valident l'apport de notre prototype, en montrant une amélioration significative de la performance d'applications typiques des systèmes partagés best-effort lors de fortes pressions mémoire. Ils soulignent également la réduction de l'impact d'applications abusives sur d'autres applications légitimes, en affichant une variabilité des durées d'exécution plus faible.

Nous terminons ce manuscrit par une synthèse de notre travail, accompagnée de perspectives d'évolutions de notre approche, notamment en la recoupant avec certaines techniques avancées par la littérature scientifique.

Chapitre 2

Motivations

Nous introduisons dans ce chapitre la notion de *système partagé best-effort*, à savoir les systèmes exécutant simultanément des applications provenant d'utilisateurs distincts et dont l'objectif est de concilier un partage équitable et une maximisation des performances globales. Ce cadre applicatif est notamment rencontré dans l'industrie du service informatique, typiquement pour les serveurs mutualisés et ceux du cloud computing.

Nous illustrons ensuite par un ensemble d'expériences les difficultés rencontrées par les systèmes partagés best-effort à assurer équité et performance pour une large variété de ressources communes. Nous montrons notamment qu'un utilisateur abusif peut prendre avantage de spécificités des couches de gestion des ressources afin de réduire significativement la qualité de service fournie à d'autres utilisateurs concurrents. Les spécificités ciblées incluent par exemple les heuristiques d'ordonnancement, l'implémentation des files d'attente ou encore les techniques de facturation.

2.1 Les systèmes partagés best-effort

La récente évolution de l'industrie du service informatique a propulsé la nécessité de partage des systèmes afin de pouvoir exécuter de manière simultanée un nombre toujours plus important d'applications clientes. Nous définissons dans ce cadre les enjeux de l'efficacité en termes de performance et d'équité de ces partages au niveau de la satisfaction des clients ainsi que du profit et de l'image des fournisseurs de services.

Nous explicitons également les technologies visées par notre travail, à savoir la virtualisation et la mutualisation basées sur des systèmes d'exploitation best-effort.

2.1.1 Nécessité du partage de systèmes

Au cours de la dernière décennie, l'industrie du service informatique a vu l'émergence puis le développement du *cloud computing*. Les services du cloud sont

déployés, délivrés et consommés à la demande et à distance par les clients. Les taxonomies communes englobent une large variété de catégories incluant notamment la mise à disposition d'applications (*Software as a Service* ou SaaS), de réseaux (NaaS) ou de plateformes (PaaS) préconfigurés, gérés et maintenus par le fournisseur d'accès cloud (*Cloud Service Provider* ou CSP) [TSS12].

Dans la pratique, cela implique l'exploitation partagée de systèmes, auxquels l'utilisateur obtient un droit d'accès temporaire pour l'exécution de ses charges de travail arbitraires. Dans cette configuration, les clients consomment le service en faisant abstraction du cadre d'exécution réel de la plateforme, sans avoir conscience de l'exploitation simultanée de l'infrastructure par d'autres clients. Ce cadre d'exécution demande une forte isolation entre les différentes charges de travail exécutées sur la plateforme afin de maintenir une qualité de service optimale pour chaque utilisateur selon les termes contractuels définis par les *Service Level Agreements* (SLA).

Or, le partage inhérent des ressources système logicielles ou matérielles par des charges de travail égoïstes, inconscientes de la concurrence et parfois nombreuses, peut engendrer des répercussions importantes sur la qualité de service délivrée dans ces systèmes partagés. Dans ce travail, nous nous intéressons à ces scénarios où un client, malveillant ou non, exploite les ressources mises à sa disposition au détriment des autres utilisateurs du système et de manière non équitable.

A l'heure où l'attente maximale tolérée par les utilisateurs finaux de systèmes informatisés est de l'ordre de quelques secondes [Bai01] et où le défi principal des CSP est d'optimiser les coûts tout en respectant la qualité de service contractuellement due aux clients [WGB11], l'équité d'accès aux ressources système est crucial afin de maximiser la satisfaction client en garantissant des réponses rapides et indépendantes de l'environnement d'exécution qui leur est invisible. Un manque d'isolation impactant de manière significative le service rendu à un client peut avoir des conséquences néfastes sur l'image du client comme du CSP [Bit09]. De manière générale, les CSP fournissent cette isolation en instaurant un certain nombre de quotas arbitraires sur la consommation des ressources classiques, par exemple en termes de bande passante ou de maximums alloués. Cette solution ne permet pas une exploitation optimisée de la capacité des ressources à disposition et induit un surcoût en termes d'infrastructures et d'administration pour le CSP. Nous montrons de plus qu'une telle réponse ne sait couvrir l'ensemble des ressources pouvant être attaquées au détriment des autres utilisateurs légitimes.

2.1.2 Technologies de partage des ressources

Les systèmes d'exploitation (OS) mis à disposition des utilisateurs distants dans le cloud sont essentiellement des systèmes d'exploitation classiques, dérivés de BSD, Linux ou Windows, déployés de manière courante en tant que serveurs. Ils sont communément décrits comme *best-effort*, c'est-à-dire qu'ils essaient de gérer les ressources à leur disposition de manière équitable entre les différents utilisateurs, la plus performante possible dans le cas général. Les systèmes best-

effort s'opposent aux systèmes à réservation, apportant des garanties a priori sur la disponibilité des ressources et les temps d'exécution, employés pour les applications temps réel notamment.

Virtualisation de systèmes. Ces systèmes ne correspondent que rarement à une entité physique unique, mais s'exécutent plutôt dans un environnement virtualisé. Dans ce cas, un système physique hôte, ou hyperviseur, gère plusieurs de ces systèmes d'exploitation classiques, dit invités. Ces environnements de virtualisation peuvent être de plusieurs types :

- les serveurs privés virtuels, utilisant des techniques de partitionnement et de restriction de privilèges afin d'offrir l'illusion de fonctionnement d'un système d'exploitation classique. Ils se servent cependant directement de fonctionnalités fournies par le système d'exploitation hôte sur lequel ils se reposent. Des exemples de serveurs privés virtuels incluent vServer [dL05] et openVZ [Kol06] pour Linux ou Virtual Server [Mic05] sous Windows.

- la paravirtualisation, technique de virtualisation dans laquelle le système hôte ne comporte qu'une couche minimale d'abstraction des ressources matérielles et gère les systèmes invités comme des processus de bas niveau. La paravirtualisation requiert cependant une adaptation des systèmes classiques invités aux interfaces de bas niveau, ou l'ajout de modules dédiés au dialogue avec le système hôte. La famille des micro-noyaux L4 est un exemple de tels systèmes, avec comme instances OKL4 [HL10], ayant des adaptations compatibles de Linux, Android et Windows, ou encore Fiasco.OC [LW09], capable de gérer des systèmes Linux ou Android invités.

- la virtualisation complète, où les hyperviseurs émulent une architecture matérielle dédiée au système invité. Ces hyperviseurs sont conçus spécialement afin de gérer efficacement de multiples systèmes invités virtualisés, et implémentent ainsi des mécanismes de bas niveau dédiés. Les hyperviseurs complets sont directement installés sur une couche physique et permettent l'installation de systèmes invités classiques sans modifications. Les systèmes virtualisés ont l'illusion d'opérer sur une couche matérielle concrète. Dans la pratique, ces hyperviseurs sont dérivés de systèmes d'exploitation classiques : par exemple, Xen [BDF+03] ou la famille ESX de VMware [VMw13] proviennent de modifications du noyau Linux, et Hyper-V [Mic12] est construit à partir de Windows. Il est à noter que Xen ou les produits VMware sont également capables, et recommandent, de s'appuyer sur des mécanismes de paravirtualisation afin d'obtenir de meilleures performances [BDF+03, SD.09].

Mutualisation de ressources. De manière orthogonale, de nombreuses plateformes d'hébergement ou de SaaS opèrent par mutualisation, c'est-à-dire que les charges de travail de multiples utilisateurs distincts s'exécutent sur le même

système d'exploitation, qu'il soit virtualisé ou non. Les utilisateurs partagent directement les ressources physiques, et la gestion des ressources se repose sur les mécanismes des systèmes hôtes en premier lieu, et ceux des systèmes invités en second lieu dans le cas de la virtualisation. Ces mécanismes se basent généralement sur un certain nombre de couches de multiplexage, dont l'objectif est d'arbitrer et d'ordonnancer le traitement des requêtes des différents utilisateurs.

La mutualisation est notamment courante pour l'exploitation partagée d'une application commune, par exemple d'un serveur Web ou d'une base de données. Ces applications sont typiquement focalisées sur l'intégrité et la confidentialité des données des différents utilisateurs d'une part et un passage à l'échelle performant permettant une gestion efficace des données d'un grand nombre d'utilisateurs d'autre part.

Systèmes partagés best-effort. Nous regroupons les différents types de plateformes partagées, maintenues par un *hébergeur* et exploitées simultanément par différents *utilisateurs* clients, sous l'appellation *systèmes partagés best-effort*. Cette notion implique notamment une volonté de la part des hébergeurs de répartir de manière équitable entre les utilisateurs l'accès aux ressources, tout en exploitant celles-ci au maximum de leur capacité. La *charge de travail* représente l'activité applicative arbitraire sollicitée par un utilisateur ; elle peut aller de l'exécution d'un simple calcul au lancement d'un système d'exploitation complet. Dans tous les cas, l'hébergeur doit être capable d'assurer de manière efficace et équitable la répartition des ressources entre les différents utilisateurs quelles que soient les charges de travail mises en concurrence.

Les systèmes partagés best-effort induisent un lien fort entre la qualité du service rendu à un utilisateur lambda d'une part, et le profil de consommation des ressources des autres charges de travail d'autre part. Dans ce contexte, l'isolation de performance des différentes charges de travail ou des systèmes virtualisés devient une problématique incontournable.

2.2 Attaques par réduction de qualité de service

Dans ce cadre d'utilisation partagée d'un système best-effort, nous nous intéressons aux ressources qui peuvent être exploitées de manière abusive par un utilisateur non-privilégié. Afin d'illustrer le pire cas d'utilisation, nous nous plaçons dans ce chapitre dans un scénario d'attaque où un utilisateur abusif, l'attaquant, essaie d'exploiter une ressource partagée au détriment d'un autre utilisateur légitime, la victime. L'impact de ces attaques peut aller d'une légère réduction de la qualité de service délivrée aux utilisateurs d'un type particulier de ressource à un déni de service généralisé du système [Bit09]. Dans le contexte contractualisé des systèmes partagés du cloud, ces réductions de qualité de service impliquent donc une violation des SLA, où le client comme l'hébergeur sont perdants.

Dans les sections suivantes, nous présentons plusieurs expérimentations qui illustrent ces attaques par réduction de qualité de service sur différents types de ressources : les ressources matérielles d'entrées/sorties, notamment les périphériques de stockage et de communication réseau, les ressources logicielles finies, limitées en nombre et consommées à la demande comme les données de capteurs, nombres aléatoires ou objets noyau, ainsi que les services transparents du système, consommant des ressources de manière implicite et transparente pour les utilisateurs, comme le sous-système de swap ou l'ordonnancement CPU. Nous présentons ici ces différentes ressources, leur gestion dans les systèmes d'exploitation, ainsi que les répercussions d'attaques ciblées sur celles-ci.

2.2.1 Inondations des ressources d'entrées/sorties

Les ressources matérielles d'entrées/sorties (E/S) réunissent la majorité des périphériques de communication et de stockage gérés par un système, comme la carte réseau ou les disques. Pour un système best-effort, le but principal de la gestion de ces périphériques est d'optimiser leur bande passante afin de traiter la plus grande quantité possible de données, tout en garantissant que chaque requête utilisateur soit traitée par le matériel dans les meilleurs temps possibles.

La gestion de ces périphériques est souvent séparée en plusieurs couches distinctes : les files d'attente et le pilote fournis par le système d'exploitation ainsi que les files d'attente et le *firmware* embarqués par le matériel. Cependant, chacune de ces couches a des connaissances et une vision différentes de l'état du matériel et des requêtes qui lui sont faites. La figure 2.1 représente la position et le rôle de ces couches lors de requêtes de processus utilisateurs à ces périphériques.

De manière générale, le processus utilisateur soumet sa requête à une couche unifiée de gestion des entrées/sorties. Cette couche ordonnance les requêtes en ayant une connaissance du nombre de requêtes en attente et des différents utilisateurs concurrents, ainsi que de l'état du matériel à haut niveau. Elle peut par exempler effectuer une approximation de la position de la tête de lecture d'un disque dur, estimation qui peut être éloignée de la réalité matérielle. Les requêtes sont ensuite transmises au pilote proprement dit, qui les remet à son tour au matériel. Elles sont enfin prises en charge par le firmware, logiciel embarqué dans le matériel, qui peut à son tour ordonnancer et traiter les requêtes en fonction de sa connaissance fine de l'état du matériel. Des couches additionnelles semblables sont insérées entre le pilote du système invité et le firmware matériel dans le cas de systèmes virtualisés, conservant les mêmes propriétés et buts.

Une telle architecture logicielle peut poser plusieurs problèmes lorsque l'on souhaite atteindre un partage équitable et performant :

- **Limites pratiques en bande passante.** Ces périphériques orientés débit ont une limite naturelle correspondant à un maximum de requêtes traitées en un temps donné. La bande passante effective du périphérique

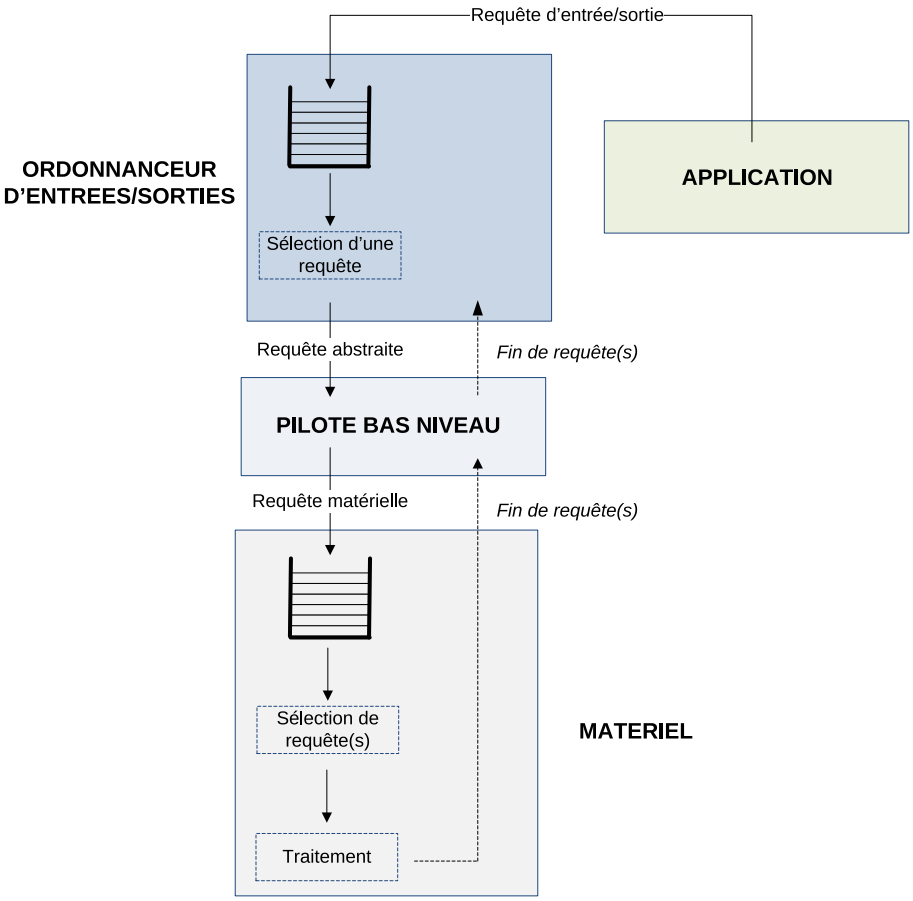


FIGURE 2.1 – Cycle de vie d'une requête d'entrée/sortie.

peut être grandement diminuée si un attaquant soumet un nombre de requêtes important qui ne maximise pas l'utilisation du périphérique en termes de débit. Un attaquant peut fortement diminuer la bande passante effective d'un disque dur mécanique en demandant des opérations sur des fichiers éloignés physiquement, maximisant le déplacement de la tête de disque. De manière orthogonale, la bande passante effective dépend également de la taille des files d'attente, aussi bien du matériel que du système, et de la capacité d'un utilisateur abusif à les remplir.

- **Ordonnancement priorisé.** Les différents réordonnancements opérés lors du traitement des requêtes sont basés sur des heuristiques visant la maximisation de l'utilisation du périphérique. Ces heuristiques sont spécifiques au matériel sous-jacent. Par exemple, dans le cas de disques durs mécaniques, les requêtes les plus proches de la position actuelle de la

tête de lecture vont être traitées en priorité. Certaines cartes réseau vont émettre en priorité des paquets de petite ou grande taille, ou vont préférer les protocoles les plus communs comme TCP. Bien que ces heuristiques maximisent l'utilisation du périphérique dans le cas général, un attaquant peut essayer de rendre prioritaire ses propres requêtes en les mettant en ligne avec les heuristiques appliquées. Ceci garantit à l'attaquant une bonne qualité de service tout en ayant un impact significatif sur le taux de service des requêtes des autres utilisateurs.

- **Réordonnancement successifs.** Comme souligné par Yu *et al.* [YSEY10] dans le cas des disques durs, les réordonnancements successifs des requêtes par le système d'exploitation et le firmware ne sont pas nécessairement basés sur les mêmes heuristiques et peuvent être contre-productifs. Ils peuvent notamment provoquer une famine d'accès à la ressource pour certaines requêtes lorsqu'une inversion de priorité entre les deux couches intervient, impliquant une attente maximale pour les requêtes concernées. Ce phénomène peut être encore plus visible dans le cas de la virtualisation, avec l'insertion d'une nouvelle couche de réordonnancement et de contrôle des requêtes par les hyperviseurs.

Ces problèmes théoriques sont modulés de manière ad hoc dans les firmwares ou les couches d'E/S des systèmes d'exploitation afin par exemple de réduire les périodes maximales de famine ou d'améliorer l'équité des réordonnancements. Ceci dit, un attaquant visant à monopoliser ces ressources orientées débit peut tout de même impacter de manière significative la qualité de service délivrée aux autres utilisateurs. Nous exposons dans les paragraphes suivants l'incidence qu'un attaquant peut avoir sur le comportement d'une charge de travail victime, sous Linux et dans le cas de transferts réseau ou disque.

Gestion des paquets réseau dans Linux. Dans Linux, comme dans la majorité des systèmes d'exploitation classiques, le modèle d'entrées/sorties utilisé pour les paquets réseau est simpliste : les cartes ne comportant généralement que deux tampons, un en émission et un en réception, les paquets sont transmis en FIFO. Chaque application ne peut effectuer qu'un transfert à la fois car elle doit attendre que son canal, communément le *socket*, soit prêt avant de pouvoir effectuer des opérations d'entrées/sorties. Ce modèle succinct restreint dans la pratique la surface d'attaque sur ces interfaces, mais d'autres heuristiques de la pile réseau permettent d'afficher des disparités de traitement significatives entre différentes charges de travail.

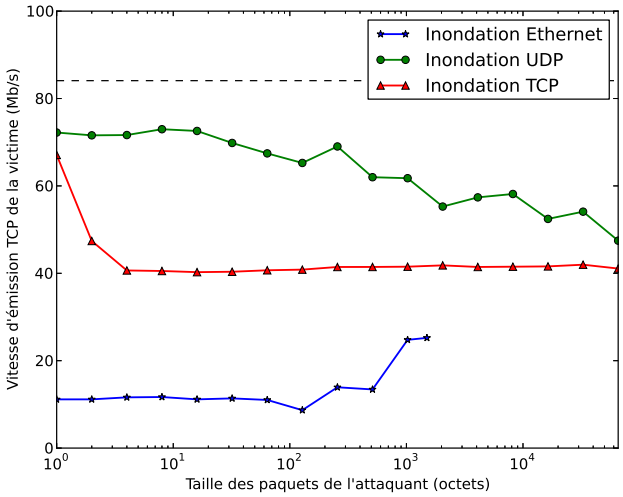


FIGURE 2.2 – Impact d'inondations réseau de l'attaquant sur la vitesse de réception d'un fichier de 25 Mo en TCP par la victime.

Attaque par inondation de paquets réseau. Afin de mettre en évidence les particularités de gestion de la carte réseau, nous utilisons un système muni d'une carte réseau ayant un débit de 100 Mb/s[1] avec une installation Linux basique. La victime transmet ou reçoit des fichiers de 25 Mo par UDP ou TCP. Les figures 2.2, 2.3 et 2.4 montrent l'évolution du débit obtenu par la victime lorsque l'attaquant inonde l'interface réseau avec des paquets de différentes tailles et par différents protocoles. Sur chaque figure, la limite en pointillé représente la vitesse de transfert obtenue par la victime sans concurrence d'accès. Pour chaque type de protocole, l'attaquant fait varier la taille de ses paquets, entre 1 octet et le maximum supporté par le protocole (65535 pour les paquets IP, 1500 pour les trames Ethernet). Le lecteur notera que l'injection de trames Ethernet nécessite en règle générale des privilèges additionnels. Bien qu'il soit possible de se les approprier dans certains cas précis [Tin09], nous avons utilisé cette injection bas niveau afin de montrer l'impact de la pile réseau de Linux en termes de performance et de décisions d'ordonnancement.

Le comportement que l'on peut attendre a priori d'un système best-effort est un partage proportionnel de la bande passante, soit une allocation de 50% à chaque utilisateur dans notre cas. De manière surprenante, seul l'envoi du fichier en TCP figure 2.2 obtient effectivement la moitié de la bande passante obtenue lorsque la charge de travail victime est exécutée seule. Ces expériences

[1] Carte réseau Broadcom NetLink BCM5906M.

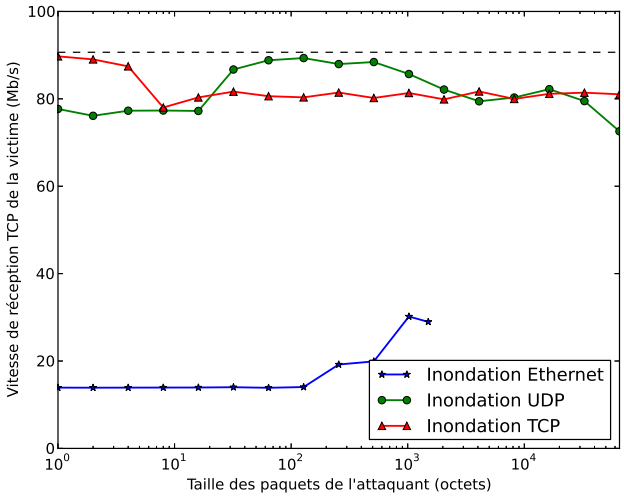


FIGURE 2.3 – Impact d'inondations réseau de l'attaquant sur la vitesse d'envoi d'un fichier de 25 Mo en TCP par la victime.

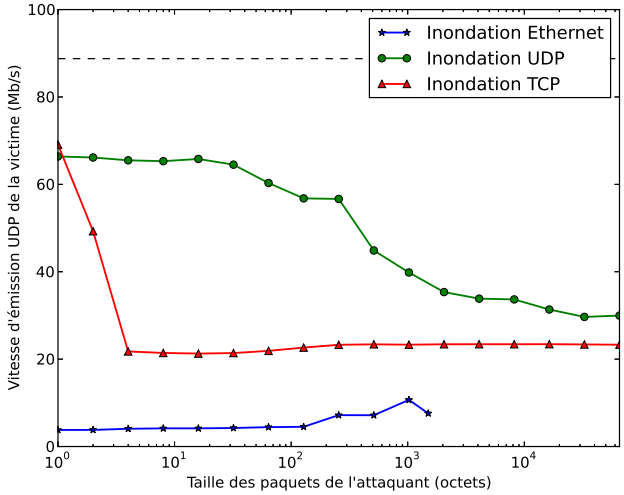


FIGURE 2.4 – Impact d'inondations réseau de l'attaquant sur la vitesse de réception d'un fichier de 25 Mo en UDP par la victime.

mettent ainsi en avant l'impact de différentes variables sur la performance de la charge de travail victime :

- La taille des paquets de l'attaquant affecte la performance de la victime. La figure 2.3 montre qu'une inondation avec de très petits ou très grands paquets UDP est plus efficace contre une réception TCP qu'avec des paquets de taille moyenne. Au contraire, une inondation Ethernet a un impact significatif lorsque les trames sont plus petites que 128 octets. Ces particularités sont notamment dues aux tailles des tampons internes du noyau et du matériel, permettant des envois et réceptions plus ou moins efficaces.

- La pile réseau de Linux différencie les protocoles. Tout d'abord, l'utilisation d'une inondation Ethernet sans protocole spécifique a un profil très différent des inondations TCP et UDP, ce qui implique une interférence de la pile réseau sur la gestion des paquets de ces protocoles de plus haut niveau. On observe également une disparité entre les performances de ces deux protocoles : sur la figure 2.4, l'envoi de paquets UDP est très fortement pénalisé par une inondation TCP simultanée et la figure 2.2 confirme qu'une inondation UDP a un impact généralement plus faible.

- Les paquets en émission et en réception ne sont pas traités de manière semblable. En effet, la réception de paquets TCP de la figure 2.3 obtient a minima 80% de la bande passante effective en concurrence avec des inondations TCP ou UDP, et propose un profil très différent de l'envoi de paquets TCP représenté sur la figure 2.2.

Dans ces expériences, la taille des fichiers échangés est fixe et l'attaquant n'utilise qu'un processus et une connexion pour son attaque. Or, si l'attaquant augmente sa quantité de processus, il diminue encore la bande passante attribuée à la victime. Une expérience complémentaire, avec un attaquant exploitant deux inondations TCP simultanées et des paquets de 1024 octets, réduit la vitesse de réception TCP de la victime à 25 Mb/s. Ceci est simplement dû au traitement FIFO des paquets, puisque l'attaquant peut obtenir autant de paquets que de processus dans la file d'attente, contrairement à l'application victime n'utilisant qu'un seul processus.

En se servant de ces informations, un attaquant sait qu'il a un impact significatif sur d'éventuelles communications concurrentes en se servant de plusieurs connexions TCP intensives en réception. Ainsi, en reproduisant une attaque à base de 5 serveurs TCP intensifs simultanés, nous observons une vitesse de téléchargement HTTP d'à peine 1 Mb/s pour la victime, contre 88 Mb/s sans concurrence.

Ces données soulignent simplement l'existence de différences de traitement, selon des caractéristiques arbitraires, opaques à l'utilisateur. D'autres heuristiques des piles réseau, visant certaines fonctionnalités des protocoles implémentés comme l'optimisation de la bande passante en fonction du taux d'erreur ou

la fragmentation IP, peuvent être détournées pour monter des dénis de service plus marquants [KK03, GBM04, GH13].

Gestion des disques sous Linux. Le stockage disque est traditionnellement la ressource la plus délicate à gérer pour un système d'exploitation, car le temps d'exécution de ses requêtes est de l'ordre de la milliseconde, tandis que les accès à la mémoire centrale sont résolus en quelques nanosecondes. Le disque est donc par défaut un goulot d'étranglement pour le système et sa gestion par les firmwares associés et les systèmes d'E/S s'est complexifiée afin d'optimiser le service des requêtes, notamment en ajoutant différents niveaux de cache et divers réordonnancements de requêtes.

Pour un utilisateur Linux non-privilégié, les requêtes d'E/S à destination des périphériques orientés bloc traversent les diverses couches représentées sur la figure 2.1 : les requêtes sont transmises à une couche unifiée d'E/S. Le système cherche ensuite à fusionner la nouvelle requête avec une requête déjà en attente qui référence par exemple un bloc physique proche. La requête est insérée dans une file d'attente et le système peut rendre la main à l'utilisateur. Par défaut, les requêtes travaillent sur un cache spécifique, le *Page Cache*. Ce cache est un mécanisme critique de performance des E/S sous Linux puisque les requêtes de bas niveau peuvent directement manipuler les données du Page Cache dont l'adressage est toujours valide quel que soit le contexte d'exécution et en particulier pendant les interruptions matérielles [BC05]. Il permet également de diminuer le nombre effectif de requêtes matérielles engendrées, puisque pouvant servir plusieurs lectures utilisateur depuis le Page Cache, même lorsque les modifications n'ont pas encore été perpétrées sur les blocs physiques. Enfin, par défaut, le système consulte des blocs supplémentaires pour les requêtes en lecture et ajoute le surplus au Page Cache, anticipant une lecture séquentielle.

L'une des particularités de l'ordonnancement de requêtes disque est l'heuristique de proximité physique. Afin de minimiser les mouvements mécaniques et lents des têtes de lecture des disques durs, les requêtes sont agrégées au niveau matériel ainsi qu'au niveau du système d'exploitation. Ceci peut poser des problèmes de famine si un attaquant réussit à engendrer un nombre important de requêtes à forte localité physique afin qu'elles soient servies en priorité. Ces heuristiques ont été étudiées en profondeur par Yu *et al.* [YSEY10] mais sont difficiles à attaquer dans la pratique. En effet, la localité physique est une notion invisible pour l'utilisateur non-privilégié, sauf cas particulier [Tin09]. De plus, les algorithmes d'ordonnancement des systèmes d'exploitation et ceux embarqués sur les disques ont mis en place des mécanismes afin de réduire les possibilités de famine. Parmi ces mécanismes, on retrouve notamment le temps d'attente maximal des requêtes, 2 secondes sous Linux par exemple, ou la mise en place de politiques d'ordonnancement équitables, comme le *Completely Fair Queuing* (CFQ) sous Linux maintenant plusieurs files d'attente ordonnancées en *round-robin* [BC05].

Nous nous intéressons ici à la robustesse du système de gestion des E/S de Linux par défaut, avec les différents types d'E/S fournies aux utilisateurs

non-privilégiés :

- E/S bufferisées : ce sont les E/S classiques, utilisant le Page Cache pour minimiser le nombre de requêtes matérielles et retardant au maximum l'envoi effectif de ces requêtes afin de permettre de potentielles agrégations.
- E/S synchronisées : les E/S synchronisées garantissent que les requêtes ont été ou sont en train d'être exécutées sur le périphérique au moment où l'appel système d'envoi de la requête retourne à l'utilisateur appelant. Ceci implique notamment le déchargement des données du Page Cache vers le fichier concerné. Ces E/S sont typiquement utilisées par les applications critiques qui souhaitent s'assurer que les modifications sont bien effectuées avant de continuer leur exécution.
- E/S directes : les E/S directes passent outre les caches et tampons du système ; les transferts mémoire sont directement opérés entre le tampon fourni par l'utilisateur et le périphérique. Ces E/S sont notamment manipulées par les applications gérant déjà un cache interne, comme les bases de données. Bien que les E/S directes n'offrent pas de garanties de synchronisation, il est possible d'utiliser des E/S directes et synchronisées.

Attaque par inondation de requêtes disque. Pour illustrer l'impact que peut avoir une attaque par inondation sur la performance d'un disque, nous utilisons un système muni d'un disque SSD[2] avec installation Linux basique. Dans notre exemple, la victime calcule un *ou exclusif* bit-à-bit sur deux fichiers déjà existants de 10 ko et place le résultat dans un troisième fichier, à créer. La victime fait donc intervenir des requêtes en lecture et en écriture de manière intensive sur des fichiers de taille plutôt faible. L'attaquant effectue sur des fichiers de taille variable des écritures intensives, qui demandent plus de travail qu'une lecture au système ainsi qu'au périphérique. Elles sont de plus aléatoires afin de minimiser l'impact du Page Cache et de la lecture anticipée. Les deux utilisateurs opèrent par E/S bufferisées. La figure 2.5 montre le temps mis par la victime à effectuer sa charge de travail selon la taille du fichier manipulé par l'attaquant.

Cette expérience met en avant la fragilité des E/S bufferisées due à la capacité limitée des files d'attente du système. En effet, les files d'attente supportent un maximum de 128 requêtes par défaut sous Linux, avec un seuil de congestion à 113 au-dessus duquel le système va limiter la vitesse de création de nouvelles requêtes et essayer de désengorger la file [BC05]. De plus, lors de ce déchargement, c'est naturellement une majorité de requêtes de l'attaquant qui sont envoyées puisqu'il manipule un fichier significativement plus large, au détriment des requêtes de la victime, moins nombreuses. On voit donc que lorsque l'attaquant manipule un fichier de 250 Mo, le temps nécessaire au travail de la victime, sur des fichiers de 10 Ko seulement, s'élève à plus de 40 secondes

[2]Disque Samsung SSD RBX Series 64GB.

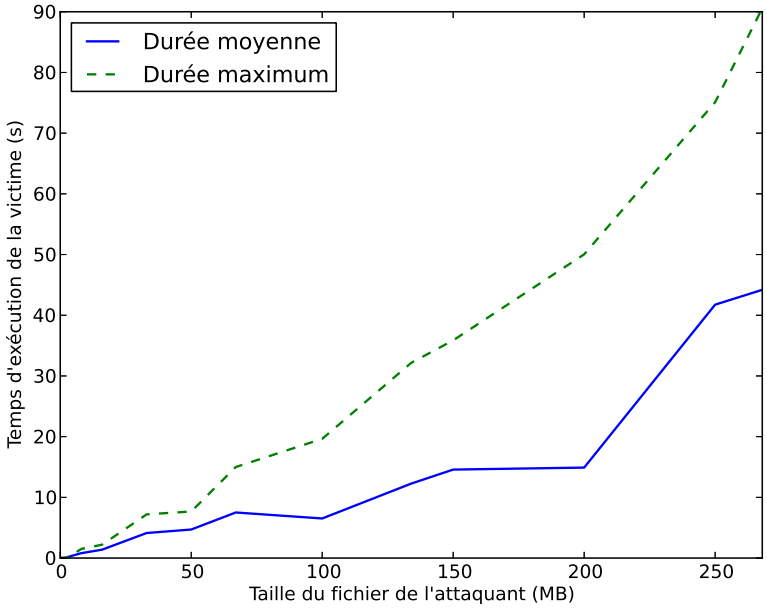


FIGURE 2.5 – Impact de la taille du fichier manipulé par l'attaquant sur la congestion des files d'attente des requêtes d'entrées/sorties du système, et donc sur les performances de la victime.

en moyenne. Si la victime est seule, ce temps est en moyenne de 10 millisecondes : la dégradation se fait donc par un facteur 4000, tout à fait contraire aux attentes best-effort d'un partage équitable et performant. Il est à noter que dans ces conditions, les différentes applications qui requièrent une attente de synchronisation des tampons d'E/S du système sont indéfiniment bloquées puisque les tampons n'ont jamais le temps d'être déchargés avant que de nouvelles requêtes n'arrivent. C'est notamment le cas des éditeurs de texte ou des lecteurs de flux multimédia.

Les E/S bufferisées constituent la meilleure attaque contre tout type d'E/S utilisé par la victime. Si la victime opère par E/S directes, le temps d'exécution est comparable mais légèrement inférieur au cas des E/S bufferisées car la victime n'utilise pas le Page Cache surchargé mais voit ses requêtes tout de même en minorité dans les files d'attente. Si la victime opère par E/S synchronisées, directes ou non, le temps de calcul du *ou exclusif* des deux fichiers de la victime est de 1300 secondes en moyenne, soit plus de 20 minutes. Ceci correspond à un débit effectif de 8 octets par seconde.

Nous notons enfin que les files d'attente CFQ du système d'E/S sont limitées en nombre, 64 par défaut, et sont liées aux processus et non aux utilisateurs.

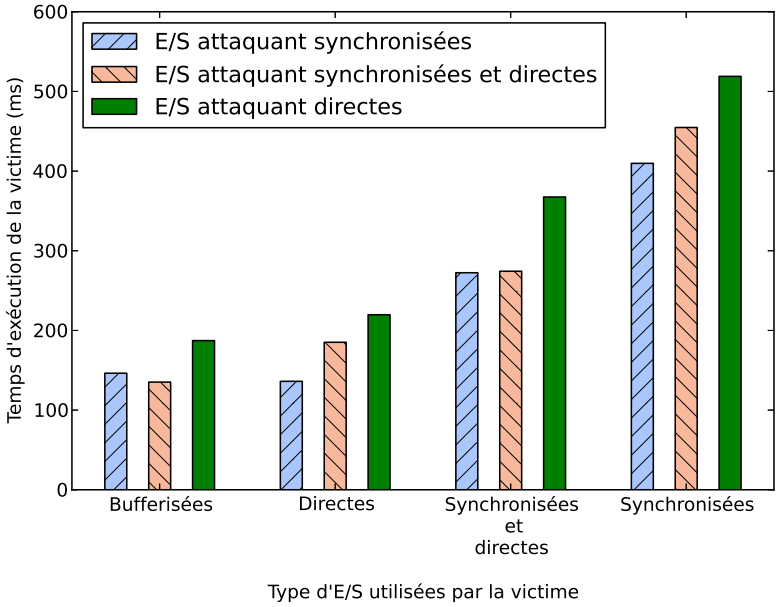


FIGURE 2.6 – Impact des types d'E/S utilisés par l'attaquant et la victime sur l'efficacité de l'attaque.

Cela signifie qu'un utilisateur ayant deux processus dont les requêtes ne sont pas placées dans les mêmes files obtient naturellement une proportion plus importante de la bande passante. Cette particularité devient problématique dans le cas de systèmes mutualisés.

La figure 2.6 a pour but d'étudier le cas des autres types d'E/S, et notamment d'observer les différences d'ordonnancement entre les E/S directes, n'utilisant pas le Page Cache, et les autres. Elle présente le temps nécessaire à la victime pour effectuer sa charge de travail alors que l'attaquant manipule un fichier d'une taille fixe de 400 Mo. Les deux utilisateurs se servent de tous les types d'E/S à leur disposition, à l'exception près que l'attaquant n'opère pas par E/S bufferisées.

Elle montre notamment que l'emploi d'E/S directes brutes est le meilleur choix pour l'attaquant quel que soit le type d'E/S de la victime. Cette observation s'explique par de multiples phénomènes : tout d'abord, les E/S directes imposent plus de travail au système qui doit mettre en place des mécanismes supplémentaires assurant que l'adresse des tampons utilisateur sera valide au moment de l'envoi de la requête au matériel. De plus, le fait de ne pas utiliser de cache maximise le nombre de requêtes effectuées, car plusieurs requêtes simultanées qui auraient été fusionnées en cache doivent être résolues indépen-

damment. Les E/S synchronisées sont le pire cas, pour l'attaquant comme pour la victime, puisque chacun doit attendre la transmission effective de chaque requête au matériel, incluant le déchargement du Page Cache et la sélection de la requête par la couche d'ordonnancement.

Discussion. Cette section met en évidence l'incapacité de Linux à partager de manière équitable l'accès aux ressources d'entrées/sorties et à protéger les utilisateurs légitimes de l'impact de charges antagonistes concurrentes. La taille des paquets, le sens de transfert ou le protocole utilisé peuvent avoir un impact significatif sur la performance d'un transfert réseau. Dans le cas de requêtes disque, la capacité des files d'attente peut être facilement mise à défaut et dégrader de manière dramatique les performances d'opérations d'entrées/sorties simples. Enfin, le traitement FIFO des paquets réseau et la conception des files d'attente CFQ des requêtes disque adoptent une équité orientée processus. Ainsi, un utilisateur unique transmettant ses opérations grâce à plusieurs processus dans un système mutualisé va naturellement accaparer une proportion plus importante de la bande passante.

2.2.2 Assèchement des ressources logicielles finies

Les ressources logicielles finies sont des ressources limitées en nombre que les utilisateurs consomment en quantité arbitraire et qui sont réapprovisionnées de manière sporadique. Elles incluent en particulier les données de capteurs ou les générateurs de nombres aléatoires basés sur un ensemble d'évènements difficilement prévisibles. L'approvisionnement de ces ressources est donc simple : lorsqu'un évènement intervient, le système collecte les données et met à jour la quantité d'information disponible pour la ressource. Côté utilisateur, une requête simple de lecture permet d'obtenir des données s'il y a suffisamment d'information pour satisfaire la requête. Si ce n'est pas le cas, la requête va être mise en attente, échouer ou renvoyer la quantité disponible d'information, l'asséchant pour les autres utilisateurs.

L'essence même de ce type de ressources entraîne des possibilités triviales de monopolisation :

- **Service aléatoire.** Que ce soient les générateurs de nombres aléatoires dans les systèmes classiques ou les services matériels comme l'instruction RdRand des processeurs Intel [Int13], il existe un facteur aléatoire dans le service rendu aux utilisateurs. En effet, de par l'arrivée sporadique des évènements, deux utilisateurs en compétition pour l'accès à l'information vont se réveiller tour à tour afin de demander la ressource, puis passer la main en attente de la ressource en cas d'échec. Il est donc impossible de prévoir lequel sera en cours de demande lorsque de l'information sera à nouveau disponible et un attaquant effectuant un large nombre de requêtes simultanées aura une meilleure probabilité d'être servi le premier, induisant une attente indéfiniment longue pour la victime.

- **Assèchement.** Puisque l'information n'est disponible qu'en quantité limitée et qu'il est le plus souvent possible pour les utilisateurs d'en demander une quantité arbitraire, un attaquant est capable d'assécher constamment la ressource en demandant systématiquement le maximum de la quantité d'information disponible, induisant également une attente de plus en plus importante pour la victime.

Afin d'illustrer le comportement de ces ressources, nous exposons dans les paragraphes suivants les cas des générateurs de nombres aléatoires et ceux des objets du noyau.

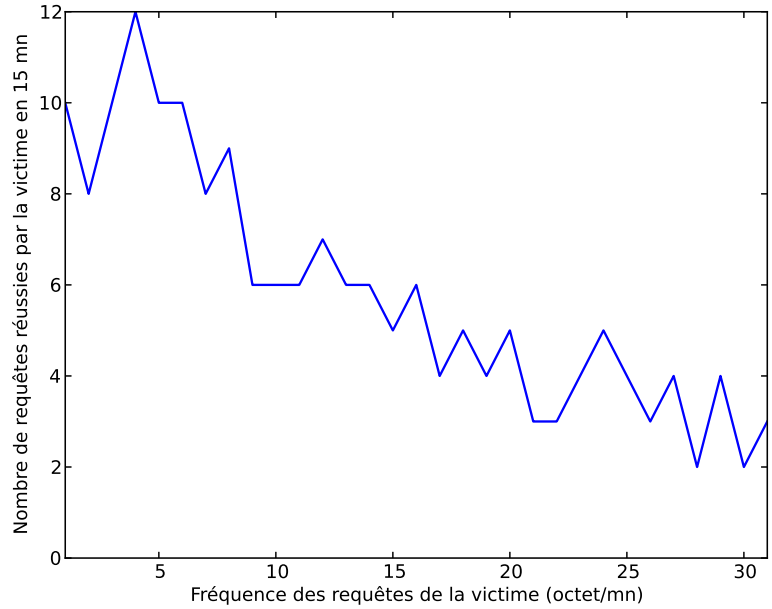
Génération de nombres aléatoires sous Linux. Afin d'étudier le comportement de ces ressources particulières, nous avons utilisé l'interface */dev/random* du générateur de nombres aléatoires intégré au noyau Linux. Ce générateur reçoit de l'entropie d'évènements système difficilement prédictibles, comme les lectures disque, les frappes clavier ou les déplacements de la souris. Il évalue l'entropie apportée par chacun de ces évènements et nourrit un *pool* d'entropie interne à partir duquel il sert ensuite les requêtes en nombres aléatoires du noyau ou des utilisateurs.

Le noyau fournit deux interfaces distinctes à ses utilisateurs : */dev/random* et */dev/urandom*. La différence essentielle entre ces deux interfaces réside dans le fait qu'en cas d'assèchement du niveau d'entropie, */dev/random* bloque les requêtes en attente jusqu'à ce que suffisamment d'entropie soit collectée, tandis que */dev/urandom* va continuer à servir les requêtes même si les nombres aléatoires servis sont cryptographiquement moins sûrs.

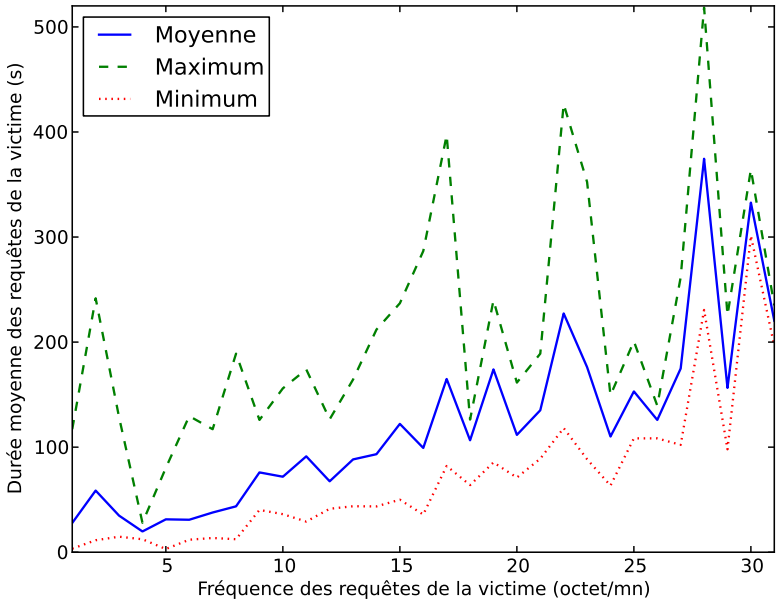
Attaque par assèchement d'entropie. Dans ce cadre, un attaquant peut continuellement demander des nombres aléatoires afin d'assécher le niveau d'entropie et forcer les autres lectures sur */dev/random* à attendre un temps indéfini. La figure 2.7 illustre ce cas d'utilisation sur un système Linux 3.9.1. L'attaquant demande de manière intensive des nombres aléatoires tandis que la victime requiert entre 1 et 32 octets aléatoires par minute (maximum recommandé [Man13]).

Comme attendu, la figure 2.7a montre que le taux de service n'est pas stable et tend vers 2 à 5 lectures réussies en 15 minutes pour plus de 120 bits aléatoires, contre 13 à 15 en utilisation normale. Les temps de lecture illustrés par la figure 2.7a, qui sont de quelques nanosecondes en utilisation normale, passent au-dessus de la minute dès 72 bits demandés et peuvent aller au-delà des 8 minutes. A partir de 128 bits, le temps minimal de lecture dépasse la minute. Ces résultats sont aggravés dans le cas où l'attaquant lance plusieurs processus effectuant de manière simultanée des lectures intensives sur */dev/random*.

A titre de comparaison, la durée par défaut d'attente d'une connexion TCP dans le noyau Linux est de 20 secondes. Cela signifie notamment que des applications dépendantes des nombres aléatoires délivrés par */dev/random*, par exemple un serveur SSH, peuvent voir leurs services complètement interrompus. Pour mieux comprendre ce problème, nous avons mené une étude sur le



(a) Nombre de requêtes de la victime satisfaites en 15 minutes.



(b) Durée moyenne de service des requêtes de la victime.

FIGURE 2.7 – Impact d'une attaque par assèchement d'entropie sur les requêtes de nombres aléatoires concurrentes.

comportement du générateur aléatoire de Linux [VGLS12]. Nous avons, entre autres, mesuré les transferts d'entropie aux entrées et aux sorties du générateur, ainsi qu'entre ses composants internes. Cette étude a permis de valider la possibilité d'attaque par assèchement décrite plus haut. Cependant, elle a également montré qu'en pratique, la surface d'attaque est limitée, car peu d'applications manipulent le périphérique bloquant */dev/random*. La plupart des programmeurs préfèrent les interfaces non-bloquantes comme */dev/urandom*, ou alors ne consultent */dev/random* que pour initialiser leur propre générateur pseudo-aléatoire logiciel. Bien que les nombres aléatoires servis par */dev/urandom* soient cryptographiquement moins sûrs en cas d'assèchement, ils sont jugés suffisamment robustes pour une majorité d'applications [Man13]. Certains systèmes d'exploitation comme FreeBSD ont d'ailleurs choisi de ne pas fournir d'interface bloquante pour la lecture de nombres aléatoires. Des générateurs de nombres aléatoires alternatifs comme HAVEGE [SS03] permettent quant à eux de fournir une bande passante en nombres aléatoires cryptographiquement sûrs bien supérieure à celle de */dev/random*.

Attaques par assèchement d'objets noyau sous Linux. Afin de gérer les applications et ressources à sa disposition, le système d'exploitation est obligé de maintenir un nombre conséquent d'objets noyau, liés à des ressources matérielles ou abstraites, représentant l'état du système et des applications. On y retrouve notamment les divers descripteurs (de fichiers, de noeuds physiques ou encore de processus), mais également les structures algorithmiques, comme les files d'attente réseau, listes de threads ou arbres de pages mémoire. Bien que la plupart de ces objets ne soient limités en nombre que par la capacité d'allocation mémoire du système, l'assèchement de certains d'entre eux est possible, et peut entraver la bonne marche du système. Nous décrivons ici deux exemples d'attaques classiques sur les objets noyau : le *SYN flooding* et la *fork bomb*. Ces attaques étant communes et leur impact très bien documenté [CER00, Wik13], nous ne retranscrivons pas ici d'exemples d'utilisation.

Le SYN flooding est une attaque se servant de ce qui fait la force du protocole TCP : le fait qu'il maintienne des connexions à états. La pile réseau doit donc stocker les informations relatives à chaque connexion TCP établie, notamment les numéros de session de chaque partie afin de pouvoir communiquer de manière robuste. Dans le but d'échanger ces informations, l'initialisation d'une connexion TCP prévoit l'échange de packets SYN : lors de la réception d'un paquet SYN, le système renvoie un paquet SYN/ACK avec son propre numéro de session, en attente d'une confirmation de réception via un paquet ACK. En l'attente de la réception du paquet ACK, les informations de connexion sont placées dans une file d'attente dédiée et plusieurs réémissions du paquet SYN/ACK sont tentées. Ainsi, un attaquant envoyant un grand nombre de paquets SYN va engendrer autant de connexions en attente de synchronisation qui n'auront jamais de réponse (paquets ACK ou RST). Lorsque la file des connexions en attente de synchronisation est remplie, le système ne peut plus gérer de nouvelles connexions et doit attendre que le délai d'attente pour cha-

cune des connexions semi-initialisées expire, soit plusieurs minutes par défaut. Aucune connexion ultérieure ne pouvant s'établir, le service réseau est hors d'usage. Cette attaque requiert un accès réseau au système ciblé.

Le concept de fork bomb repose quant à lui sur une boucle infinie de création de processus et emprunte son nom à l'appel système Unix de duplication d'exécution, `fork()`. Lors de la mise en place d'une telle attaque, la table des processus du système, la mémoire et l'ordonnancement CPU sont naturellement saturés, jusqu'à ce que le système ne soit plus capable d'effectuer de nouvelles allocations et que les processus déjà existants n'obtiennent le CPU que de manière sporadique. Notamment, toute nouvelle création de processus est impossible et la saturation n'a d'autre issue que la réinitialisation complète du système.

Discussion. Les ressources d'entrées/sorties ne sont pas les seules à être susceptibles à un écroulement en cas d'une utilisation intensive. Les ressources logicielles finies, limitées en nombre et souvent consommables à la demande par les utilisateurs, peuvent être asséchées facilement par un attaquant. Un assèchement se traduit par des latences importantes sur la résolution de requêtes à ces ressources et peut provoquer un déni de service complet du système, sans possibilité de récupération.

2.2.3 Monopolisation des services transparents du système

L'un des buts du système d'exploitation est de gérer les périphériques matériels et d'en présenter une vue abstraite aux processus. Quel que soit le niveau d'abstraction choisi, les systèmes offrent implicitement aux applications un ensemble de services transparents, permettant la cohabitation des différents utilisateurs ou processus et le partage des ressources. Des exemples classiques de tels services incluent l'ordonnancement CPU, qui va répartir selon une politique arbitraire le temps processeur, ou encore le service de *swap*, qui permet au système de fournir à ses applications plus de mémoire virtuelle au total que de mémoire physique disponible, en émargeant de manière temporaire les pages peu utilisées sur un périphérique de stockage annexe.

L'implémentation de ces services peut également être détournée afin de monopoliser ces ressources gérées de manière transparente par le système, comme nous le montrons dans les paragraphes suivants avec les cas de l'ordonnancement CPU et du système de swap.

Ordonnancement CPU à ticks. De manière classique, les évènements temporels dans les systèmes d'exploitation sont basés sur un seul timer matériel fixe [Bar02] initialisé au lancement du système : le *tick*. A chaque interruption de ce timer, les timers logiciels sont mis à jour et les handlers associés sont exécutés si nécessaire. L'ordonnanceur CPU n'échappe pas à cette règle : à chaque tick, il vérifie s'il est nécessaire d'effectuer un changement de contexte afin de donner la main à un autre processus. En *round-robin* tel qu'implémenté dans

Linux 2.4, chaque processus prend la main pour un nombre de ticks fixe, appelé quantum de temps, à moins qu'il ne rende la main avant ou qu'un processus de priorité supérieure ne demande le processeur. Dans l'ordonnancement moderne, des paramètres supplémentaires modulent le nombre de ticks dans un quantum, comme l'interactivité d'un processus (sa tendance à s'endormir avant d'être suspendu par l'ordonnanceur), ou la proportion d'utilisation récente du processeur par rapport aux autres processus en attente [Rob03, Aas05, RSI12].

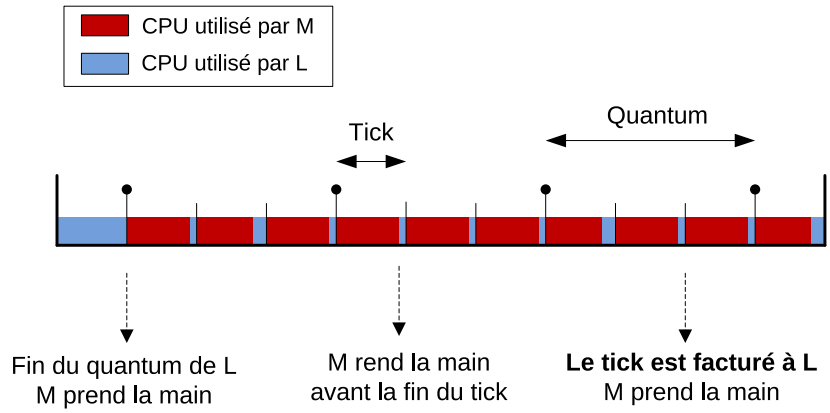


FIGURE 2.8 – Attaque par monopolisation de tick sur un ordonnanceur à base de facturation discrète.

Le problème de l'échantillonnage discret de l'utilisation du CPU a été soulevé par Tsafrir *et al.* [TEF07], qui soulignent qu'un processus peut tirer parti de cette évaluation discrète pour s'accaparer près de 100% du temps processeur sans que l'ordonnanceur ne s'en aperçoive. Leur technique s'appuie sur un calcul préalable du nombre de cycles processeur dans un tick. Ensuite, un processus exécuté autant de travail CPU que souhaité, dans la limite de ce nombre de cycles observé, puis rend la main volontairement. La figure 2.8 illustre de manière simplifiée la répartition du CPU dans un tel scénario, avec un processus malveillant, M, et un autre légitime, L. En effet, le processus M n'ayant idéalement jamais la main au moment du tick est considéré comme fortement interactif et n'est jamais facturé pour son utilisation du processeur. Il est donc favorisé par les ordonnancements ultérieurs puisque n'ayant pas eu sa part de temps processeur du point de vue de l'ordonnanceur.

Attaque par monopolisation de tick. Cette faiblesse ayant été corrigée dans l'ordonnancement Linux, nous avons implémenté cette attaque sur le système L4Re, basé sur le micro-noyau Fiasco.OC [LW09]. Ce noyau effectue par défaut un ordonnancement périodique et supporte les quantums dynamiques selon l'utilisation récente du CPU via WFQ [Zha90], un algorithme de répartition

de charge emprunté à l'ordonnancement réseau. Fiasco.OC ne fournit pas d'appel système permettant de rendre la main tout en demeurant exécutable dès le prochain tick, à l'instar de `sched_yield()` sous Unix ou `NTYieldExecution()` sous Windows NT. Nous avons cependant pu exploiter l'expiration de temps d'attente des communications inter-processus (IPC) pour obtenir un mécanisme similaire. Ceci induit un léger biais, car une IPC avec un temps d'attente nul retourne immédiatement et ne stoppe pas le processus. Il est donc nécessaire de s'endormir pour 1 microseconde au moins. La figure 2.9 retrace le résultat de cette attaque, avec deux contextes d'exécution isolés : la victime mesurant le temps nécessaire au calcul d'un membre de la suite de Fibonacci via un algorithme récursif et l'attaquant tentant de monopoliser une quantité variable de cycles CPU.

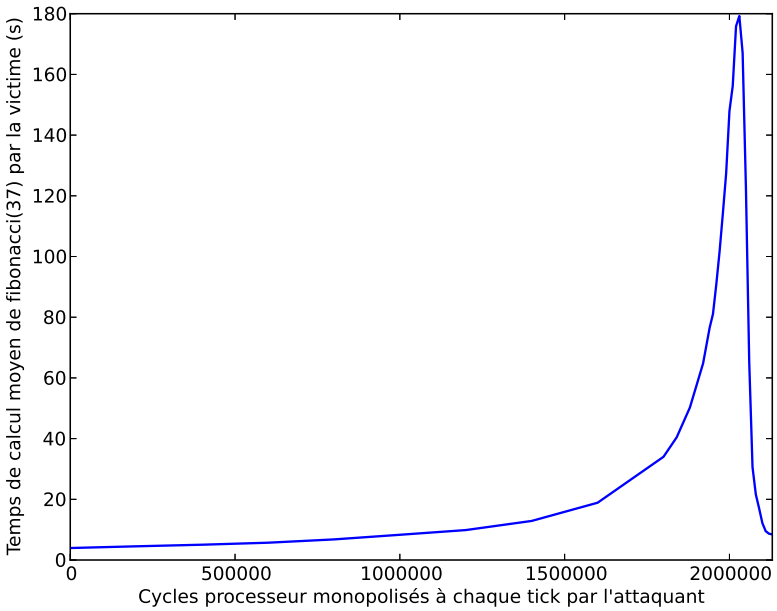


FIGURE 2.9 – Impact de la monopolisation de tick sur l'ordonnancement CPU.

Le temps de calcul du 37e membre de la suite de Fibonacci par la victime est d'à peine 4 secondes sans concurrence. Ce temps se voit doublé lorsque le processus malveillant monopolise 1 million de cycles CPU par tick avant de s'endormir. L'attaquant maximise son utilisation du tick tout en évitant la facturation en monopolisant 2 030 000 cycles. Dans ce cas, le travail de la victime est effectué en 180 secondes en moyenne, ce qui révèle une dégradation par un facteur 45. Au-delà de ces 2 030 000 cycles, l'attaquant ne peut pas éviter sys-

tématiquement la facturation et perd ainsi très rapidement la monopolisation du temps CPU.

Une telle attaque rend le système inutilisable en pratique pour les autres applications ayant le même niveau de priorité, car elles ne prennent jamais la main sur le processeur, ou alors de manière éphémère pour la fin d'un tick. Ce type de vulnérabilité se transpose à n'importe quel niveau d'ordonnancement : par exemple, l'ordonnanceur à crédit de Xen pondère les priorités des machines virtuelles grâce à un échantillonnage périodique, permettant à une machine virtuelle d'être toujours créditrice de l'ordonnanceur au détriment des autres [ZGDS11]. Certaines techniques d'ordonnancement permettent de combler cette faiblesse, par exemple en effectuant une facturation à grain fin mise à jour à chaque fois qu'un processus s'endort comme Solaris, ou encore en s'affranchissant de l'utilisation des ticks à la manière de Mac OS X [TEF07]. Linux a plus récemment introduit le Completely Fair Scheduler [Pab09] qui se sert notamment du temps d'attente de chaque processus en état prêt à l'exécution afin de répondre à cette attaque.

Mécanisme de swap. Le sous-système de swap permet au système d'exploitation de gérer plus de mémoire virtuelle que la quantité de mémoire physique (RAM) dont il dispose réellement. Lorsque le taux d'utilisation de la mémoire physique est haut, le système de swap choisit les pages mémoire les moins utilisées par les applications et les décharge de la RAM en les stockant temporairement sur un périphérique de stockage annexe, comme le disque dur. Lorsqu'une application a de nouveau besoin d'une page déchargée, le système rapatrie de manière transparente la page en mémoire centrale, émargeant si nécessaire de nouvelles pages, et permettant ainsi à l'application de continuer son exécution normalement. Pour plus de détails, nous analysons plus en profondeur les techniques de gestion de la mémoire virtuelle et d'application du mécanisme de swap dans le chapitre 5.

Cependant, comme nous l'avons signalé plus tôt, les temps d'accès aux périphériques de stockage annexes sont de plusieurs ordres de grandeur supérieurs aux accès RAM. Ainsi, sous forte contention mémoire de la part de plusieurs processus, le système peut être significativement ralenti, puisqu'effectuant un nombre important de requêtes disque, lentes, en lieu et place d'accès mémoire rapides.

Attaque par thrashing d'un système. Afin d'illustrer l'impact potentiel de la swap sur les performances d'un processus, nous utilisons un système Linux 3.5 basique, ayant 512 Mo de RAM à sa disposition. La victime calcule le *ou exclusif* bit-à-bit sur deux tableaux en mémoire de 50 Mo chacun et stocke le résultat de l'opération dans l'un des tableaux. L'attaquant alloue une zone mémoire de taille variable et effectue en boucle une opération mémoire sur chaque page, de manière séquentielle. La figure 2.10 montre les dégradations de performance subies par la victime en fonction de la taille de la zone mémoire sur laquelle l'attaquant travaille.

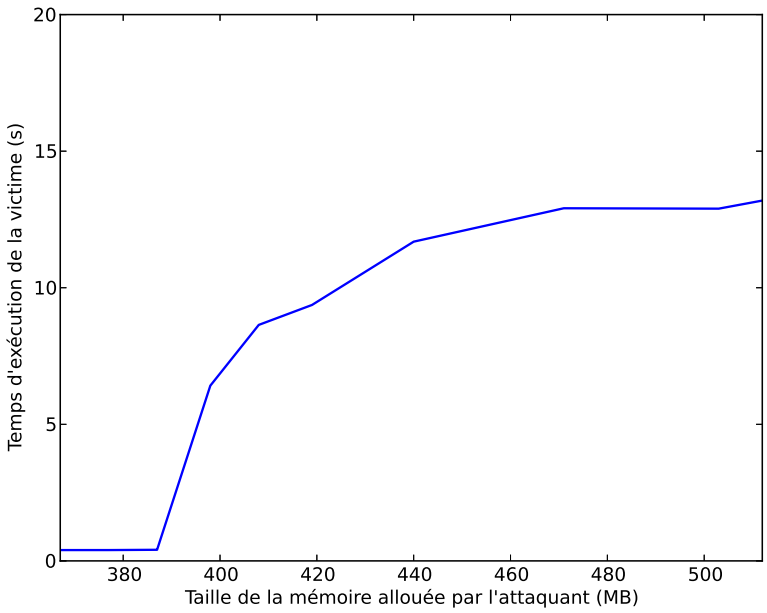


FIGURE 2.10 – Impact du système de swap sur les performances sous contention mémoire.

L'apparition du phénomène de swap est très clair sur la figure, lorsque l'attaquant alloue plus de 390 Mo de mémoire. En effet, la victime ayant des besoins dépassant les 100 Mo et le reste du système nécessitant légèrement plus de 20 Mo, la quantité totale de mémoire manipulée par le système atteint la taille de la RAM, soit 512 Mo. Cette dégradation originale réduit l'utilisation du CPU à seulement 4% pour la victime, au lieu des 50% normalement attendus et observés lorsque l'attaquant alloue moins de mémoire. La dégradation opérée par l'attaquant est plus lente pour des allocations plus fortes, la différence résidant essentiellement dans l'usage d'un espace de swap plus grand.

Le fait de ramener une page nécessaire à l'exécution en mémoire centrale se fait de manière synchrone : le processus concerné est bloqué jusqu'à ce que la page soit rapatriée via une E/S standard. De ce fait, attaquer avec plusieurs processus permet à l'attaquant d'émettre plusieurs requêtes d'E/S simultanées à tout moment et donc d'augmenter encore le temps nécessaire au rechargement des pages mémoire. La figure 2.11 rapporte les dégradations de performance subies par la même victime en fonction de la taille totale de la mémoire allouée par l'attaquant, partagée par un nombre de processus variable.

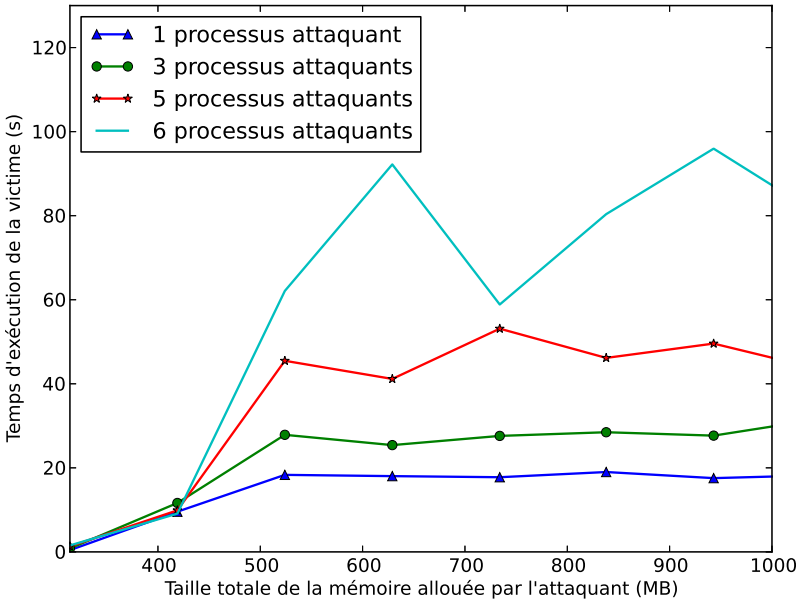


FIGURE 2.11 – Impact du nombre de processus se disputant la mémoire sur les performances du système de swap.

Comme attendu, augmenter le nombre de processus dégrade encore plus les performances de la charge de travail de la victime. Lorsque l'attaquant utilise moins de 5 processus dans notre exemple, la dégradation subie par la victime demeure proportionnelle : par exemple, les temps d'exécution à 600 Mo restent environ 40 fois supérieurs aux temps d'exécution obtenus à 300 Mo. Pour 6 processus, on peut observer à la fois une dégradation plus forte, ainsi que l'apparition d'une grande incertitude sur les durées d'exécution. Pour 600 Mo par exemple, les temps d'exécution s'étalent de 50 à 170 secondes. Ceci est dû au phénomène de *thrashing* de la mémoire virtuelle. Ce phénomène apparaît lorsque la contention mémoire est tellement forte que même les pages mémoire fréquemment utilisées par un processus sont déchargées de la mémoire centrale, entre deux prises de CPU par celui-ci. Chaque processus est donc dans un état constant de pagination et d'attente de requêtes disque, réduisant de manière dramatique l'exploitation effective du CPU. Cet état a la particularité de s'auto-alimenter, car chaque réveil d'un processus différent recharge un ensemble de pages en mémoire centrale et entraîne l'éviction des pages des autres processus qui devront être rechargées lorsque ceux-ci prendront la main. Ce phénomène devient plus constant et visible avec l'ajout de processus supplémentaires. Par exemple, les temps d'exécution de la charge de travail de la victime en concur-

rence avec 8 processus attaquants partageant 600 Mo de mémoire s'étale de 120 à 300 secondes, ce qui correspond à un usage du CPU entre 0.07 et 0.175% du temps. Sous ces conditions, le système devient difficilement utilisable, imposant des latences très importantes pour l'exécution de toute charge de travail.

Discussion. Les systèmes d'exploitation fournissent certains services transparents aux processus. C'est notamment le cas de l'ordonnancement CPU et du mécanisme de swap, gérant respectivement l'utilisation du processeur et des surplus de mémoire, via une politique inconnue des applications. Ces politiques sont généralement implémentées via un certain nombre d'algorithmes non-triviaux : l'ordonnancement CPU moderne se base par exemple sur une facturation de l'utilisation du processeur afin d'ajuster le temps alloué à chaque processus, et le mécanisme de swap adopte un remplacement de pages émargeant en priorité celles qui n'ont pas été accédées récemment. Dans cette section, nous avons montré qu'une connaissance plus ou moins fine de l'implémentation de ces services peut permettre à un attaquant de s'accaparer une ressource ou de dégrader de manière significative les performances du système.

2.3 Conclusion

Un utilisateur malveillant ou abusif dans un système partagé best-effort peut interférer de manière significative avec la performance des autres charges de travail, voire du système entier, en attaquant les ressources. Malgré la conception de systèmes d'entrées/sorties équitables ou l'optimisation des algorithmes de gestion des ressources, il est possible d'exploiter les heuristiques de réordonnancement ou les limites naturelles des files d'attente pour réduire la bande passante délivrée aux utilisateurs concourants. De plus, les entrées/sorties ne sont pas les seules ressources qui peuvent être détournées dans le but d'entraver la qualité de service d'autres applications.

Les ressources logicielles finies, comme les nombres aléatoires ou les objets noyau, sont limitées en quantité et consommées à la demande, souvent sans restriction. Cela permet aux attaquants de forcer des famines d'accès aux ressources pour tous les autres clients. Enfin, certains services rendus de manière transparente par le système afin de gérer de multiples utilisateurs concurrents, comme le mécanisme de swap ou l'ordonnancement CPU, peuvent être manipulés par des attaquants comprenant leurs mécanismes internes afin de pénaliser les autres utilisateurs.

Nous avons montré que des abus dans l'usage de ces ressources peuvent avoir un impact très significatif sur la performance de charges de travail légitimes concurrentes. Nos expériences, majoritairement effectuées sur des systèmes Linux basiques, se transposent à l'identique aux cas d'hyperviseurs ou systèmes virtualisés, qui emploient des techniques de gestion de ressources similaires. Dans ce contexte et avec la contractualisation de la qualité de service due aux clients via les SLA, il paraît nécessaire de proposer des techniques génériques permettant le partage équitable et le contrôle de ces ressources. Ces techniques

doivent cependant veiller à exploiter les ressources au maximum de leur capacité, critère essentiel de réduction des coûts pour les hébergeurs de services informatiques.

Chapitre 3

État de l'art

Parmi les points centraux dans la définition des systèmes partagés best-effort se retrouvent la performance dans le cas général et l'équité de distribution des ressources entre les différents utilisateurs. Comme illustré dans le chapitre précédent, les techniques de gestion de ressources employées par les systèmes d'exploitation classiques ne parviennent pas à appliquer ce compromis de manière satisfaisante, puisqu'un nombre conséquent de ressources peuvent être monopolisées, violant la propriété d'équité et dégradant les performances de façon combinée.

Ce chapitre présente la littérature scientifique autour des techniques et architectures de gestion de ressources dans les systèmes d'exploitation. Nous exposons tout d'abord les algorithmes de partage de ressources, hérités et dérivés essentiellement des premiers routeurs réseaux et systèmes à temps partagé. Ces algorithmes peuvent se présenter comme un compromis entre équité parfaite et optimisation de la capacité du matériel. Les algorithmes développés ont chacun leurs avantages dans les systèmes best-effort, mais souffrent toujours de limites, notamment d'implémentation, ne leur permettant pas d'offrir les mêmes propriétés sous toutes les charges de travail.

Ces faiblesses ont amené la communauté scientifique à formuler le besoin d'architectures complètes de gestion et de contrôle des ressources, dans le but d'une répartition et d'une cohabitation optimales de charges de travail potentiellement antagonistes. Un premier élément est l'ajout de couches de facturation justes de la consommation de chaque ressource en temps réel. Ces couches peuvent ensuite répartir les charges ou réguler leur usage des ressources. La répartition de charge va essayer a priori, à partir de contraintes contractuelles ou de profils de consommation préalables, de maximiser l'exploitation des ressources à la disposition de l'hébergeur tout en garantissant un niveau de qualité de service fixe au client. Ce modèle n'est cependant pas adapté au profil de déploiement à la demande des charges de travail typiques des systèmes mutualisés ou du cloud, puisqu'il sous-utilise intrinsèquement les capacités des ressources. Ainsi, un certain nombre d'architectures évoluées, se basant sur une facturation à l'exécution afin d'adapter la priorité ou le taux de requêtes de clients abusifs

ou souffrant de la concurrence, permettent d'obtenir des réponses plus proches des besoins des systèmes partagés best-effort. Ces architectures sont cependant difficilement configurables dans le cas général, et doivent être adaptées pour chaque type de ressource.

3.1 Algorithmes de partage des ressources

Afin de respecter les deux principes de maximisation des performances globales et d'équité d'accès aux ressources, les systèmes best-effort ont naturellement affaire à un compromis difficile, car ces deux principes sont potentiellement antagonistes du point de vue de la gestion des ressources. Les algorithmes d'équité sont essentiellement dérivés des travaux autour des files d'attente équitables réseau, qui ont très tôt fait face au besoin de répartition égale de charges égoïstes et multiples. Ces travaux ont ensuite été adaptés à l'ordonnancement des autres ressources d'un système, comme les requêtes d'entrées/sorties ou l'utilisation du processeur. Bien que les algorithmes théoriques offrent une équité idéale et optimale, la réalité des implémentations, notamment le besoin d'efficacité et les limites en termes d'espace mémoire, ne permet dans la pratique qu'une équité probabiliste.

A l'opposé, un nombre important de travaux ont cherché à maximiser l'utilisation du matériel. Par exemple, traiter toutes les requêtes d'entrées/sorties vers un disque dur une à une peut sous-utiliser gravement le matériel, à cause de ses particularités mécaniques et géométriques. Dans ce cas, des algorithmes ont cherché à minimiser les mouvements de la tête de disque tout en servant un maximum de requêtes. Il en va de même pour le CPU, qui doit faire face aux particularités de certaines charges de travail, comme celles effectuant beaucoup d'entrées/sorties et donc renonçant à leurs périodes de temps attribuées, ou pour les noeuds réseau qui peuvent exploiter la sémantique de leurs protocoles pour faire face aux problèmes de congestion.

Cependant, les utilisateurs abusifs peuvent se servir des cas particuliers de ces techniques pour les rendre inutiles, en exploitant les limites d'implémentation des algorithmes d'équité ou les heuristiques d'optimisation des ressources comme exposé dans la section 2.2.

3.1.1 Équité d'accès aux ressources

Avec les premiers systèmes à temps partagé sont venus les deux manières naturelles de partager le processeur et les entrées/sorties : le *round-robin* et les files d'attente FCFS, *First Come First Serve*. Le round-robin, utilisé initialement pour le partage du temps processeur, maintient une liste cyclique des concurrents d'une ressource, et la parcourt de manière séquentielle, servant une requête de chacun, ou offrant un service exclusif pour un quantum de temps donné, tour à tour. Pour le réseau, c'est une approche FCFS qui a d'abord été retenue, afin de réduire les tâches algorithmiques et donc de maximiser l'efficacité des noeuds réseau. Ces solutions ne sont cependant pas équitables, puisque

FCFS va offrir une bande passante directement liée à la fréquence des requêtes de chaque client, et que le round-robin va offrir plus de temps aux utilisateurs intensifs qu'à ceux qui rendent la main avant la fin de leur quantum.

Files d'attente équitables pour les noeuds réseau. La première expression du besoin d'algorithmes d'équité des systèmes à base de files d'attente vient de Nagle [Nag87]. Nagle a au préalable montré que le modèle FCFS utilisé dans les routeurs pouvait causer une grande disparité dans l'attribution de la bande passante aux différentes sources, en fonction notamment de la taille et de la fréquence d'arrivée des paquets, dépendant elle-même de l'implémentation et de l'efficacité de la pile réseau de la source [Nag84]. Nagle a ainsi proposé d'adapter le round-robin naturel du CPU à la gestion des flux réseaux par les routeurs : chaque flux, identifié par le couple source/destination, utilise ainsi une file d'attente distincte de taille fixe, et le routeur transmet un paquet de chaque flux tour à tour.

Cette première solution a une faiblesse essentielle : la probable disparité dans la taille des paquets. En effet, pour deux sources concurrentes $N1$ et $N2$ envoyant respectivement des paquets de taille moyenne $S1$ et $S2$, la bande passante attribuée à $N1$ sera $\frac{S1}{S2}$ celle de $N2$, ce rapport n'étant borné que par celui entre les tailles extrêmes $SMax/SMin$. Cette solution a cependant permis la naissance des files d'attente équitables, ou *fair queuing* (FQ), proposées par Demers *et al.* [DKS89]. Afin de résoudre la faiblesse de l'algorithme de Nagle, Demers *et al.* proposent un algorithme théorique idéal qui fait un round-robin bit-à-bit et non paquet par paquet. De cette manière, chaque flux reçoit une bande passante équitable. Cet algorithme n'est cependant pas implémentable en pratique, et les auteurs proposent une approximation pratique par simulation : le routeur calcule, en fonction de la taille des paquets en attente, la date à laquelle chaque paquet aurait été transmis si l'algorithme théorique était exécuté, et utilise cette information afin d'insérer les paquets dans une file d'attente ordonnée par date de transmission estimée. Demers *et al.* soulignent également la possibilité d'ajouter un ratio arbitraire à chaque flux, lorsque certains noeuds ont besoin d'une bande passante plus grande, permettant ainsi une équité pondérée, nommée *Weighted Fair Queuing* (WFQ).

FQ a cependant une complexité algorithmique non satisfaisante pour des noeuds comme les routeurs basés sur l'efficacité, à cause notamment du nombre de flux potentiellement infini. Diverses extensions ont ainsi été proposées afin d'approximer l'algorithme théorique idéal tout en ayant de meilleures propriétés d'implémentation. Les files d'attente équitables stochastiques (SFQ) [McK90] maintiennent un nombre de files restreint, et chaque flux est assigné à une file par hashage, d'où une complexité d'accès en $\mathcal{O}(1)$. L'équité des SFQ n'est donc pas garantie, mais se base sur la faible probabilité de plusieurs flux distincts à se retrouver simultanément dans la même file d'attente.

Shreedhar et Varghese ont également proposé une technique avec une complexité algorithmique en $\mathcal{O}(1)$, le Deficit Round Robin [SV95]. Leur technique se base sur les files d'attente stochastiques, mais ajoute au round-robin em-

ployé pour servir les différentes files la notion de déficit : lorsqu'une file n'a pas consommé toute sa bande passante allouée à un certain tour du round-robin car le prochain paquet à envoyer était trop gros, elle se voit attribuer un déficit, égal à la bande passante non consommée, qu'elle pourra utiliser dans ses prochains tours. Le Deficit Round Robin introduit ainsi notamment l'idée d'un ordonnancement basé sur les évènements passés, afin de viser une équité globale sur la durée, et non pas sur un seul tour du round-robin.

Transposition aux autres ressources. Ces algorithmes issus de la théorie des files d'attente se sont ensuite transposés naturellement à la gestion des ressources partagées dans un système. Linux implémente depuis 2004 et sa version 2.6.10 le *Completely Fair Queuing* (CFQ) pour la gestion des requêtes sur les périphériques à blocs comme les disques durs [Axb04]. CFQ est une adaptation directe de SFQ à la gestion des requêtes disque, avec un nombre de files fini (64 par défaut), ordonnancées en round-robin. Les requêtes de chaque processus sont insérées dans une file en fonction d'un hashage de leur PID.

Ces algorithmes ont aussi été adaptés à l'ordonnancement CPU. Le micro-noyau Fiasco.OC [LW09] permet par exemple l'utilisation de WFQ comme algorithme équitable permettant un partage pondéré du temps processeur. Linux implémente le *Completely Fair Scheduling* (CFS) pour son ordonnancement CPU [Pab09]. CFS se base sur un temps virtuel, commun à tous les processus du système, et ordonnance les processus en privilégiant ceux ayant le plus attendus le processeur par le passé. Contrairement au round-robin qui effectue ses décisions d'ordonnancement de manière périodique, CFS a besoin de maintenir une facturation à grain fin du temps virtuel consommé par chaque processus. Il doit notamment prendre en compte tous les évènements où un processus rend la main, que ce soit de manière explicite ou lors d'entrées/sorties par exemple. CFS ordonne la file d'attente des processus en fonction du temps pendant lequel ils auraient dû avoir le CPU avec un ordonnancement FQ idéal, permettant de choisir le processus qui a eu le moins de CPU dans les tours passés en $\mathcal{O}(1)$. CFS implique une très bonne réactivité pour les processus interactifs et utilisateurs d'entrées/sorties, faiblesse principale du round-robin. De la même manière, l'ordonnanceur à crédit de Xen garde en mémoire les déficits de temps de chaque machine virtuelle qui rend la main avant la fin de son quota de temps, afin de lui autoriser une consommation plus importante du CPU dans les tours futurs [CGV07]. Ces deux algorithmes sont donc des applications directes de FQ et du Deficit Round Robin au partage équitable du temps processeur. L'ordonnanceur à crédit de Xen maintient cependant une facturation qui n'évalue l'utilisation du CPU que de manière périodique, et non pas à grain fin comme CFS. Cet échantillonnage discret peut ainsi être exploité par un attaquant pour fausser la facturation et monopoliser le processeur [ZGDS11].

Répartition dynamique équitable. D'autres algorithmes efficaces adoptent une approche différente pour le partage équitable, notamment les ordonnancements loterie et pas à pas de Waldspurger [WW94]. L'ordonnancement loterie

se base sur une répartition statistique afin de garantir l'équité en nombre de requêtes traitées sur une période de temps donné. L'ordonnanceur distribue à chaque client potentiel un nombre de tickets égal au poids arbitraire attribué à chacun. Il sélectionne ensuite en boucle et de manière aléatoire un ticket gagnant, puis traite une requête du client concerné. Cette technique simple permet de respecter naturellement la proportion de traitement des requêtes pour chaque client en compétition et de manière efficace. La répartition initiale des tickets et la période de renouvellement sont cependant difficiles à paramétrer de manière générique.

Le deuxième algorithme est l'allocation "pas à pas", où le système estime l'intervalle de temps virtuel, le *pas*, qu'un client doit attendre entre deux allocations d'une ressource. Ce pas peut être finalement vu comme un poids d'utilisation pour chaque client, où le poids le plus faible est le plus prioritaire. Pour chaque client est maintenue une valeur d'utilisation, la *passe*, initialisée à son pas. A chaque allocation, le client avec la plus petite passe est servi, et sa passe est incrémentée par son pas. Cet algorithme, pouvant être considéré comme une amélioration algorithmique de WFQ, propose ainsi une allocation proportionnelle aux pas de chaque client, tout en gérant de manière dynamique le problème de priorité. Ces deux algorithmes partagent cependant la faiblesse de ne s'intéresser qu'à l'équité sur le nombre de requêtes, qui ne saurait être suffisante puisque indépendante de la taille des paquets ou des particularités de traitement de certaines requêtes par les périphériques.

3.1.2 Optimisation de l'utilisation des ressources

Le but de ces algorithmes est un partage équitable sur le temps, mais pas nécessairement une utilisation maximisée de la ressource. En effet, les particularités de chaque périphérique doivent aussi être prises en compte afin de ne pas sous-utiliser la ressource. Il est par exemple nécessaire de prendre en charge de l'interactivité des processus pour le CPU, la minimisation des mouvements de la tête de lecture pour les disques durs mécaniques, ou la congestion des autres noeuds pour les interfaces réseau. L'optimisation de l'utilisation de la mémoire et les problèmes inhérents à son partage sont débattus plus en profondeur dans le chapitre 5.

Intégration des profils applicatifs dans l'ordonnancement CPU. L'ordonnancement classique souffre de la difficulté de concilier les applications ayant des profils d'utilisation du processeur très différents : les applications interactives effectuent généralement peu de calculs mais ont besoin d'une forte réactivité, les applications de calcul ont des besoins importants en temps CPU cumulé, et les applications temps réel ont besoin de garanties sur leurs temps d'exécution [SCG+00].

Afin de répondre à ces spécificités de l'ordonnancement, les systèmes d'exploitation cherchent à isoler les différents profils d'applications afin de leur assigner le processeur de manière optimale. Par exemple, les processus Linux obtiennent des priorités dynamiques, attribuées et évoluant différemment selon

leur profil détecté [BC05]. Un processus gagne ainsi des bonus de priorité s'il s'endort fréquemment, ce qui lui garantit une exécution rapide après son réveil. Avant l'introduction de CFS, Linux maintenait aussi deux files d'attente : les processus ayant rendu la main avant la fin de leur dernier quantum de temps, les processus *actifs*, et ceux ayant utilisé l'intégralité de leur quantum, les processus *expirés*. Ceci permettait de prioriser les processus actifs, tout en veillant à ce que l'attente des processus expirés ne passe pas au-dessus d'une limite arbitraire. Les processus temps réel bénéficient eux d'un ordonnancement séparé basé sur les priorités et un round-robin strict, préemptif sur les autres files d'attente afin de leur garantir les meilleurs temps d'exécution possible.

Windows prévoit également un ensemble de bonus de priorité destinés à améliorer la réactivité des applications interactives [RSI12]. Les threads se voient attribuer des bonus différents lors de la fin d'opérations d'entrées/sorties, d'entrées destinées à une interface graphique, de l'arrivée d'évènements système comme l'expiration d'un timer ou la libération d'un lock, ou encore lors d'attentes trop longues sur une ressource exécutive. Lorsque certains seuils arbitraires sur le temps d'attente du CPU sont dépassés, les threads concernés voient également leur priorité augmentée, afin d'éviter une famine d'accès au processeur. Ces mécanismes ad hoc, configurés de manière empirique, permettent d'offrir une réactivité améliorée aux cas communs d'utilisation de Windows, notamment l'usage des interfaces graphiques ou l'attente dans les zones critiques des librairies système communes.

Ces cas particuliers ont été appliqués de manière similaire aux systèmes virtualisés où les machines virtuelles gagnent des bonus de priorité lorsqu'elles reçoivent des entrées/sorties par exemple. La transposition directe est cependant coûteuse en performance car imposant un nombre important de changements de contexte d'exécution [OCR08].

L'ordonnancement à base de priorités pose cependant des problèmes supplémentaires, par exemple lorsqu'une application à forte priorité communique avec une application de faible priorité. Plutôt que de forcer une hausse de priorité en cascade des dépendances d'une application à forte priorité, Stoess [Sto07] propose ainsi un ordonnancement hiérarchique, coopératif, entièrement en espace utilisateur. Ceci permet à chaque noeud de l'arbre d'ordonnancement de prendre les meilleures décisions, connaissant les besoins des applications en utilisant leurs retours. Ceci permet notamment à une application de forte priorité de laisser explicitement la main à une application de plus faible priorité. Cet ordonnancement, implémenté au-dessus de Fiasco.OC, se base sur la transmission hiérarchique d'IPC de préemption et de notification d'expiration des quantum de temps, et donc une coopération complète des différents composants. Le coût supplémentaire en temps d'exécution dû à la multiplication des IPC de contrôle de l'ordonnancement et des changements de contexte demeure cependant significatif.

Gestion coopérative de la congestion réseau. Bien que les algorithmes dérivés des files d'attente équitables soient implémentables dans les piles ré-

seau des systèmes, la complexité a traditionnellement été poussée dans des noeuds réseau dédiés, les routeurs, afin de permettre une gestion efficace et équitable des flux réseau, quels que soient l'efficacité et le profil des sources. Par exemple, bien que Linux ou Windows soient capables d'utiliser SFQ ou d'autres techniques d'équité en sortie de la pile réseau, leurs configurations communes emploient de simples piles FIFO [BC05, RSI12].

La communauté a cependant notamment choisi d'utiliser les spécificités des protocoles afin de contrôler les besoins de QoS et la qualité des liens réseau. Le mécanisme de fenêtre de réception TCP, contenue dans les en-têtes du protocole, permet de contrôler la taille des paquets envoyés par la source, afin de ne pas surcharger le destinataire et le nombre d'accusés de réception [MMFR96]. La fenêtre est initialisée à une valeur basse et augmentée rapidement par la source pendant la transmission des premiers paquets d'une connexion. En cas de congestion, le destinataire peut ainsi réduire la taille de la fenêtre pour demander explicitement à la source de limiter la taille des paquets futurs. La fenêtre de congestion TCP offre un mécanisme similaire pour gérer la congestion sur le parcours entre la source et la destination : la source étudie notamment les durées de transmission des paquets afin de mettre à jour la fenêtre de congestion, des durées trop importantes sans modification de la fenêtre de réception indiquant une congestion sur le réseau [APB09]. Dans la pratique, la plus petite des deux fenêtres de réception et de congestion doit être utilisée par la source pour une transmission optimale.

Certains fournisseurs d'accès cloud permettent seulement l'usage de protocoles proches de TCP, afin de pouvoir utiliser ces mécanismes de congestion de manière efficace et sans interférences, comme Windows Azure [GB13] ou Google App Engine [Goo13].

Un mécanisme de contrôle de congestion a également été ajouté au protocole IP : la notification explicite de congestion (ECN) [RFB01]. Ce mécanisme est implémenté par les noeuds du réseau et non par les sources ou destinataires : lors de l'apparition de congestion, plutôt que d'ignorer les paquets en surplus, les noeuds réseau les transmettent en modifiant deux bits du champ de différentiation de services (DSCP) de l'entête IP. Cela permet à une source coopérative de diminuer sa fréquence d'envoi de paquets, sous peine de voir plus tard ses paquets ignorés par les noeuds surchargés. Les autres bits du champ DCSP permettent de définir une classe de service, taux de QoS souhaité, pour un paquet [Sys08]. Ceci permet aux noeuds réseau compatibles de prioriser les paquets selon leur classe, puis d'utiliser les protocoles d'équité commun pour l'ordonnancement interne à une classe.

Ces mécanismes intégrés aux protocoles réseau dépendent cependant de l'implémentation des piles réseau des différents acteurs. L'usage de piles réseau personnalisées ou l'injection de paquets peut permettre d'abuser ces mécanismes, en utilisant par exemple la notification explicite de congestion pour forcer une source bienveillante à diminuer fortement sa fréquence d'envoi et donc sa bande passante [KK03, GBM04].

Ceci pose notamment problème dans les datacenters du cloud, où les attaquants peuvent instancier des machines virtuelles à pile réseau modifiée, ayant

un accès très haut débit permettant d'attaquer ou de noyer les autres utilisateurs [Bit09]. Seawall est une proposition permettant le contrôle de congestion dans ces environnements fortement virtualisés [SKGK10]. Seawall effectue une supervision par les hyperviseurs, jugés de confiance, de la fréquence des paquets envoyés par les systèmes invités. Seawall se base sur les mécanismes classiques de gestion de congestion comme ECN ou les fenêtres TCP, et force les systèmes invités à réduire leur fréquence d'envoi lors de l'apparition de congestion, en limitant le débit des interfaces réseau virtuelles associées. L'ajout d'une coopération entre les différents hyperviseurs permet également de détecter les noeuds potentiellement compromis, qui ne répondent pas correctement aux demandes de réduction du trafic.

Optimisation de la bande passante des disques durs. La nature mécanique des disques durs apporte une spécificité très forte à ces périphériques de stockage secondaires. Puisque les durées des mouvements mécaniques sont incompressibles et usent le périphérique, de nombreux travaux proposent des algorithmes optimisés, dédiés à la gestion des requêtes disque.

Ces algorithmes sont essentiellement dérivés des deux algorithmes pionniers de minimisation des mouvements rotationnels, SSTF et SCAN. SSTF, pour *Shortest Seek Time First*, emploie une heuristique simple consistant à minimiser le mouvement de la tête de disque à tout moment. Il sélectionne ainsi toujours la requête qui se fait sur le bloc physique le plus proche du bloc courant. Cet algorithme présente cependant un risque de famine élevé, puisque des accès intensifs localisés physiquement vont être servis en priorité, ne permettant pas à des requêtes sur des blocs plus lointains d'être exécutées. Afin d'améliorer l'équité tout en minimisant les mouvements mécaniques, Denning propose SCAN [Den67]. L'algorithme SCAN parcourt intégralement le disque dans un sens puis dans l'autre, servant les requêtes séquentiellement du point de vue de la position physique des blocs. SCAN borne ainsi la durée de résolution d'une requête au double de la durée de parcours séquentiel du disque entier. Il ne permet cependant pas un accès équitable à la bande passante disque, puisque un accès séquentiel intensif augmente de manière significatif le temps nécessaire à l'accès d'autres blocs plus éloignés.

Ces bornes sur le temps d'exécution d'une requête permettent cependant l'intégration de SCAN dans l'optimisation des requêtes ayant des contraintes temps réel. Molano *et al.* proposent ainsi un algorithme "juste à temps" de service des requêtes disque, basé sur SCAN et EDF [MJR97]. Leur algorithme sépare les requêtes d'entrées/sorties en requêtes unitaires contiguës physiquement, et les ordonnance en EDF, c'est-à-dire selon leur priorité temps réel. Lorsque les contraintes temps réel le permettent, les requêtes de priorité moins hautes sur les blocs physiquement proches sont servies à la manière de SCAN. Ceci leur permet de respecter les contraintes temps réel des requêtes tout en optimisant le parcours du disque.

Wu et Brandt proposent une couche de contrôle d'accès des requêtes disque dans les systèmes où cohabitent des applications à temps réel mou et des ap-

plications best-effort [WB04]. Leur couche d'accès gère deux files d'attente distinctes pour les requêtes disque des deux classes d'application, en effectuant une optimisation classique des requêtes disque selon la localité. En fonction de l'obtention ou non des garanties de temps des applications temps réel, la couche d'accès adapte dynamiquement le ratio de requêtes best-effort autorisées. Après une diminution de ce ratio, celui-ci augmente de nouveau avec le temps, si les contraintes de temps des applications temps réel sont satisfaites. Cette technique permet une exploitation maximisée du disque lorsqu'il n'y a pas de contention, même pour les requêtes best-effort, tout en améliorant la qualité de service des requêtes temps réel lors de l'apparition de contention.

Ces techniques ne sont cependant pas applicables dans les systèmes partagés best-effort où les requêtes ne sont pas différentiables en termes de priorité, et les algorithmes dérivés de SCAN sont trop enclins aux possibilités de famine. L'ordonnancement anticipatoire proposé par Iyer et Druschel [ID01] offre une réponse élégante à ce problème. L'ordonnancement anticipatoire diffère brièvement la transmission de nouvelles requêtes disque, afin de laisser la possibilité à de nouvelles requêtes d'être prises en compte et agrégées lorsqu'elles sont proches physiquement. En forçant l'attente des requêtes, l'ordonnancement anticipatoire utilise l'heuristique intuitive voulant que les applications effectuent communément plusieurs accès physiquement proches sur une courte période de temps. Le Completely Fair Queuing (CFQ) implémente ainsi l'ordonnancement anticipatoire comme technique de performance spécifique aux périphériques à blocs, et SFQ comme technique de file d'attente offrant une équité d'accès au medium [BC05]. Paradoxalement, forcer les requêtes à attendre un délai minimum avant transmission aux périphériques permet donc une amélioration de performances significative dans le cas général.

Les disques constituant un goulot d'étranglement naturel de la performance des systèmes, les vendeurs de matériel ont amélioré les algorithmes embarqués afin que les contrôleurs de disques puissent recevoir de multiples requêtes en parallèle et les ordonnancer eux-mêmes. Ainsi, les spécifications du protocole SATA 2 prévoient l'utilisation du *Native Command Queuing* (NCQ) pour réordonnancer les requêtes au sein des disques [HC03]. Le réordonnancement sur le disque permet notamment la prise en compte de caractéristiques spécifiques comme les vitesses rotationnelle et angulaire et les positions géométriques relatives de chaque bloc. Ces grandeurs sont inconnues du système d'exploitation, qui ne peut qu'en faire des estimations à partir des identifiants de blocs ou d'étalonnages empiriques. Le problème posé par NCQ est sa superposition avec les couches d'ordonnancement des requêtes des systèmes d'exploitation, comme pointé par Yu *et al.* [YSEY10]. Les auteurs observent ainsi une contre-productivité des deux couches sous certaines charges de travail, mais également des apports significatifs de chaque technique : par exemple, les accès aléatoires sont gérés de manière optimale par NCQ, tandis que l'ordonnancement anticipatoire est le meilleur pour les accès séquentiels. Ils proposent ainsi l'intégration d'un module d'ordonnancement chargé de désactiver l'ordonnanceur logiciel ou NCQ selon les charges de travail détectées. Leur module se base essentiellement

sur l'évolution de la taille des requêtes disque afin de déterminer si les charges de travail sont à tendance séquentielle ou non. Au-delà d'un certain seuil maximum, la charge est déclarée séquentielle et NCQ est paramétré pour n'agir qu'en FIFO. En dessous d'un seuil minimum, la charge de travail est déclarée aléatoire et l'ordonnanceur système agit comme une FIFO. Entre ces deux seuils, les deux ordonnanceurs sont utilisés en parallèle. Bien que NCQ borne le temps d'attente maximum d'une requête à 2 secondes, les auteurs fournissent une technique logicielle permettant de détecter les requêtes qui sont victimes des stratégies d'ordonnancement du matériel. Pour chaque requête transmise au disque, un âge lui est associé. Cet âge est incrémenté lors de chaque notification de fin de requête par le matériel, suivant ainsi le nombre de fois où une requête a été doublée. Avant de transmettre de nouvelles requêtes au matériel, le système peut ainsi vérifier s'il existe des requêtes actives ayant un âge supérieur à un seuil arbitraire, et bloquer la transmission de nouvelles requêtes tant que la famine perdure le cas échéant. Leur module permet ainsi de bénéficier des avantages des ordonnancements matériel et logiciel tout en éliminant le potentiel antagoniste sur certaines charges de travail et en réduisant la durée de famine des requêtes. Il se base cependant sur un nombre important de seuils arbitraires, définis de manière empirique, ce qui implique une configuration délicate dans un cas général.

Stanovich *et al.* ont également montré que la seule protection contre la famine de NCQ, l'attente maximale de 2 secondes, s'applique très mal aux applications temps réel mou cohabitant avec des applications best-effort, et augmente de manière significative le pire cas d'utilisation, comparativement à un disque sans NCQ [SBW08]. Afin d'offrir de meilleures garanties en termes de temps de service aux requêtes temps réel, les auteurs limitent le nombre de requêtes concurrentes transmises au disque, voire bloquent totalement la transmission de nouvelles requêtes lorsque la contrainte de temps d'une requête transmise au disque est en danger. Cette technique leur permet ainsi de borner l'impact de NCQ sur le temps de service des requêtes, correspondant à la durée maximale de traitement des requêtes actives.

Boutcher et Chandra ont récemment étudié l'impact de l'ordonnancement des disques dans les environnements virtualisés [BC10]. En effet, les environnements de virtualisation imposent une nouvelle couche d'ordonnancement des requêtes entre les systèmes invités et le matériel, qui peut être contre-productive avec les deux couches d'ordonnancement déjà existantes. Leurs évaluations montrent que l'apport de l'ordonnancement embarqué dans les systèmes invités reste inchangé dans un environnement virtuel : par exemple, l'ordonnancement anticipatoire pour les charges de travail séquentielles demeure le plus efficace quel que soit le type d'ordonnancement effectué par l'hyperviseur. En termes de performance, l'emploi d'un algorithme minimal comme FIFO par l'hyperviseur permet une allocation maximisée de bande passante dans le cas général. Les files d'attente équitables comme CFQ permettent un partage équitable de la bande passante entre les différents systèmes invités, mais réduisent la bande passante et donc la latence moyenne des accès disque.

3.1.3 Discussion

La gestion des ressources matérielles dans les systèmes d'exploitation best-effort a toujours dû faire face à un compromis entre l'équité du partage de la ressource et l'optimisation de l'utilisation du périphérique. La formulation des algorithmes de files d'attente équitable a permis une amélioration significative dans l'implémentation de politiques d'équité d'accès aux ressources partagées. Des implémentations efficaces comme SFQ ont pu en être dérivées et appliquées aux différentes ressources, par exemple à l'orchestration des requêtes disque avec CFQ ou à l'ordonnancement CPU avec CFS. D'un autre côté, des techniques dédiées ont été mises en place afin d'optimiser l'exploitation des différentes ressources. Les ordonnanceurs CPU prennent notamment en compte les profils des applications, comme leur interactivité et ont ajusté les techniques de facturation afin de pouvoir attribuer justement le processeur aux processus qui le reçoivent le moins. Les piles réseau s'appuient sur les protocoles afin d'implémenter des politiques distribuées et coopératives de gestion de ressources, particulièrement efficaces dans le cadre d'environnements massivement virtualisés, où l'hyperviseur de confiance peut forcer les systèmes invités à respecter ces politiques. Enfin, de nombreux travaux ont visé à optimiser la gestion des disques durs, depuis le matériel ou le logiciel, en minimisant les mouvements des têtes de disque, tout en essayant de conserver une équité dans la distribution de la bande passante.

Dans les deux cas, l'implémentation de ces politiques passe par des biais qui mettent en péril leur efficacité et leur équité. Par exemple, le seul algorithme efficace qui a pu être dérivé de la théorie des files d'attente équitables demeure SFQ, comportant un nombre limité de files d'attente ordonnancées en round-robin. Seulement, les limites pratiques, aussi bien du nombre de files que du nombre de requêtes pouvant être insérées dans chacune, vont permettre à un utilisateur abusif d'écrouler le système d'entrées/sorties, comme démontré dans la section 2.2.1, ou d'exploiter les heuristiques de placement des processus dans les différentes files afin de surcharger celle d'un processus victime et de diviser naturellement son accès au périphérique. De la même façon, la connaissance des heuristiques d'optimisation dédiées peut favoriser un attaquant qui force ses requêtes à suivre la fréquence ou la localité préférée par la couche d'ordonnancement logicielle ou matérielle.

3.2 Facturation de l'utilisation des ressources

La facturation de l'utilisation des ressources est un élément essentiel de toute architecture de gestion de ressources. En effet, que ce soit pour offrir à l'hébergeur une vision globale de l'état du système, ou pour permettre une régulation plus fine de la consommation des ressources dans les systèmes partagés, un suivi en temps réel paraît incontournable. Cependant, la mise en place d'une facturation de l'utilisation des ressources dépend notamment du choix des métriques de facturation, ainsi que des unités de consommation de ressources, qui peuvent varier d'un cadre applicatif à un autre. Les travaux passés ont notamment mis

en avant la besoin de définitions propres d'unités de facturation distinctes de l'activité d'exécution, ainsi que la difficulté dans un système partagé d'effectuer une facturation juste, notamment du temps d'utilisation du processeur.

Dans un contexte d'utilisation contractualisée d'un système, les couches de facturation sont centrales dans la vérification du respect des SLA, accords préalables entre utilisateurs et hébergeurs sur la qualité de service due.

3.2.1 Unité de consommation des ressources

Historiquement, les deux ressources principalement concurrentielles dans les systèmes informatisés ont tout d'abord été le processeur et l'espace mémoire, ces systèmes étant en premier lieu utilisés à des fins de calculs complexes. Cet héritage historique s'est naturellement transposé dans les systèmes partagés avec l'utilisation d'une unité d'activité unique, le processus ou *thread*, représentant in fine l'unité d'exécution et d'utilisation du processeur. Le partage du processeur entre ces unités d'exécution en *round-robin* consiste à donner la main sur le processeur à chacun à tour de rôle et pour une période de temps définie, le quantum. L'intuition veut qu'un tel ordonnancement induise une gestion globalement efficace et équitable du système et de ses ressources, puisque chaque processus a la même période de temps pour effectuer ses actions. Cependant, les années 1990 ont vu l'émergence des applications dynamiques notamment, comme la gestion de flux multimédia ou les serveurs Web, ayant des besoins changeants et parfois importants en ressources. La réalité de l'exécution des ces applications impose une utilisation non maximisée des quantum de temps, et une disparité des requêtes aux ressources, que ce soit en taille ou en fréquence.

Séparation des unités d'exécution et de consommation des ressources. Mach 3 [Loe92], l'un des premiers systèmes à micro-noyaux à tendance minimaliste, prône la séparation entre les *threads*, unité d'exécution, regroupés en *tasks*, unité de facturation et de consommation des ressources, eux-mêmes regroupés en processeurs virtuels utilisés comme unité d'ordonnancement CPU. Les périphériques matériels sont accessibles via une couche unifiée, les *ports*, et chaque task a un ensemble de *capabilities* l'autorisant à recevoir et/ou à émettre sur un certain nombre de ces ports. Ces capabilities sont distribuées et maintenues par un périphérique virtuel maître, permettant un contrôle transparent et générique, mais à grain fin, des droits d'accès aux ressources matérielles. Bien que la couche de contrôle d'accès ne permette qu'une administration booléenne de l'accès aux ports, Mach 3 fait un premier pas en liant conceptuellement l'utilisation des ressources par une entité abstraite groupant un nombre arbitraire d'unités d'exécution.

Les premières définitions approfondies d'unités de consommation de ressources sont les *resource containers* définis par Banga *et al.* [BDM99], ainsi que les unités de performance logicielles (SPU) de Verghese *et al.* [VGR98]. Le travail de Banga *et al.* part du constat de l'émergence d'une classe d'applications qui n'est alors pas gérée de manière optimale par les systèmes existants : les serveurs Web. Ils pointent notamment l'inadéquation du modèle classique de

gestion des ressources utilisant une unité d'exécution et de facturation unique, le processus. En effet, l'impossibilité de définir des groupes de threads ou de processus, représentant une consommation en ressources ayant un but commun, ne permet pas de répondre aux réalités des serveurs Web et d'optimiser l'ordonnancement des requêtes. Leurs resource containers regroupent toutes les ressources d'une activité précise : par exemple, le temps CPU, les sockets et buffer réseaux utilisés par un groupe de threads pour servir un même site Web. Un container est paramétré avec des valeurs souhaitées de qualité de service, par exemple la bande passante réseau, ainsi que des quotas de consommation arbitraires. Les entités d'exécution, processus ou threads, peuvent être rattachées à un container de manière dynamique et initiée par l'espace utilisateur. Les resource containers sont hiérarchiques, c'est-à-dire qu'un processus fils s'attachant à un nouveau resource container, ne pourra obtenir qu'une fraction des ressources allouées au resource container du parent. La nécessité de hiérarchisation des mécanismes de facturation et de contrôle a notamment été énoncée par Liedtke *et al.* [LIJ97], puisqu'elle permet d'endiguer un comportement excessif aux processus contenus dans le sous-arbre de facturation concerné. En utilisant cette couche de contrôle de ressources, Banga *et al.* ont montré leur capacité à contenir des attaques SYN Flood, ainsi qu'à améliorer le partage de bande passante dans un serveur HTTP mutualisé.

L'inadéquation du modèle unique de facturation et d'exécution est aussi pointée par Verghese *et al.*, qui s'appuient sur l'émergence des systèmes à plusieurs processeurs partagés. En effet le partage des caches et autres ressources matérielles par ces coeurs exécutant des charges de travail de manière simultanée apporte naturellement de nouveaux facteurs d'interférences. Ils ajoutent également que les alternatives classiques que sont la définition de quotas ou de garanties formelles et calculées a priori sur l'utilisation des ressources ne peuvent convenir aux applications dynamiques dont les besoins sont changeants et non prédictibles. Afin d'offrir un modèle générique d'abstraction et de contrôle des performances, ils ajoutent une abstraction au niveau noyau : l'unité de performance logicielle (SPU). La SPU regroupe un ensemble de processus auxquels est attribuée une quotité arbitraire pour chaque ressource. Le système ordonnance ensuite l'utilisation de ressources entre chaque SPU. Chaque SPU est capable de prêter des ressources dont il ne se sert pas de manière temporaire aux autres SPU, mais n'y est pas obligé. Leur définition des SPU propose ainsi une isolation de performance prédéfinie, tout en offrant la possibilité d'un équilibrage dynamique et coopératif des ressources. Verghese *et al.* mettent par ailleurs l'accent sur la nécessité de deux mécanismes incontournables dans l'optique d'une bonne isolation de performance : une facturation juste et un contrôle à l'exécution du respect de la part attribuée à chaque SPU.

Intégration tardive dans les systèmes classiques. Cependant, la transposition de ces principes dans les systèmes d'exploitation classiques comme Linux n'a été effectuée que 10 ans plus tard. Leur introduction a notamment été initiée par l'apparition des serveurs privés virtuels, ayant besoin de mé-

canismes efficaces des noyaux pour isoler les charges de travail invitées. Par exemple, les *cgroups* ou groupes de contrôle, unité de consommation de ressources de Linux [Men08], n'ont été ajoutés au noyau que fin 2007 dans sa version 2.6.24. Les cgroups permettent aujourd'hui de facturer l'utilisation des ressources, notamment le temps CPU, l'espace mémoire ou la bande passante des périphériques d'entrées/sorties, à des groupement arbitraires de processus. Chaque type de ressource est géré par un contrôleur dédié, ainsi *cpuacct* permet la facturation de l'utilisation des processeurs pour les différents groupes, tandis que le contrôleur *blkio* va gérer les périphériques de type bloc comme les disques durs. L'intégration de cette unité de facturation n'est cependant pas intégrée de manière systématique à l'ordonnancement des requêtes : par exemple, le système d'entrées/sorties maintient des files d'attente liées aux processus, indépendamment de la définition des cgroups. Le gestionnaire de ressources de Windows introduit avec Windows Server 2008 ne permet quant à lui qu'une gestion de la quantité de mémoire et de l'allocation CPU à des entités plus abstraites que l'unité d'exécution [RSI12].

Discussion. La séparation de l'unité de consommation de l'unité d'exécution est un pas essentiel dans la mise en place d'une facturation adaptée : en effet, l'unité d'exécution représente la consommation d'une ressource particulière, le CPU, et les mêmes paradigmes de facturation et de partage doivent être appliqués aux autres ressources dont la réalité de consommation est potentiellement décorrélée de l'usage du CPU.

Ce paradigme a cependant tardé à être intégré aux systèmes classiques, et certains systèmes d'exploitation ne fournissent que des mécanismes de facturation restreints.

3.2.2 Obstacles à la justesse de la facturation

Au-delà du concept d'unité de consommation, Verghese *et al.* [VGR98] et Banga *et al.* [BDM99], soulignent également un autre challenge pour la facturation de l'utilisation des ressources : la justesse. Un des premiers obstacles à la justesse de la facturation dans les systèmes traditionnels est l'effet de *QoS crosstalk*, originalement identifié par Leslie *et al.* [LMB+96]. Ce phénomène est dû au partage des gestionnaires de ressources par les différentes applications. Non seulement le maintien de structures algorithmiques partagées dégrade naturellement la performance, mais en plus l'exécution des algorithmes de partage est facturée au gestionnaire de la ressource, qui occupe du temps CPU au nom des applications utilisatrices. Enfin, certains traitements liés aux ressources sont effectués lors d'interruptions matérielles, dénuées de tout contexte d'exécution, et provoquent la facturation du traitement au processus arbitraire réveillé lorsque l'interruption matérielle survient.

Réduction des abstractions de haut niveau. Afin de réduire le QoS crosstalk au minimum, Leslie *et al.* prônent via leur système d'exploitation Nemesis [LMB+96] une interface de gestion des ressources minimale, épurée des

abstractions de haut niveau, et la déportation de la complexité algorithmique au sein des applications elles-mêmes. Cette solution est notamment dérivée des principes de l'exokernel [EKO95], qui remettait déjà en cause les abstractions de haut niveau des ressources matérielles faites par les noyaux, empêchant les applications d'obtenir des performances optimales. L'exokernel préfère fournir des librairies en espace utilisateur, implémentant différentes abstractions de haut niveau et politiques de gestion de ressources. Les applications sont ainsi libres d'utiliser l'abstraction qui leur convient le mieux, dans leur propre contexte d'exécution. Ce principe sera ensuite étendu dans le système EROS [SH02], dont le noyau ne fournit que des mécanismes permettant aux gestionnaires en espace utilisateur d'implémenter une facturation et une gestion sur-mesure de la ressource.

Dans les premiers systèmes conçus pour être massivement partagés, comme le noyau d'isolation Denali [WSG02], le QoS crosstalk est ainsi réduit en fournissant à chaque système invité un espace de noms privé, contenant notamment un gestionnaire de ressource, répliqué pour chaque instance. Cependant, le QoS crosstalk dans ce cas réside dans le fait que les accès effectifs au matériel sont réalisés exclusivement par une machine virtuelle de supervision privilégiée, qui devient donc un point de congestion algorithmique et de mauvaise facturation.

La technique du *lazy receiver processing* [DB96] (LRP) permet d'atténuer le QoS crosstalk pour certaines ressources, notamment le traitement de paquets réseau. Le LRP diffère la majorité du travail CPU lié aux requêtes d'entrées/sorties, à la lecture effective des données par l'utilisateur. Par exemple, la dissection des entêtes protocolaires n'est effectuée qu'au moment de la lecture via socket, et donc facturée à l'application qui reçoit effectivement au paquet, et non pas au processus victime d'une interruption matérielle ou à la pile réseau.

Le QoS crosstalk s'est ensuite naturellement transposé dans les environnements virtualisés, où le gestionnaire de ressources devient un système invité à part entière, dédié aux opérations privilégiées : c'est le cas de la machine virtuelle de supervision du noyau d'isolation Denali [WSG02], ou de la VM Dom0 dans Xen [BDF+03]. Les travaux ultérieurs sur ce problème des systèmes virtualisés ont conduit aux mêmes conclusions que celles de Leslie *et al.* : la nécessité pour les hyperviseurs d'offrir aux systèmes invités des interfaces de bas niveau, et l'introduction de la notion de notification, virtualisation des interruptions matérielles en LRP, afin de réduire les changements de contexte et de facturer justement les traitements liés à ces interruptions [SVL02, LHAP06, OCR08].

Ajustement de la facturation grâce aux traces. Ces solutions ont le désavantage de coûter très cher en termes d'implémentation, puisque nécessitant la reconstruction de la gestion des ressources des applications et des gestionnaires. Pour pallier à ce problème, Gupta *et al.* [GCGV06] proposent l'ajout à Xen de couches de facturation à partir de traces des gestionnaires de ressource, et de l'observation dynamique de l'utilisation des ressources par les différents systèmes invités [GGC05]. Les traces de haut niveau produits par les gestionnaires de ressources permettent de facturer correctement le temps de

traitement des requêtes aux systèmes invités initiateurs ou destinataires. Cette technique leur permet d'équilibrer efficacement les bandes passantes attribuées à deux serveurs Web concurrents, en n'imposant qu'une modification minimale, la production de traces, aux gestionnaires de ressources. La facturation à base de traces a également été définie de manière plus générique dans le cadre des systèmes à micro-noyaux L4 par Stoess *et al.* [SU06]. Stoess *et al.* exposent notamment des techniques permettant d'optimiser la production et le traitement des traces, comme le filtrage ou la modification à chaud des routines de production de traces. Leur travail se base sur une production coopérative des traces entre les différentes entités, que ce soient les applications, systèmes invités ou *middlewares*. Dans le cas général de systèmes partagés ayant des charges de travail égoïste, les seuls composants de confiance sont les gestionnaires de ressource, ce qui correspond au travail de Gupta *et al.*.

Prêt et transfert de ressources. Lemerre *et al.* [LDVN09] proposent de réduire le QoS crosstalk en supprimant l'utilisation de ressources effectuée par les gestionnaires, ayant lieu par effet de bord de requêtes des applications. Leur solution combine les concepts avancés par L4 [Lie95] et EROS [SH02] notamment, à savoir les mécanismes de transfert et de prêt de ressources. Ces transferts peuvent être implicites ou explicites : un processus envoyant un paquet réseau va prêter de manière implicite au gestionnaire réseau son temps CPU afin de gérer la requête ou d'analyser les entêtes et devra avoir alloué au préalable, puis prêté de manière explicite, la mémoire nécessaire au traitement et à l'envoi de la requête. Ceci permet d'éliminer une partie importante du QoS crosstalk et donc de la facturation erronée due aux actions effectuées par le noyau.

Facturation basée sur les évènements. Du côté des techniques de facturation, l'attaque de Tsafrir *et al.* [TEF07], ensuite appliquée à l'ordonnancement de machines virtuelles dans Xen [ZGDS11] a montré qu'une facturation juste de l'utilisation d'une ressource doit être basée sur les évènements, et non pas échantillonnée. En effet, comme démontré dans la section 2.2.3, une facturation à base d'échantillonnage périodique peut mesurer des parts d'utilisation mesurées opposées à la réalité : non seulement un attaquant peut monopoliser la ressource, mais sa consommation est facturée à d'autres utilisateurs n'ayant que peu accès à celle-ci.

3.2.3 Discussion

Les travaux passés sur la facturation des ressources ont prôné la définition d'unités de consommation de ressources indépendantes de l'unité d'exécution, permettant une gestion plus fine de l'utilisation des ressources, et plus adaptée aux applications dynamiques. Ce principe vient notamment du fait que le CPU n'est devenu qu'une ressource parmi les autres, et que ces dernières se sont révélées être des goulots d'étranglement significatifs pour certaines applications.

Ce paradigme n'a été implémenté que très tard dans les systèmes d'exploitation classiques, un exemple étant les cgroups de Linux ajoutés au noyau fin 2007, ou le très restreint gestionnaire de ressources Windows introduit avec Windows Server 2008.

D'un autre côté, les travaux passés dans le domaine de la justesse de facturation ont pointé du doigt les problèmes rencontrés dans la facturation du temps CPU, mais se sont peu intéressés à la facturation de l'utilisation des autres ressources, jugeant que les algorithmes d'équité et d'optimisation présents permettaient un contrôle a priori suffisant. Pourtant, les réordonnancements et heuristiques dédiées décrits dans la section 3.1.2 révèlent qu'une facturation réellement représentative de l'utilisation du périphérique par une ressource induit également des problèmes non-triviaux. Il en va de même pour la difficulté de facturation des optimisations génériques du système, comme l'utilisation du Page Cache partagé décrit dans la section 2.2.1. Par exemple, les cgroups Linux ne sont pas capables de facturer correctement les requêtes d'entrées/sorties bufferisées [Doc09].

3.3 Répartition de charge et contrôle d'accès

Au-dessus des algorithmes de partage et de gestion de ressources, la répartition des charges de travail et l'ajout de couches de contrôle d'accès sont devenus essentiels pour la préservation de la qualité de service des différentes applications concurrentes. Le challenge principal réside ainsi dans l'adaptation des systèmes aux besoins réels des applications.

Un premier ensemble d'efforts vise à garantir a priori un niveau d'utilisation des ressources à chaque application, par l'usage de priorités ou de besoins quantitatifs connus à l'avance. Ces besoins, souvent dérivés des SLA ou de l'observation au préalable du profil de consommation, permettent l'expression de problèmes de contraintes résolus à l'avance afin de répartir les charges de manière optimale. Ces techniques ne peuvent cependant pas s'adapter aux besoins changeants ou non prédictibles des applications, et doivent ainsi souvent sous-utiliser la capacité des ressources à disposition.

D'autres techniques, plus adaptées aux réalités des systèmes partagés best-effort, s'appuient notamment sur la facturation en temps réel, afin d'adapter l'allocation et l'ordonnancement des requêtes à l'état réel du système. Bien qu'elles promettent des résultats attractifs, ces approches ne sont pas satisfaisantes car reposant sur l'utilisation de quotas arbitraires, difficilement paramétrables et dépendantes du type de ressources, à ajuster finement à la main.

3.3.1 Garanties a priori sur l'utilisation des ressources

A l'opposé des systèmes best-effort, les systèmes à réservation sont adaptés aux contextes applicatifs où les besoins des charges de travail sont connus à l'avance. Ce contexte permet souvent la résolution a priori d'un problème à base de contraintes afin de garantir une qualité de service arbitraire à chaque application. Un certain nombre de travaux se sont inspirés de ce paradigme et l'ont

appliqué aux systèmes best-effort, en se servant de besoins définis à l'avance, à l'aide de priorités, de modélisation des charges de travail et d'observations préalables notamment.

La difficulté de détermination de ces besoins et le caractère dynamique du déploiement de services dans les systèmes partagés best-effort rendent cependant difficile l'application pratique de ces techniques, et provoquent une sous-utilisation des ressources afin de permettre une gestion naturelle des pires cas d'exécution.

Répartition arbitraire prédéfinie. Contrairement aux travaux étudiant le QoS crosstalk notamment, Rajkumar *et al.* [RJMO01] posent le noyau comme le garant de la qualité de service des applications. Chaque application doit selon eux simplement pouvoir spécifier ses besoins en temps d'utilisation des ressources puis laisser au noyau le soin d'ordonnancer et d'allouer l'accès aux ressources en conséquence, celui-ci ayant une vue globale du système. Déléguer toute la politique de gestion des ressources est selon eux la seule façon d'éviter les effets de bord induits par les applications. Dans la pratique, leurs *resource kernels* se basent sur un mécanisme de réservation de ressources unifié, ainsi qu'une vérification à l'exécution du respect du temps d'utilisation des ressources. Le noyau permet également aux applications de connaître l'état de charge des ressources et leur utilisation nominale, afin que chaque application puisse adapter ses demandes.

La vision de Rajkumar *et al.* est typique des systèmes à réservation, en utilisant des besoins en ressources connus avant l'exécution. Cela permet de réserver les ressources nécessaires à chaque application et d'implémenter une couche de contrôle imposant le respect des réservations. Cette idée s'est traduite dans les systèmes best-effort partagés par une volonté de garanties a priori sur l'utilisation des ressources, par la définition de priorités et de pondérations.

Un exemple de cette vision est le système QLinux [SCG+00], un dérivé de Linux dont l'objectif principal est de répondre à l'existence de plusieurs classes d'applications, aux besoins très hétérogènes en termes de ressources. Les applications interactives désirent des temps de réponse faibles, les applications intensives en ressources ont besoin d'une bande passante cumulée importante, et les applications temps réel souple ont besoin de garanties de performance ou de temps d'exécution. Afin de supporter l'exécution simultanée de ces charges de travail hétérogènes, QLinux embarque dans son noyau la notion de classe d'application. Chaque classe d'application est liée aux différentes ressources par un poids spécifique, déterminé arbitrairement a priori. Les ordonnanceurs liés aux requêtes de chaque ressource essaient ainsi de répartir l'utilisation selon ces poids prédéfinis. L'ordonnancement peut être enrichi de manière hiérarchique, c'est-à-dire que chaque classe d'application peut être divisée en un certain nombre de groupes d'applications plus spécifiques, se partageant l'accès aux ressources dans les limites des accès permis à la classe d'application mère. Dans QLinux, cette technique d'ordonnancement est notamment appliquée au CPU, à l'envoi de paquets dans la pile réseau et à un premier niveau

d'ordonnancement des requêtes disque.

Kaneko *et al.* [KKS03] s'intéressent eux au cas des systèmes partagés où il est possible de différencier les priorités des applications. Leur travail vise à empêcher le déni de service d'applications légitimes, à cause de l'abus d'applications moins prioritaires. Ils proposent ainsi un ordonnancement des requêtes matérielles à base de priorités et de préemption. En cas de contention autour d'une ressource, les requêtes des processus de priorité supérieure sont servies au détriment de celles des processus de priorité inférieure, les processus non-sûrs ayant la priorité la plus basse. Ceci revient essentiellement à appliquer l'idée des priorités d'ordonnancement du processeur à l'ordonnancement des requêtes d'entrées/sorties. Ce modèle permet notamment une gestion élégante des problèmes liés à l'exécution de code externe, comme les applets Java ou les applications Flash dans un système de bureau. Il n'est cependant pas possible de contenir les processus abusifs dans le cas général, autrement qu'en effectuant une opération d'administration humaine de diminution de priorité, et ce modèle ne convient donc pas réellement aux systèmes partagés best-effort où un nombre important d'utilisateurs indépendants ont la même priorité d'exécution.

Une autre solution plus triviale est la mise en place de quotas simplistes, divisant chaque ressource en parts fixes et équitables. Cette approche permet de borner de manière directe l'utilisation des ressources par chaque utilisateur et ainsi d'éviter les contentions brutales. Ceci dit, ces seuils arbitraires sont difficiles à émettre a priori par les hébergeurs, et sous-utilisent la capacité du matériel dans le cas général, tant que chaque utilisateur ne dépense pas entièrement son quota de ressources.

Modélisation et observation préalable. Dans les systèmes partagés communs, il n'est pas possible de déterminer a priori l'utilisation effective des ressources par les nouvelles charges de travail. De ce fait, un nombre important de travaux ont cherché à estimer ou à modéliser le comportement attendu des applications, à partir d'observations étalon effectuées avant l'exécution réelle sur le système partagé. La connaissance du profil attendu d'utilisation des ressources permet de ramener ce problème à une résolution de contraintes de placement d'applications sur un certain nombres de systèmes mutualisés, ou de systèmes invités sur un ensemble d'hyperviseurs.

Un exemple direct de ces travaux est l'outil Minerva [ABG+01], cherchant à répartir un ensemble de charges de travail sur différents systèmes de stockage, en fonction de la capacité de ceux-ci et des besoins des charges de travail. La caractérisation de ces besoins est dérivée de traces d'exécutions préalables, regroupant des critères complexes et spécifiques aux ressources utilisées, comme la localité des accès disque et la corrélation entre les différentes charges de travail. La résolution de ce problème de contrainte leur permet ainsi de déduire une répartition optimale de maximisation de la performance et du taux d'utilisation des systèmes de stockage.

De manière semblable, Urgaonkar *et al.* [USR02] présentent une méthode

statistique permettant de déduire les besoins en termes de chaque application en termes de qualité de service (QoS). Ces besoins, également dérivés d'une observation étalon, expriment la fréquence et la variabilité d'utilisation de chaque ressource. Ils permettent de grouper les applications sur différents hôtes, de manière à ce que les besoins de chaque application soient garantis. Les auteurs montrent également les bénéfices potentiels de légères sur-réservations, c'est-à-dire un groupement des applications tel que la somme des besoins individuels dépasse la quantité de ressources réellement disponibles sur un hôte, tout en conservant une faible probabilité statistique de contention réelle.

Koh *et al.* [KKB+07] se sont quant à eux attachés à modéliser les différents profils d'utilisation des ressources, afin de pouvoir classer les applications en fonction de l'observation préalable. Leur travail part du constat que les différents types d'applications ont des utilisations très différentes des ressources à leur disposition, pouvant être intensives en bande passante réseau, en temps processeur ou encore manipulant intensivement des données ayant une forte localité en mémoire. Ces différents profils d'utilisation peuvent cependant être fortement antagonistes, ou au contraire cohabiter aisément. Par exemple, une charge de travail intensive en mémoire va engendrer des interférences négligeables sur une charge de travail intensive en entrées/sorties. Les auteurs modélisent ainsi l'interférence de performance entre les systèmes invités dans un environnement virtualisé, en fonction de leur profil d'utilisation des ressources. A partir du profil des systèmes invités et de ces différences d'interférences, dépendant notamment du type de ressource mais aussi du type d'environnement de virtualisation [MHH+07, PLM+10], Koh *et al.* sont capables de répartir ces systèmes sur plusieurs hôtes, en réduisant au maximum l'interférence de performances tout en maximisant l'utilisation globale des ressources.

Nathuji *et al.* sont les premiers à combiner cet aspect rigide de besoins dérivées d'une observation étalon avec la réalité du comportement des applications concurrentes à l'exécution, dans la spécification de leurs Q-clouds [NKG10]. Leur approche est surtout focalisée sur le lissage des différences de performance naturelles entre une exécution solitaire et une autre au sein d'un système virtualisé partagé. L'observation étalon leur permet de dériver les besoins de QoS minimaux des applications virtualisées, ainsi que le surplus en ressources destiné à combler la dégradation de performance induite par le partage. Les Q-clouds introduisent cependant une certaine notion de flexibilité à l'exécution, avec la possibilité de détecter à l'exécution les ressources libres et de les exploiter pour élever le niveau de QoS garanti à un système invité, dans le cas où le client est prêt à payer plus pour de meilleures garanties de performance. Les Q-clouds apportent ainsi un certain niveau de garantie de QoS minimal, potentiellement élevé en fonction de la situation à l'exécution, au prix d'une sous-répartition initiale des systèmes invités par rapport aux ressources.

Discussion. L'ensemble de ces techniques cherchant à réguler l'utilisation des ressources avant l'exécution dans l'environnement final ne s'adaptent pas aux réalités d'exécution des systèmes partagés best-effort. Les services mutualisés

ou du cloud sont consommés en temps réel et à la demande, rendant impossible toute utilisation de traces d'exécution. De plus, la réservation de ressources implique une sous-utilisation vis-à-vis de la capacité maximale, alors même que l'optimisation de la répartition des ressources est un facteur essentiel de réduction des coûts pour les hébergeurs.

3.3.2 Régulation dynamique et adaptative

Les algorithmes de partage équitables et optimisés décrits dans la section 3.1 ne sont pas suffisants à assurer un partage équitable des ressources entre différents domaines dans certains cas particuliers, comme démontré par nos résultats de la section 2.2. Ces problèmes se posent à l'identique dans les environnements virtualisés, qui emploient des techniques de gestion similaires. Ongaro *et al.* [OCR08], ou Kim *et al.* [KKC12], pointent par exemple l'incapacité de l'ordonnancement de Xen à assurer un partage équitable dans l'utilisation des ressources. Indépendamment des couches d'optimisation de l'exploitation des périphériques, des efforts plus réalistes cherchent à assurer un partage équitable des ressources d'un système, s'adaptant en fonction d'une facturation de la consommation des ressources lors de l'exécution réelle.

Un premier exemple tourné vers l'adaptation dynamique de l'ordonnancement des requêtes est Façade [LMA03], qui interpose une couche d'arbitrage entre les ressources matérielles et les charges de travail, dans le cadre de l'utilisation de larges systèmes de stockage à travers le réseau. Façade comporte trois composants essentiels : un ordonnanceur de requêtes, un contrôleur d'utilisation des ressources et un système d'observation regroupant les statistiques de consommation. L'ordonnanceur scrute les files d'attente de requêtes de chaque charge de travail et les transfère à la file d'attente du périphérique matériel, à partir de délais de réponses maximums déduits des objectifs contractuels des SLA, imposés par le client. Le système d'observation regroupe les temps d'arrivée et de fin de requêtes, permettant au contrôleur d'estimer les temps moyens d'exécution et le nombre de requêtes en attente sur chaque périphérique. Le contrôleur peut de cette manière paramétrer l'ordonnanceur et lui permettre d'ajuster son approximation des dates de départ maximums de chaque requête, en prenant en compte les latences moyennes de transmission et d'exécution de celles-ci.

Le système Tessellation [CEH+13] adopte également une approche à base de besoins de QoS prédéfinis à partir des SLA, tout en redistribuant les ressources périodiquement en fonction de rapports de performance des applications. La gestion de ressources de Tessellation est basée sur deux niveaux d'ordonnancement, en séparant gestion de l'allocation et équité d'utilisation. Le premier répartit les ressources partitionnables entre différentes unités de consommation, les cellules. Les cellules peuvent être vues comme un environnement virtuel isolé, ayant accès aux ressources via une interface dédiée. Le deuxième niveau d'ordonnancement a lieu à l'intérieur des cellules, qui regroupent des applications ayant des buts similaires en termes de consommation de ressources comme les serveurs réseau, et ordonnance les requêtes internes selon une politique ajustée

aux besoins de la classe d'application spécifique. L'adaptivité de Tessellation réside dans la redistribution périodique des allocations, en fonction de rapports de performance via un certain nombre de métriques arbitraires émanant à la fois des cellules, par exemple le délai moyen de traitement de requêtes pour les applications, mais également à partir de mesures de performance globales, comme les statistiques d'accès aux caches. Cette redistribution essaie ainsi de concilier performance globale du système et maintient d'un niveau de QoS minimal pour chaque type d'application. L'utilisation d'un nombre de métriques spécifiques important, la complexité de la résolution multi-critères et la définition a priori des besoins de QoS compromettent cependant l'application d'une telle architecture dans le cas général des systèmes partagés.

Pour le cas plus général des environnements de virtualisation, Gulati *et al.* proposent mClock [GMV10], un ordonnanceur d'entrées/sorties destiné aux hyperviseurs. Il tente d'assurer des niveaux de QoS à un certain nombre de machines virtuelles (VM) invitées via trois critères principaux, définis par VM et pour chaque ressource : une allocation minimum, un quota maximum et un poids. mClock ordonnance et facture ensuite les requêtes d'entrées/sorties de chaque VM en priorisant les VM qui n'ont pas encore reçu leur allocation minimum, en ne traitant pas les requêtes des VM qui ont déjà reçu le maximum permis, puis en round-robin pondéré pour les requêtes des VM ne correspondant pas aux deux premiers cas. Les garanties de QoS sont définies de manière arbitraire et sont ajoutées en tant que métadonnées aux requêtes. Les résultats expérimentaux montrent une bonne isolation des VM abusives ainsi qu'un respect de la proportionnalité définie par les poids attribués à chaque VM. Gulati *et al.* fournissent donc un mécanisme d'équité travaillant exclusivement sur les requêtes d'E/S, et non sur l'ordonnancement processeur des différentes VM. Cette architecture de respect de QoS forme notamment la base de la gestion des ressources dans les systèmes de virtualisation VMware [GHJ+12]. Bien que mClock se base toujours sur un ensemble de valeurs arbitraires difficilement définissables en dehors de l'existence des SLA, il permet la maximisation de l'utilisation des ressources dans le cas général, puis suivant des poids arbitrairement définis lors de contention.

AutoControl [PHS+09] est un autre exemple d'architecture de gestion de ressources, visant à réguler les requêtes de systèmes invités virtualisés, en minimisant l'intervention humaine et en s'adaptant aux charges dynamiques des machines virtuelles. Leur architecture ajoute deux ensembles de contrôleurs à un système Xen classique : les contrôleurs d'applications, chargés de mesurer la consommation des ressources de chaque application et d'estimer ses besoins futurs à court terme, et les contrôleurs de noeuds, contrôlant l'utilisation agrégée des ressources de plusieurs applications. Leur objectif est de respecter les SLA définis pour chaque application et pour chaque ressource : la demande client exprimée spécifiquement pour la ressource, par exemple en termes de délai maximum de réponse ou de bande passante minimum. A partir d'une estimation linéaire des besoins des applications à court terme, les contrôleurs de noeuds sont capables de déterminer si une ressource en particulier est sous contention ou non. Ils résolvent dans ce cas une optimisation multi-critères

visant à minimiser localement la différence entre l'allocation souhaitée et l'allocation effective à chaque application, pondérée par la priorité définie par les SLA. AutoControl est ainsi capable d'adapter dynamiquement les allocations des différents noeuds et différentes applications lorsqu'une contention intervient. La résolution de l'équation d'optimisation, faite dans leurs travaux pour le CPU et l'utilisation du disque seulement, se complexifie significativement avec l'ajout de types de ressources supplémentaires à l'équation. Enfin, la précision des sondes de consommation de ressources sur lesquelles se basent les contrôleurs d'applications est un point crucial du modèle d'AutoControl, et les auteurs précisent que Xen ne fournit pas de mécanismes suffisamment précis pour mesurer l'utilisation de certaines ressources, notamment le réseau. Ces imprécisions des couches de facturation de l'hyperviseur, rapportées par Gupta *et al.* [GCGV06] ou Zhou *et al.* [ZGDS11], peuvent engendrer des erreurs d'ordonnancement significatives. Cette architecture hiérarchique permet tout de même d'effectuer un contrôle théoriquement générique, pour les systèmes virtualisés et/ou mutualisés en fonction des SLA.

Ces approches différentes ont en commun la nécessité d'une paramétrisation extérieure non triviale en termes de maximum de bande passante ou de latence, supposée dérivée des SLA. Au contraire, la couche d'analyse à grain fin de la distribution des E/S dans Xen de Kim *et al.* [KKC12] s'affranchit de ces quotas maximum et minimum arbitraires. Kim *et al.* prouvent qu'il est possible d'obtenir une équité empirique dans l'utilisation des entrées/sorties en manipulant la priorité des requêtes des systèmes invités, en fonction de la détection d'un état de famine ou de monopolisation de la ressource. Leur détection se base sur une facturation à grain fin de la bande passante utilisée par chaque VM, sur une période de temps définie arbitrairement à 30 millisecondes. Lorsqu'une VM obtient plus de 80% ou moins de 5% de la bande passante de la ressource sur l'intervalle de facturation, la priorité de ses requêtes est respectivement diminuée ou augmentée. Ajuster ainsi les priorités de manière périodique leur permet d'obtenir une distribution équitable de la bande passante disque, contrairement aux résultats de l'ordonnanceur CFQ par défaut sous Xen. Ce mécanisme simple a notamment l'avantage de n'avoir besoin que de la définition de deux paramètres externes, à savoir la période de facturation et la capacité maximale de la ressource contrôlée.

Un autre cas intéressant est le travail de Hu *et al.* [HLZ+10], qui exploitent les architectures multicoeurs, en groupant différents profils de machines virtuelles sur des coeurs indépendants. Un coeur est réservé aux pilotes afin de minimiser les changements de contexte, un coeur regroupe les VM utilisant les entrées/sorties de manière intensive, et un dernier coeur est réservé aux autres VM plus généralistes. Le placement des VM sur chaque coeur est effectué de manière dynamique à l'exécution, en fonction du dépassement ou non de seuils arbitraires d'utilisation de la bande passante. Chaque coeur suit un ordonnancement CPU dédié et optimisé : l'ordonnanceur à crédit de Xen est utilisé pour les VM généralistes, et un ordonnanceur à tick rapide, priorisant les VM venant de recevoir un évènement, est utilisé pour les VM intensives en E/S. Bien que cette solution ne maximise pas l'utilisation des coeurs à disposition

de l'hébergeur, elle ne nécessite pas la définition de seuils de bande passante. Elle se base au contraire sur une facturation en temps réel afin de caractériser les applications et de leur appliquer une politique d'ordonnancement CPU adaptée.

3.3.3 Discussion

Dans les systèmes partagés best-effort, afin de permettre aux hébergeurs de respecter un partage des ressources selon les demandes des clients, deux approches principales s'opposent. D'un côté, une répartition définie préalablement garantit aux applications un accès quantifié aux ressources. De l'autre côté, un contrôle dynamique se basant sur une facturation en temps réel de la consommation supporte une adaptation de l'ordonnancement des requêtes à la réalité de l'état du système. La première approche se base notamment sur des segmentations a priori des charges de travail, comme les quotas de consommation ou la modélisation par observation préalable. Ces techniques sont non seulement peu réalistes dans les cas d'utilisation de systèmes partagées best-effort qui nous intéressent, mais sous-utilisent de manière importante la capacité des ressources à leur disposition.

D'un autre côté, un certain nombre de travaux ont implémenté des architectures de contrôle complètes, se basant sur une facturation de la consommation effective pour réguler en temps réel l'utilisation des ressources par les différentes applications. Certaines de ces approches conservent l'usage de quotas maximum difficiles à déterminer a priori, mais permettent une maximisation de l'exploitation du matériel lorsqu'il n'y a pas de contention, ainsi qu'un partage équitable lorsqu'une contention intervient. L'approche de Kim *et al.* [KKC12] est plus particulièrement intéressante, car leur facturation à base de pourcentage d'utilisation de la bande passante leur permet de détecter les utilisateurs abusifs et les utilisateurs victimes de famine, puis d'adapter dynamiquement la priorité des requêtes de ces utilisateurs particuliers. Ainsi, leur couche de facturation ne nécessite pas de paramètres arbitraires potentiellement dérivés des SLA, mais seulement la définition d'une période interne de facturation.

Enfin, toutes ces approches passées continuent à se baser sur des métriques propres à chaque ressource, comme la latence des paquets réseau ou la bande passante des périphériques d'entrées/sorties. Cette particularité impose une implémentation différente des couches de facturation et de contrôle pour chaque type de ressource et un paramétrage différent selon les capacités effectives des différents systèmes.

3.4 Conclusion

La gestion des ressources dans les systèmes best-effort doit faire face à un compromis difficile entre l'équité d'accès et la performance dans le cas général. Cet arbitrage se retrouve naturellement dans le contexte de services contractualisés, où les besoins de garanties en termes d'allocation et de performance des clients

contrastent avec la réduction des coûts et donc l'optimisation d'utilisation de la capacité souhaitée par les hébergeurs.

Les limites d'implémentation des algorithmes de gestion des ressources ne peuvent pas à eux seuls garantir des propriétés d'équité ou de maximisation de capacité sous toutes les charges de travail. Ainsi, l'ajout de couches de facturation et de mécanismes de répartition et de contrôle de l'utilisation de ressources par les charges de travail est devenu une nécessité. La littérature scientifique a montré la nécessité de techniques de facturation spécifiques aux différentes ressources, ainsi qu'un certain nombre de challenges concernant la justesse de celles-ci.

La facturation de l'utilisation des ressources est notamment la base des couches de contrôle adaptatives, permettant de réguler la consommation des charges de travail concurrentes, tout en tirant le maximum des capacités matérielles. Cependant, ces architectures de contrôle s'appuient souvent sur un ensemble de valeurs arbitraires difficilement paramétrables, et souvent spécifiques à chaque type de ressources, comme la bande passante réseau, la latence d'accès aux systèmes de stockage ou le maximum de temps processeur alloué.

Chapitre 4

Contrôle unifié et générique de l'utilisation des ressources

Comme nous l'avons vu dans le chapitre précédent, les techniques de gestion des ressources dans les systèmes partagés se basent typiquement sur le couplage de méthodes d'optimisation dédiées et de couches de distribution équitable. De plus, ces techniques se reposent communément sur des métriques arbitraires, exprimées par exemple en termes de bande passante ou d'une quantité maximum d'allocation, afin de segmenter l'accès effectif aux ressources. En premier lieu, ces valeurs sont délicates à mettre au point pour les hébergeurs, car elles dépendent de facteurs multiples, notamment du profil des applications ainsi que des caractéristiques des ressources. Enfin, elles sous-utilisent intrinsèquement l'usage des ressources afin de garantir un taux d'accès minimal aux différentes applications. D'un autre côté, les limites naturelles dans l'implémentation des algorithmes d'optimisation et d'équité permettent toujours à une application abusive de réduire la qualité de service fournie aux autres applications s'exécutant dans le même environnement, comme démontré dans la section 2.2.

Dans ce chapitre, nous introduisons une métrique générique de caractérisation de l'usage des ressources : le temps d'utilisation. Cette métrique est facilement calculable à grain fin moyennant une modification minimale des gestionnaires de ressources, que ce soit pour le processeur, les entrées/sorties, les services système ou les ressources logicielles finies. Elle permet de plus la définition de politiques de partage simples, dénuées de paramètres arbitraires ou spécifiques aux ressources supervisées.

Ces caractéristiques nous permettent de proposer une architecture générique de gestion et de contrôle des ressources qui intercepte, et éventuellement diffère, les requêtes de manière transparente pour les applications. Son objectif n'est pas de se substituer aux ordonnancements et/ou optimisations, mais de borner l'impact des applications abusives sur les autres applications concurrentes. Cette architecture s'adapte naturellement aux micro-noyaux de type L4, mais peut être transposée aux systèmes monolithiques en conservant ses propriétés d'équité et de maximisation de l'exploitation des ressources.

4.1 Temps d'utilisation comme métrique générique

Les métriques typiquement employées pour facturer et contrôler l'équité sont généralement entièrement spécifiques aux ressources surveillées. Les travaux passés manient par exemple la bande passante pour les transferts réseau ou disque, le temps d'utilisation pour le CPU, la quantité de mémoire allouée, ou encore le nombre d'accès ou de fautes de page sur une période arbitraire pour le service de swap. Les hébergeurs doivent ainsi paramétrer ces quotas arbitraires de manière pointue et en fonction de chaque ressource.

De plus, ces valeurs ne sont pas nécessairement représentatives du travail réellement imposé au système : un envoi important de paquets de petite taille confère une bande passante moins importante à un utilisateur lambda, par rapport à un autre envoyant un nombre similaire de paquets de plus grande taille. Cependant, les deux utilisateurs dans ce cas imposent une pression globalement comparable au périphérique, puisque celui-ci passe autant de temps à traiter les requêtes de chacun.

La métrique sous-jacente à ce problème est le pourcentage de temps d'utilisation de la ressource, et est applicable de manière générique aux différentes ressources typiques des systèmes partagés best-effort.

4.1.1 Le temps : métrique intuitive de l'utilisation du CPU

Le temps d'utilisation est d'ores et déjà employé pour la répartition du CPU dans les systèmes partagés. Les ordonnanceurs équitables, comme CFS, se basent par exemple sur le temps pendant lequel un utilisateur est en attente du CPU. Dans ce cas, les utilisateurs ayant attendu le plus sont prioritaires aux prochains ordonnancements. Ceci revient à donner la main aux utilisateurs ayant le plus faible pourcentage du temps processeur sur la période de facturation.

La figure 4.1 rapporte la répartition effective du temps CPU lors de l'attaque par monopolisation de ticks décrite dans la section 2.2.3. Cette figure montre une relation linéaire claire entre le nombre de cycles monopolisés par tick et le pourcentage du temps CPU obtenu par le processus abusif. C'est pourquoi une facturation plus précise, dirigée par les événements et non par échantillonnage, permet aux ordonnanceurs équitables de contrer naturellement une monopolisation de ticks.

Nous remarquons ici que le temps d'utilisation a toujours été appliqué au partage du CPU, et non une valeur spécifique à la ressource en question, qui pourrait être le nombre d'instructions dans ce cas précis. En effet, les différentes instructions imposent un travail différent au processeur et un temps d'exécution plus ou moins long. De la même manière, le traitement de requêtes d'entrées/sorties peut être plus ou moins long selon la taille ou la position physique des données demandées.

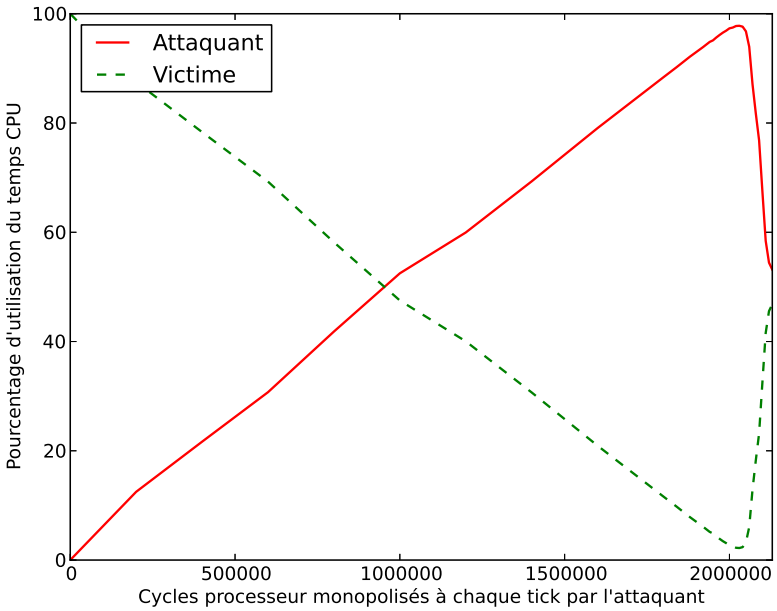


FIGURE 4.1 – Distribution du temps CPU lors d'une attaque par monopolisation de cycles.

4.1.2 Temps d'utilisation des périphériques d'entrées/sorties

Cette métrique est simple à calculer à grain fin dans le cas du CPU, puisqu'elle correspond à la somme des périodes pendant lesquelles un processus a la main sur le processeur. Il suffit donc simplement de mettre à jour la facturation lorsque les processus prennent ou rendent la main sur le processeur. La gestion des périphériques d'entrées/sorties est cependant plus complexe, puisque les requêtes traversent de multiples couches d'ordonnancement, en particulier celles du système d'exploitation et du matériel. Dans ce contexte, la différence de temps entre l'arrivée d'une requête à la couche de gestion des entrées/sorties et sa fin de traitement n'est pas significative de la pression réellement imposée au périphérique, notamment à cause des réordonnancements potentiellement effectués par chaque couche.

Réordonnancement de requêtes et justesse de facturation. Afin d'illustrer ce point, nous reproduisons ici l'attaque par congestion des files d'attente de l'ordonnanceur d'entrées/sorties Linux exposée dans la section 2.2.1. Un processus abusif fait ainsi pression sur les files d'attente en utilisant des entrées/sorties bufferisées sur un fichier de 50 Mo, tandis que l'utilisateur légitime

manipule deux fichiers de 10 Ko. Les cgroups Linux, unité de facturation de la consommation des ressources, permettent de suivre le temps passé par les requêtes d'un cgroup particulier dans les files d'attente de l'ordonnanceur, ainsi que la durée de traitement de ces requêtes par le matériel. En plaçant chaque utilisateur dans un cgroup différent, il est ainsi possible d'étudier l'évolution de leur consommation du temps d'utilisation du système d'entrées/sorties. La facturation des cgroups ne gérant pas convenablement deux domaines différents effectuant des requêtes bufferisées de manière simultanée [Doc09], nous utilisons ici des entrées/sorties synchronisées directes pour la victime.

La figure 4.2 rapporte l'évolution, en pourcentage, des temps d'utilisation du système d'entrées/sorties complet pendant l'exécution de la charge légitime, c'est-à-dire le temps entre la soumission des requêtes par les utilisateurs et l'interruption matérielle de fin de traitement.

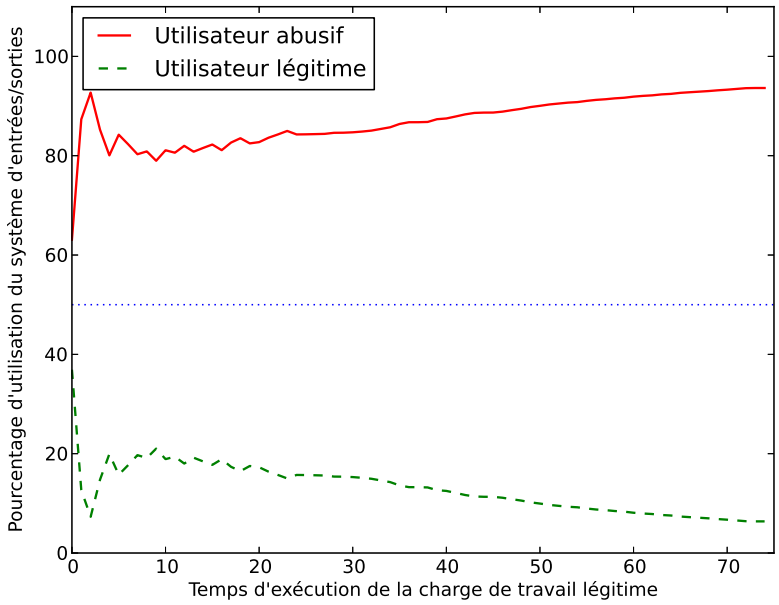


FIGURE 4.2 – Distribution du temps d'utilisation du système d'entrées/sorties complet (ordonnancement + traitement par le matériel) lors d'une attaque sur les files d'attente de l'ordonnanceur d'entrées/sorties.

Cette figure expose déjà l'abus dont est victime le processus légitime, mais pas dans les proportions réelles de l'attaque : la charge de travail légitime s'exécute en 0.3 secondes seule, et en plus de 70 secondes avec le processus abusif.

Or, la dégradation observée relève d'un facteur 5 à 10. La figure 4.3 rapporte quant à elle seulement l'utilisation du temps matériel à chaque seconde.

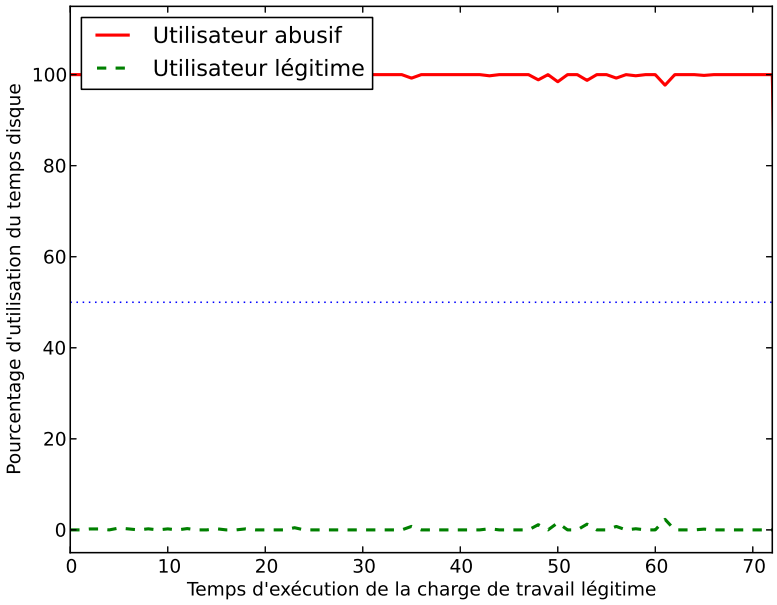


FIGURE 4.3 – Distribution du temps d'utilisation du matériel lors d'une attaque sur les files d'attente de l'ordonnanceur d'entrées/sorties.

La monopolisation relevée est cette fois supérieure à 99% du temps matériel, et permet de mieux évaluer la dégradation de performance exceptionnelle engendrée par le processus abusif et subie par le processus légitime. Ceci prouve notamment qu'une facturation juste du temps d'utilisation doit être effectuée au plus près du traitement de la ressource, afin d'obtenir une évaluation de la pression indépendante des biais introduits par les réordonnancements ou l'état de congestion des files d'attente.

Calcul générique du temps d'utilisation de requêtes. Cependant, la facturation du temps matériel des cgroups souffre exactement du problème que nous venons de souligner : si les réordonnancements sont problématiques pour une facturation juste dans les files d'attente du système, il en va de même pour les réordonnancements dans les files d'attente du matériel. Malgré le fait que les matériels ne permettent pas d'évaluer l'ordre ou le temps nécessaire au traitement de chaque requête, il est possible de déduire de manière fine ce temps

d'utilisation réel du matériel pour chaque requête, à partir des évènements de transmission et de fin de traitement des requêtes.

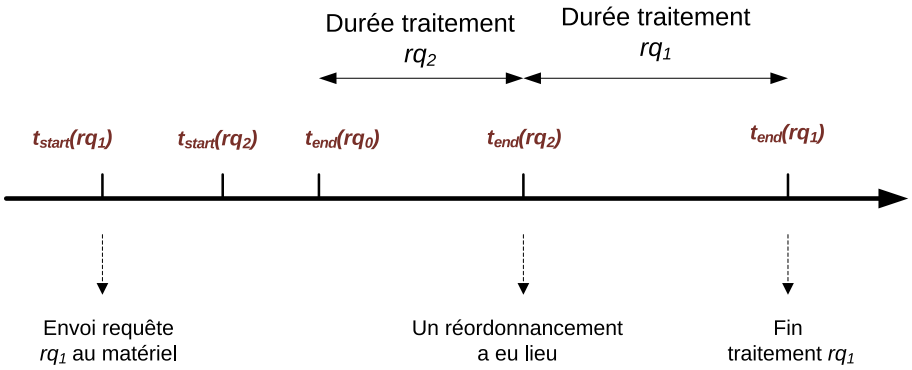


FIGURE 4.4 – Inversion de priorité et durée de requêtes.

Supposons que le matériel possède une file d'attente interne dont l'ordre ne peut être prédit, et qu'il envoie une interruption de fin de traitement pour chaque requête. Soit une requête rq_i, i-ème requête transmise au matériel au temps $t_{start}(rq_i)$. Sa fin de traitement est notifiée au temps $t_{end}(rq_i)$. Nous pouvons distinguer trois cas vis-à-vis de l'ordre de traitement de rq_i par le matériel :

1. Le matériel n'est pas en cours de traitement d'une autre requête au temps $t_{start}(rq_i)$, donc rq_i ne peut pas subir de réordonnancement. La durée d'utilisation du matériel se calcule alors simplement par $t_{end}(rq_i) - t_{start}(rq_i)$.

2. Le matériel est en cours de traitement de la requête rq_j au temps $t_{start}(rq_i)$, et rq_i ne subit pas de réordonnancement. Le début effectif de traitement de la requête rq_i est alors la fin de traitement de rq_j, et la durée d'utilisation du matériel est $t_{end}(rq_i) - t_{end}(rq_j)$.

3. Le matériel est en cours de traitement de la requête rq_j, et rq_i subit un réordonnancement. La figure 4.4 illustre cette situation, où rq_{i+1} est traitée avant rq_i. Le début effectif de traitement de la requête rq_i est alors la fin de traitement de rq_{i+1}, et la durée d'utilisation du matériel est $t_{end}(rq_i) - t_{end}(rq_{i+1})$.

Notons $t_{last}(t_{end}(rq_i))$ le temps de la dernière interruption matérielle de fin de requête avant $t_{end}(rq_i)$. Dans le cas du réordonnancement subi par rq_1 sur la figure 4.4, $t_{last}(t_{end}(rq_1))$ correspond ainsi à $t_{end}(rq_2)$. De la même façon, $t_{last}(t_{end}(rq_2)) = t_{end}(rq_0)$. Ainsi, $t_{last}(t_{end}(rq_i))$ représente le début effectif de l'utilisation du matériel par rq_i dans les cas 2 et 3. Dans le cas 1, le début

de traitement est $t_{start}(rq_i)$ qui est nécessairement supérieur à $t_{last}(t_{end}(rq_i))$, puisque le matériel n'est pas en cours d'utilisation lors de la soumission de rq_i. Nous pouvons déduire $Duration(rq_i)$, la durée d'utilisation du matériel par une requête rq_i, comme étant le maximum entre $t_{start}(rq_i)$ et $t_{last}(t_{end}(rq_i))$, d'où la formule suivante :

$$Duration(rq_i) = t_{end}(rq_i) - max(t_{start}(rq_i), t_{last}(t_{end}(rq_i)))$$

Dans la réalité, certains matériels peuvent cependant envoyer une interruption de fin de traitement agrégeant plusieurs requêtes. Dans ce cas, la durée de traitement de ces requêtes peut être approximée en divisant $Duration(rq_i)$ par le nombre de requêtes agrégées.

4.1.3 Temps d'utilisation du mécanisme de swap

Nous montrons ici que cette métrique peut également être appliquée à des ressources dont la facturation a traditionnellement été difficile, comme l'utilisation de la mémoire virtuelle et du mécanisme de swap par exemple. La taille de mémoire allouée à une application, usitée pour la mise en place de quotas dans les systèmes partagés best-effort, est potentiellement décorrélée de la pression réellement imposée au système. En effet, un processus manipulant une large quantité de mémoire de manière efficace peut ne nécessiter qu'un nombre restreint et raisonnable de pages en mémoire centrale à la fois.

La pression effectivement imposée au système lors de l'utilisation du mécanisme de swap est directement liée au traitement des requêtes d'entrées/sorties nécessaires au rapatriement de pages préalablement déchargées. Le déchargement de page ayant lieu de manière globale et périodique dans la majorité des systèmes d'exploitation, celui-ci ne peut être facturé à une entité spécifique. Au contraire, le rapatriement de pages est effectué à la demande, lorsqu'un processus précis effectue un accès mémoire provoquant une faute de page.

Afin d'observer la distribution de l'utilisation du mécanisme de swap sous diverses charges de travail, nous avons instrumenté le noyau Linux afin de facturer de manière fine la somme des durées nécessaires à l'exécution des requêtes d'entrées/sorties dues aux fautes de page de chaque processus. Sur un système doté de 512 Mo de RAM, nous avons tout d'abord exécuté de manière simultanée deux charges de travail similaires, intensives en mémoire car utilisant chacune 400 Mo. La distribution de l'utilisation dans ce cas est rapportée sur la figure 4.5.

Cette figure démontre que le temps d'utilisation du mécanisme de swap relève bien une répartition globalement équitable entre deux processus similaires, puisque le pourcentage de consommation de chacun oscille autour des 50%. La métrique permet donc de bien représenter un cas équitable.

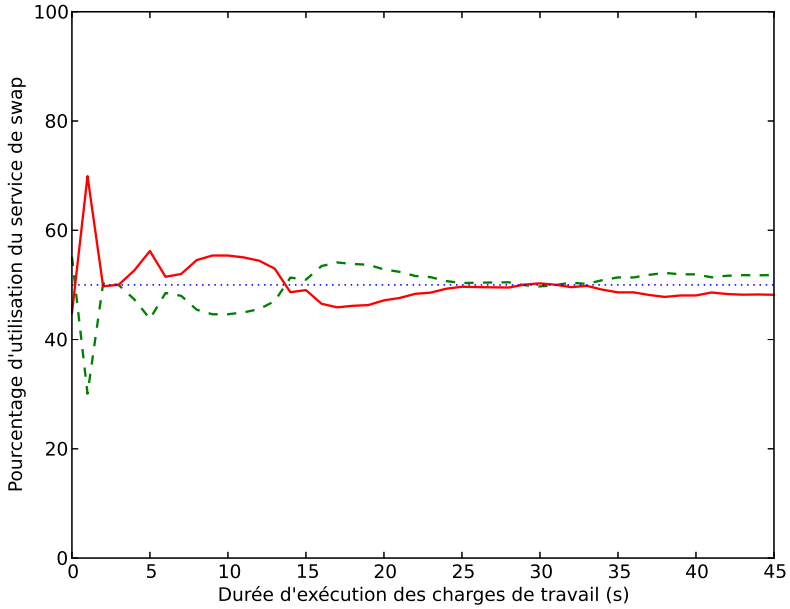
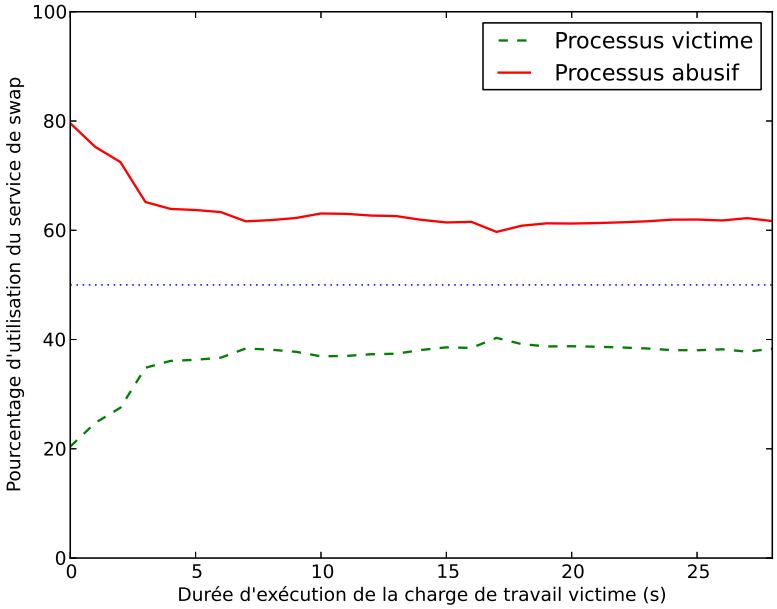


FIGURE 4.5 – Distribution de l'utilisation du système de swap lors de l'exécution simultanée de deux charges de travail similaires.

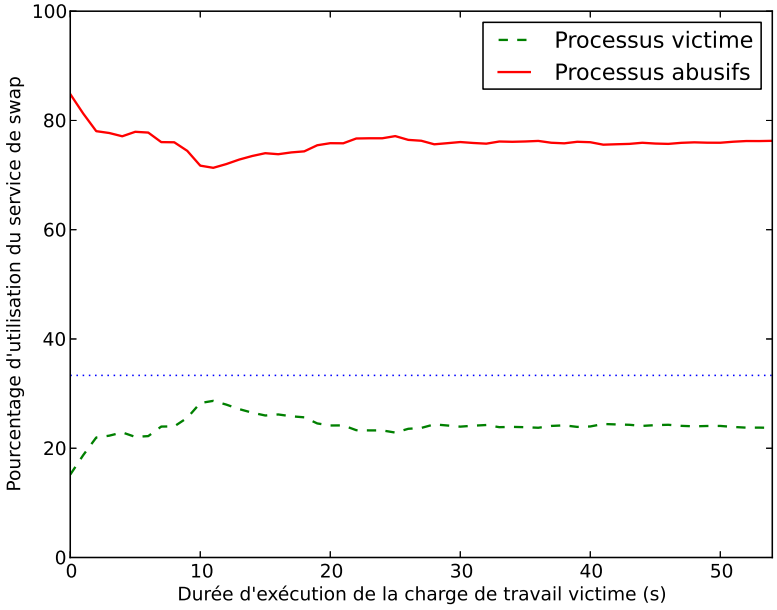
Afin d'étudier le cas d'une forte contention mémoire, nous avons reproduit une attaque par thrashing similaire à celle présentée dans la section 2.2.3. La figure 4.6 rapporte la répartition du temps du service de swap dans ce cas, avec un à deux processus abusifs effectuant une utilisation cumulée de 600 Mo de mémoire, contre 100 Mo pour le processus légitime.

La figure 4.6a montre que le processus abusif monopolise le système de swap plus de 60% du temps, bien que les requêtes disque sous-jacentes soient ordonnées par CFQ. En effet, afin de prolonger l'état de thrashing, l'attaquant a besoin de décharger les pages légitimes de manière efficace, et donc d'imposer un ratio de fautes de page par instruction le plus haut possible. Un processus légitime, qui effectue plus d'instructions de traitement de données entre les accès mémoire, ne va pas engendrer autant de fautes de page et donc induire moins de requêtes disque. Ceci implique une utilisation plus importante du système de swap pour un processus abusif qu'un processus légitime et la pression sur le système de swap est retranscrite directement sur le pourcentage d'utilisation du service.

La figure 4.6b illustre le fait qu'avec deux processus abusifs, le pourcentage d'utilisation du système de swap pour le processus légitime passe en dessous de 25%. Ceci expose deux problèmes notables avec la gestion de l'équité du



(a) Distribution de l'utilisation du système de swap entre un processus abusif et un processus légitime.



(b) Distribution de l'utilisation du système de swap entre deux processus abusifs et un processus légitime.

FIGURE 4.6 – Distribution de l'utilisation du système de swap lors d'attaques par thrashing.

temps de swap et des requêtes disque sous Linux, à base de CFQ uniquement. En premier lieu, un ordonnancement CFQ des requêtes ne suffit pas à forcer une équité effective de l'utilisation du disque, régie en grande partie par la fréquence des requêtes, et donc celle des fautes de page pour le cas du système de swap. De plus, l'ordonnancement CFQ, maintenant une file par processus, induit une inéquité importante dans les systèmes mutualisés où la balance doit être effectuée en priorité entre les utilisateurs et non les processus.

Bien qu'un processus abusif force un processus légitime à effectuer plus de fautes de page en temps de pression mémoire qu'en exécution isolée, le fait que celui-ci doive imposer une pression encore plus forte au mécanisme de swap rend la métrique de temps d'utilisation significative de l'abus.

FIGURE 4.7 – Distribution de l'utilisation du PRNG lors d'une attaque par assèchement.

4.1.4 Temps d'utilisation d'un générateur de nombres aléatoires

Le cas des ressources logicielles finies comme les générateurs de nombres aléatoires est différent, puisque les processus utilisateurs sont simplement mis en attente, pendant que des mécanismes d'arrière-plan approvisionnent la ressource pour le système entier. Il n'est donc pas possible de relier de manière directe

l'utilisation de la ressource à son réapprovisionnement ou son traitement. Cependant, dans ce cas, un utilisateur abusif doit nécessairement se placer en attente de la ressource plus fréquemment que les utilisateurs légitimes, si ces derniers sont victimes d'un assèchement.

Afin d'observer la distribution de l'utilisation du générateur de nombres aléatoires, nous avons suivi le rapport entre le temps d'attente de chaque utilisateur et le temps d'attente total sur le service du PRNG Linux pendant l'attaque par assèchement de la section 2.2.2. La figure 4.7 expose cette distribution, en fonction de la quantité d'information demandée par l'utilisateur légitime.

Cette figure expose de nouveau l'adéquation de la métrique de temps d'utilisation avec le cas de la génération de nombres aléatoires. En effet, on peut observer une utilisation plus importante de la part du processus abusif, et une monopolisation d'autant plus forte que le processus légitime demande une quantité plus faible de nombres aléatoires.

4.1.5 Discussion

L'usage du pourcentage du temps d'utilisation d'une ressource, qu'elle soit matérielle ou logicielle, nous permet donc de mettre en avant les phénomènes d'équité d'accès et de monopolisation. En effet, le temps d'utilisation est directement lié à la pression subie, ainsi que la seule métrique réellement représentative du travail imposé aux ressources du système, indépendamment de valeurs arbitraires comme la bande passante, la quantité d'information lue ou le nombre d'appels à la ressource. Le calcul de cette métrique dépend du type de la ressource suivie : il se fait de manière intuitive et simple pour le CPU, il utilise les évènements au plus près du traitement matériel pour les entrées/sorties ainsi que dans le cas du mécanisme de swap, et suit le temps total de résolution d'une requête dans le cas d'une ressource logicielle dépendante d'évènements externes sporadiques.

4.2 Partage de ressources équitable basé sur le temps d'utilisation

Cette métrique permet la mise en place d'une facturation générique et de politiques d'utilisation des ressources simples et dénuées de valeurs arbitraires spécifiques aux ressources supervisées. Dans cette section, nous introduisons les couches architecturales nécessaires au calcul du temps d'utilisation dans un cas général, ainsi que des exemples de politiques équitables d'accès aux ressources.

4.2.1 Modèle générique de consommation d'une ressource

Dans cette section, nous désignons l'unité d'exécution d'un système par le terme générique *application*. La définition d'une couche de facturation se repose en

premier lieu sur son unité de facturation. Comme souligné par les travaux précédents présentés dans la section 3.2, l'unité de facturation doit être plus générique que l'unité d'exécution, de manière semblable aux tasks Mach 3 [Loe92], containers de Banga *et al.* [BDM99] ou cgroups Linux.

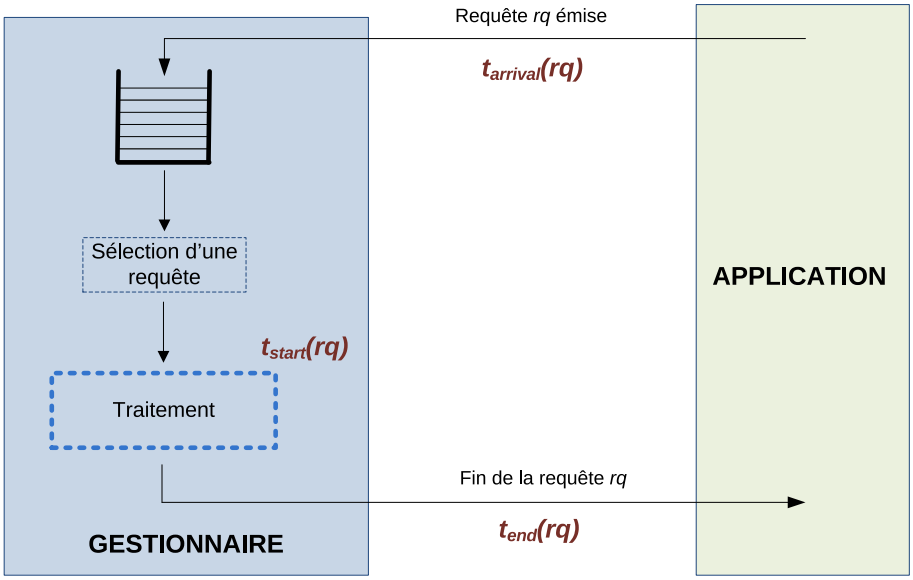


FIGURE 4.8 – Modèle générique de consommation d'une ressource.

Nous retenons ici l'appellation *domaine de facturation*, employée notamment par Stoess *et al.* [SU06], pour désigner un groupement arbitraire d'applications. La définition effective des domaines de facturation est directement liée au contexte applicatif : chaque application peut être liée à un domaine dans les systèmes personnels, les systèmes mutualisés vont eux considérer comme un même domaine l'ensemble des applications appartenant à un utilisateur ou groupe d'utilisateurs, tandis que dans les environnements virtualisés, un domaine regroupera les applications liées à une même machine virtuelle.

Les applications consomment les ressources par le biais de *requêtes*. Les requêtes peuvent être explicites, comme dans le cas des entrées/sorties, ou implicites, comme une faute de page pour le mécanisme de swap ou le passage à l'état "prêt à l'exécution" impliquant une demande du processeur. Chaque requête est liée au domaine de facturation de l'application émettrice, noté $Domain(rq)$.

Ces requêtes sont traitées par un *gestionnaire de ressource*. Le gestionnaire maintient une file de requêtes en attente en entrée, et sélectionne arbitrairement les requêtes à transmettre au bloc de *traitement interne*. Pour les ressources les plus simples comme le processeur, la file d'attente en entrée de la ressource est une FIFO et le bloc de traitement interne suit un algorithme simple et ne

résout qu'une requête à la fois. Pour le cas des entrées/sorties par exemple, la procédure de sélection effectuée par l'ordonnanceur d'entrées/sorties peut décider de réordonnancements. Le bloc de traitement interne correspond alors aux échanges avec le périphérique matériel, maintenant sa propre file d'attente et entraînant potentiellement un réordonnancement supplémentaire imprévisible pour le système d'exploitation.

Les requêtes sont temporalisées, c'est-à-dire qu'il est possible de déterminer les dates des trois évènements du parcours d'une requête rq : la date de soumission de la requête au gestionnaire notée $t_{arrival}(rq)$, la date de transmission d'une requête au bloc de traitement interne du gestionnaire, $t_{start}(rq)$, et enfin la date de fin de traitement de la requête par le gestionnaire $t_{end}(rq)$.

La figure 4.8 rapporte ces différentes notions de notre modèle de consommation d'une ressource.

4.2.2 Facturation générique de l'utilisation des ressources

Les couches de facturation existantes dans les systèmes traditionnels rapportent l'usage des ressources d'une manière spécifique à chaque type. Par exemple, les cgroups utilisés dans Linux ont besoin d'un nombre de contrôleurs important pour suivre les différentes ressources du système : les entrées/sorties de type bloc sont gérées par le module *blkio* qui suit notamment la bande passante et le nombre d'octets transmis, le processeur est géré par le couple *cpusets* et *cpuacct*, le trafic réseau par *net_cls*, etc...

Le temps d'utilisation nous permet au contraire de définir une couche de facturation générique de la consommation des ressources.

Facturation juste du temps d'utilisation. Les attaques sur les ordonnancements à base de facturation [TEF07, ZGDS11] nous ont appris qu'une facturation juste doit être effectuée à grain fin et pilotée directement par les évènements de début et de fin d'exploitation d'une ressource. Comme montré dans la section 4.1, la facturation du temps d'utilisation d'une ressource peut être effectuée à grain fin pour des ressources très diverses, cependant cette facturation ne se base pas sur les mêmes évènements pour chaque type de ressource.

En effet, la facturation du temps d'utilisation implique notamment la connaissance de la date de l'évènement de soumission effective de la requête au composant central de traitement de la ressource, par exemple le périphérique dans le cas des entrées/sorties détaillé dans la section 4.1.2. La remontée de cet évènement à une couche de facturation indépendante requiert ainsi une légère modification des gestionnaires de ressources, à des noeuds d'exécution d'ores et déjà identifiés par les couches de facturation existantes dans les systèmes d'exploitation classiques. Cependant, pour les ressources dont la gestion n'impose pas de réordonnancements, les évènements de soumission et de fin de requête suffisent à une facturation valide.

La figure 4.9 représente la position d'une telle couche de facturation centralisée du temps d'utilisation d'une ressource.

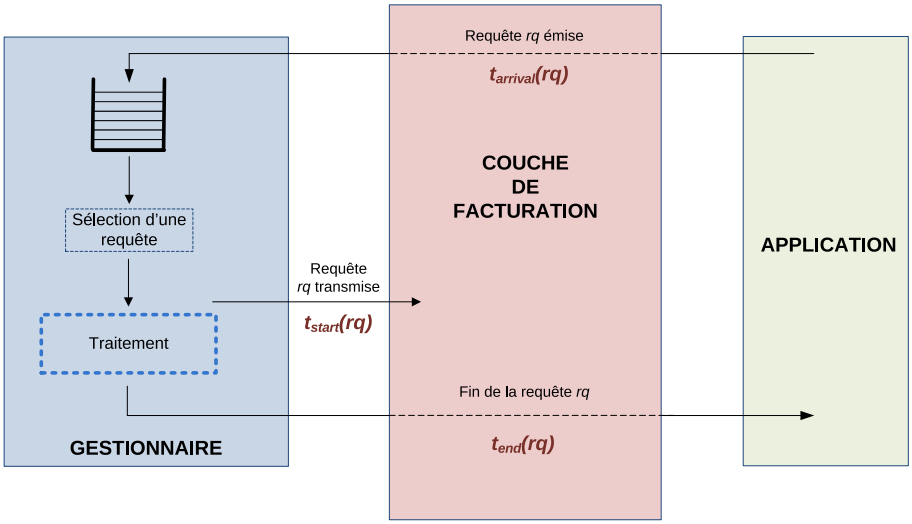


FIGURE 4.9 – Couche de facturation du temps d'utilisation.

Notre couche de facturation se positionne donc en interception des requêtes afin de déduire trivialement les dates $t_{arrival}(rq)$ et $t_{end}(rq)$ d'une requête rq. La date $t_{start}(rq)$ est transmise par le gestionnaire de ressource, et est égale à $t_{arrival}(rq)$ par défaut. Ceci permet de directement supporter les ressources ne pratiquant pas de réordonnancements. Les dates $t_{start}(rq)$ et $t_{end}(rq)$ suffisent à la déduction de l'utilisation effective du temps d'une ressource en présence ou non de réordonnancement, comme décrit dans la section 4.1.2.

Algorithme de facturation. L'algorithme 4.10 représente l'implémentation simplifiée d'une facturation juste et fine du temps cumulé d'utilisation de la ressource pour chaque domaine.

Cet algorithme présente ainsi les procédures liées aux trois évènements de soumission de requête, début de traitement et fin de traitement présentés sur la figure 4.9. L'utilisation par le gestionnaire de ressource de la procédure DÉBUT_TRAITEMENT() est optionnelle pour les ressources les moins complexes n'opérant pas de réordonnancements, puisque le début de traitement est alors égal à la date de soumission de la requête. Ces dates permettent de déduire la durée effective d'utilisation de la ressource par la requête rq, notée $Duration(rq)$ lors de la fin de traitement de celle-ci. Il est ensuite possible de facturer cette durée aux domaines liés à la requête rq, en ajoutant $Duration(rq)$ à la durée totale d'utilisation pour un domaine D, $Bill(D)$. La somme des durées d'utilisation par tous les domaines, notée S, est également mise à jour. La part d'utilisation de la ressource pour un domaine D, $Share(D)$, est ensuite aisément calculée puisqu'égale au rapport $\frac{Bill(D)}{S}$. Il est à noter que cette couche de facturation se pose en interception des requêtes pour une ressource

Algorithme 4.10 Facturation fine de l'utilisation du temps d'une ressource dirigée par les évènements.

procedure SOUMISSION_REQUÊTE(rq)
  $t_{arrival}(rq) \leftarrow now()$
  $t_{start}(rq) \leftarrow now()$
  Transmettre rq au gestionnaire de ressource
end procedure

procedure DÉBUT_TRAITEMENT(rq)
  $t_{start}(rq) \leftarrow now()$
end procedure

procedure FIN_TRAITEMENT(rq)
  $t_{end}(rq) \leftarrow now()$
  $Duration(rq) \leftarrow t_{end}(rq) - max(t_{start}(rq), t_{last})$
  $t_{last} \leftarrow now()$
  $D \leftarrow Domain(rq)$
  $Bill(D) \leftarrow Bill(D) + Duration(rq)$ ▷ Facturation au domaine
  $S \leftarrow S + Duration(rq)$
  for all $X \in AllDomains()$ **do**
    $Share(X) \leftarrow \frac{Bill(X)}{S}$
  end for
  Transmettre l'évènement de fin de requête rq à l'application
end procedure

en particulier, et doit être dupliquée devant chaque ressource différente dont la consommation doit être facturée.

La facturation du temps d'utilisation d'une ressource est ainsi implémentable de manière simple et générique, en imposant une modification minimale à certains gestionnaires de ressource afin de connaître la date de début de traitement réel d'une requête. La liaison d'une requête à un domaine de facturation, ainsi que la prise en compte de la part d'utilisation de chaque domaine, dépendent de la politique de gestion des ressources souhaitée.

4.2.3 Politique de partage de ressources transparente

Cette architecture de facturation peut être aisément améliorée afin de réguler l'utilisation de la ressource par les différents domaines concurrents. En effet, l'utilisation facturée aux domaines permet de discriminer ceux qui sont *abusifs* de ceux qui ne le sont pas, selon une politique de partage arbitraire. L'application de cette politique requiert l'ajout de deux composants simples à l'archi-

tecture de facturation décrite par la figure 4.9 : un composant responsable du contrôle d'accès et du blocage des requêtes liées à un domaine abusif, en amont de la couche de facturation, ainsi qu'un composant de mise à jour du statut abusif ou non de chaque domaine, en aval.

Ceci permet l'implantation d'une politique de partage de manière simple et indépendante des mécanismes internes des gestionnaires, comme le traitement et l'ordonnancement.

Éviter la famine du gestionnaire de ressource. Un point souvent délaissé par les couches de respect des politiques de partage est la maximisation de l'exploitation dans le cas général, qui doit empêcher les états de famine du gestionnaire. Notre facturation en pourcentage d'utilisation n'embarque pas la notion de capacité de traitement du gestionnaire de la ressource : deux domaines peuvent ainsi avoir consommé respectivement 20% et 80% du temps d'utilisation d'une ressource, tout en n'ayant exploité qu'une faible part de la capacité de celle-ci. Dans ce cas, les requêtes "abusives" ne doivent pas nécessairement être bloquées puisqu'elles n'en pénalisent pas d'autres.

Pour éviter cette famine du gestionnaire, nous suivons le nombre de requêtes en cours de traitement pour chaque domaine D, noté $PendingRq(D)$. A l'arrivée d'une requête liée à un domaine abusif, la couche de contrôle vérifie s'il existe au moins une requête en attente émise par un domaine plus raisonnable. Si c'est le cas, la requête est bloquée, garantissant le traitement de celles en attente et une mise à jour de la facturation. Si ce n'est pas le cas, la requête peut être transmise, afin de ne pas la différer inutilement et indéfiniment.

Équité persistante de l'utilisation d'une ressource. L'algorithme 4.11 donne ainsi un exemple de l'application d'une politique répartissant de manière pondérée l'utilisation d'une ressource, en respectant un poids arbitraire $Weight(D)$ associé à chaque domaine D.

Le fait de placer le composant de contrôle en interception de la soumission de requête permet de bloquer la transmission d'une requête liée à un domaine abusif, en attendant que les autres domaines aient obtenu leur part d'utilisation de la ressource. En aval du traitement de la requête, après l'évaluation de $Share(D)$ pour chaque domaine D par la couche de facturation, le composant de mise à jour des statuts peut évaluer selon sa politique arbitraire pondérée le fait que chaque domaine soit abusif ou non.

Il est à noter que cet exemple précis requiert une intervention humaine, puisque nécessitant la détermination d'un poids arbitraire pour chaque domaine. Cette mise au point est envisageable dans un cadre contractuel où le prix d'une prestation est lié à une qualité de service quantifiée. Cependant, dans le cas générique d'un système partagé best-effort où l'on souhaite répartir la ressource de manière équitable entre chaque domaine, il suffit de leur assigner à chacun un poids égal à 1. Ainsi, la couche de contrôle ne transmet que les requêtes des domaines D respectant $Share(D) <= \frac{1}{N}$, avec N le nombre de domaines en concurrence sur la ressource. Ceci permet l'implémentation d'une

Algorithme 4.11 Partage pondéré de l'utilisation du temps d'une ressource.

procedure SOUMISSION_REQUÊTE(rq)
  $D \leftarrow Domain(rq)$
  while $Abusive(D) == True$ **and** $\exists X \in AllDomains() : (Share(X) < Share(D)$ and $PendingRq(X) > 0)$ **do** ▷ Contrôle anti-famine
    Placer rq en attente
  end while
  $PendingRq(D) \leftarrow PendingRq(D) + 1$
  Transmettre rq à la couche de facturation
end procedure

procedure FIN_TRAITEMENT(rq)
  $D \leftarrow Domain(rq)$
  $PendingRq(D) \leftarrow PendingRq(D) - 1$
  $WeightTotal \leftarrow \sum_{X \in AllDomains()} Weight(X)$
  for all $X \in AllDomains()$ **do** ▷ Mise à jour des statuts d'utilisation
    if $Share(X) > \frac{Weight(X)}{WeightTotal}$ **then**
      $Abusive(X) \leftarrow True$
    else
      $Abusive(X) \leftarrow False$
    end if
  end for
  Transmettre l'évènement de fin de requête rq à l'application
end procedure

politique de partage équitable de la ressource, sans administration ou configuration quantitative. Une telle politique maximise dynamiquement l'usage de la ressource dans les limites acceptables d'un partage équitable de celle-ci. Elle évite les possibilités de famine dues aux domaines abusifs, tout en n'imposant pas un arrêt définitif de l'exploitation de la ressource par ceux-ci.

Implémenter la mise en attente des requêtes. L'une des difficultés pratiques de la mise en place de cet algorithme est la mise en attente des requêtes. La nature centralisée des systèmes monolithiques permet naturellement une implémentation plus simple de ce contrôle d'accès lui-même centralisé, mais celle-ci demeure possible et efficace dans le contexte des micro-noyaux :

- Dans un système monolithique, les applications emploient des interfaces unifiées pour l'appel aux ressources : les exceptions logicielles comme les appels système, ou les interruptions matérielles comme les fautes de page. Ces appels sont résolus dans le contexte de l'application appelante. De cette manière, mettre en attente une requête synchrone abusive revient à bloquer l'appelant, tant que le domaine lié à la requête demeure abusif.

Une requête asynchrone abusive ne peut pas être placée en attente et doit être refusée, afin de respecter les attentes applicatives.

- Les micro-noyaux manient quant à eux des composants indépendants ayant des contextes d'exécution propres et communiquant par IPC synchrones. Utiliser une procédure SOUMISSION_REQUÊTE() bloquante engendrerait dans ce cas un déni de service pour les requêtes ultérieures. Une première solution est de placer les requêtes liées aux domaines abusifs dans une file d'attente spécifique, et différer la transmission de ces requêtes au moment où les domaines concernés perdent ce statut. Cependant, l'implémentation effective des files d'attente pose une problème d'espace mémoire dans le cas où elles sont dynamiques, et un problème de congestion semblable aux files CFQ démontré dans la section 2.2.1 dans le cas où elles sont bornées. Une deuxième solution est de créer un contexte d'exécution séparé pour la résolution de chaque requête, à la manière d'un serveur réseau, ce qui permet de bloquer le contexte d'exécution spécifique à la requête, comme pour le cas monolithique. Les requêtes ultérieures liées à un domaine en possédant déjà au moins une autre différée doivent être refusées, afin de limiter au nombre de domaines gérés la quantité de contextes d'exécution simultanés. Les mécanismes de prêt de ressources, issus des systèmes à micro-noyaux comme L4 [Lie95] ou EROS [SH02], permettent au contexte d'exécution principal de la couche d'interception de déléguer les droits de communication nécessaires à la gestion de la requête.

Équité périodique de l'utilisation d'une ressource. Telle quelle, la combinaison des algorithmes 4.10 et 4.11 applique une politique de partage persistante où l'utilisation de la ressource est facturée à un domaine de sa création à sa destruction. Dans certains cas, cette configuration n'est pas désirable, si l'on ne souhaite pas qu'un domaine qui était fortement abusif dans le passé mais a correctement auto-régulé son utilisation soit toujours fortement pénalisé lors d'une nouvelle contention.

Pour résoudre ce problème, il est nécessaire de restreindre la facturation sur une période commune τ, en ne prenant en compte au temps t que les requêtes ayant été traitées dans l'intervalle de temps $t - \tau$. Ceci impose la conservation de l'historique des requêtes sur la période τ pour chaque domaine, ainsi que l'ajout d'un démon de mise à jour périodique de la facture $Bill(D)$ associée à chaque domaine D. L'algorithme 4.12 applique cette technique.

Cet algorithme maintient notamment un ensemble $History(D)$ pour chaque domaine D, contenant les requêtes passées pour le domaine, et mis à jour de manière périodique par l'appel à la procédure FACTURATION_PÉRIODIQUE(). L'ajout de cette procédure décharge également la procédure de fin de traitement de la complexité algorithmique de mise à jour de la part d'utilisation et du statut de chaque domaine.

Différer la complexité est souhaitable pour deux raisons pratiques. En premier lieu, la fin de traitement des requêtes est potentiellement exécutée dans

Algorithme 4.12 Partage du temps d'utilisation d'une ressource sur une périodique arbitraire τ.

procedure SOUMISSION_REQUÊTE(rq)
  $D \leftarrow Domain(rq)$
  while $Abusive(D) == True$ **and** $\exists X \in AllDomains() : (Share(X) < Share(D)$ and $PendingRq(X) > 0)$ **do**
    Placer rq en attente
  end while
  $t_{arrival}(rq) \leftarrow now()$
  $t_{start}(rq) \leftarrow now()$
  $PendingRq(D) \leftarrow PendingRq(D) + 1$
  Transmettre rq au gestionnaire de ressource
end procedure
procedure DÉBUT_TRAITEMENT(rq)
  $t_{start}(rq) \leftarrow now()$
end procedure
procedure FIN_TRAITEMENT(rq)
  $D \leftarrow Domain(rq)$
  $PendingRq(D) \leftarrow PendingRq(D) + 1$
  $t_{end}(rq) \leftarrow now()$
  $Duration(rq) \leftarrow t_{end}(rq) - max(t_{start}(rq), t_{last})$
  $t_{last} \leftarrow now()$
  $History(D) \leftarrow History(D) \cup \{rq\}$
  Transmettre l'évènement de fin de requête rq à l'application
end procedure

procedure FACTURATION_PÉRIODIQUE()
  $S \leftarrow 0$
  $WeightTotal \leftarrow 0$
  for all $D \in AllDomains()$ **do** ▷ Facturation sur $[now() - \tau, now()]$
    $Bill(D) \leftarrow 0$
    for all $rq \in History(D)$ **do**
      if $now() - Duration(rq) > \tau$ **then**
        $History(D) \leftarrow History(D) \setminus \{rq\}$
      else
        $Bill(D) \leftarrow Bill(D) + Duration(rq)$
      end if
    end for
    $S \leftarrow S + Bill(D)$
    $WeightTotal \leftarrow WeightTotal + Weight(D)$
  end for
  for all $D \in AllDomains()$ **do** ▷ Mise à jour des statuts
    $Share(D) \leftarrow \frac{Bill(D)}{S}$
    if $Share(D) > \frac{Weight(D)}{WeightTotal}$ **then**
      $Abusive(D) \leftarrow True$
    else
      $Abusive(D) \leftarrow False$
    end if
  end for
end procedure

un contexte d'interruption matérielle pour un nombre important de ressources. Ces contextes d'interruption opèrent en exclusion mutuelle, ce qui signifie que deux fins de traitement ne peuvent être évaluées de manière simultanée et qu'il est difficile de manipuler des structures de données partagées nécessitant des mécanismes de synchronisation. Déporter les blocs algorithmiques complexes permet une conclusion rapide de l'interruption matérielle et réduit le nombre de structures de données à manier.

D'autre part, une mise à jour des parts d'utilisation et statut des domaines à chaque fin de traitement de requête saccade les exécutions, car un domaine abusif va de manière successive obtenir une part d'utilisation légèrement inférieure puis légèrement supérieure à la limite autorisée. Une transmission saccadée des requêtes réduit fortement les performances en sous-utilisant la capacité des files d'attente et heuristiques de performance globales potentiellement employées par le gestionnaire. Ainsi, différer et grouper la facturation des requêtes introduit une propriété d'hystérésis souhaitable pour une performance maximisée de la couche de facturation.

Dans cette configuration, la période τ est le seul paramètre nécessaire à la mise en oeuvre du partage équitable des ressources. τ est réglé en fonction de la ressource sous-jacente, et doit être d'un ou deux ordres de grandeur supérieurs au temps moyen de traitement d'une requête, afin de capter les variations sur un échantillon significatif. Ceci implique des périodes allant de quelques dizaines de millisecondes à une seconde pour la majorité des ressources. Par exemple, Kim *et al.* [KKC12], utilisent une période de 30 millisecondes dans leurs travaux sur l'équité des entrées/sorties dans Xen. Certains cas de ressources potentiellement lentes, comme les nombres aléatoires, peuvent exiger des périodes de l'ordre de quelques minutes.

4.2.4 Discussion

La facturation du temps d'utilisation d'une ressource est implémentable en ramenant l'acte de consommation à un modèle simple où une application émet explicitement ou implicitement un certain nombre de requêtes au gestionnaire de la ressource en question. Pour certaines ressources complexes comme les entrées/sorties, il est nécessaire de modifier légèrement les gestionnaires afin de connaître la date effective de transmission de la requête au bloc de traitement, et ainsi éviter une facturation erronée due aux réordonnancements.

A partir de ce modèle, nous avons défini une couche générique de facturation et de contrôle, fonctionnant en interception de ces requêtes. Cette architecture permet l'implémentation de politiques de partage équitable, dénuées de paramètres arbitraires dédiés, comme les quotas en termes de bande passante par exemple ; déterminer le pourcentage de temps d'utilisation de chaque domaine ouvre la porte à une discrimination entre les domaines qui sont abusifs et ceux consommant la ressource de manière plus raisonnée. Les requêtes liées à un domaine abusif sont différées, jusqu'à ce que les autres aient consommé leur part légitime d'utilisation de la ressource. La centralisation des données de facturation permet également de maximiser l'usage de la ressource contrôlée, en

autorisant tout de même un domaine abusif à transmettre des requêtes, lorsque les domaines ayant moins exploité la ressource n'en ont pas besoin.

4.3 Architecture de contrôle de ressources transparente et générique

L'emploi d'une métrique générique, transverse aux différentes ressources, permet de concevoir une architecture unifiée d'accès et de contrôle des ressources dans les systèmes d'exploitation, et ce, de manière transparente pour les applications. Dans cette section, nous exposons les différentes briques architecturales nécessaires et nous détaillons sa mise en application, notamment au sein de systèmes à micro-noyaux et monolithiques.

Accès aux ressources selon les architectures des systèmes d'exploitation. Les architectures matérielles communes fournissent différents niveaux de privilège des processeurs, afin de permettre aux systèmes d'exploitation de réserver l'accès et la configuration effective des ressources matérielles à une base d'instructions de confiance. Dans les systèmes monolithiques comme Linux, le noyau seul opère en mode privilégié, et rassemble toutes les couches nécessaires à la gestion de ressources. A l'opposé, les systèmes à micro-noyaux, parmi lesquels EROS [SH02] ou la famille L4 [Lie95], font l'effort de définir les concepts minimaux devant être implémentés en mode privilégié, afin de déléguer les autres traitements à des gestionnaires dédiés et en leur attribuant le minimum de privilèges requis.

La figure 4.13 détaille l'accès aux ressources et services du système dans les deux cas extrêmes des architectures monolithiques et des noyaux de type L4 :

- Système monolithique. Lorsqu'une application a besoin d'accéder à une ressource quelconque, le noyau est informé via une exception logicielle comme un appel système, ou une interruption matérielle, par exemple une faute de page. Le noyau peut ensuite résoudre en interne la requête de l'application, par exemple via son interface unifiée d'accès aux entrées/sorties, qui après d'éventuels ordonnancements internes va finalement transmettre la requête au pilote du périphérique.

- Système à micro-noyaux L4. Au contraire, les noyaux à la L4 ne supportent que quatre fonctionnalités jugées minimales : la gestion de la mémoire, la gestion de l'unité d'exécution, la communication inter-processus (IPC) et les *capabilities*. Les capabilities représentent un identifiant non forgeable, associé à un certain nombre de droits d'accès sur les actions implémentées par le noyau. A son initialisation puis lors de son exécution, chaque composant se voit attribuer un ensemble de ces capabilities, qui lui fournissent par exemple la capacité de communiquer avec d'autres composants, de gérer un ensemble de pages mémoire spécifiques ou de gérer le cycle de vie de certains threads. Les pilotes s'exécutent dans un contexte d'exécution dédié, et possèdent les capabilities d'accès nécessaires à la

gestion de leur ressource, par exemple les plages de mémoire physique de configuration d'un périphérique. Pour qu'une application puisse effectuer une requête, elle doit posséder les capabilities de communication au pilote.

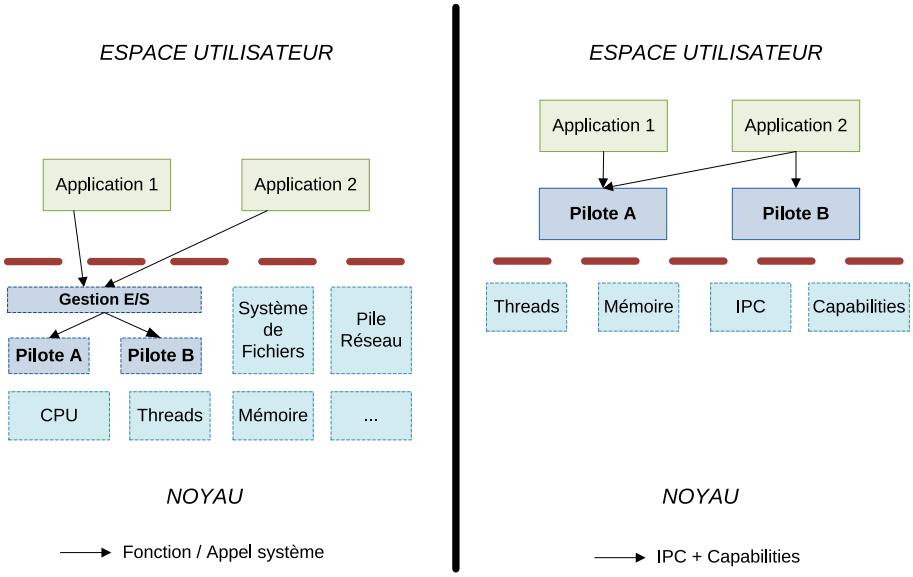


FIGURE 4.13 – Accès aux ressources dans les systèmes monolithiques (à gauche) et les systèmes à micro-noyaux L4 (à droite).

Dans la pratique, les capabilities d'accès aux ressources matérielles sont souvent toutes attribuées à un composant unique, l'annuaire de ressources, qui va ensuite déléguer les droits nécessaires à l'utilisation de chaque ressource à d'autres composants de confiance comme les pilotes, appelés les *serveurs*. Les applications souhaitant l'accès à une ressource en particulier effectuent donc une requête à l'annuaire de ressources, qui peut leur fournir les capabilities nécessaires à la communication avec le serveur abritant le pilote concerné.

Contrôle générique, transparent et automatisé dans les micro-noyaux. Cette architecture de communication est très intéressante pour l'insertion de couches d'interposition transparentes entre différents composants, puisque les applications ne connaissent les serveurs que par les capabilities qui leur sont attribuées. Vogt *et al.* profitent par exemple de ce paradigme pour ajouter des composants à l'interface entre les applications et les serveurs, qui sont capables de détecter les états d'erreur des serveurs et d'opérer une réinitialisation transparente pour l'application finale [VDL10]. De la même manière, notre couche

de contrôle générique intercepte les requêtes des applications à destination des serveurs, afin d'effectuer une facturation et un contrôle transparents du temps d'utilisation de chaque ressource. Notre architecture complète de gestion de ressources est exposée sur la figure 4.14.

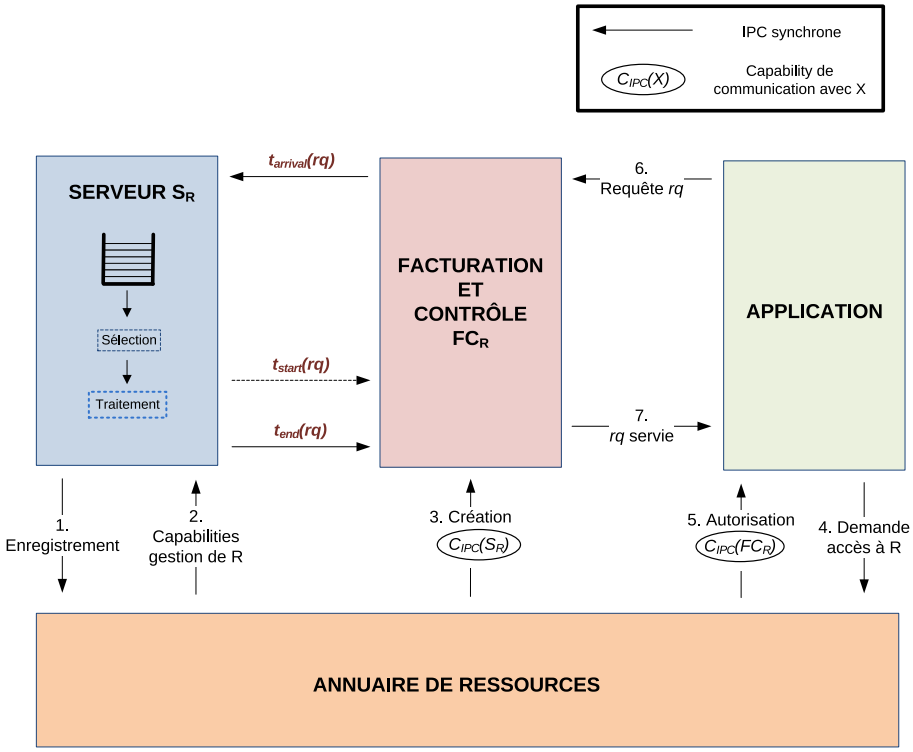


FIGURE 4.14 – Architecture générique de gestion et de contrôle de l'utilisation des ressources dans les micro-noyaux.

Dans cette configuration, l'annuaire de ressources centralise les accès aux différentes ressources du système et est accessible par tous les composants applicatifs. La mise en place et l'accès aux ressources sont régis par un protocole simple :

1. Enregistrement de S_R, serveur de gestion de la ressource R. Cette étape est généralement dérivée de la configuration du système, mais peut également être opérée à l'exécution, par exemple dans le cas de ressources logicielles comme les générateurs de nombres aléatoires.

2. Délégation de la gestion de R à S_R. Si l'annuaire accepte l'enregistrement de la ressource, il délègue les capabilities nécessaires à sa gestion. Ceci

peut représenter l'accès à des plages de mémoire physique spécifiques pour les périphériques matériels, ou la communication avec d'autres composants, par exemple le serveur gérant le disque dans le cas du mécanisme de swap.

3. Instanciation de la couche de facturation et de contrôle. Comme décrit dans la section 4.2, la facturation et le contrôle sont uniques à chaque ressource. L'annuaire de ressources transmet à cette couche les capabilities nécessaires à la communication avec le serveur S_R. Seule la couche de facturation est ainsi capable de communiquer avec celui-ci.

4. L'application souhaite accéder à R. Elle effectue une demande d'accès au composant de gestion des ressources, supposé connu par l'ensemble des composants du système.

5. L'annuaire accepte la demande. Il transfère à l'application les capabilities nécessaires à la communication avec la couche de facturation et de contrôle.

6. L'application transmet sa requête rq, destinée de son point de vue au serveur S_R. La requête est interceptée et traitée de manière transparente par la couche de facturation et de contrôle, éventuellement différée selon la politique de gestion de la ressource souhaitée. Du point de vue de l'application, il n'y a pas de différence entre un délai dû à l'ordonnancement interne du serveur S_R, et une attente imposée par la couche de contrôle.

Utiliser métrique de facturation et politique de partage génériques ouvre ainsi la possibilité d'une gestion centralisée et automatisée des ressources, permettant une protection des serveurs sans que ni eux-mêmes, ni les applications n'en soient nécessairement conscients.

Contrôle générique et transparent dans les systèmes monolithiques. Bien que la définition de cette architecture se base sur les concepts propres aux micro-noyaux, elle reste implémentable dans le cadre de systèmes plus monolithiques. Dans ce cas, c'est le caractère automatique de notre architecture qui se trouve fragilisé, puisqu'il n'existe pas de couche de communication unifiée interceptant la gestion de tous les types de ressources.

Porter cette architecture vers un système monolithique impose essentiellement l'insertion explicite, pour chaque ressource gérée, d'appels aux fonctions de la couche de facturation et de contrôle, à l'arrivée et à la fin du traitement de chaque requête. Cependant, ces points particuliers sont souvent bien identifiés, puisque les couches de facturation existantes, comme le cgroups sous Linux, y insèrent leurs propres fonctionnalités de suivi des ressources. Le problème de la mise en attente de requêtes liées à un domaine abusif est toutefois simplifié dans ce cas, car la réception des requêtes est toujours exécutée dans le contexte de l'application cliente. Cette particularité permet à la couche de contrôle de n'avoir qu'à bloquer le processus courant, en attente du changement de statut de son domaine de facturation.

4.4 Conclusion

Dans ce chapitre, nous avons présenté notre approche au problème de la gestion des ressources dans les systèmes partagés best-effort. Elle consiste à se reposer sur les techniques de maximisation de l'utilisation des ressources embarquées dans les pilotes et autres composants de gestion des ressources, tout en permettant une implémentation facilitée de politiques de distribution équitable des ressources.

Nous introduisons ainsi une métrique unique pour la caractérisation de l'usage des ressources : le temps d'utilisation. Nous montrons notamment que cette métrique, déjà couramment usitée pour le partage du temps processeur, peut s'adapter aux autres ressources, à condition de la calculer au plus près possible du traitement et en prenant en compte les possibilités de réordonnancement. A partir de cette facturation juste et à grain fin, il est possible d'exprimer des politiques de partage équitable, sans avoir à spécifier de quotas arbitraires ou spécifiques à la ressource supervisée.

La facturation et le contrôle des ressources via le temps d'utilisation peuvent être effectués en interception des requêtes, de manière transparente pour les composants de gestion comme pour les applications clientes. Ceci nous a permis de spécifier une architecture générique de contrôle de ressources dans les systèmes à micro-noyaux qui protège les gestionnaires de manière automatisée et sans possibilité de contournement par les applications. Les différents éléments de cette architecture demeurent cependant implémentables dans les systèmes monolithiques en conservant leur potentiel de partage équitable et de maximisation de l'utilisation des ressources, mais perdent cette caractéristique d'automatisation.

Afin d'établir l'apport d'une facturation et d'un contrôle basés sur le temps d'utilisation des ressources, nous nous intéressons dans les deux chapitres suivants à une ressource souvent ignorée par les couches de gestion des ressources : la mémoire virtuelle et son mécanisme de swap. Cette ressource est particulièrement intéressante car la mémoire, virtuellement illimitée, est centrale dans la performance des systèmes informatisés, mais dissimule une grande inconnue en termes de qualité de service lors de surcharges importantes.

Chapitre 5

Utilisation partagée de la mémoire

Dans ce chapitre, nous nous focalisons sur les mécanismes de gestion de la mémoire, ressource à la fois singulière car nécessairement sur-allouée, et critique car centrale dans l'exécution des applications et des systèmes. Ces particularités ont justifié l'apport de mécanismes complexes afin d'optimiser et de simplifier son utilisation par les applications et de permettre le travail sur des zones de plus en plus importantes. De plus, son allocation a priori et les accès directs au matériel ne permettent pas aux systèmes d'exploitation classiques de s'interposer afin de réguler les flux.

En termes d'impact sur les interférences de performance entre plusieurs applications concurrentes, deux mécanismes sont particulièrement problématiques : les caches matériels et le système de swap. Le mécanisme de swap, permettant le support de l'adressage virtuel, constitue l'un des goulots d'étranglement les plus sévères lors de fortes contentions mémoire. Les techniques d'optimisation du remplacement de pages, en cache ou en swap, ont vu le jour afin de gérer au mieux les charges de travail communes dans les systèmes best-effort, mais peuvent causer un effondrement dramatique des performances lorsque des charges de travail antagonistes s'exécutent en parallèle.

Certains travaux ont même montré, à l'encontre des principes d'équité, que la favorisation de l'utilisation mémoire par les charges de travail les plus abusives pouvait soulager le système à moyen terme. De telles techniques sont naturellement limitées dans les systèmes partagés où les charges abusives permanentes vont s'exécuter au détriment de charges raisonnables. La gestion de la contention mémoire peut être améliorée dans le cas des environnements de virtualisation, qui peuvent implémenter des mécanismes permettant d'adapter dynamiquement, à l'exécution, la quantité de mémoire allouée aux systèmes invités, afin de maximiser l'utilisation de la mémoire tout en minimisant les appels au mécanisme de swap.

5.1 Impact des caches matériels

En premier lieu, les particularités matérielles de certains types de mémoire comme la DRAM permettent à plusieurs processus s'exécutant de manière simultanée sur différents coeurs d'avoir des effets significatifs sur leurs performances respectives, si l'un d'entre eux accède à des domaines à très forte localité de manière intensive [MM07]. De plus, l'observation de l'utilisation du cache et la pollution de ceux-ci peut permettre d'avoir un impact néfaste significatif sur des charges concurrentes, voire même de porter atteinte à la confidentialité des données [Per05]. Enfin, les attaques par libération de ressources [VKF+12] peuvent permettre à un attaquant de monopoliser l'utilisation du cache en forçant une charge de travail victime à atteindre ses quotas de consommation sur d'autres ressources système, dégradant de manière significative ses performances tout en boostant celles de l'attaquant.

L'utilisation de ces caches est un mécanisme critique de la performance d'un système, par conséquent leur usage est régulé essentiellement par le matériel. Toute interposition logicielle sur la gestion de ces caches dégraderait de manière significative leur performance, et empoisonnerait potentiellement leurs données. Dans ce contexte, les seules solutions permettant de borner les interférences sont les groupements de charges de travail collaboratrices sur des coeurs indépendants ou la réservation de portions du cache pour chaque charge de travail [RNSE09].

Le partage des caches ne pouvant être résolus que de manière statique et dépendante de l'architecture matérielle, nous n'avons pas traité ce problème plus en détail dans notre travail.

5.2 Optimisation du remplacement de pages

Le concept de mémoire virtuelle, où le noyau expose aux applications une vision de la mémoire décorrélée de la réalité de l'utilisation de la mémoire physique, a permis une simplification de la gestion mémoire pour les applications. Le système d'exploitation maintient pour chaque processus une table de relations entre un ensemble d'adresses virtuelles et leurs adresses physiques réelles. Un composant matériel, l'unité de gestion mémoire (MMU), est ensuite chargé de traduire les adresses virtuelles utilisées par les applications en adresses physiques de manière transparente, en se servant de ces tables de relation.

Dans un contexte où la taille des mémoires centrales était très faible, l'adressage virtuel a notamment été possible grâce à l'introduction du mécanisme de swap [KELS62], permettant l'extension de la quantité de mémoire disponible à un système d'exploitation au-delà de la quantité de mémoire physique effectivement présente. Puisque seul le système d'exploitation connaît la position réelle de chaque page de mémoire virtuelle, il est libre d'en décharger une quantité arbitraire sur un espace de stockage annexe lorsque la mémoire physique vient à manquer. Lorsqu'une application accède à une adresse virtuelle valide qui n'est plus liée à une page en mémoire centrale, la MMU déclenche une faute de

page, synonyme de violation d'accès mémoire. Si le système reconnaît l'accès à une page qui a été déchargée, il peut replacer celle-ci en mémoire centrale et relancer l'instruction de manière transparente pour l'application.

Politiques de pagination. Il est donc nécessaire pour le système d'exploitation d'implémenter des politiques de pagination efficaces, lui permettant de sélectionner les pages à décharger et à charger de manière optimale et à faible coût algorithmique. Dans ce cadre, la difficulté principale pour le système d'exploitation réside dans l'algorithme de remplacement des pages, qui sélectionne les pages à décharger lorsqu'une contention mémoire intervient. Le remplacement de pages optimal, OPT, décrit par Belady [Bel66], déchargerait toujours la page utilisée le plus tard dans le futur. Cet algorithme n'étant pas implémentable en pratique sans une connaissance précise des motifs futurs d'accès mémoire, il est généralement approximé par les algorithmes LRU et LFU.

Ces deux algorithmes déchargent respectivement la page accédée la moins récemment, et celle utilisée la moins fréquemment dans le passé. Dans la pratique, les algorithmes LRU et LFU sont également approximés, puisqu'il n'est pas efficace de conserver le détail de tous les accès mémoire passés. Deux algorithmes implémentés très tôt en lieu et place de LRU sont NRU et Clock. NRU choisit arbitrairement une page parmi celles qui n'ont pas été référencées récemment, information accessible directement par le système d'exploitation en utilisant les bits d'accès des pages mémoire. Clock effectue quant à lui un remplacement FIFO, mais vérifie avec toute éviction si la page a été accédée récemment. Si c'est le cas, la page est réinsérée au début de la file, et le système essaie de décharger la suivante. Le remplacement de pages adaptatif (ARC) combine LRU et LFU afin d'obtenir un compromis empiriquement meilleur [MM04]. CAR, associant Clock et ARC, permet des performances comparables à ARC, mais également une gestion de cas supplémentaires spécifiques, comme l'itération séquentielle [BM04].

Remplacement de pages adaptatif. Cependant, ces algorithmes traditionnels de remplacement de pages, conçus à des fins d'optimisation globale de la marche du système, peuvent être contre-productifs en présence de charges de travail spécifiques [JZ02a]. Par exemple, LRU est inefficace pour la gestion d'accès séquentiels à de larges portions mémoire, alors qu'un algorithme MRU, déchargeant les pages les plus récemment accédées, devient optimal. Afin de résoudre ce problème, Midorikawa *et al.* ont proposé une extension de LRU, LRU-WAR [MPC08], qui détecte et adapte son heuristique de remplacement de pages lorsqu'une utilisation séquentielle de la mémoire est détectée [MPC08]. Ils se servent notamment du nombre de page accédées entre deux fautes pour déterminer une tendance séquentielle. Dans ce cas, LRU-WAR se base sur une liste MRU, ordonnant les pages par date d'accès, les pages accédées les plus récemment étant le plus haut dans la liste et étant déchargées en priorité. Cet algorithme a ensuite été étendu en considérant la fréquence d'accès aux pages, à la manière de LFU et ARC [CPM10]. Leurs expérimentations

affichent une gestion efficace des accès mémoire séquentiels, et une meilleure capacité de prédiction de l'utilisation de pages qui n'ont pas été récemment accédées. Cette approche, validée par simulation seulement, paraît cependant difficilement adaptable dans un environnement partagé, où les accès globaux effectués par l'ensemble des applications ne permettent pas nécessairement la détection de l'utilisation d'accès séquentiels dans une seule application.

Zhou *et al.* ont proposé l'utilisation d'une nouvelle métrique, le *Miss Ratio Curve* (MRC), afin de déterminer les demandes mémoire des applications de manière dynamique [ZPS+04]. Le MRC représente l'évolution du nombre de fautes de page d'une application, en fonction de la taille de la mémoire physique qui lui est allouée sur une période donnée. Son suivi permet ainsi de définir un optimum local pour chaque application, au-delà duquel l'ajout de mémoire physique supplémentaire n'aura pas d'impact significatif sur le pourcentage de fautes de page. Puisque les systèmes d'exploitation ne sont notifiés que des fautes de page et non des accès, le pourcentage de fautes de page n'est pas une métrique facilement déterminable par un OS. Zhou *et al.* exploitent les mécanismes de protection de page, afin de forcer l'apparition de fautes, à la manière d'un Copy-on-Write, et ainsi pouvoir suivre l'évolution du nombre de pages accédées. Ils intègrent ensuite notamment ces données à l'algorithme de remplacement de pages de Linux, afin de décharger en priorité les pages des applications dont le MRC sera le moins impacté. Les auteurs précisent que cette technique ne s'applique favorablement que pour les applications dont le MRC est convexe, c'est-à-dire ayant une utilisation stable de la mémoire sur le temps.

Discussion. L'implémentation du mécanisme de swap nécessite une politique de pagination adaptée, qui sélectionne arbitrairement les pages mémoire à décharger de la RAM lorsque le système est surchargé. L'algorithme théoriquement optimal, OPT, n'est pas implémentable en pratique, et par conséquent, différentes approximations ont été définies. Les algorithmes les plus courants sont dérivés de LRU, et cherchent à décharger les pages mémoire qui n'ont pas été accédées dans un passé récent. L'heuristique sous-jacente, à savoir le fait qu'une application consulte un nombre restreint de pages sur une période de temps donnée, n'est pas valide pour toutes les applications. D'autres recherches ont donc porté sur l'adaptation dynamique de la politique de pagination, en essayant de détecter les tendances d'accès à la mémoire, par exemple les accès séquentiels qui ne sont pas bien gérés par LRU. Ces techniques demeurant difficilement implémentables ou spécifiques de certaines charges de travail, les systèmes classiques utilisent toujours des algorithmes basés sur LRU, notamment Clock.

5.3 Réduction du thrashing

Comme démontré dans la section 2.2.3, une utilisation excessive et prolongée du mécanisme de swap peut faire chuter de manière dramatique les performances

générales d'un système, puisque la latence des accès mémoire devient liée à celle des accès au périphérique de stockage externe. Ce phénomène oblige notamment les hébergeurs à sur-provisionner les besoins mémoire de leurs clients afin de réduire voire d'empêcher l'utilisation du système de swap. Certains travaux ont d'ailleurs proposé le déchargement de la mémoire en excédent vers des systèmes proches ayant de la mémoire inutilisée, via le réseau [LNP05].

Le *working set* comme modèle de consommation mémoire. Les limites des mécanismes combinés d'adressage virtuel et de swap ne sont pas nouvelles et ont été traitées en profondeur depuis un demi-siècle. Denning explique dès 1968 [Den68b] le besoin pour un système d'exploitation multi-processus de maintenir le *working set* de chaque processus en mémoire centrale. Le working set $W(t,\tau)$ d'un processus P est défini comme l'ensemble des pages mémoire accédées par P à l'instant t sur l'intervalle $[t-\tau,t]$. Suivant le principe de localité des accès mémoire par le programme, $W(t,\tau)$ est une bonne estimation de $W(t+\alpha,\tau)$ pour $\alpha << \tau$. Denning montre notamment que la connaissance de $w(t,\tau)$, égal au nombre de pages présentes dans l'ensemble $W(t,\tau)$, suffit à une gestion efficace de l'allocation des pages de mémoire physique et à la mise en place d'une politique efficace de partage de la mémoire et du processeur. Cependant, la maintenance du working set de chaque processus ainsi que l'estimation de la période τ idéale ne peuvent être implémentées de manière efficace qu'à l'aide de mécanismes matériels dédiés [Den68b, Mor72], non présents dans les architectures communes.

A la rencontre de Clock, approximation de LRU axée sur l'efficacité et une optimisation globale de l'utilisation mémoire du système, et des algorithmes d'approximation du working set fournissant de meilleures propriétés d'isolation et de protection contre le thrashing, Carr et Henessy proposent WSClock [CH81]. WSClock ordonne les pages par une liste cyclique à la manière de Clock et les examine en FIFO, mais ajoute à chaque page p résidente en mémoire centrale une approximation de son dernier accès, notée $LR(p)$. $LR(p)$ est exprimée via le temps virtuel de chaque application, correspondant à la somme des durées pendant lesquelles le processus est à l'état d'exécution. A chaque examen d'une page p, WSClock observe la valeur du bit d'accès associé et le repositionne à 0. Si le bit d'accès était à 1, donc que la page p a été accédée depuis le dernier examen, $LR(p)$ se voit attribuer la valeur actuelle du temps virtuel de l'application, VT. Ceci permet ensuite à WSClock d'éviter le remplacement des pages pour lesquelles $VT - LR(p) > \tau$, avec τ la période d'échantillonnage du working set de l'application. WSClock fournit ainsi une approximation de LRU efficace, simple à implémenter et approximant le working set des applications afin d'optimiser et d'isoler un peu plus leurs performances individuelles.

Une autre possibilité est l'utilisation d'une politique de remplacement locale : à chaque faute de page, c'est une page encore en mémoire centrale appartenant au processus fautif qui est déchargée et remplacée. Ceci borne naturellement l'impact de chaque processus sur les autres, mais requiert des mécanismes de partitionnement ou de réservation de la mémoire spécifiques,

difficiles à configurer [Laz79] et ne pouvant s'adapter aux applications ayant des besoins dynamiques.

Réduction du niveau de concurrence. Au delà des algorithmes et politiques de remplacement de cache, les travaux passés ont notamment étudié la possibilité de réduire le niveau de concurrence d'un système, ou *Multi-Programming Level* (MPL). Comme observé initialement par Denning [Den68a], il est possible d'éviter tout thrashing si la somme des working sets des applications en cours d'exécution est plus petite ou égale à la quantité de mémoire physique disponible. Lorsqu'une contention mémoire intervient, réduire le MPL signifie réduire le nombre de working sets concurrents, donc d'applications s'exécutant simultanément. Denning a ainsi proposé un modèle de suspension de threads afin d'éviter le thrashing, en ajoutant une file de threads inactifs, et en réévaluant périodiquement l'ensemble des threads capables de s'exécuter afin de maximiser l'utilisation du CPU [Den80]. Certains systèmes d'exploitation ont adopté des mécanismes similaires afin de supprimer les possibilités de thrashing [RRD73, MBKQ96, HP03].

Plus récemment, Reuven *et al.* proposent de réduire le MPL par l'ajout d'un deuxième niveau d'ordonnancement des processus, au-dessus de l'ordonnanceur CPU [RW06]. Les processus sont groupés afin que chaque groupe requière le maximum de mémoire possible, dans les limites de la quantité de mémoire physiquement disponible. Ainsi, un ordonnanceur à moyen-terme peut exécuter chaque groupe tour à tour, en round-robin. Les processus du groupe choisi peuvent ensuite être ordonnancés de manière classique. Cette méthode élimine le thrashing de manière définitive, et le système ne paye le transfert de mémoire que lors du changement de groupe par l'ordonnanceur moyen-terme. Ceci ne peut cependant résoudre les problèmes posés par les processus demandant plus de mémoire que disponible en mémoire centrale, et ceux ayant une importante demande mémoire mais ne travaillant effectivement que sur des ensembles de pages réduits. De plus, le problème de groupement des processus, ou *bin packing*, est connu pour être NP-complet, ce qui rend en pratique difficile l'application de cet ordonnancement à moyen terme dans le cas où les charges de travail sont dynamiques.

Protection des processus intensifs : le swap token. L'approche novatrice de Jiang et Zhang [JZ02b] a choisi de concilier les deux approches en visant la réduction du MPL via une modification de l'algorithme de remplacement de page. Leur solution, TPF pour *Thrashing Protection Facility*, ne va pas chercher à offrir des garanties immédiates sur le MPL, mais à le réduire à moyen terme. Lors de la détection du thrashing, c'est-à-dire lorsque l'exploitation du CPU passe sous un seuil arbitraire, TPF choisit parmi les processus ayant produit un nombre de fautes de page au-dessus de la normale celui qui monopolise le moins de mémoire, et le protège. Un processus protégé ne contribue pas à la LRU tant que son seuil de fautes de page n'est pas revenu à la normale, c'est-à-dire que ses pages sont verrouillées en mémoire centrale. Cette approche permet

de réduire le nombre de fautes de page de manière effective dès l'apparition du thrashing, mais surtout de permettre aux applications intensives en mémoire de terminer leur exécution plus rapidement afin de diminuer le MPL à moyen terme. TPF peut cependant aggraver une situation où le processus abusif est perpétuel, et se base sur des seuils arbitraires d'utilisation CPU et de nombre de fautes de page difficiles à déterminer.

Leur proposition a évoluée avec l'introduction de l'algorithme *Token-ordered* LRU [JZ05]. L'algorithme Token-ordered ajoute un jeton de protection unique à l'algorithme de remplacement de pages. A chaque faute de page, le système vérifie si le jeton est déjà utilisé par un processus. Si ce n'est pas le cas, le jeton est affecté au processus ayant fait la faute de page. Lorsqu'un processus possède le jeton, il ne contribue par à la LRU conformément au principe de protection prôné par TPF. L'apport de Token-ordered LRU est l'ajout d'un mécanisme d'observation de l'usage du jeton : si le programme continue à créer un nombre important de fautes de page, cela signifie qu'il requiert plus de mémoire que disponible en mémoire centrale, le jeton lui est donc retiré ; de plus, un seuil de durée d'usage du jeton est défini au-delà duquel le jeton est automatiquement retiré. Dans ces deux cas, le processus est inéligible à l'usage du jeton pendant une période arbitraire. En s'affranchissant des seuils d'utilisation du CPU et du nombre de fautes de page, Token-ordered LRU permet une gestion des pics de demande mémoire dès l'apparition de fautes de page ainsi que de la mise au point de ces variables. Enfin, le contrôle dynamique de l'usage du jeton permet de diminuer l'impact de certains cas abusifs.

L'algorithme du remplacement de pages dans Linux a implémenté le jeton de la Token-ordered LRU sous l'appellation *swap token* à partir de sa version 2.6.9 [BC05, Jia09]. L'implémentation originale ne permet pas plus d'une affectation du jeton toutes les deux secondes, empêche un processus de le recevoir deux fois de suite, et les processus ne reçoivent le jeton que pour une période de temps réduite à un tick par défaut à partir de la version 2.6.11. Ces choix d'implémentation sont très discutables, puisque la protection maximale attribuée à un processus est de l'ordre de quelques millisecondes toutes les 4 secondes. L'implémentation a évolué à partir de 2006, en se déchargeant de cette durée de conservation du jeton difficile à configurer[1] : une priorité d'affectation est calculée pour chaque processus et augmente notamment avec le nombre de fautes de page, un processus ayant une priorité plus forte préemptant le jeton. Dans la pratique, ceci permet à un processus abusif de conserver le jeton de manière indéfinie. Cette particularité et l'incapacité du swap token à prendre en compte les cgroups, unités de facturation des ressources dans Linux, ont entraîné sa suppression du noyau à partir de la version 3.5 [vR12].

Discussion. Le mécanisme de swap pose naturellement un problème de performance lors de fortes contentions mémoire, le thrashing, comme exposé dans la section 2.2.3. Un premier ensemble d'approches gravite autour du modèle de consommation mémoire d'une application introduit par Dening, le working

[1] http ://lxr.free-electrons.com/source/mm/thrash.c ?v=3.4

set [Den68b]. Ce modèle définit l'ensemble des pages mémoire nécessaires pour qu'une application effectue ses traitements sur une période donnée. La résolution du thrashing dans ce modèle cherche ainsi à conserver au maximum les différents working sets des applications concurrentes en mémoire centrale. L'approximation des working sets est cependant coûteuse en espace mémoire, et le problème de thrashing demeure lorsque la taille des working sets dépasse de manière trop importante la quantité de mémoire physique.

Différents travaux ont ainsi pointé la nécessite de réduire le niveau de concurrence, c'est-à-dire le nombre d'applications exécutées simultanément en temps de contention mémoire. La sélection des processus à stopper dans ce cas n'est cependant pas simple, et les techniques proposées sont arbitraires ou reposant sur des calculs algorithmiques complexes a effectuer a priori comme le bin-packing.

Au contraire, le swap token de Jiang et Zhang [Jia09] propose une protection du processus le plus pesant sur la mémoire centrale, en espérant améliorer sa vitesse de travail et ainsi réduire au plus tôt la contention mémoire. Cette technique a été implémentée dans Linux pendant plusieurs années, mais a été abandonnée en 2012, n'étant compatible qu'avec une vision processus du système. De plus, l'hypothèse de terminaison des processus les plus lourds peut avoir des conséquences néfastes significatives sur le fonctionnement d'un système lorsqu'elle n'est pas valide, comme démontré par notre exemple de la section 2.2.3.

5.4 Élasticité mémoire

Ce besoin d'équilibrage dynamique des charges mémoire s'applique plus difficilement au contexte des environnements virtualisés. En effet, afin de gérer la mémoire de manière optimale, un hyperviseur peut souhaiter exploiter les ressources mémoire inutilisées par certaines machines virtuelles pour permettre à d'autres machines plus intensives d'obtenir de meilleures performances, tout en réduisant les accès disque propres aux mécanismes de swap internes au système invité. Ceci permet notamment aux hyperviseurs de disposer de leur capacité mémoire de manière optimale en sur-réservant [SD.09]. Cette fonctionnalité permettant à un hyperviseur d'adapter dynamiquement la quantité de mémoire allouée aux systèmes invités est couramment désignée par le terme d'*élasticité mémoire*.

Mécanismes d'élasticité dans les hyperviseurs. Très tôt, certains hyperviseurs, comme VMware ESX Server [Wal02], puis Xen [BDF+03] ont implémenté la possibilité d'une modification à l'exécution de la mémoire physique allouée à une machine virtuelle. La technique privilégiée par ces hyperviseurs, le *balooning*, consiste en l'ajout d'un pilote dans le système invité, par modifications explicites du code, techniques de paravirtualisation ou installation explicite depuis le système invité. Ce pilote sert de pont de communication privé entre l'hyperviseur et le système invité, concernant les besoins mémoire de chacun. Le pilote de balooning gère un ensemble de pages physiques attri-

buées originalement au système invité, qui sont disponibles pour une utilisation libre par l'hyperviseur, le ballon. En fonction de la pression mémoire observée, l'hyperviseur ajuste donc la taille du ballon demandée à chaque système invité, qui eux essaient de répondre à la demande de l'hyperviseur de manière optimale. Ainsi, l'hyperviseur transfère une partie de sa propre pression mémoire vers les systèmes invités qui en sont capables. Le balooning requiert cependant une coopération du système invité, dépend de la limite que s'impose le système invité en termes de taille du ballon, et n'offre pas de garanties en termes de réactivité aux systèmes invités.

Lorsque le balooning ou le partage de pages similaires, efficace lorsque deux multiples instances d'un même système sont présentes, ne permettent pas de soulager la pression mémoire de l'hyperviseur, celui-ci doit se résoudre à utiliser un mécanisme de swap classique, où la mémoire physique des systèmes invités est déchargée vers un système de stockage secondaire. Cependant, l'hyperviseur n'ayant aucune connaissance de la disposition de la mémoire des systèmes invités, le mécanisme de swap peut engendrer des pertes de performance significatives, en déchargeant des pages fréquemment accédées ou essentielles au système d'exploitation, celles du noyau par exemple. Le problème du double swapping, où l'hyperviseur décharge une page qui est ensuite également déchargée par le système invité, peut également engendrer un trafic de swap encore plus important [SD.09]. Du point de vue des systèmes invités comme de l'hyperviseur, le mécanisme de swap est donc utilisé en dernier ressort, puisque dangereux pour les performances.

Ces mécanismes de réduction et d'élasticité mémoire permettent la mise en place de politiques plus fines d'équilibrage de la mémoire entre les machines virtuelles invitées. Une première approche est de dériver des SLA et d'une observation préalable les besoins mémoire maximums de chaque machine virtuelle, comme appliquée par Heo *et al.* à Xen [HZPW09]. Les nouvelles machines virtuelles sont ainsi exécutées dans des conditions variables d'utilisation mémoire et CPU, afin de déterminer les tendances en termes de taux de fautes de page et de temps moyen de réponse des services notamment. En se servant de ces relations, leur contrôleur décide à l'exécution et de manière périodique, des allocations mémoire et CPU de chaque machine virtuelle, afin de maximiser l'exploitation de la mémoire tout en minimisant les temps de réponse de chaque service. Bien que cette technique permette de sur-réserver la mémoire d'un hyperviseur de manière efficace, elle dépend fortement du profil déterminé empiriquement et non nécessairement représentatif de l'exécution dynamique réelle du programme.

Prédiction des besoins en mémoire. MEB, introduit par Zhao *et al.*, permet la prédiction à l'exécution des besoins en mémoire des systèmes invités dans Xen [ZW09]. MEB estime ces besoins à partir d'histogrammes LRU, qui suivent le rapport entre le nombre de pages accédées et les fautes de page pour un système en particulier, en exploitant les mécanismes de protection des pages, à la manière de Zhou *et al.* pour le calcul du MRC [ZPS+04]. Ceci leur

permet d'estimer la taille du working set d'un système en particulier, et donc de détecter ceux ayant de la mémoire libre, même s'ils ont effectivement alloué la majorité de leur espace mémoire réservé. Les histogrammes LRU sont particulièrement efficaces pour prévoir une hausse dans la taille du working set, et permettent donc d'ajuster la mémoire allouée avant les premiers symptômes de contention mémoire. Cette solution impose cependant une perte de performance pour les applications intensives en CPU uniquement, qui payent le surplus d'exécution nécessaire à la construction des histogrammes LRU, sans gagner en performance.

Baruchi *et al.* ont proposé l'adaptation à la prévision des besoins mémoire de la moyenne mobile exponentielle, technique employée dans le monde de la finance pour analyser les tendances des prix du marché [BM11]. Leur moyenne mobile exponentielle se base sur un échantillonnage discret de l'utilisation mémoire d'une machine virtuelle dans le passé. Une moyenne, pondérée de manière exponentielle, les valeurs les plus récentes ayant le poids le plus fort, permet de dériver une estimation des besoins futurs de l'application. Cette technique s'adapte mieux aux charges de travail périodiques que celles basées sur celles le nombre de fautes de page, mais gère naturellement moins bien les pics soudains d'utilisation mémoire. La moyenne mobile exponentielle permet une allocation a priori, contrairement aux techniques à bases de fautes de page qui sont réactives, et donc une adaptation plus fluide aux fluctuation de l'utilisation de la mémoire. Elle nécessite cependant plus d'appels aux mécanismes d'ajustement des allocations, comme le balooning, imposant un surplus en temps d'exécution parfois significatif.

Discussion. L'un des problèmes de la gestion mémoire est son allocation a priori, qu'il n'est pas possible pour un système de moduler de manière dynamique sans mettre en danger le fonctionnement des applications. Les systèmes virtualisés permettent une solution élégante à ce problème avec l'introduction du concept d'élasticité mémoire. Dans ce cas, une coopération entre l'hyperviseur et les systèmes invités permet de moduler de manière dynamique le nombre de pages allouées à chaque système, en fonction des besoins de chacun à une date précise.

Ce mécanisme sert de base à la définition de techniques plus évoluées, ayant pour but de prévoir les besoins des différents systèmes invités dans un futur proche, et permettent une allocation adaptée aux tendances de consommation détectées. L'inconvénient principal réside cependant dans la volonté de coopération des systèmes invités, qui peuvent ne pas être assez réactifs, ou ne pas mettre de pages à disposition de l'hyperviseur. De plus, les appels fréquents aux mécanismes d'ajustement de l'allocation induisent en pratique des surplus en temps d'exécution significatifs.

5.5 Conclusion

Du fait de son caractère central dans l'utilisation et la performance des systèmes d'exploitation, la mémoire est une ressource problématique dont l'usage doit être optimisé. Contrairement à d'autres ressources de type flux, les allocations mémoire sont effectuées a priori, et ne peuvent ensuite être adaptées par le système d'exploitation sans mettre en danger la performance des applications. Un nombre important d'optimisations génériques sur les mécanismes de remplacement de pages permettent de bonnes performances dans le cas général, en offrant plus de mémoire aux processus intensifs tout en déchargeant les pages qui présentent la plus faible probabilité d'être accédées dans le futur.

Le mécanisme de swap, permettant déchargements et rechargements transparent de pages mémoire vers et depuis un système de stockage secondaire, peut engendrer des pertes de performance dramatiques pour les applications voire le système. Ce phénomène impose le besoin d'un certain degré d'isolation dans les performances mémoire de chaque application pour les systèmes partagés. Bien que certaines techniques novatrices comme le swap token ou l'ordonnancement à moyen terme offrent des solutions élégantes à ce problème, elles s'adaptent difficilement aux systèmes partagés best-effort et leurs applications concurrentes ayant des besoins changeants en mémoire.

Dans le cadre de la virtualisation, les hyperviseurs cherchent à minimiser les risques de swapping, tout en sur-réservant la mémoire physique afin d'exploiter leurs ressources de manière optimale. Le balooning permet notamment aux hyperviseurs d'ajuster les demandes mémoire des systèmes invités si ceux-ci sont coopératifs, mais peut induire des surplus en temps d'exécution si les ajustements sont trop fréquents. Cette technique permet cependant l'application de techniques d'estimation des besoins futurs des systèmes invités, à base du ratio de fautes de page ou de la consommation mémoire passée, induisant une gestion de la mémoire plus efficace mais ayant leurs limites en termes de charges de travail supportées.

Chapitre 6

Application à la régulation du mécanisme de swap

Dans ce chapitre, nous appliquons notre approche de contrôle, définie au chapitre 4, à la régulation équitable du mécanisme de swap de Linux. En effet, la mémoire virtuelle, dont la surcharge peut conduire à une dégradation sévère de la performance du système, est une ressource peu étudiée du point de vue de l'équité. Les techniques d'élasticité appliquées dans le cas des environnements de virtualisation reposent sur une coopération non garantie dans les systèmes partagés best-effort, et les systèmes d'exploitation classiques n'incluent aucun mécanisme de régulation.

Afin d'évaluer notre approche par rapport aux techniques employées dans ces systèmes, nous avons développé un prototype au sein du noyau Linux. Nos expériences permettent de valider les bénéfices de notre couche de contrôle lors de contentions mémoire, en montrant une performance d'applications étalon significativement améliorée, tout en affichant une incertitude réduite sur les durées d'exécution.

6.1 Implémentation

Notre prototype de régulation du mécanisme de swap est implémenté au sein du noyau Linux. Nous avons notamment choisi ce système d'exploitation pour sa maturité et la disponibilité d'une technique de gestion de la surcharge de mémoire virtuelle à l'état de l'art, à savoir le swap token décrit dans la section 5.3.

Nous fournissons dans cette section une vue globale de l'intégration de notre couche de contrôle au mécanisme de swap de Linux. Nous détaillons le cycle de vie d'une requête de swap dans Linux, ainsi que les emplacements auxquels nous avons positionné les différents éléments de notre approche, notamment la facturation et le contrôle d'accès. Nous discutons également des structures de données et techniques de synchronisation manipulées afin de ne pas perturber l'efficacité des traitements de bas niveau du système.

L'implémentation complète est disponible en annexe A.

6.1.1 Choix du système d'exploitation cible

Notre approche définie dans le chapitre 4 prône la mise en place d'une facturation et d'un contrôle unifié de l'usage des ressources, en interception des requêtes. Elle est rendue possible par l'évaluation du temps d'utilisation comme métrique générique. Comme détaillé dans la section 4.3, les spécificités des systèmes à micro-noyaux, de type L4 en particulier, permettent d'implémenter une architecture de régulation transparente et automatisée. Cependant, les différents dérivés de L4 que nous avons considérés ne sont pas encore assez matures : nous n'avons notamment pas trouvé d'instance qui implémente suffisamment de pilotes matériels ainsi que des techniques de gestion de ressources évoluées, comme le mécanisme de swap. De plus, l'absence de couches de contrôle ou d'optimisation de l'usage des ressources ne permet pas une appréciation adéquate de l'apport de notre approche.

Nous avons ainsi opté pour une implémentation au sein du noyau Linux. Ce choix ne nous permet pas d'offrir l'automatisation possible dans les micro-noyaux, présentée dans la section 4.3, mais permet d'évaluer notre approche dans un système très répandu, représentatif des systèmes partagés best-effort. En effet, Linux est couramment utilisé dans les offres de serveurs dédiés ou systèmes invités virtualisés et forme une brique de base pour plusieurs hyperviseurs, parmi lesquels Xen et VMware ESX.

Mécanisme de swap dans Linux. Dans Linux, les fautes de page sont résolues à la demande, dans le contexte d'exécution du processus fautif. Le noyau forge une requête d'entrée/sortie classique puis la soumet à l'ordonnanceur d'entrées/sorties comme toute requête ordinaire. Par défaut, cet ordonnancement dans Linux est basé sur les files d'attente équitables CFQ, implémentation de l'algorithme SFQ détaillé dans la section 3.1.1. Le processus est bloqué en attente de la fin de traitement de la requête, puis l'instruction ayant provoqué la violation d'accès mémoire est relancée. Le déchargement des pages mémoire est quant à lui effectué de manière globale au système, en suivant l'algorithme Clock comme approximation de LRU. Un démon dédié observe de manière périodique la charge mémoire du système, et décide s'il est nécessaire de décharger des pages de la mémoire centrale ou non. L'éviction de pages peut également être déclenchée par une allocation après laquelle la quantité de pages physiques disponibles passe en-dessous d'un seuil arbitraire, jugé insuffisant.

Le déchargement de pages est ainsi anonymisé, et ne peut être rapporté à un domaine particulier. Au contraire, le rechargement est effectué à la demande et peut être aisément facturé au processus fautif.

Enfin, nous employons pour nos expériences le noyau Linux dans sa version 3.5, avant la suppression du swap token [vR12]. Cela nous permet de comparer trois approches différentes, à savoir le mécanisme de swap de Linux par défaut, le même mécanisme en présence du swap token, ainsi que notre couche de contrôle, et d'évaluer l'impact de ces différentes techniques.

6.1.2 Mécanismes internes de l'implémentation

Puisque le mécanisme de swap est un service fourni de manière transparente par le système, les applications ne soumettent pas de requêtes explicites. Le déchargement de page est déclenché de manière périodique par le système ou lorsqu'un niveau anormalement bas de mémoire physique disponible est détecté, ce qui le rend opaque du point de vue de l'espace utilisateur. De ce fait, le rechargement de page est opéré de manière transparente et synchrone lorsqu'un accès mémoire fautif est détecté par le matériel : si une faute de page est bien liée à une page préalablement déchargée, le noyau effectue les opérations d'entrées/sorties nécessaires afin de rapatrier la page en mémoire centrale, puis relance le processus à l'instruction fautive.

Cette description du mécanisme souligne l'adéquation avec le modèle de ressources présenté dans la définition de notre couche de contrôle en section 4.2 : la résolution d'une requête de swap est accomplie de manière synchrone et dans le contexte du processus fautif, puis fait appel à un traitement interne induisant potentiellement des réordonnancements. Ceci nous permet d'effectuer un contrôle a priori, avant création des requêtes d'entrées/sorties nécessaires, et d'éventuellement bloquer le processus appelant si celui-ci appartient à un domaine abusif. La facturation du temps d'utilisation peut se faire de manière comparable au cas des entrées/sorties, mais en prenant seulement en compte les requêtes issues de fautes de page, puisque nous considérons le service de swap comme une ressource disjointe des entrées/sorties.

Intégration dans Linux. Comme nous l'avons abordé dans la section 4.3, l'application de notre couche de contrôle dans un système monolithique passe par l'ajout explicite d'appels aux fonctions de facturation, lors des évènements de soumission de requête, de début et de fin de traitement. La figure 6.1 résume le cycle de vie d'une requête de swap dans la noyau Linux, et les endroits où nous avons inséré nos appels de fonctions explicites.

Nous avons donc modifié trois fonctions du noyau afin de suivre l'évolution du traitement d'une requête rq :

1. `do_swap_page()`[1], exécutée dans le contexte du processus fautif, est la première fonction au sein de laquelle est déterminé le besoin d'effectuer un rapatriement de page depuis l'espace de swap. Elle nous permet de marquer la date $t_{arrival}(rq)$ d'arrivée d'une requête. C'est d'ailleurs au même endroit que Linux appelle la fonction `grab_swap_token()`, évaluant la priorité d'accès au jeton de swap afin de protéger les pages du processus si nécessaire.

2. `blk_start_request()`[2], exécutée par le noyau dans un contexte arbitraire, est la dernière fonction du système d'entrées/sorties bloc avant de fournir une requête au pilote de bas niveau pour transmission au matériel.

[1] Fonction implémentée dans le fichier mm/memory.c des sources du noyau Linux.
[2] Fonction implémentée dans le fichier block/blk-core.c des sources du noyau Linux.

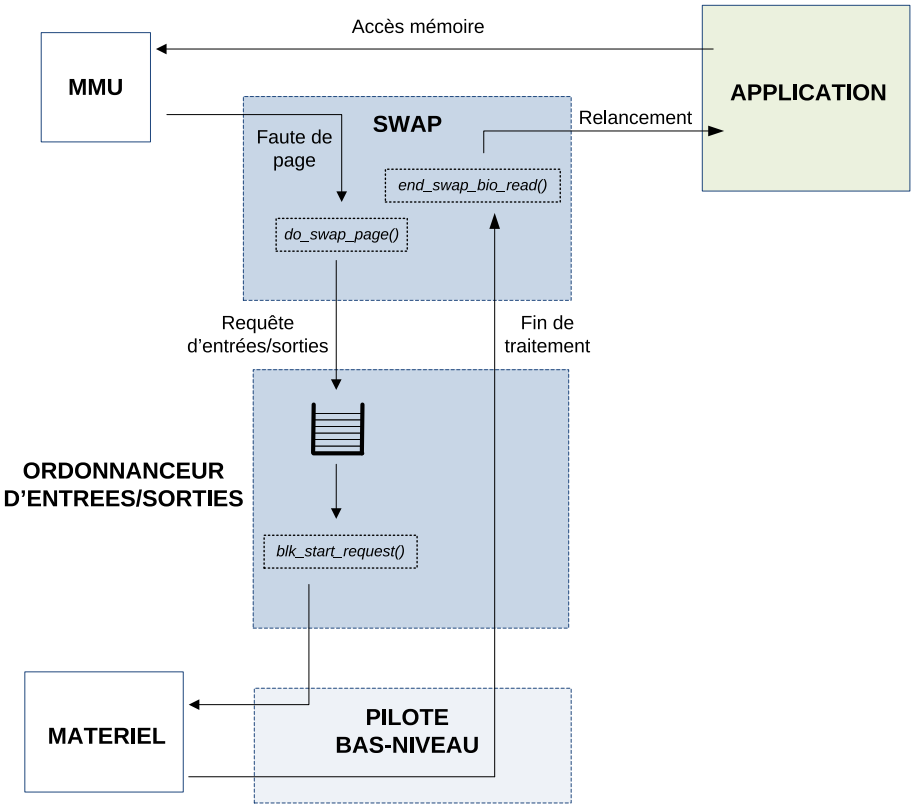


FIGURE 6.1 – Appels interceptés au cours du cycle de vie d'une requête de swap dans le noyau Linux.

Elle permet de capturer le temps effectif de transmission $t_{start}(rq)$ de la requête au matériel.

3. `end_swap_bio_read()`[3], exécutée dans un contexte d'interruption matérielle, note la terminaison de rapatriement d'une page depuis l'espace de swap. Elle indique ainsi le temps de fin de traitement de la requête, $t_{end}(rq)$.

Le premier appel, captant la date $t_{arrival}(rq)$, nous permet également de bloquer le processus fautif si nécessaire, puisque `do_swap_page()` est exécutée dans le contexte de celui-ci. De plus, cet appel est effectué avant que toute structure de données ne soit verrouillée et que toute requête d'entrée/sortie ne soit créée, évitant ainsi d'éventuels problèmes de synchronisation sur une page déchargée accédée de manière concurrente. Les différentes métadonnées de

[3]Fonction implémentée dans le fichier mm/page_io.c des sources du noyau Linux.

facturation, comme $t_{start}(rq)$ et $t_{end}(rq)$, sont taguées, c'est-à-dire ajoutées aux structures de données de chaque requête. Ceci est possible car notre facturation est effectuée au cours du cycle de vie normal des requêtes. Les métadonnées sont initialisées lors de la création des requêtes d'entrées/sorties engendrées par une faute de page, dans la fonction `swap_readpage()`[3].

Domaines de facturation. Conformément aux travaux de recherche passés, présentés dans la section 3.2.1, nos domaines de facturation sont flexibles puisqu'ils peuvent correspondre à un seul processus, regrouper les processus d'un utilisateur ou encore rassembler les processus d'un groupe. Un domaine est associé à un identifiant numérique unique, respectivement le PID (*Process ID*), UID (*User ID*) et GID (*Group ID*). De ce fait, l'ajout futur de tout groupement arbitrairement défini, par exemple un cgroup Linux, peut facilement être intégré dans notre notion de domaine de facturation.

Facturation différée. Il est à noter que le traitement de fin de requête est effectué en contexte d'interruption, ce qui signifie que les accès non atomiques aux structures de données partagées doivent être prohibés. Ceci force une implémentation différée de la mise à jour de la facture des domaines, ainsi que de l'évaluation du statut abusif, de manière similaire à l'algorithme 4.12 présenté dans la section 4.2.3. Le noyau Linux fournit plusieurs facilités afin d'implémenter de tels traitements à déplacer en dehors des contextes d'interruption, parmi lesquels les *work queues*. Les work queues permettent de planifier l'exécution d'une procédure à effectuer au plus tôt en dehors d'un contexte d'interruption ou de manière périodique. Elles sont exécutées dans un contexte arbitraire et ne sont pas réentrantes, ce qui convient parfaitement à notre cas d'utilisation.

Nous avons implémenté la communication des résultats entre la fin d'interruption matérielle en gérant deux listes chaînées différentes : l'une contient les requêtes terminées et est mise à jour en contexte d'interruption uniquement, et l'autre les requêtes en cours de traitement par la procédure de facturation aux domaines. La liste contenant les requêtes terminées est référencée par un pointeur, modifiable de manière atomique par le processus de facturation qui interchange les deux listes avant d'effectuer ses traitements, et vide la liste des requêtes en cours de traitement une fois son travail terminé. Cette technique impose une facturation groupée des requêtes lorsque plusieurs interruptions matérielles interviennent successivement. Cela induit à la fois une réduction du temps passé en contexte d'interruption, afin de ne pas ralentir les entrées/sorties, mais également une propriété d'hystérésis naturelle puisque la facture et les statuts des domaines ne sont pas modifiés en continu, comme expliqué en section 4.2.3. Par ailleurs, d'autres expérimentations non présentées ici nous ont confirmé qu'une facturation implémentée entièrement en contexte d'interruption pouvait avoir une incidence significativement négative sur la performance des processus. Au contraire, l'impact d'une facturation différée et groupée n'est généralement pas tangible.

Structures de données manipulées. Les structures de données représentant les domaines ne sont initialisées que lors de la facturation de leur première requête, afin de réduire l'espace mémoire réservé. De manière similaire, ces structures de données peuvent être libérées lors de la destruction ou de l'inactivité des domaines, par exemple la fin d'un processus ou une facture nulle sur la période de suivi, selon le type de regroupement. L'empreinte mémoire en termes de variables est relativement faible, le surplus dans notre implémentation étant de moins de 60 octets par domaine de facturation et moins de 50 octets par requête, auxquels il faut ajouter environ 100 octets pour les données globales aux algorithmes[4]. Cette caractéristique n'est pas négligeable, puisque l'usage du mécanisme de swap implique une forte pression mémoire et donc une quantité réduite de pages disponibles.

Pour plus de détails, nous rapportons le lecteur à l'implémentation de notre prototype, consignée dans l'annexe A.

6.2 Choix d'applications étalon pour l'évaluation expérimentale

Afin d'évaluer notre couche de contrôle de manière reproductible, nous avons exploité la suite d'applications SPEC CPU2006 [Hen06]. Nous avons sélectionné diverses applications que nous considérons représentatives de charges de travail des systèmes partagés best-effort, et qui présentent des profils d'utilisation de la mémoire et du processeur variés.

Nous décrivons dans cette section le profil d'allocation mémoire de chaque application sélectionnée selon différentes charges de travail fournies en entrée, ainsi que leur comportement sous contention mémoire dans Linux.

6.2.1 Profils mémoire des applications sélectionnées

Pour chaque charge de travail, nous précisons les profils mémoire en exécution solitaire, sur un système Dell Latitude D520 avec 512 Mo de RAM et une installation Linux 3.5 basique. Les profils décrivent l'évolution moyenne, au cours d'une exécution étalon, de la taille de mémoire virtuelle, quantité totale de mémoire allouée à l'application, ainsi que de la taille résidente, c'est-à-dire la quantité de RAM occupée.

[4] Tailles définies pour une architecture 32-bits.

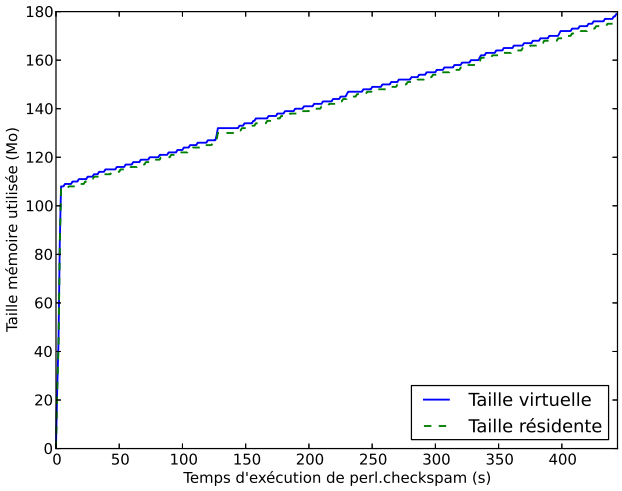


FIGURE 6.2 – Profil d'utilisation de la mémoire par la charge de travail *perl.checkspam* exécutée seule.

400.perlbench. Cette application est une version minimaliste de Perl 5.8.7, et les trois scripts d'entrée fournis représentent des opérations de gestion de mails, à savoir la détection de spam, la conversion d'e-mails au format HTML, ainsi que l'utilisation de l'outil *specdiff* calculant une "différence spectrale" entre plusieurs ensembles de mails. Dans la suite de ce chapitre, nous référons respectivement à ces trois scripts par les appellations *perl.checkspam*, *perl.splitmail* et *perl.diffmail*.

Les profils d'utilisation mémoire de ces trois charges sont respectivement représentés sur les figures 6.2, 6.3 et 6.4.

perl.checkspam s'exécute en 445 secondes, et demande une allocation en rampe de la mémoire jusqu'à un maximum de 180 Mo, pour une moyenne de 140 Mo.

perl.splitmail s'exécute en 290 secondes. et a une demande mémoire globalement constante de 480 Mo en moyenne, agrémentée de légers pics allant jusqu'à 550 Mo. Ses besoins dépassant les 512 Mo de RAM du système, sa taille résidente est plus faible, à 440 Mo en moyenne.

perl.diffmail s'exécute en 170 secondes et présente un profil d'allocation en escalier, culminant à 290 Mo pour une moyenne de 190 Mo.

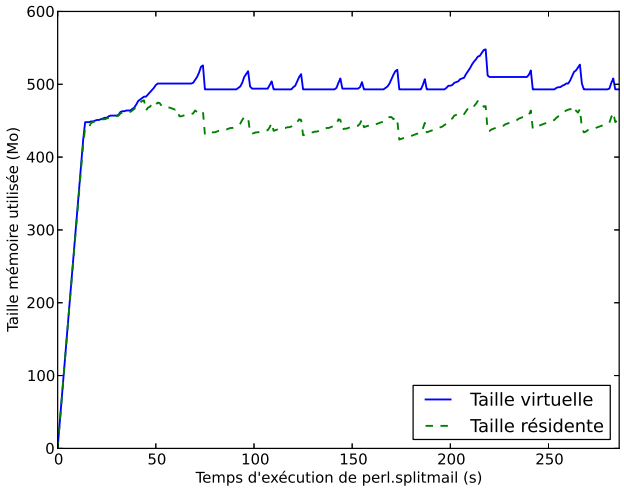


FIGURE 6.3 – Profil d'utilisation de la mémoire par la charge de travail *perl.splitmail* exécutée seule.

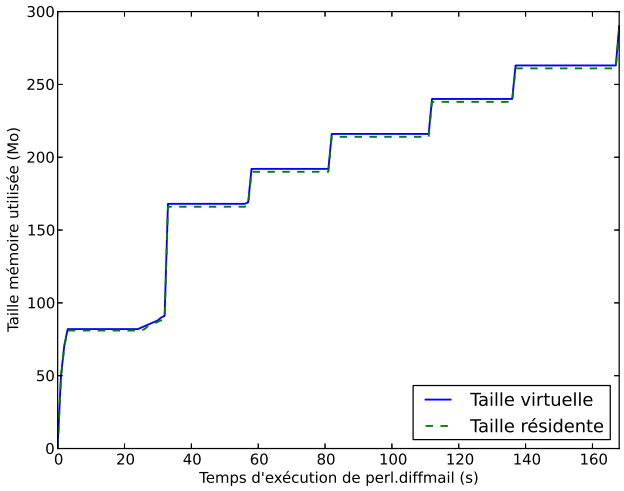


FIGURE 6.4 – Profil d'utilisation de la mémoire par la charge de travail *perl.diffmail* exécutée seule.

401.bzip2. Cette application, basée sur l'utilitaire de compression bzip2 1.0.3, applique plusieurs niveaux de compression à des fichiers de taille variable. Nous avons retenu les deux fichiers *liberty.jpg* et *input.combined*. Le premier est une image au format JPEG, et le deuxième une archive rassemblant à la fois des contenus hautement et faiblement compressibles. Ces deux fichiers sont respectivement référés ci-après par *bzip.liberty* et *bzip.combined*.

Les profils mémoire des compressions de ces deux fichiers sont représentés sur les figures 6.5 et 6.6. Les deux charges de travail affichent un profil similaire, avec une allocation globalement constante, qui diminue très légèrement pendant les périodes de décompression. *bzip.liberty* s'exécute en 185 secondes et demande environ 100 Mo de mémoire, tandis que *bzip.combined* s'exécute en 205 secondes et travaille sur un maximum de 610 Mo.

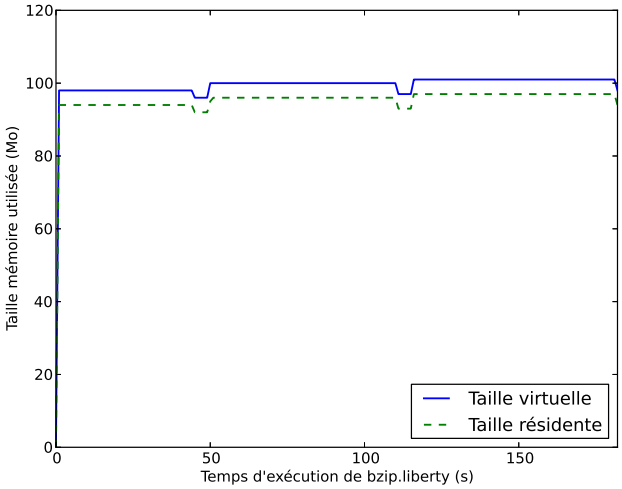


FIGURE 6.5 – Profil d'utilisation de la mémoire par la charge de travail *bzip.liberty* exécutée seule.

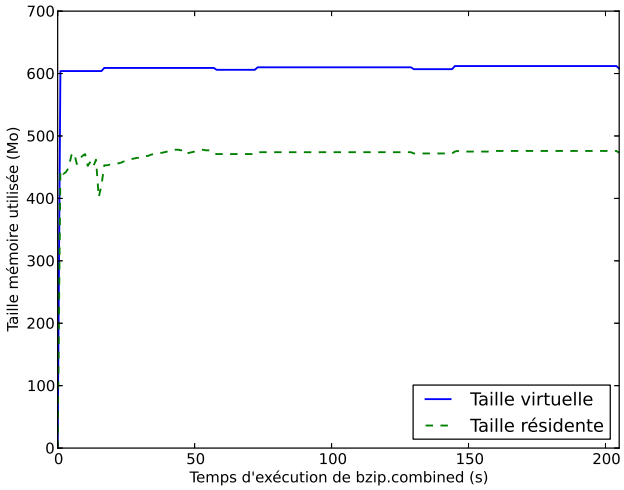


FIGURE 6.6 – Profil d'utilisation de la mémoire par la charge de travail *bzip.combined* exécutée seule.

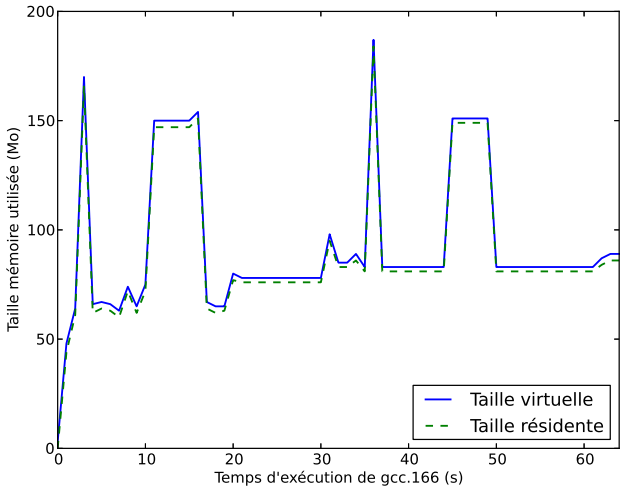


FIGURE 6.7 – Profil d'utilisation de la mémoire par la charge de travail *gcc.166* exécutée seule.

403.gcc. Cette application est basée sur le compilateur gcc 3.2. Parmi les sources C fournies, nous avons sélectionné trois fichiers arborant différents profils d'allocation mémoire : *166.in*, *200.in* et *c-typeck.in*, que nous dénommons respectivement *gcc.166*, *gcc.200* et *gcc.typeck*.

Les figures 6.7, 6.8 et 6.9 représentent l'utilisation mémoire lors de la compilation de ces différents fichiers. *gcc.166* s'exécute en 65 secondes, *gcc.typeck* en 80 secondes et *gcc.200* en 100 secondes.

Les charges de travail *gcc.166* et *gcc.typeck* affichent un profil chaotique, avec la présence de multiples pics importants de mémoire. *gcc.166* a besoin de 90 Mo de mémoire en moyenne, avec une demande maximum de 190 Mo. Le pic d'allocation pour *gcc.typeck* est à 400 Mo, avec une demande moyenne bien plus faible de 132 Mo.

Au contraire, *gcc.200* présente un profil global de type rampe, avec 170 Mo de demande mémoire maximum pour 85 Mo de moyenne.

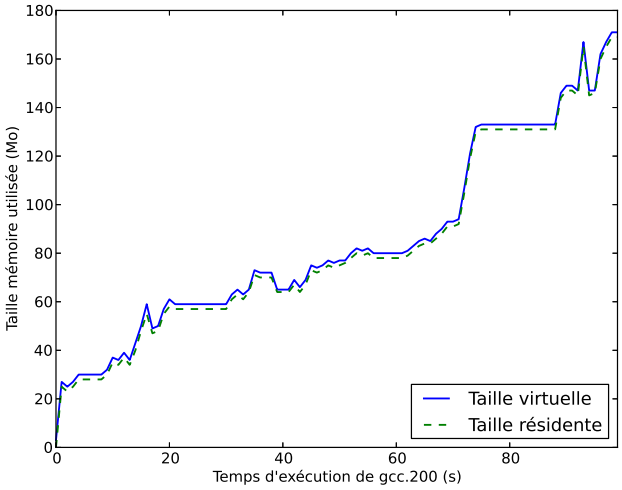


FIGURE 6.8 – Profil d'utilisation de la mémoire par la charge de travail *gcc.200* exécutée seule.

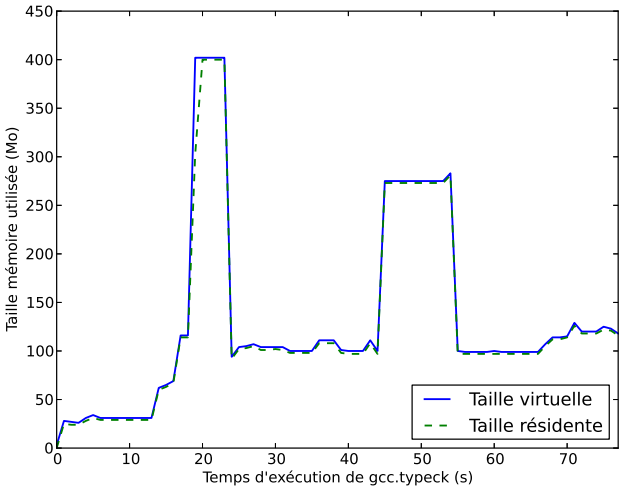


FIGURE 6.9 – Profil d'utilisation de la mémoire par la charge de travail *gcc.typeck* exécutée seule.

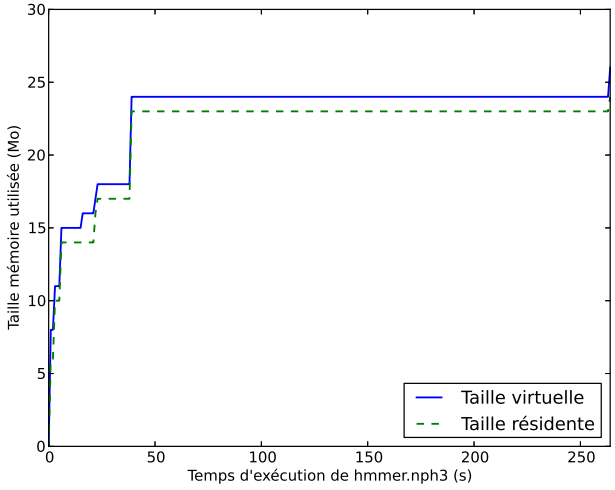


FIGURE 6.10 – Profil d'utilisation de la mémoire par la charge de travail *hmmer.nph3* exécutée seule.

456.hmmer. HMMER est un outil d'analyse de séquences ADN. Nous utilisons la charge de travail *nph3.hmm* qui produit, à partir d'une base de données de séquences ADN, un tri classifié des séquences s'approchant le plus de l'ensemble de séquences arbitraires recherchées. Nous y référons ensuite par l'appellation *hmmer.nph3*.

Le profil mémoire est rapporté sur la figure 6.10, et montre une allocation en escalier initialement, puis constante, avec 26 Mo au maximum et 23 Mo en moyenne. Le temps d'exécution, de 265 secondes, témoigne d'un travail CPU important sur cette quantité de mémoire relativement faible.

464.h264ref. Cette application est une implémentation du standard de compression vidéo H.264/AVC. La charge choisie, que nous appellons *h264.sss*, effectue la compression d'une séquence de 171 images de 512x320 pixels provenant d'un jeu vidéo.

Comme rapporté par la figure 6.11, *h264.sss* s'exécute en 1 400 secondes, malgré sa demande mémoire relativement faible de 63 Mo en moyenne, constante avec de légers pics.

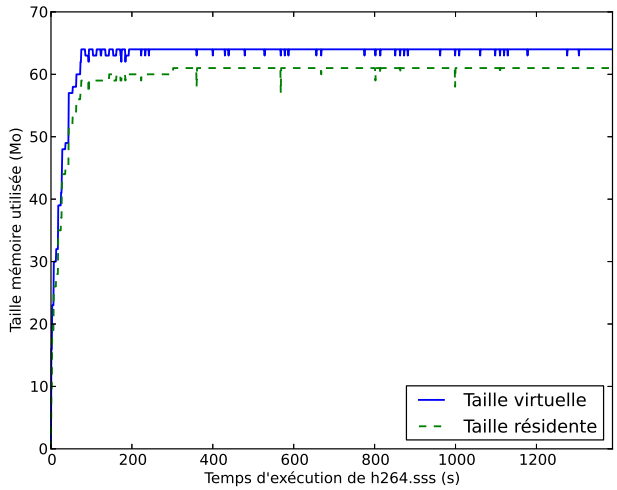


FIGURE 6.11 – Profil d'utilisation de la mémoire par la charge de travail *h264.sss* exécutée seule.

Dénomination	**Max. mémoire virtuelle** (Mo)	**Allocation mémoire**	**Temps d'exécution** (s)
hmmer.nph3	26	Escalier	265
h264.sss	65	Légers pics	1 400
bzip.liberty	100	Constante	185
gcc.200	170	Rampe	100
perl.checkspam	180	Rampe	445
gcc.166	190	Larges pics	65
perl.diffmail	290	Escalier	170
gcc.typeck	400	Larges pics	80
perl.splitmail	550	Légers pics	290
bzip.combined	610	Constante	205

TABLE 6.12 – Caractéristiques en termes d'utilisation mémoire et de durée d'exécution en environnement isolé d'applications étalon extraites de la suite SPEC CPU2006.

La table 6.12 résume les caractéristiques principales de ces observations étalon choisies.

Cet ensemble d'applications nous permet ainsi d'évaluer notre couche de contrôle avec divers profils d'utilisation de la mémoire, à savoir des demandes constantes, avec des pics légers ou importants, en escalier ou encore en rampe. La distribution des temps d'exécution relative à la mémoire demandée, parfois très courts pour une large utilisation comme *gcc.typeck*, ou longs avec une faible quantité allouée pour *h264.sss*, nous offre une bonne couverture du ratio instructions CPU par opération mémoire. Enfin, 4 de ces 10 applications effectuent une demande mémoire supérieure à 256 Mo. En considérant l'ajout d'une application concurrente, ces applications sont ainsi candidates à un statut abusif vis-à-vis des quotas classiques, car nécessitant plus de la moitié des 512 Mo du système à au moins un point de leur exécution.

6.2.2 Comportement sous contention mémoire dans Linux

Afin de caractériser mieux ces applications étalon, nous les avons exécutées sous deux types de contention mémoire. Nous utilisons pour cela une application abusive, désignée par le terme *swappeur*. Le swappeur alloue une quantité fixe de mémoire au début de son exécution, puis effectue une boucle infinie consistant en une opération mémoire arbitraire sur chaque page. Ce profil applicatif est contraire aux hypothèses des algorithmes de remplacement de pages type LRU, car les pages les plus récemment accédées deviennent celles ayant la plus faible probabilité d'être requises dans un futur proche.

Configuration Linux par défaut. Nous avons mis les applications étalon en concurrence avec deux types de swappeurs, allouant respectivement 256 et 512 Mo. Le système utilisé est une installation basique de Linux 3.5, sans swap token. Un swappeur utilisant 256 Mo permet d'observer le comportement des applications étalon en concurrence avec une application respectant un quota de mémoire alloué à hauteur de 50% de la RAM. L'utilisation de 512 Mo permet d'évaluer le cas d'une surcharge grave, où l'application concurrente essaye de s'accaparer toute la RAM. La table 6.13 représente la moyenne ainsi que l'incertitude des temps de 10 exécutions indépendantes pour chaque charge de travail.

Dénomination	**Sans concurrence** (s)	***Sw256*** (s)	***Sw512*** (s)
hmmer.nph3	265	568 ± 0,5%	1 487 ± 5%
h264.sss	1 400	1 475 ± 0,4%	4 406 ± 6%
bzip.liberty	185	288 ± 0,2%	370 ± 4%
gcc.200	100	125 ± 0,3%	14 886 ± 5%
perl.checkspam	445	497 ± 0,4%	6 094 ± 3%
gcc.166	65	91 ± 0,8%	1 883 ± 35%
perl.diffmail	170	223 ± 1%	2 412 ± 7%
gcc.typeck	80	266 ± 6%	7 191 ± 11%
perl.splitmail	290	13 773 ± 4%	42 174 ± 9%
bzip.combined	205	421 ± 5%	1 801 ± 5%

TABLE 6.13 – Durées moyennes d'exécution des applications étalon sans concurrence, ou en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire. Les applications sont exécutées sur Linux 3.5 et le swap token est désactivé.

Le cas du swappeur travaillant sur 256 Mo montre des temps d'exécution accrus légèrement, par des facteurs de 1 à 3 dans le cas général. Ceci est lié à l'apparition d'un léger phénomène de swapping dans certains cas, ou au partage du processeur dans d'autres. Les deux exceptions notables sont *hmmer.nph3* et *perl.splitmail*. Le premier double son temps d'exécution malgré une allocation de mémoire plutôt faible (26 Mo), tandis que le deuxième subit une pénalisation à hauteur de 8 000%. Dans le cas de *hmmer.nph3*, ce résultat indique un usage intensif du processeur. Comme la mise en concurrence divise le temps d'accès au CPU et réduit l'effet des caches de données du processeur, son temps d'exécution est naturellement doublé. Pour *perl.splitmail*, le ralentissement observé est la conséquence d'une utilisation non seulement intensive, mais probablement séquentielle de la mémoire, impliquant un nombre important de fautes de page.

Comme attendu, cette table met également en évidence le phénomène de thrashing présenté dans la section 2.2.3. En effet, les concurrences avec un swappeur de 512 Mo obligent un surplus sur l'exécution allant de 120% dans le cas de *bzip.liberty*, à 15 000% et 25 000 % pour *gcc.200* et *perl.splitmail* respectivement. Les variations importantes dans les temps d'exécution typiques du thrashing sont clairement observables, *gcc.166* subissant par exemple une incertitude à hauteur de 35% du temps moyen. Les dégradations les plus sérieuses, par exemple celles subies par les trois charges de travail *gcc*, indiquent une certaine séquentialité dans les accès mémoire, incompatible avec LRU, de la même façon que *perl.splitmail*.

Impact du swap token de Linux sur les performances. Nous avons répété cette expérience en activant le swap token de Linux, décrit dans la section 5.3. Ceci nous permet d'observer l'impact d'une des seules techniques d'optimisation du mécanisme de swap implémentées à ce jour dans les systèmes classiques. La table 6.14 rapporte l'évolution des durées moyennes, toujours sur 10 exécutions, par rapport au cas où le swap token est désactivé.

Dénomination	**Durée moyenne** *Sw256*	**Durée moyenne** *Sw512*
hmmer.nph3	-	-
h264.sss	-	+69%
bzip.liberty	-	+10%
gcc.200	-	+101%
perl.checkspam	-	+54%
gcc.166	-	+117%
perl.diffmail	-	+5%
gcc.typeck	-6%	+1%
perl.splitmail	-	> +400% *
bzip.combined	-2%	-2%

* Exécutions stoppées prématurément

TABLE 6.14 – Impact de l'activation du swap token sur le temps moyen d'exécution des applications étalon. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire. Les tirets représentent des variations négligeables, inférieures à 1%. Une valeur repérée en vert ou en rouge souligne respectivement une évolution significativement positive ou négative.

Comme attendu, l'activation du swap token n'affecte pas les cas où la concurrence mémoire est faible, observables pour toutes les applications nécessitant moins de 256 Mo de mémoire lorsqu'elles sont mises en concurrence avec un swappeur de 256 Mo. Les bénéfices du swap token sont déjà visibles

pour *gcc.typeck* et *bzip.combined* lorsque mises en concurrence avec le swappeur de 256 Mo. Cela n'est pas surprenant, car *gcc.typeck* présente deux pics d'utilisation de la mémoire supérieurs à 256 Mo, et *bzip.combined* une demande constante de mémoire au-delà de 600 Mo. Pendant ces utilisations intensives de la mémoire, supérieures à la consommation du swappeur, le swap token va protéger les pages mémoire des charges de travail et leur permettre de progresser plus rapidement.

Cependant, lorsque l'application est globalement moins intensive que sa concurrente, ce qui est le cas lorsque le swappeur alloue 512 Mo, le swap token s'avère contre-productif. En effet, son hypothèse de base voulant que les charges de travail les plus intenses soient éphémères n'est plus valable dans ce cas. Ceci pénalise particulièrement les charges de travail qui présentent une faible localité mémoire, comme *h264.sss*, *gcc.200* ou *gcc.166*. La charge de travail *bzip.combined* fait office d'exception, puisque le swap token semble globalement améliorer sa vitesse d'exécution.

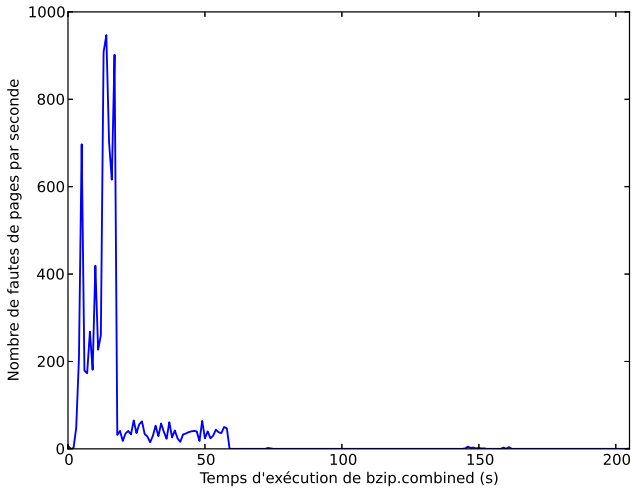


FIGURE 6.15 – Évolution du nombre de fautes de page déclenchées par *bzip.combined* lors d'une exécution sur un système ne possédant que 512 Mo de RAM.

Ceci est probablement dû au profil même de l'application concernée : la compression est très intensive en accès mémoire au début de son exécution, puisqu'elle effectue ses traitements sur la totalité du fichier en entrée. Ensuite, l'application travaille sur une taille mémoire de plus en plus réduite, au fur et à mesure de l'amélioration du taux de compression. Ce phénomène est visible sur la figure 6.15 rapportant le nombre de fautes de pages de *bzip.combined*

au cours d'une exécution normale. Ceci signifie que le swap token supporte *bzip.combined* pendant sa charge mémoire initiale, et que cette aide est plus importante que la pénalisation ensuite induite par la protection des pages du swappeur.

Le cas d'applications fortement intensives, mais n'obtenant pas un rapport fautes de page par instruction aussi élevé que le swappeur, est cristallisé par *perl.splitmail*. Un tel scénario implique une perte de performance dramatique, supérieure à 400%, soit au moins deux jours et demi, au lieu de 290 secondes sans concurrence et d'une demi-journée avec le même environnement mais sans le swap token.

Ces résultats mettent en avant l'incompatibilité du swap token avec les surcharges mémoire intentionnelles et/ou durables. Dans les systèmes partagés best-effort, où les charges de travail sont égoïstes et n'ont pas un profil prédictible, son activation ne paraît donc pas adaptée.

Dénomination	**Coefficient de variation** *Sw256*	**Coefficient de variation** *Sw512*
hmmer.nph3	-	+3%
h264.sss	-	+21%
bzip.liberty	-	+2%
gcc.200	-	-
perl.checkspam	-	+22%
gcc.166	-	+22%
perl.diffmail	-	+2%
gcc.typeck	-	+17%
perl.splitmail	+9%	N/A
bzip.combined	-	+2%

TABLE 6.16 – Impact de l'activation du swap token sur le coefficient de variation des temps d'exécution des applications étalon. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire. Les tirets représentent des variations négligeables, inférieures à 1%. Une valeur repérée en vert ou en rouge souligne respectivement une évolution significativement positive ou négative.

Impact du swap token de Linux sur la dispersion des durées d'exécution. La table 6.16 rapporte par ailleurs l'effet du swap token sur la dispersion des durées d'exécution observées. L'incertitude sur les durées d'exécution est une deuxième manière d'étudier l'influence des applications abusives sur les applications légitimes, puisque la dispersion doit être d'autant plus faible que l'impact est borné. La métrique présentée est le coefficient de variation, correspondant à l'écart-type normalisé par la moyenne des durées d'exécution.

Ceci nous permet de comparer les résultats pour les différentes applications, indépendamment des ordres de grandeur.

Cette table appuie une nouvelle fois l'inadéquation du swap token avec la présence de charges de travail abusives persistantes. Son activation impose en effet une dispersion significativement accrue des durées d'exécution dans le cas général, avec un coefficient de variation de 45% dans le cas de *gcc.166*, soit 22% de plus que l'installation de Linux basique. En effet, le fait d'imposer aux processus plus légers un nombre supérieur de requêtes d'entrées/sorties induit une plus grande incertitude sur les temps d'exécution pour la majorité de nos applications. De plus, l'attribution du swap token étant directement liée à la fréquence des fautes de page, elle est dépendante des latences de résolution des requêtes d'entrées/sorties, qui ne sont pas constantes d'une exécution à l'autre. Nous notons également que le swap token ne réduit pas non plus l'incertitude sur les charges de travail dont les performances sont significativement améliorées, comme *perl.diffmail* ou *bzip.combined*.

Cette caractéristique repousse encore les garanties en termes de performance que les hébergeurs peuvent proposer aux utilisateurs de systèmes partagés best-effort.

6.2.3 Discussion

Les applications que nous avons sélectionnées dans la suite SPEC CPU2006 sont intéressantes pour notre évaluation expérimentale. En premier lieu, elles sont représentatives de charges de travail typiques des systèmes partagés, à savoir la compilation, la compression de fichiers ou de flux vidéos, le traitement de mails ou encore le séquençage ADN. De plus, les 10 charges de travail choisies présentent des caractéristiques variées, que ce soit en termes d'allocation mémoire, de temps d'exécution moyen, ou encore de comportement sous différents niveaux de contention mémoire.

Enfin, nous avons montré que l'activation du swap token de Linux, destiné à optimiser les performances lors de contentions mémoire, s'avère contre-productif lorsque des applications légitimes sont mises en concurrence avec des applications abusives pendant une période prolongée. Les charges de travail les plus affectées subissent un surplus d'exécution dû à l'activation du swap token supérieur à 100%, et un coefficient de variation de 45%. Cette technique ne paraît donc pas adaptée à l'arbitrage de charges de travail inconscientes de la concurrence et potentiellement antagonistes qui cohabitent dans les systèmes partagés best-effort.

6.3 Évaluation expérimentale de notre couche de contrôle

Afin d'évaluer l'apport de notre prototype, nous avons reproduit des situations de contention mémoire pendant lesquelles sont exécutées les charges de travail SPEC CPU2006 détaillées dans la section précédente. Nous étudions dans cette

section l'impact de notre prototype sur ces charges de travail, en termes de performance et de variabilité d'exécution. Le coeur de notre approche est dérivé de la mesure du temps d'utilisation, ce qui ouvre la porte à l'implantation de diverses politiques de gestion, comme définies dans la section 4.2.

La première politique testée applique un contrôle orienté vers une équité stricte et persistante. La persistance signifie que la facture d'un domaine correspond à son temps d'utilisation cumulé de la ressource sur la totalité de sa durée de vie. Le caractère strict renvoie au fait que les domaines abusifs ne peuvent pas soumettre de requêtes tant que leur statut ne change pas. Les résultats de cette politique à équité stricte et persistante permettent notamment de déterminer l'intensité mémoire des différentes applications ainsi que les carences en termes d'équité et de performance du mécanisme de swap par défaut de Linux.

La deuxième politique est best-effort et persistante. Son aspect best-effort est lié à la maximisation de l'utilisation de la ressource en évitant les famines, comme décrit dans la section 4.2.3. Un domaine abusif peut ainsi soumettre des requêtes tant qu'aucun domaine moins abusif n'a besoin de la ressource.

Enfin, la troisième et dernière politique évaluée applique un contrôle best-effort basé sur une facturation périodique. Cela signifie notamment que le contrôle d'accès est effectué à partir d'une facture ne dépendant que d'un passé récent.

Ces trois politiques nous permettent de souligner les répercussions sur les performances de l'incapacité du mécanisme de swap de Linux à assurer une équité d'accès, que ce soit avec ou sans le swap token. Toutes les expériences ont été menées sur un système Dell Latitude D520 doté de 512 Mo de RAM.

6.3.1 Équité stricte et persistante

Cette section rapporte l'évaluation expérimentale de notre prototype implémenté dans Linux 3.5, décrit dans la section 6.1, configuré pour assurer une équité sur le cycle de vie complet des applications, conformément aux algorithmes 4.10 et 4.11 de la section 4.2. Le prototype respecte une équité stricte, c'est-à-dire qu'un domaine abusif ne peut avoir accès au mécanisme de swap tant que son statut ne change pas. Cette solution ne maximise pas l'exploitation de la ressource, mais permet d'observer l'incidence de l'équité des temps d'utilisation du mécanisme de swap sur le comportement des applications étalon lors de contentions mémoire.

Impact sur les performances. La table 6.17 rapporte l'impact de cette couche de contrôle sur la durée moyenne d'exécution sous contention mémoire des applications étalon décrites dans la section 6.2.1. Les résultats sont comparés aux valeurs obtenues dans la section 6.2.2, avec un système Linux basique ainsi qu'avec le même système agrémenté du swap token.

Comme attendu, la régulation n'a pas d'effet pour les applications nécessitant moins de 256 Mo mises en concurrence avec un swappeur de 256 Mo

Dénomination	Par rapport à Linux basique		Par rapport au swap token	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-62%	-	-62%
h264.sss	-	-68%	-	-81%
bzip.liberty	-	-44%	-	-49%
gcc.200	-	-99%	-	-99%
perl.checkspam	-	-91%	-	-94%
gcc.166	-	-94%	-	-97%
perl.diffmail	+2%	-91%	+2%	-92%
gcc.typeck	-1%	-98%	+7%	-98%
perl.splitmail	-	-16%	+1%	N/A
bzip.combined	-	-80%	+3%	-80%

TABLE 6.17 – Impact d'une politique d'équité stricte et persistante des temps d'utilisation du mécanisme de swap sur la durée d'exécution moyenne des applications étalon. Les durées moyennes sont comparées à celles obtenues avec un système Linux basique, avec ou sans swap token. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire. Les tirets représentent des variations négligeables, inférieures à 1%. Une valeur repérée en vert ou en rouge souligne respectivement une évolution significativement positive ou négative.

puisqu'il n'y a pas de contention mémoire. En revanche, lorsque ces mêmes applications sont exécutées de manière simultanée avec un swappeur de 512 Mo, les résultats exposent amélioration radicale allant de 45% à 99% de réduction des durées moyennes, que ce soit par rapport à un système avec ou sans swap token.

Notre couche de régulation agit comme prévu, puisqu'elle détecte le swappeur comme étant abusif, et bloque son exécution tant que l'application étalon n'a pas exploité au moins autant le mécanisme de swap. Certaines applications obtiennent même de meilleures performances avec un swappeur plus abusif. Par exemple, *h264.sss* s'exécute en 1474 secondes en moyenne avec un swapeur de 256 Mo, et en 1397 secondes avec un swappeur de 512 Mo. En effet, lorsque le swappeur est bloqué, l'application étalon peut jouir d'une meilleure cohérence des caches de données et ne souffre d'aucune concurrence d'accès au processeur.

Pour les applications plus lourdes, notre couche de contrôle a toujours un impact relativement faible à 256 Mo. Certaines applications jouissent de légères améliorations par rapport à un système Linux basique, comme *gcc.typeck*, ou de légères dégradations, par exemple pour *perl.diffmail*. Bien que ces différences soient difficilement significatives sur 10 exécutions, elles soulignent tout de même une tendance compréhensible. *gcc.typeck* ne présente en effet que deux pics d'allocation mémoire allant au-delà de 256 Mo, pendant moins de

25% de son temps d'exécution ; il paraît donc normal que l'application soit légèrement moins intensive qu'un swappeur sur sa durée complète d'exécution. Au contraire, *perl.diffmail* semble très intensif dans ses accès mémoire et effectue une allocation en rampe qui oblige la couche de contrôle à ralentir considérablement la fin de son exécution. Ces observations sont d'ailleurs appuyées par le fait que notre couche de contrôle a un effet contraire à celui du swap token à 256 Mo, et pénalise donc bien les applications les plus intensives.

Le cas d'un swappeur de 512 Mo a les mêmes conséquences sur les applications lourdes que sur celles nécessitant moins de 256 Mo, à savoir une amélioration significative des performances. Ceci souligne notamment la décorrélation entre la pression émise par une charge de travail sur le mécanisme de swap et son besoin en allocation mémoire. Par exemple, bien que *perl.splitmail* ait une allocation mémoire constamment au-dessus de 512 Mo, elle n'atteint pas le ratio de fautes de page par instruction d'un swappeur de 512 Mo.

Impact sur la dispersion des durées d'exécution. La table 6.18 rapporte l'impact de cette couche de contrôle sur la dispersion des durées moyennes d'exécution de l'expérience ci-dessus. Les coefficients de variation sont comparés à ceux obtenus dans la section 6.2.2, avec un système Linux basique ainsi qu'avec le même système agrémenté du swap token.

Dénomination	Par rapport à Linux basique		Par rapport au swap token	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-1%	-	-4%
h264.sss	-	-4%	-	-25%
bzip.liberty	-	-2%	-	-3%
gcc.200	-	+2%	-	+2%
perl.checkspam	-	-	-	-22%
gcc.166	-	-13%	-	-35%
perl.diffmail	-	-1%	-	-3%
gcc.typeck	-2%	-4%	-1%	-17%
perl.splitmail	-3%	-7%	-12%	N/A
bzip.combined	-	+2%	-	-

TABLE 6.18 – Impact d'une politique d'équité stricte et persistante des temps d'utilisation du mécanisme de swap sur le coefficient de variation des temps d'exécution des applications étalon. Les coefficients de variation sont comparés à ceux obtenus avec un système Linux basique, avec ou sans swap token. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire. Les tirets représentent des variations négligeables, inférieures à 1%. Une valeur repérée en vert ou en rouge souligne respectivement une évolution significativement positive ou négative.

De manière décorrélée de l'amélioration ou de la dégradation des performances, cette table montre qu'une telle politique d'équité réduit globalement la dispersion des durées d'exécution. La différence avec les coefficients de variation des expériences effectuées sous Linux basique est cependant globalement faible. Ceci est notamment dû à la grande diminution des moyennes d'exécution qui dissimule une réduction très importante des écarts-types. Par exemple, *gcc.200* présente une légère augmentation en termes de coefficient de variation, alors que l'écart-type de 530 secondes initialement est fortement amoindri, à 11 secondes seulement.

La comparaison avec le swap token est plus nette et la différence des coefficients de variation culmine à 35% dans le cas de *gcc.166*. Le coefficient de variation est également réduit pour *perl.diffmail*, *gcc.typeck* et *perl.splitmail* avec un swappeur de 256 Mo, alors que notre couche de contrôle dégrade leurs performances. Ceci montre la capacité d'une équité sur les temps d'utilisation à réguler justement l'usage du mécanisme de swap, puisque même les charges abusives jouissent d'une meilleure prédictabilité dans leurs exécutions.

6.3.2 Équité best-effort et persistante

Cette section rapporte l'évaluation expérimentale de notre prototype implémenté dans Linux 3.5, décrit dans la section 6.1, configuré pour assurer une équité sur le cycle de vie complet des applications conformément aux algorithmes 4.10 et 4.11 de la section 4.2.

Le prototype respecte une équité best-effort, c'est-à-dire qu'un domaine abusif peut avoir accès au mécanisme de swap s'il n'existe pas de requêtes en attente émanant d'autres domaines ayant moins exploité la ressource. Cette solution permet de maximiser l'exploitation de la ressource tout en conservant la propriété d'équité désirable pour les systèmes partagés best-effort.

Impact sur les performances. La table 6.19 rapporte l'impact de cette couche de contrôle sur la durée moyenne d'exécution sous contention mémoire des applications étalon décrites dans la section 6.2.1. Les résultats sont comparés aux valeurs obtenues dans la section 6.2.2, avec un système Linux basique ainsi qu'avec le même système agrémenté du swap token.

Comme attendu, notre heuristique best-effort qui évite la famine du mécanisme de swap en transmettant certaines requêtes des applications abusives ne permet pas d'obtenir d'aussi bonnes performances qu'une équité stricte. En effet, la résolution d'une faute de page implique également le déchargement d'une ou plusieurs autres pages, appartenant probablement au processus légitime. Les requêtes supplémentaires obligent donc naturellement ce processus légitime à effectuer lui-même plus de fautes de page.

Dénomination	Par rapport à Linux basique		Par rapport au swap token	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-	-	-
h264.sss	-	-35%	-	-62%
bzip.liberty	-	-17%	-	-24%
gcc.200	-	-56%	-	-78%
perl.checkspam	-	-8%	-	-41%
gcc.166	-	-64%	-	-83%
perl.diffmail	+2%	-72%	+1%	-73%
gcc.typeck	-	-80%	+7%	-80%
perl.splitmail	+2%	-7%	+2%	N/A
bzip.combined	-	-60%	+3%	-59%

TABLE 6.19 – Impact d'une politique d'équité best-effort et persistante des temps d'utilisation du mécanisme de swap sur la durée d'exécution moyenne des applications étalon. Les durées moyennes sont comparées à celles obtenues avec un système Linux basique, avec ou sans swap token. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire.

L'apport de notre couche de contrôle demeure cependant significativement positif en termes de performance et les durées moyennes d'exécution des processus les plus lourds demeurent comparables aux valeurs obtenues avec une politique d'équité stricte. Notre heuristique apparaît donc comme un compromis raisonnable et performant entre la maximisation de l'utilisation de la ressource et la conservation d'une répartition juste de la capacité de traitement.

Impact sur la dispersion des durées d'exécution. La table 6.20 rapporte l'impact de cette couche de contrôle sur la dispersion des durées moyennes d'exécution de l'expérience ci-dessus. Les coefficients de variation sont comparés à ceux obtenus dans la section 6.2.2, avec un système Linux basique ainsi qu'avec le même système agrémenté du swap token.

En termes de dispersion des temps d'exécution, cette politique conserve son caractère proche d'un système Linux basique, mais la réduction de la dispersion n'est pas aussi nette qu'avec une équité stricte. Ceci signifie notamment que notre couche de contrôle se rapproche d'un système Linux basique en termes de proportion d'entrées/sorties effectuées tout en affichant des gains en performance significatifs. Notre heuristique best-effort semble donc appropriée car elle maximise l'usage de la ressource supervisée de manière semblable à une installation basique, tout en induisant un ordonnancement des requêtes plus équitable.

Dénomination	Par rapport à Linux basique		Par rapport au swap token	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-	-	-
h264.sss	-	-3%	-	-24%
bzip.liberty	-	-	-	-1%
gcc.200	-	+1%	-	+2%
perl.checkspam	-	-	-	-20%
gcc.166	-	-15%	-	-37%
perl.diffmail	-	-2%	-	-4%
gcc.typeck	-2%	+2%	-1%	-14%
perl.splitmail	-	-7%	-10%	N/A
bzip.combined	-	-	-	-2%

TABLE 6.20 – Impact d'une politique d'équité best-effort et persistante des temps d'utilisation du mécanisme de swap sur le coefficient de variation des temps d'exécution des applications étalon. Les coefficients de variation sont comparés à ceux obtenus avec un système Linux basique, avec ou sans swap token. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire.

6.3.3 Équité best-effort et périodique

Cette section rapporte l'évaluation expérimentale de notre prototype implémenté dans Linux 3.5, décrit dans la section 6.1, configuré pour assurer une équité entre les applications sur une période de temps arbitrairement définie à 1 seconde, conformément à l'algorithme 4.12 de la section 4.2.3.

Le prototype respecte une équité best-effort, c'est-à-dire qu'un domaine abusif peut avoir accès au mécanisme de swap s'il n'existe pas de requêtes en attente émanant d'autres domaines ayant moins exploité la ressource.

Impact sur les performances. La table 6.21 rapporte l'impact de cette couche de contrôle sur la durée moyenne d'exécution sous contention mémoire des applications étalon décrites dans la section 6.2.1. Les expériences précédentes offrent une vision claire des apports et limites du swap token lors d'une contention mémoire prolongée. De ce fait, cette table compare l'équité best-effort et périodique au cas de Linux basique ainsi qu'au cas de l'équité best-effort et persistante présenté dans la section précédente. Ceci nous permet d'évaluer directement l'impact du caractère périodique de la facturation.

La comparaison avec un système Linux basique suit la tendance des politiques d'équité persistantes, à savoir une amélioration significative des performances des charges de travail les moins intensives. Cette observation valide notre approche avec une politique d'équité réaliste et applicable dans le cas général. La charge de travail *perl.splitmail* fait cependant office d'exception

Dénomination	Par rapport à Linux basique		Par rapport à équité persistante	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-1%	-	-1%
h264.sss	-	-35%	-	-
bzip.liberty	-	-15%	-	+1%
gcc.200	-	-40%	-	+39%
perl.checkspam	-	-7%	-	+1%
gcc.166	-	-64%	-	-
perl.diffmail	+2%	-62%	-	+35%
gcc.typeck	+1%	-71%	-	+49%
perl.splitmail	-1%	+4%	-4%	+12%
bzip.combined	-	-60%	-	-

TABLE 6.21 – Impact d'une politique d'équité best-effort et périodique des temps d'utilisation du mécanisme de swap sur la durée d'exécution moyenne des applications étalon. Les durées d'exécution sont comparées à celles obtenues avec un système Linux basique ainsi qu'avec notre couche de contrôle best-effort et persistante. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire.

puisque sa performance est dégradée de 4% lorsque mise en concurrence avec un swappeur de 512 Mo. Bien que ce faible pourcentage puisse être dû à la taille relativement faible de notre échantillon, nous pensons qu'il est significatif et lié à deux facteurs principaux : l'implémentation de notre prototype et le caractère statique de la période de facturation.

Au niveau de l'implémentation, notre heuristique best-effort implique un nombre plus important de fautes de page, donc un appel plus fréquent à la mise à jour de la facture et du statut de chaque domaine. Or, une vérification de l'existence d'un domaine actif moins monopolisateur est nécessaire à chaque requête émise par un domaine abusif ainsi qu'à chaque mise à jour des factures. Cette recherche est effectuée en $\mathcal{O}(n)$ lors de l'arrivée de nouvelles requêtes abusives, mais en $\mathcal{O}(n^2)$ pour la mise à jour de la facturation. Il est donc probable que cette implémentation sous-optimale provoque l'apparition d'un léger phénomène de QoS crosstalk pendant les contentions sévères.

De plus, le fait d'utiliser une période arbitraire de 1 seconde pénalise particulièrement certaines charges de travail. *perl.splitmail* par exemple n'est que légèrement moins intensive que le swappeur exécuté simultanément. Ce fait est particulièrement visible dans l'évaluation de performance des deux autres politiques, puisque *perl.splitmail* ne bénéficie que d'améliorations modestes. Dans ce cadre, une facturation sur une période trop courte ne permet pas à la couche de contrôle d'avoir une vision suffisamment précise de la pression imposée par chaque application. Le profil d'allocation mémoire de *perl.splitmail*, présenté

plus tôt sur la figure 6.3 de la section 6.2.1, montre des pics d'allocation espacés de 10 secondes au minimum et une période globale de 50 secondes sur ses allocations. Une période plus proche de ces valeurs permettrait ainsi d'obtenir une facturation plus réaliste de sa consommation.

Cette table confirme cependant l'application d'un compromis généralement satisfaisant entre les performances d'un système Linux basique et celles du cas de l'équité best-effort persistante.

Impact sur la dispersion des durées d'exécution. La table 6.22 rapporte l'impact de cette politique périodique sur la dispersion des durées moyennes d'exécution de l'expérience ci-dessus. Les coefficients de variation sont comparés à ceux obtenus pour un système Linux basique ainsi que pour notre couche assurant une équité best-effort persistante.

Dénomination	Par rapport à Linux basique		Par rapport à équité persistante	
	Sw256	*Sw512*	*Sw256*	*Sw512*
hmmer.nph3	-	-	-	-
h264.sss	-	-2%	-	-
bzip.liberty	-	-	-	-
gcc.200	-	-2%	-	-3%
perl.checkspam	-	+2%	-	+1%
gcc.166	-	-6%	-	+9%
perl.diffmail	-	+3%	-	+5%
gcc.typeck	-	-	-	-2%
perl.splitmail	-2%	-7%	-	-
bzip.combined	-	-2%	-	-2%

TABLE 6.22 – Impact d'une politique d'équité best-effort et périodique des temps d'utilisation du mécanisme de swap sur le coefficient de variation des temps d'exécution des applications étalon. Les coefficients de variation sont comparés à ceux obtenus avec un système Linux basique ainsi qu'avec notre couche de contrôle best-effort et persistante. Les applications sont en concurrence avec des swappeurs utilisant 256 (*Sw256*) et 512 (*Sw512*) Mo de mémoire.

Le caractère périodique de la facturation atténue encore les différences avec un système Linux basique. Seulement deux charges de travail permettent d'observer une variation significative de la dispersion des durées moyennes d'exécution. Pour *perl.splitmail*, le coefficient de variation obtenu est similaire à celui observé pour le cas de l'équité persistante. Le cas de *gcc.166* est conforme à nos attentes, puisque la dispersion observée est comprise entre celle d'un système Linux basique et celle fournie par une politique d'équité best-effort persistante.

Cette politique d'équité best-effort, basée sur une facturation périodique, offre donc une dispersion des durées d'exécution semblable à un système Linux basique, voire plus faible pour certaines charges de travail. Cependant notre couche de contrôle réduit considérablement les durées moyennes d'exécution. Cela signifie qu'elle diminue significativement l'incertitude absolue sur la durée d'exécution d'une charge de travail, même lors de contentions sévères.

6.3.4 Discussion

Notre évaluation expérimentale nous permet de comparer l'impact de notre prototype sur le comportement d'applications intensives en mémoire lors de contentions, par rapport au mécanisme de swap de Linux par défaut, ainsi que par rapport à l'état de l'art représenté par le swap token. Afin de résumer les résultats obtenus, la figure 6.23 rapporte pour chacun de ces systèmes la durée moyenne d'exécution de l'ensemble de nos applications étalon, ainsi que l'incertitude observée sur 10 exécutions.

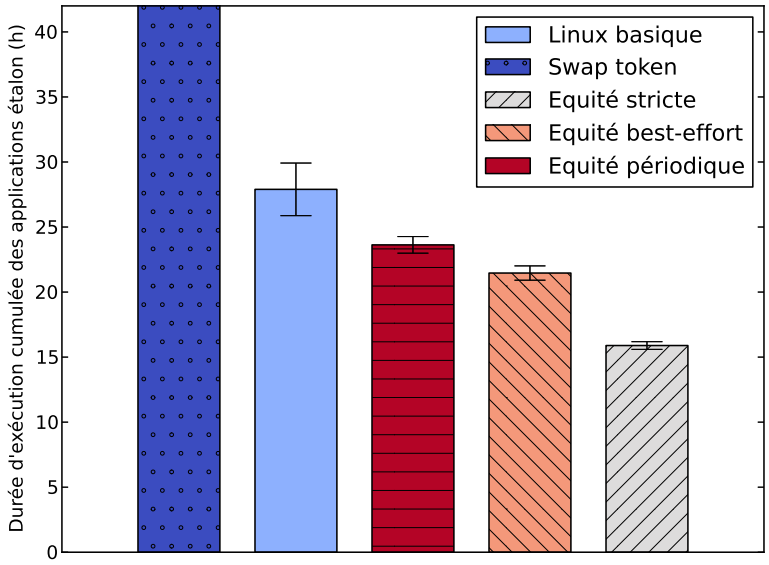


FIGURE 6.23 – Comparaison de la durée moyenne d'exécution de l'ensemble des applications étalon entre les différents systèmes évalués.

Pour Linux basique, dont l'équité du mécanisme de swap se base sur son ordonnanceur d'entrées/sorties CFQ, la durée d'exécution cumulée s'élève à 1

jour et 4 heures en moyenne, plus ou moins 2h. Cette figure montre que le swap token n'est pas adapté aux surcharges lourdes et prolongées, le temps moyen d'exécution moyen étant de minimum 3 jours et 11 heures. Même en ignorant les résultats pour l'application la plus intensive, *perl.splitmail*, le swap token induit une dégradation des performances de 40% et une incertitude 5 fois plus grande sur les durées d'exécution.

Afin d'évaluer notre approche, nous avons instancié trois politiques d'équité sur les temps d'utilisation du mécanisme de swap. La première, l'équité stricte, facture les applications sur leur temps de vie complet et ne permet pas aux charges de travail abusives de consommer la ressource tant que leurs concurrentes ne l'ont pas exploitée au moins autant. Dans ce contexte, la durée d'exécution d'une application légitime mise en concurrence avec une charge de travail abusive est naturellement fortement réduite. La durée d'exécution moyenne de la totalité des applications étalon est de seulement 16 heures sous cette politique, avec 17 minutes d'incertitude.

L'ajout de notre heuristique best-effort visant à maximiser l'utilisation de la ressource obtient toujours de meilleures performances qu'une installation Linux basique, avec 23 heures et 30 minutes en moyenne pour l'exécution de l'ensemble des applications étalon. L'incertitude est légèrement accrue par rapport à une équité stricte, aux environs de 30 minutes. Cependant, le détail au cas par cas effectué au sein de la section 6.3.2 montre que la variation sur les temps d'exécution n'est proportionnellement pas très éloignée de celle obtenue pour une installation Linux basique. Ceci implique un nombre d'entrées/sorties effectuées comparable pour les deux systèmes et valide l'impact souhaité de notre heuristique best-effort en termes de maximisation de l'utilisation de la ressource.

Enfin, notre dernière instance ne maintient qu'une facture sur une période passée courte, fixée arbitrairement à une seconde. Cette politique est ainsi applicable dans le cas général d'un système partagé best-effort. La figure 6.23 montre qu'une telle politique demeure plus efficace lors de contentions qu'un système Linux basique. Les résultats de la section 6.3.3 exposent cependant des pertes de performance pour certaines charges de travail particulièrement intensives en accès mémoire. Ceci est probablement dû à une implémentation non-optimisée de notre facturation, ainsi qu'à l'inadéquation de la période fixée arbitrairement. L'incertitude absolue globalement induite demeure tout de même plus faible, de l'ordre de 40 minutes.

L'ensemble de ces résultats valide ainsi notre approche, visant à assurer un ordonnancement équitable et performant de requête simultanées vers une ressource système surchargée.

6.4 Conclusion

Afin d'évaluer notre approche de contrôle des ressources par facturation du temps d'utilisation, nous avons développé un prototype permettant la régulation du mécanisme de swap dans le noyau Linux. Ce système d'exploitation a

notamment été choisi en raison de sa popularité, de son déploiement parmi les systèmes partagés best-effort, et de la disponibilité de techniques avancées de gestion de la mémoire virtuelle.

Nous avons sélectionné une dizaine de charges de travail, typiques des systèmes partagés best-effort, parmi celles disponibles dans la suite SPEC CPU2006 [Hen06], représentant un panel varié en termes de consommation et de fréquence d'accès mémoire. Notre évaluation expérimentale étudie le comportement de ces charges de travail lors de fortes contentions mémoire, sous 5 configurations du système Linux et de notre prototype.

La première configuration étudiée est celle par défaut du système Linux dont l'équité du mécanisme de swap se repose sur son ordonnanceur d'entrées/sorties CFQ. La deuxième représente l'état de l'art académique, via l'activation du swap token [Jia09]. Enfin, nous avons configuré notre prototype afin de respecter trois politiques distinctes d'équité sur les temps d'utilisation du mécanisme de swap. La première est une équité stricte où les applications abusives sont mises en attente tant que leurs concurrentes n'ont pas consommé une part égale de la ressource. La deuxième est une équité best-effort qui permet aux applications abusives de tout de même émettre des requêtes lorsque celles-ci ne pénalisent pas d'autres applications. Ces deux politiques sont persistantes, c'est-à-dire qu'une facture correspond au temps d'utilisation cumulé de toutes les requêtes d'une application sur son cycle de vie. Notre dernière politique se libère de cette limitation en ne prenant en compte que la consommation sur une durée passée fixée arbitrairement à 1 seconde et conserve le caractère best-effort de notre deuxième politique.

Nos expériences montrent en premier lieu l'inadéquation du swap token avec les contentions mémoire importantes ou prolongées puisqu'il favorise les charges de travail intensives. Les gains en performance obtenus pour les applications intensives sont modestes, tandis que les dégradations observées pour les charges plus légères sont particulièrement sévères. De plus, les incertitudes sur les temps d'exécution demeurent très importantes, même pour les charges de travail favorisées. Au contraire, notre prototype configuré pour le respect d'une équité stricte démontre un potentiel d'amélioration et de cloisonnement des performances très significatif. Ceci prouve qu'un ordonnancement équitable est crucial pour une ressource comme le mécanisme de swap. En effet, une requête provenant d'une application abusive a un double effet négatif : non seulement elle exploite la ressource à la place de concurrents plus raisonnables, mais elle provoque de plus le déchargement de leurs pages mémoire, les forçant à émettre des fautes de page supplémentaires.

L'apport de notre prototype est également validé pour les configurations best-effort puisque les performances des charges de travail testées sont nettement améliorées par rapport à un système Linux basique. Les charges de travail les plus intensives mettent cependant en avant certaines limites de notre prototype, notamment vis-à-vis de l'implémentation de notre heuristique best-effort ainsi que du caractère arbitraire et statique de notre période de facturation. Au-delà des performances, notre couche d'équité permet une réduction visible des incertitudes sur les durées d'exécution des charges de travail. Le nombre d'en-

trées/sorties effectuées par seconde semble proportionnellement comparable au cas d'un système Linux basique, ce qui confirme l'impact souhaité de notre heuristique en termes de maximisation de l'exploitation.

Chapitre 7

Bilan et perspectives

7.1 Synthèse

Avec cette thèse, notre objectif a été d'étudier le problème posé par la concurrence d'accès aux ressources dans les systèmes partagés best-effort. Un nombre important des ressources typiques de ces systèmes, comme les entrées/sorties, les services transparents du système ou encore les ressources finies, s'avèrent particulièrement problématiques dans la mise au point d'un compromis entre un usage maximisé et une répartition équitable des accès. Nous avons notamment mis en évidence la possibilité pour un utilisateur abusif voire malveillant de réduire significativement la qualité de service délivrée à d'autres utilisateurs concurrents, en prenant avantage de faiblesses dans les heuristiques d'optimisation ou dans l'implémentation des gestionnaires de ressources. Cette observation demeure valide pour les différents types de systèmes partagés best-effort déployés dans l'industrie du service informatique, comme les systèmes mutualisés ou les environnements de virtualisation.

Les algorithmes destinés à un ordonnancement équitable des requêtes simultanées, ou ceux dédiés à l'optimisation du taux de service d'une ressource particulière, ne permettent pas à eux seuls de fournir des interfaces justes et robustes aux comportements abusifs. La littérature scientifique pointe la nécessité d'introduction de couches de facturation et de contrôle de la consommation de chaque ressource par les différents utilisateurs d'un système. Cependant, les instances de ces travaux se focalisent souvent sur une ressource en particulier et nécessitent une mise au point non triviale de paramètres de configuration dédiés. Les hébergeurs de systèmes partagés best-effort résolvent typiquement cette problématique de manière statique, en restreignant chaque utilisateur à un quota de consommation fixe. Une telle approche sous-utilise les ressources à disposition et nécessite toujours une configuration spécifique à chaque ressource, par exemple en termes de bande passante minimum ou d'allocation maximum.

L'introduction d'une métrique générique de consommation des ressources paraît ainsi nécessaire à la mise en place de couches de contrôle unifiées et

simples à administrer. Nous montrons que le temps d'utilisation est une métrique applicable aux différentes ressources typiques des systèmes partagés best-effort, et qu'elle permet une représentation précise de la pression imposée par chaque utilisateur lorsque rapportée en pourcentage de l'utilisation totale. Cette métrique nous permet de définir un modèle de facturation et de contrôle générique de la consommation, dénué de paramètres de configuration dédiés à une ressource en particulier. Nous proposons la protection des gestionnaires de ressource par une couche de facturation, interceptant les requêtes et ne nécessitant qu'une légère modification des couches logicielles existantes, afin de déduire le temps d'utilisation effectif des requêtes. Ceci nous permet de distinguer les utilisateurs légitimes de ceux monopolisant la ressource. Il est ainsi possible d'implémenter de façon simple diverses politiques d'équité en différant la transmission des requêtes émises par les utilisateurs abusifs afin de favoriser celles des utilisateurs plus raisonnables. Afin de tirer profit de la ressource au maximum, il est possible de servir les requêtes des utilisateurs abusifs lorsque celles-ci n'en pénalisent pas d'autres.

Nous avons validé cette approche en l'appliquant à la régulation du mécanisme de swap de Linux. Une surcharge de consommation mémoire est un phénomène ayant un impact particulièrement fort sur la performance des applications dans un système partagé best-effort. Un accès non équitable au mécanisme de swap permet aux applications abusives de monopoliser l'occupation de la mémoire centrale au détriment de leurs concurrentes. Pourtant, peu de travaux se sont intéressés à l'équité de ce service système et certaines techniques d'optimisation à l'état de l'art, comme le swap token, prônent au contraire la protection des charges de travail les plus intensives. Nos expériences mettent en avant l'inadéquation de telles techniques avec les contentions intensives ou durables dont peuvent être victimes les systèmes partagés best-effort.

Au contraire, notre prototype montre que le respect d'un partage équitable du temps d'utilisation du mécanisme de swap permet d'améliorer de manière significative la performance de charges de travail intensives en accès mémoire lors de fortes contentions. Il permet également de cloisonner les performances de chaque application en induisant une réduction des incertitudes sur les durées d'exécution. La dispersion statistique des durées d'exécutions observées avec notre prototype demeure proportionnellement comparable à celle observée avec un système Linux basique. Ce fait permet de déduire l'usage d'une quantité comparable d'entrées/sorties et donc un taux d'utilisation du mécanisme de swap similaire. Notre couche de contrôle évite ainsi la sous-utilisation tout en offrant une répartition équitable des accès à une ressource surchargée.

7.2 Perspectives

Nos travaux permettent d'envisager diverses améliorations et nouvelles orientations de notre approche, notamment en l'associant avec certaines propositions de la littérature.

Amélioration de notre prototype. En premier lieu, notre évaluation expérimentale a mis en avant une marge d'amélioration existante pour certains algorithmes implémentés dans notre prototype. Par exemple, notre heuristique best-effort, évaluant l'aptitude des domaines de facturation abusifs à tout de même émettre des requêtes, exhibe une complexité en $\mathcal{O}(n^2)$ vis-à-vis du nombre de domaines. Or, cet algorithme peut aisément être allégé en maintenant une structure de données ordonnant efficacement les domaines selon leur consommation. L'ordonnanceur CFS de Linux manie par exemple des arbres rouges et noirs, dont les noeuds représentent les processus et sont triés en fonction du temps pendant lequel ils ont attendu le processeur [Pab09]. Les opérations de recherche et d'insertion dans une telle structure de données présentent une complexité logarithmique, plus adéquate. Cette structure de données offrirait également de meilleures performances pour les autres opérations manipulant les factures, comme la mise à jour des statuts.

De plus, une telle modification paraît nécessaire pour le passage à l'échelle de notre couche de contrôle. En effet, bien que certaines expériences non présentées dans ce travail suggèrent que notre prototype supporte aisément une dizaine d'utilisateurs concurrents, certains systèmes mutualisés gèrent une quantité d'utilisateurs d'un ou deux ordres de grandeur supérieure, ce qui renforce le besoin d'algorithmes efficaces.

Facturation hiérarchique et interactive. Certains travaux de la littérature soulignent les bénéfices d'ordonnancements hiérarchiques et interactifs avec les applications, et cette technique a été adoptée par de nombreuses architectures de contrôle des ressources [LIJ97, VGR98, SCG+00, Sto07]. Notre prototype pourrait ainsi être enrichi en supportant une hiérarchisation des domaines de facturation, potentiellement dirigée par les applications. Un serveur HTTP mutualisé pourrait ainsi définir et modifier de manière dynamique un certain nombre de sous-domaines, chacun lié aux threads servant les requêtes d'un site en particulier. Notre couche de facturation serait ainsi capable de niveler la consommation en fonction du besoin spécifique de l'application. De la même façon, la possibilité de consulter factures et statuts permettrait aux applications d'adapter leur usage, pour leur propre performance ainsi que celle du système.

Généralisation de l'approche. De plus, notre prototype ne profite pas du caractère générique du temps d'utilisation en tant que métrique de consommation. Il serait ainsi pertinent de poursuivre ces travaux par l'application de notre couche de contrôle aux multiples ressources gérés par les systèmes partagés best-effort. La régulation des entrées/sorties serait effectuée de manière proche de l'implémentation déjà en place pour le mécanisme de swap, puisque son traitement interne est similaire. Cependant, l'apport d'une telle couche sur le service de ressources logicielles finies comme les nombres aléatoires reste à évaluer. De plus, il serait particulièrement intéressant d'étudier l'impact de l'application simultanée de notre approche à plusieurs ressources. En effet, les

applications peuvent se retrouver dans des situations où leur caractère abusif par rapport à une ressource les pénalise dans l'usage d'une autre. La résolution de contentions simultanées autour de plusieurs ressources a notamment été étudié de manière formelle par Padala *et al.* [PHS+09]. Ce travail pourrait spécialement être mené en instanciant l'architecture générique de contrôle automatisé des ressources dans les micro-noyaux, que nous avons définie dans la section 4.3.

Évolution de la facturation périodique. Enfin, parmi les politiques d'équité que nous avons instanciées, celle basée sur une facturation périodique paraît la plus aisément applicable au cas général des systèmes partagés best-effort. Cependant, elle nécessite l'emploi d'un paramètre arbitraire fixe, la période de facturation. Or, toute période arbitraire peut se révéler sous-optimale en fonction du profil des charges de travail arbitrées. Afin d'atténuer ce problème, nous proposons l'introduction d'une période de facturation dynamique, qui évoluerait en fonction de l'usage constaté de la ressource. En effet, une période de facturation optimale doit induire une utilisation maximisée de la ressource à moyen terme. Ainsi, la couche de facturation peut sporadiquement tenter un agrandissement ou une diminution de la période de facturation et surveiller l'impact de ce changement sur les périodes à venir. Si le temps d'utilisation de la ressource augmente, alors la modification peut être considérée comme positive et répétée. Dans le cas contraire, un retour à l'ancienne période serait opéré. En ajoutant à ce mécanisme certaines techniques avancées d'évaluation et d'estimation des besoins, comme la moyenne mobile exponentielle [BM11], il serait possible d'offrir une évolution perpétuelle de la période de facturation vers un optimum local dépendant de l'environnement d'exécution courant.

Toutes ces pistes de développement et de recherche fournissent des perspectives intéressantes, à considérer dans une optique de poursuite de nos travaux.

Annexe A

Sources du prototype

A.1 Fichiers ajoutés au noyau Linux v3.5

A.1.1 include/linux/task_swap_accounting.h

```
#ifndef __TASK_SWAP_ACCOUNTING_H
#define __TASK_SWAP_ACCOUNTING_H

#include <linux/spinlock.h>
#include <linux/wait.h>

#define _SWPACC_MIN_UID 1000

#define _SWPACC_ANTI_STARVATION
#define _SWPACC_PERIODIC_ACCOUNTING

#ifdef _SWPACC_PERIODIC_ACCOUNTING
 #define _SWPACC_HISTORY_SIZE 10
 #define _SWPACC_PERIOD_MS 1000
#endif

struct swap_domain {
  struct list_head domains_list;
  spinlock_t domain_lock;
  u64 swap_time;
  atomic_t abusive;
  int is_active;
  int domain_owner;
  atomic_t pending_requests;
#ifdef _SWPACC_ANTI_STARVATION
  atomic_t starvation_bypass;
#endif
#ifdef _SWPACC_PERIODIC_ACCOUNTING
```

```
  u64 history[_SWPACC_HISTORY_SIZE];
#endif
  struct page * last_page;
  wait_queue_head_t abusive_tasks;
};

#endif
```

A.1.2 include/linux/task_swap_accounting_ops.h

```
#ifndef __TASK_SWAP_ACCOUNTING_OPS_H
#define __TASK_SWAP_ACCOUNTING_OPS_H

#include <linux/blkdev.h>

/* Delay if current domain is abusive */
void swap_acct_delay_if_abusive(void);

/* Initialize bio for swap accounting */
void swap_acct_bio_check(struct bio* bio);

/* Callback for accounted thread destruction */
void swap_acct_destroy(struct task_struct *process);

/* Callback before request is passed to hardware */
void swap_acct_start(struct request * rq);

/* Callback at the end of a swapin operation */
void swap_acct_end(struct bio * bio);

#endif
```

A.1.3 include/linux/swap_domains.h

```
#ifndef __SWAP_ACCOUNTING_DOMAINS_H
#define __SWAP_ACCOUNTING_DOMAINS_H

int __init init_domains(void);

struct swap_domain * get_domain_locked(
  int id,
  int create_if_none,
  int force);

struct swap_domain * get_domain(int id);
```

```
void release_domain(struct swap_domain * domain, int);

void update_domains(void);

void rotate_histories(unsigned int window);

void reevaluate_starving(struct swap_domain * domain, int, int);

void swap_domain_start_request(struct swap_domain * domain);

void swap_domain_end_request(struct swap_domain * domain);

#endif
```

A.1.4 mm/swap_domain.c

```
#include <linux/spinlock.h>
#include <linux/slab.h>
#include <linux/sched.h>

#include <asm/atomic.h>

#include <linux/task_swap_accounting.h>

struct list_head domains_list;
spinlock_t domains_list_lock;

static unsigned int nb_current_domains;
#ifdef _SWPACC_PERIODIC_ACCOUNTING
static unsigned int current_active_domains;
#endif
#define MAX_SWAP_DOMAINS (PAGE_SIZE/sizeof(struct swap_domain))

static struct swap_domain * domain_pool;

int __init init_domains(void) {
        int i;

        INIT_LIST_HEAD(&domains_list);
        spin_lock_init(&domains_list_lock);

        if (!(domain_pool = kmalloc(
                                MAX_SWAP_DOMAINS*sizeof(struct swap_domain),
                                    GFP_KERNEL))) {
                printk("[SWPACC] Unexpected error: "
                        "could not allocate domain pool\n");
```

```
                return 1;
        }
        for (i=0;i<MAX_SWAP_DOMAINS;i++) {
                domain_pool[i].is_active = 0;
                spin_lock_init(&domain_pool[i].domain_lock);
        }

        nb_current_domains = 0;
#ifdef _SWPACC_PERIODIC_ACCOUNTING
        current_active_domains = 0;
#endif

        return 0;
}

static void lock_domains_list(void) {
        spin_lock(&domains_list_lock);
}
static void unlock_domains_list(void) {
        spin_unlock(&domains_list_lock);
}

void reevaluate_starving(
   struct swap_domain * origin,
   int have_domains_lock,
   int have_orig_lock) {
#ifdef _SWPACC_ANTI_STARVATION
        struct swap_domain * domain;
        struct list_head * lh_domain;

        if (!have_domains_lock)
                lock_domains_list();
        if (!have_orig_lock)
                spin_lock(&origin->domain_lock);
        atomic_set(&origin->starvation_bypass, 1);
        list_for_each(lh_domain, &domains_list) {
                domain = list_entry(lh_domain,
                                struct swap_domain, domains_list);
                if (domain == origin) continue;
                spin_lock(&domain->domain_lock);
                if (atomic_read(&domain->pending_requests)
                    && domain->swap_time < origin->swap_time)
                        atomic_set(&origin->starvation_bypass, 0);
                spin_unlock(&domain->domain_lock);
                if (!atomic_read(&origin->starvation_bypass))
                        break;
```

```
        }
        if (!have_domains_lock)
                unlock_domains_list();
        if (!have_orig_lock)
                spin_unlock(&origin->domain_lock);
#endif
}
void swap_domain_start_request(struct swap_domain * domain) {
  atomic_inc(&domain->pending_requests);
}
void swap_domain_end_request(struct swap_domain * domain) {
  atomic_dec(&domain->pending_requests);
}

struct swap_domain * get_domain_locked(
   int id,
   int create_if_none,
   int force) {
        struct swap_domain * domain;
        struct list_head * lh_domain;
        int i;

        if (!id || (!force && current_uid() < _SWPACC_MIN_UID))
                return NULL;

        if (!create_if_none && !nb_current_domains)
                return NULL;

        lock_domains_list();
        list_for_each(lh_domain, &domains_list) {
                domain = list_entry(lh_domain,
                            struct swap_domain, domains_list);
                spin_lock(&domain->domain_lock);
                if (domain->domain_owner == id) {
                        unlock_domains_list();
                        return domain;
                }
                spin_unlock(&domain->domain_lock);
        }
        unlock_domains_list();

        if (!create_if_none || !domain_pool)
                return NULL;

        /* Need to create a new domain */
        i = (id % MAX_SWAP_DOMAINS);
```

```
        do {
                spin_lock(&domain_pool[i].domain_lock);
                if (!domain_pool[i].is_active) {
                   domain_pool[i].is_active = 1;
                   domain_pool[i].domain_owner = id;
                   atomic_set(&domain_pool[i].abusive, 0);
                   domain_pool[i].swap_time = 0;
#ifdef _SWPACC_ANTI_STARVATION
                   atomic_set(&domain_pool[i].pending_requests, 0);
                   atomic_set(&domain_pool[i].starvation_bypass, 0);
#endif
#ifdef _SWPACC_PERIODIC_ACCOUNTING
                   memset(&domain_pool[i].history,
                                   0, sizeof(&domain_pool[i].history));
#endif
                   init_waitqueue_head(&domain_pool[i].abusive_tasks);

                   lock_domains_list();
                   list_add(&domain_pool[i].domains_list, &domains_list);
                   nb_current_domains++;
#ifdef _SWPACC_DEBUG_DOMAINS
                   printk("New domain for pid %d(%d): %p (%d)\n",
                           current->pid,  id,
                         &domain_pool[i], nb_current_domains);
#endif
                   unlock_domains_list();

                   return &domain_pool[i];
                }
                spin_unlock(&domain_pool[i].domain_lock);

                i++;
                if (++i == MAX_SWAP_DOMAINS)
                        i = 0;
        } while (i != (id % MAX_SWAP_DOMAINS));

        return NULL;
}

/* Wrapper to simply check for a domain */
struct swap_domain * get_domain(int id) {
        struct swap_domain * domain;
        if ((domain = get_domain_locked(id, 0, 0)) != NULL)
                spin_unlock(&domain->domain_lock);
        return domain;
}
```

```
extern u64 total_swap_duration;
extern spinlock_t global_swap_time_lock;

void release_domain(
   struct swap_domain * domain,
   int got_domains_lock) {
        BUG_ON(!spin_is_locked(&domain->domain_lock));

        if (!got_domains_lock)
          lock_domains_list();
        atomic_set(&domain->abusive, 0);
        domain->is_active = 0;
        list_del(&domain->domains_list);
        nb_current_domains--;
#ifdef _SWPACC_DEBUG_DOMAINS
        printk("Destroyed a domain for pid %d: %p (%d, %llu)\n",
                domain->domain_owner, domain,
                nb_current_domains, total_swap_duration);
#endif
        unlock_domains_list();
        if (!got_domains_lock)
          spin_unlock(&domain->domain_lock);
}

#ifdef _SWPACC_DEBUG_DOMAINS
void debug_domains(void) {
        struct list_head * lh_domain;
        struct swap_domain * domain;

        lock_domains_list();
        list_for_each(lh_domain, &domains_list) {
                domain = list_entry(lh_domain,
                                struct swap_domain, domains_list);
                printk("Domain %p\n", domain);
                spin_lock(&domain->domain_lock);
                printk("Locked domain lock\n");
                printk("Active: %d, abusive: %d, owner: %d\n",
                        domain->is_active,
                        atomic_read(&domain->abusive),domain->domain_owner);
                printk("Last page: %p\n", domain->last_page);
                spin_unlock(&domain->domain_lock);
        }
        unlock_domains_list();
}
#endif
```

```
void update_domains(void) {
        struct list_head * lh_domain;
        struct swap_domain * domain;
        unsigned int nb_doms;

        lock_domains_list();
#ifdef _SWPACC_PERIODIC_ACCOUNTING
        nb_doms = current_active_domains;
#else
        nb_doms = nb_current_domains;
#endif
        list_for_each(lh_domain, &domains_list) {
                domain = list_entry(
                        lh_domain, struct swap_domain, domains_list);

                spin_lock(&domain->domain_lock);
                /*
                 * Quota exceeded if
                 * total_time/time_for_domain < nb_domains
                 */
                    spin_lock(&global_swap_time_lock);
                atomic_set(&domain->abusive, nb_doms > 1
                    && (nb_doms*domain->swap_time > total_swap_duration));
                    spin_unlock(&global_swap_time_lock);
                reevaluate_starving(domain,1,1);
                wake_up(&domain->abusive_tasks);


                spin_unlock(&domain->domain_lock);
        }
        unlock_domains_list();

}

#ifdef _SWPACC_PERIODIC_ACCOUNTING
void rotate_histories(unsigned int window) {
        struct swap_domain * domain;
        struct list_head * lh_domain;
        unsigned int i;
        unsigned int active_doms = 0;
        u64 tmp_time;

        if (window > _SWPACC_HISTORY_SIZE)
                window = _SWPACC_HISTORY_SIZE;
```

```
        else if (!window)
                return;

        lock_domains_list();
        list_for_each(lh_domain, &domains_list) {
                domain = list_entry(
                        lh_domain, struct swap_domain, domains_list);

                spin_lock(&domain->domain_lock);
                for (i=_SWPACC_HISTORY_SIZE - window
                                ;i< _SWPACC_HISTORY_SIZE;i++){
                        tmp_time = domain->history[i];
                            domain->swap_time -= tmp_time;
                            spin_lock(&global_swap_time_lock);
                            total_swap_duration -= tmp_time;
                            spin_unlock(&global_swap_time_lock);
                }

                if (domain->swap_time
                        || atomic_read(&domain->pending_requests))
                        active_doms ++;

                for (i=_SWPACC_HISTORY_SIZE - 1; i >= window; i--)
                        domain->history[i] = domain->history[i-window];
                memset(domain->history, 0, window*sizeof(domain->history[0]));
                spin_unlock(&domain->domain_lock);
        }
        current_active_domains = active_doms;
        unlock_domains_list();
}
#endif
```

A.1.5 mm/swap_accounting.c

```
#include <linux/sched.h>
#include <linux/blk_types.h>
#include <linux/blkdev.h>
#include <linux/spinlock.h>
#include <linux/workqueue.h>
#include <linux/proc_fs.h>


#include <linux/cred.h>

#include <asm/atomic.h>
```

```
#include <linux/task_swap_accounting.h>
#include <linux/swap_domains.h>

/* Kernel worker */
void do_swap_acct_update(struct work_struct *);
DECLARE_DELAYED_WORK(update_work, do_swap_acct_update);
void schedule_swpacc_update(struct work_struct *);
DECLARE_WORK(schedupdate_work, schedule_swpacc_update);

#ifdef _SWPACC_DEBUG_TIMEOUT
void do_swap_acct_timeout(struct work_struct *);
DECLARE_DELAYED_WORK(timeout_work, do_swap_acct_timeout);
#endif

struct workqueue_struct *swapacc_worker;

/* TIMES */
#define UNIFORM_TIME cpu_clock(0)
u64 total_swap_duration;
spinlock_t global_swap_time_lock;

/* Requests processed */
struct list_head processed_list_1, processed_list_2;
atomic_t current_processed_list_ptr; /* struct list_head* */

#ifdef _SWPACC_PERIODIC_ACCOUNTING
u64 periods_baseline;
#endif

/* Data structures initialization */
static int __init swap_acc_init(void) {
  int ret;
  total_swap_duration = 0;
  spin_lock_init(&global_swap_time_lock);

  INIT_LIST_HEAD(&processed_list_1);
  atomic_set(&current_processed_list_ptr, (int)&processed_list_1);

  swapacc_worker = create_singlethread_workqueue("kswapacc");

  ret = init_domains();

#ifdef _SWPACC_PERIODIC_ACCOUNTING
  periods_baseline = UNIFORM_TIME;
#endif
  return ret;
```

```
}
subsys_initcall(swap_acc_init);


static void schedule_update(void) {
  queue_work(swapacc_worker,&schedupdate_work);
}


#define SWAP_BIO_PAGE(bio) (bio->bi_io_vec[0].bv_page)

void swap_acct_delay_if_abusive(void) {
#ifndef _SWPACC_ACCOUNT_ONLY
  struct swap_domain * domain;

  if ((domain = get_domain(current->pid))) {
    reevaluate_starving(domain,0,0);
#ifdef _SWPACC_ANTI_STARVATION
    wait_event(domain->abusive_tasks,
      atomic_read(&domain->abusive) == 0
        || atomic_read(&domain->starvation_bypass));
#else
    wait_event(domain->abusive_tasks,
      atomic_read(&domain->abusive) == 0);
#endif
  }
#endif
}
void swap_acct_bio_check(struct bio *bio) {
  struct swap_domain * domain;

#ifdef _SWPACC_DEBUG_INFO
  printk("[INFO] Request for page %p for pid %d\n",
        SWAP_BIO_PAGE(bio), current->pid);
#endif

  if ((domain = get_domain_locked(current->pid, 1, 0))) {
    bio_get(bio);
    bio->swap_domain = domain;
    swap_domain_start_request(domain);

    spin_unlock(&domain->domain_lock);
  }
}
```

```
void swap_acct_destroy(struct task_struct * task) {
  struct swap_domain * domain;

  if ((domain = get_domain_locked(task->pid, 0, 1))) {
    printk("Destroying domain %p\n", domain);
    /* may add a check for sleeping tasks */
    spin_lock(&global_swap_time_lock);
    total_swap_duration -= domain->swap_time;
    spin_unlock(&global_swap_time_lock);

    release_domain(domain,0);
    schedule_update();
  }

}

void swap_acct_start(struct request * rq) {
 struct bio * bio;
 int i=0;

  __rq_for_each_bio(bio, rq)
  i++;
  __rq_for_each_bio(bio, rq)
    if (bio->swap_domain) {
      bio->request_start = UNIFORM_TIME;
      bio->req_cnt = i;
    }
}

#ifdef _SWPACC_DEBUG_TIMEOUT
void schedule_swpacc_timeout(void) {
 cancel_delayed_work(&timeout_work);
 queue_delayed_work(swapacc_worker, &timeout_work, 200*HZ);
}
#endif

#ifdef _SWPACC_PERIODIC_ACCOUNTING
#define _SWPACC_SAMPLE_PERIOD_NS \
  (_SWPACC_PERIOD_MS*1000/_SWPACC_HISTORY_SIZE)
#define _SWPACC_SAMPLE_PERIOD_JF \
  ((HZ*_SWPACC_PERIOD_MS)/(1000*_SWPACC_HISTORY_SIZE))
void schedule_periodic_update(void) {
 cancel_delayed_work(&update_work);
 queue_delayed_work(swapacc_worker,
      &update_work, _SWPACC_SAMPLE_PERIOD_JF);
```

```
}
#endif

void schedule_swpacc_update(struct work_struct * work) {
 cancel_delayed_work(&update_work);
 queue_delayed_work(swapacc_worker, &update_work, 0);
}

#ifdef _SWPACC_DEBUG_TIMEOUT
extern void debug_domains(void);
void do_swap_acct_timeout(struct work_struct * work) {
  debug_domains();
}
#endif
void do_swap_acct_update(struct work_struct * work) {
  struct list_head * current_list_ptr, *next_list_ptr;
  struct list_head *lh_req, *tmp;
  struct swap_domain * domain;
  struct bio * bio;
  u64 tmp_time;
#ifdef _SWPACC_PERIODIC_ACCOUNTING
  u64 tmp_baseline, tmp_timediff;
  unsigned int window;
#endif

#ifdef _SWPACC_DEBUG_INFO
    printk("[INFO] START KTHREAD SWAP ACC");
#endif
  current_list_ptr = (struct list_head *)
        atomic_read(&current_processed_list_ptr);
  if (current_list_ptr == (next_list_ptr = &processed_list_1))
    next_list_ptr = &processed_list_2;

  INIT_LIST_HEAD(next_list_ptr);
  atomic_set(&current_processed_list_ptr, (int)next_list_ptr);


#ifdef _SWPACC_PERIODIC_ACCOUNTING
  tmp_baseline = UNIFORM_TIME;
  tmp_timediff = tmp_baseline - periods_baseline;

  for (window=0;window <= _SWPACC_HISTORY_SIZE;window++) {
    if (tmp_timediff < _SWPACC_SAMPLE_PERIOD_NS)
      break;
    tmp_timediff -= _SWPACC_SAMPLE_PERIOD_NS;
  }
```

```
  if (window) {
    rotate_histories(window);
    periods_baseline = (window <= _SWPACC_HISTORY_SIZE
          ? window*_SWPACC_SAMPLE_PERIOD_NS
          : tmp_baseline);
  }
#endif

  /* Iterate over processed requests, add to sum */
  list_for_each_safe(lh_req, tmp, current_list_ptr) {
    bio = list_entry(lh_req, struct bio, swap_acct_list);
    domain = bio->swap_domain;
    spin_lock(&domain->domain_lock);
    if (likely(domain->is_active)) {
      tmp_time = bio->request_duration;
#ifdef _SWPACC_PERIODIC_ACCOUNTING
      tmp_timediff = (
        periods_baseline > bio->request_start + tmp_time
        ? periods_baseline - bio->request_start - tmp_time
        : 0);
      for (window=0;window < _SWPACC_HISTORY_SIZE;window++) {
        if (tmp_timediff < _SWPACC_SAMPLE_PERIOD_NS)
          break;
        tmp_timediff -= _SWPACC_SAMPLE_PERIOD_NS;
      }
      if (window < _SWPACC_HISTORY_SIZE) {

        domain->history[window] += tmp_time;
#endif
        domain->swap_time += tmp_time;
        spin_lock(&global_swap_time_lock);
        total_swap_duration += tmp_time;
        spin_unlock(&global_swap_time_lock);
#ifdef _SWPACC_PERIODIC_ACCOUNTING
      }
#endif
    }
    spin_unlock(&domain->domain_lock);
    list_del(lh_req);
    bio_put(bio);
  }

  update_domains();
#ifdef _SWPACC_PERIODIC_ACCOUNTING
  /*
   * Could reschedule in periods_baseline +
```

```
   * _SWPACC_SAMPLE_PERIOD_NS - tmp_baseline for added precision
   */
  schedule_periodic_update();
#endif
#ifdef _SWPACC_DEBUG_INFO
    printk("[INFO] END KTHREAD SWAP ACC");
#endif
#ifdef _SWPACC_DEBUG_TIMEOUT
  schedule_swpacc_timeout();
#endif
}

void swap_acct_end(struct bio * bio) {
#ifdef _SWPACC_DEBUG_INFO
  printk("[INFO] Swapped in bio %p (page %p)\n",
      bio, SWAP_BIO_PAGE(bio));
#endif

  if (bio->swap_domain) {
    struct list_head * process_list;

    swap_domain_end_request(bio->swap_domain);
    bio->request_duration = UNIFORM_TIME - bio->request_start;
    process_list = (struct list_head *)
      atomic_read(&current_processed_list_ptr);
    list_add_tail(&bio->swap_acct_list, process_list);
    schedule_update();
  }

}
```

A.2 Fichiers du noyau Linux v3.5 modifiés

```
--- orig/linux-3.2.5/include/linux/blk_types.h
+++ linux-3.2.5/include/linux/blk_types.h
@@ -8,6 +8,7 @@
 #ifdef CONFIG_BLOCK

 #include <linux/types.h>
+#include <linux/spinlock.h>

 struct bio_set;
 struct bio;
@@ -71,6 +72,15 @@ struct bio {
 #endif
```

```
   bio_destructor_t  *bi_destructor; /* destructor */
+
+  /* Swap accounting data */
+  struct list_head  swap_acct_list;
+  struct swap_domain  *swap_domain;
+  u64       request_start;
+  u64       request_duration;
+  int       req_cnt;

   /*
--- orig/linux-3.2.5/mm/memory.c
+++ linux-3.2.5/mm/memory.c
@@ -58,6 +58,8 @@
 #include <linux/elf.h>
 #include <linux/gfp.h>

+#include <linux/task_swap_accounting_ops.h>
+
 #include <asm/io.h>
 #include <asm/pgalloc.h>
 #include <asm/uaccess.h>
@@ -2880,7 +2882,8 @@ static int do_swap_page(struct mm_struct
   delayacct_set_flag(DELAYACCT_PF_SWAPIN);
   page = lookup_swap_cache(entry);
   if (!page) {
-    grab_swap_token(mm); /* Contend for token _before_ read-in */
+    swap_acct_delay_if_abusive();
+    //grab_swap_token(mm);
     page = swapin_readahead(entry,
           GFP_HIGHUSER_MOVABLE, vma, address);
     if (!page) {
--- orig/linux-3.2.5/mm/page_io.c
+++ linux-3.2.5/mm/page_io.c
@@ -20,6 +20,8 @@
 #include <linux/writeback.h>
 #include <asm/pgtable.h>

+#include <linux/task_swap_accounting_ops.h>
+
 static struct bio *get_swap_bio(gfp_t gfp_flags,
         struct page *page, bio_end_io_t end_io)
 {
@@ -36,6 +38,11 @@ static struct bio *get_swap_bio(gfp_t gf
     bio->bi_idx = 0;
     bio->bi_size = PAGE_SIZE;
```

```
     bio->bi_end_io = end_io;
+
+    /* Swap accounting init */
+    bio->swap_domain = NULL;
    }
    return bio;
 }
@@ -81,6 +88,7 @@ void end_swap_bio_read(struct bio *bio,
    } else {
      SetPageUptodate(page);
    }
+   swap_acct_end(bio);
    unlock_page(page);
    bio_put(bio);
 }
@@ -129,7 +137,8 @@ int swap_readpage(struct page *page)
      goto out;
    }
    count_vm_event(PSWPIN);
-   submit_bio(READ, bio);
+   swap_acct_bio_check(bio);
+        submit_bio(READ, bio);
 out:
    return ret;
 }
--- orig/linux-3.2.5/kernel/exit.c
+++ linux-3.2.5/kernel/exit.c
@@ -52,6 +52,8 @@
 #include <linux/hw_breakpoint.h>
 #include <linux/oom.h>

+#include <linux/task_swap_accounting_ops.h>
+
 #include <asm/uaccess.h>
 #include <asm/unistd.h>
 #include <asm/pgtable.h>
@@ -198,6 +200,7 @@ repeat:
        leader->exit_state = EXIT_DEAD;
    }

+   swap_acct_destroy(p);
    write_unlock_irq(&tasklist_lock);
    release_thread(p);
    call_rcu(&p->rcu, delayed_put_task_struct);
--- orig/linux-3.2.5/fs/bio.c
+++ linux-3.2.5/fs/bio.c
```

```
@@ -256,6 +256,7 @@ void bio_init(struct bio *bio)
   memset(bio, 0, sizeof(*bio));
   bio->bi_flags = 1 << BIO_UPTODATE;
   atomic_set(&bio->bi_cnt, 1);
+  bio->swap_domain = NULL;
 }
 EXPORT_SYMBOL(bio_init);

@@ -458,6 +459,7 @@ void __bio_clone(struct bio *bio, struct
   bio->bi_vcnt = bio_src->bi_vcnt;
   bio->bi_size = bio_src->bi_size;
   bio->bi_idx = bio_src->bi_idx;
+  bio->swap_domain = NULL;
 }
 EXPORT_SYMBOL(__bio_clone);

--- orig/linux-3.2.5/block/blk-core.c
+++ linux-3.2.5/block/blk-core.c
@@ -30,6 +30,8 @@
 #include <linux/list_sort.h>
 #include <linux/delay.h>

+#include <linux/task_swap_accounting_ops.h>
+
 #define CREATE_TRACE_POINTS
 #include <trace/events/block.h>

@@ -2016,6 +2018,7 @@ void blk_start_request(struct request *r
     req->next_rq->resid_len = blk_rq_bytes(req->next_rq);

   blk_add_timer(req);
+  swap_acct_start(req);
 }
 EXPORT_SYMBOL(blk_start_request);

--- orig/linux-3.2.5/Makefile
+++ linux-3.2.5/Makefile
@@ -1,7 +1,7 @@
 VERSION = 3
 PATCHLEVEL = 2
 SUBLEVEL = 5
-EXTRAVERSION =
+EXTRAVERSION = -swp-acct
 NAME = Saber-toothed Squirrel

 # *DOCUMENTATION*
```

```
--- orig/linux-3.2.5/mm/Makefile
+++ linux-3.2.5/mm/Makefile
@@ -25,7 +25,7 @@ endif
 obj-$(CONFIG_HAVE_MEMBLOCK) += memblock.o

 obj-$(CONFIG_BOUNCE)  += bounce.o
-obj-$(CONFIG_SWAP)  += page_io.o swap_state.o swapfile.o thrash.o
+obj-$(CONFIG_SWAP)  += page_io.o swap_state.o swapfile.o thrash.o
+obj-$(CONFIG_SWAP)  += swap_accounting.o swap_domain.o
 obj-$(CONFIG_HAS_DMA)  += dmapool.o
 obj-$(CONFIG_HUGETLBFS)  += hugetlb.o
 obj-$(CONFIG_NUMA)   += mempolicy.o
```

Bibliographie

[Aas05] Josh Aas. Understanding the Linux 2.6.8.1 CPU scheduler. Technical report, Silicon Graphics Inc., February 2005.

[ABG+01] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva : An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4) :483–518, November 2001.

[APB09] M. Allman, V. Paxson, and E. Blanton. RFC 5681 : TCP congestion control, September 2009.

[Axb04] Jens Axboe. Linux block IO—present and future. In *Proceedings of the fifth Ottawa Linux Symposium*, pages 51–61, 2004.

[Bai01] Bob Bailey. Acceptable computer response times [en ligne]. `http://www.humanfactors.com/downloads/apr01.asp` (consulté le 20/10/2013), April 2001.

[Bar02] Michael Barr. Introduction to counter/timers [en ligne]. `http://www.embedded.com/electronics-blogs/beginner-s-corner/4024440/Introduction-to-Counter-Timers` (consulté le 20/10/2013), 2002.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3 edition, 2005.

[BC10] David Boutcher and Abhishek Chandra. Does virtualization make disk scheduling passé ? *ACM SIGOPS Operating Systems Review*, 44(1) :20–24, March 2010.

[BDF+03] 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. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers : a new facility for resource management in server systems. In *Proceedings of the third USENIX symposium on Operating systems design and implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2) :78–101, June 1966.

[Bit09] BitBucket. On our extended downtime, Amazon and what's coming [en ligne]. `http://blog.bitbucket.org/2009/10/04/on-our-extended-downtime-amazon-and-whats-coming/` (consulté le 20/10/2013), October 2009.

[BM04] Sorav Bansal and Dharmendra S. Modha. CAR : Clock with adaptive replacement. In *Proceedings of the third USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[BM11] Artur Baruchi and Edson Toshimi Midorikawa. A survey analysis of memory elasticity techniques. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par '10, pages 681–688, Berlin, Heidelberg, 2011. Springer-Verlag.

[CEH+13] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation : refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of the fiftieth Annual Design Automation Conference*, DAC '13, pages 76 :1–76 :10, New York, NY, USA, 2013. ACM.

[CER00] CERT. Advisory CA-1996-21 TCP SYN flooding and IP spoofing attacks [en ligne]. `http://www.cert.org/advisories/CA-1996-21.html` (consulté le 20/10/2013), September 2000.

[CGV07] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Performance Evaluation Review*, 35(2) :42–51, September 2007.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK — a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, 1981. ACM.

[CPM10] Jacinto C.A. Cansado, João H.S. Pereira, and Edson T. Midorikawa. Considering the frequency dimension into on demand adaptive algorithms. *ACM SIGOPS Operating Systems Review*, 44(1) :110–115, March 2010.

[DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp) : a network subsystem architecture for server systems. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 261–275, New York, NY, USA, 1996. ACM.

[Den67] Peter J. Denning. Effects of scheduling on file memory operations. In *Proceedings of the 1967 AFIPS spring joint computer conference*, AFIPS '67 (Spring), pages 9–21, New York, NY, USA, 1967. ACM.

[Den68a] Peter J. Denning. Thrashing : its causes and prevention. In *Proceedings of the 1968 AFIPS fall joint computer conference, part I*, AFIPS '68 (Fall), pages 915–922, New York, NY, USA, 1968. ACM.

[Den68b] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5) :323–333, May 1968.

[Den80] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1) :64–84, January 1980.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the 1989 ACM Symposium on Communications architectures & protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM.

[dL05] Benoît des Ligneris. Virtualization of Linux based computers : the Linux-VServer project. In *Proceedings of the nineteenth IEEE International Symposium on High Performance Computing Systems and Applications*, HPCS '05, pages 340–346, Washington, DC, USA, 2005. IEEE Computer Society.

[Doc09] Linux Kernel Documentation. Block IO controller [en ligne]. `https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt` (consulté le 20/10/2013), 2009.

[EKO95] Dawson R. Engler, Frans Kaashoek, and James 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, New York, NY, USA, 1995. ACM.

[GB13] Glenn Gailey and Alva Jaime Bravo. Microsoft software developer network : Ports requirements in Windows Azure [en ligne]. `http://msdn.microsoft.com/fr-fr/library/windowsazure/jj136814.aspx` (consulté le 20/10/2013), January 2013.

[GBM04] Mina Guirguis, Azer Bestavros, and Ibrahim Matta. Exploiting the transients of adaptation for RoQ attacks on internet resources. In

Proceedings of the twelfth IEEE International Conference on Network Protocols, ICNP '04, pages 184–195, Washington, DC, USA, 2004. IEEE Computer Society.

[GCGV06] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[GGC05] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon : QoS monitoring and performance profiling tool. Technical Report HPL-2005-187, Hewlett-Packard Labs, 2005.

[GH13] Yossi Gilad and Amir Herzberg. Fragmentation considered vulnerable. *ACM Transactions on Information and System Security*, 15(4) :16 :1–16 :31, April 2013.

[GHJ+12] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. Vmware distributed resource management : Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1) :45–64, 2012.

[GMV10] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock : handling throughput variability for hypervisor IO scheduling. In *Proceedings of the ninth USENIX conference on Operating systems design and implementation*, OSDI '10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[Goo13] Google. Google App Engine : Socket API overview [en ligne]. `https://developers.google.com/appengine/docs/java/sockets/` (consulté le 20/10/2013), September 2013.

[HC03] Amber Huffman and Joni Clark. Serial ATA : Native command queuing–an exciting new performance feature for serial ATA. Technical report, Intel Corp. and Seagate Technology LLC, 2003.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4) :1–17, September 2006.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 microvisor : convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific Workshop on Systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[HLZ+10] Yanyan Hu, Xiang Long, Jiong Zhang, Jun He, and Li Xia. I/O scheduling model of virtual machine based on multi-core dynamic partitioning. In *Proceedings of the nineteenth ACM International*

Symposium on High Performance Distributed Computing, HPDC '10, pages 142–154, New York, NY, USA, 2010. ACM.

[HP03] Hewlett-Packard. HP-UX 11i version 2 : serialize(1) [en ligne]. `http://nixdoc.net/man-pages/HP-UX/man1/serialize.1.html` (consulté le 20/10/2013), August 2003.

[HZPW09] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *Proceedings of the 2009 IFIP/IEEE International Symposium on Integrated Network Management*, IM '09, pages 630–637, Washington, DC, USA, 2009. IEEE Computer Society.

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling : a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 117–130, New York, NY, USA, 2001. ACM.

[Int13] Intel. Intel 64 and IA-32 architectures software developer's manual [en ligne]. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf` (consulté le 20/10/2013), September 2013.

[Jia09] Song Jiang. *Advanced Operating Systems and Kernel Applications : Techniques and Technologies*, chapter Swap Token : Rethink the Application of the LRU Principle on Paging to Remove System Thrashing. Information Science Reference - Imprint of : IGI Publishing, Hershey, PA, 2009.

[JZ02a] Song Jiang and Xiaodong Zhang. LIRS : an efficient low interreference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 31–42, New York, NY, USA, 2002. ACM.

[JZ02b] Song Jiang and Xiaodong Zhang. TPF : a dynamic system thrashing protection facility. *Software : Practice and Experience*, 32(3) :295–318, March 2002.

[JZ05] Song Jiang and Xiaodong Zhang. Token-ordered LRU : an effective page replacement policy and its implementation in linux systems. *Performance Evaluation*, 60(1-4) :5–29, May 2005.

[KELS62] Tom Kilburn, David BG Edwards, MJ Lanigan, and Frank H Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, 2 :223–235, April 1962.

[KK03] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted denial of service attacks : the shrew vs. the mice and elephants. In *Proceedings of the 2003 ACM conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 75–86, New York, NY, USA, 2003. ACM.

[KKB+07] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '07, pages 200–209, Washington, DC, USA, 2007. IEEE Computer Society.

[KKC12] Sewoog Kim, Dongwoo Kang, and Jongmoo Choi. Fine-grained I/O fairness analysis in virtualized environments. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, pages 403–408, New York, NY, USA, 2012. ACM.

[KKS03] Wataru Kaneko, Kenji Kono, and Kentaro Shimizu. Preemptive resource management : Defending against resource monopolizing DoS. *Applied Informatics*, pages 662–669, 2003.

[Kol06] Kirill Kolyshkin. Virtualization in Linux. Technical report, OpenVZ, 2006.

[Laz79] Edward D. Lazowska. The benchmarking, tuning and analytic modeling of VAX/VMS. In *Proceedings of the 1979 ACM SIGMETRICS conference on Simulation, measurement and modeling of computer systems*, SIGMETRICS '79, pages 57–64, New York, NY, USA, 1979. ACM.

[LDVN09] Matthieu Lemerre, Vincent David, and Guy Vidal-Naquet. A communication mechanism for resource isolation. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 1–6, New York, NY, USA, 2009. ACM.

[LHAP06] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference*, ATEC '06, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[LIJ97] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing Denial-of-Service attacks on a μ-kernel for WebOSes. In *Proceedings of the sixth Workshop on Hot Topics in Operating Systems*

(HotOS-VI), HOTOS '97, pages 73–, Washington, DC, USA, 1997. IEEE Computer Society.

[LMA03] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade : Virtual storage devices with performance guarantees. In *Proceedings of the second USENIX Conference on File and Storage Technologies*, FAST '03, pages 131–144, Berkeley, CA, USA, 2003. USENIX Association.

[LMB+96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7) :1280–1297, 1996.

[LNP05] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over infiniband : An approach using a high performance network block device. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, pages 1–10, Washington, DC, USA, 2005.

[Loe92] Keith Loepere. *Mach 3 kernel principles*. Open Software Foundation Mach 3. Open Software Foundation and Carnegie Mellon University, July 1992.

[LW09] Adam Lackorzynski and Alexander Warg. Taming subsystems : capabilities as universal resource access control in L4. In *Proceedings of the second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM.

[Man13] Linux Programmer's Manual. random, urandom - kernel random number source devices [en ligne]. `http://man7.org/linux/man-pages/man4/random.4.html` (consulté le 20/10/2013), March 2013.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[McK90] Paul E McKenney. Stochastic fairness queueing. In *Proceedings of the ninth Annual Joint Conference of the IEEE Computer and Communication Societies.'The Multiple Facets of Integration'*, INFOCOM '90, pages 733–740, Washington, DC, USA, 1990. IEEE Computer Society.

[Men08] Paul Menage. Cgroups [en ligne]. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt` (consulté le 20/10/2013), 2008.

[MHH+07] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[Mic05] Microsoft. Microsoft Virtual Server 2005 R2 [en ligne]. `http://www.microsoft.com/windowsserversystem/virtualserver/` (consulté le 20/10/2013), 2005.

[Mic12] Microsoft. Microsoft Hyper-V Server 2012 [en ligne]. `http://www.microsoft.com/en-us/server-cloud/hyper-v-server/` (consulté le 20/10/2013), 2012.

[MJR97] Anastasio Molano, Kanaka Juvva, and Ragunathan Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. In *Proceedings of the eighteenth IEEE Real-Time Systems Symposium*, RTSS '97, pages 155–, Washington, DC, USA, 1997. IEEE Computer Society.

[MM04] Nimrod Megiddo and Dharmendra S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4) :58–65, April 2004.

[MM07] Thomas Moscibroda and Onur Mutlu. Memory performance attacks : denial of memory service in multi-core systems. In *Proceedings of the sixteenth USENIX Security Symposium*, SS '07, pages 18 :1–18 :18, Berkeley, CA, USA, 2007. USENIX Association.

[MMFR96] M Mathis, J. Mahdavi, S Floyd, and A. Romanov. RFC 2018 : TCP selective acknowledgment options, October 1996.

[Mor72] James B. Morris. Demand paging through utilization of working sets on the MANIAC II. *Communications of the ACM*, 15(10) :867–872, October 1972.

[MPC08] Edson T. Midorikawa, Ricardo L. Piantola, and Hugo H. Cassettari. On adaptive replacement based on LRU with working area restriction algorithm. *ACM SIGOPS Operating Systems Review*, 42(6) :81–92, October 2008.

[Nag84] John Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Computer Communication Review*, 14(4) :11–17, October 1984.

[Nag87] John Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4) :435–438, 1987.

[NKG10] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds : managing performance interference effects for QoS-aware clouds. In *Proceedings of the fifth European conference on Computer systems*, EuroSys '10, pages 237–250, New York, NY, USA, 2010. ACM.

[OCR08] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[Pab09] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184), August 2009.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proceedings of the third BSDCan annual conference*, BSDCan '05, May 2005.

[PHS+09] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the fourth ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.

[PLM+10] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. Understanding performance interference of I/O workload in virtualized cloud environments. In *Proceedings of the third IEEE International Conference on Cloud Computing*, CLOUD '10, pages 51–58, Washington, DC, USA, 2010. IEEE Computer Society.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. RFC 3168 : The addition of explicit congestion notification (ECN) to IP, September 2001.

[RJMO01] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Readings in multimedia computing and networking. chapter Resource kernels : a resource-centric approach to real-time and multimedia systems, pages 476–490. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[RNSE09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 77–84, New York, NY, USA, 2009. ACM.

[Rob03] Jeff Roberson. ULE : a modern scheduler for FreeBSD. In *Proceedings of the BSD Conference 2003 on BSD Conference*, BSDC '03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[RRD73] Juan Rodriguez-Rosell and Jean-Pierre Dupuy. The design, implementation, and evaluation of a working set dispatcher. *Communications of the ACM*, 16(4) :247–253, April 1973.

[RSI12] Mark Russinovich, David A. Salomon, and Alex Ionescu. *Windows Internals*, volume I. Microsoft Press, 6 edition, 2012.

[RW06] Moses Reuven and Yair Wiseman. Medium-term scheduler as a solution for the thrashing effect. *Computer Journal*, 49(3) :297–309, May 2006.

[SBW08] Mark J. Stanovich, Theodore P. Baker, and An-I Andy Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 331–341, Washington, DC, USA, 2008. IEEE Computer Society.

[SCG+00] Vijay Sundaram, Abhishek Chandra, Pawan Goyal, Prashant Shenoy, Jasleen Sahni, and Harrick Vin. Application performance in the QLinux multimedia operating system. In *Proceedings of the eighth ACM international conference on Multimedia*, MULTIMEDIA '00, pages 127–136, New York, NY, USA, 2000. ACM.

[SD.09] VMware Technical Marketing SD. Understanding memory resource management in VMware ESX server. Technical report, VMware Inc., 2009.

[SH02] Jonathan S Shapiro and Norman Hardy. EROS : A principle-driven operating system from the ground up. *Software*, 19(1) :26–33, 2002.

[SKGK10] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall : performance isolation for cloud datacenter networks. In *Proceedings of the second USENIX conference on Hot topics in cloud computing*, HotCloud '10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[SS03] André Seznec and Nicolas Sendrier. HAVEGE : A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation*, 13(4) :334–346, October 2003.

[Sto07] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM SIGOPS Operating Systems Review*, 41(4) :59–68, July 2007.

[SU06] Jan Stoess and Volkmar Uhlig. Flexible, low-overhead event logging to support resource scheduling. In *Proceedings of the twelfth International Conference on Parallel and Distributed Systems - Volume 2*, ICPADS '06, pages 115–120, Washington, DC, USA, 2006. IEEE Computer Society.

[SV95] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of the 1995 ACM conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '95, pages 231–242, New York, NY, USA, 1995. ACM.

[SVL02] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2002. USENIX Association.

[Sys08] Cisco Systems. Implementing quality of service policies with DSCP [en ligne]. `http://www.cisco.com/en/US/tech/tk543/tk757/technologies_tech_note09186a00800949f2.shtml` (consulté le 20/10/2013), February 2008.

[TEF07] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *Proceedings of the sixteenth USENIX Security Symposium on USENIX Security Symposium*, SS '07, pages 17 :1–17 :18, Berkeley, CA, USA, 2007. USENIX Association.

[Tin09] Julien Tinnes. Bypassing Linux' NULL pointer dereference exploit prevention (mmap_min_addr) [en ligne]. `http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html` (consulté le 20/10/2013), June 2009.

[TSS12] Focus Group on Cloud Computing Telecommunication Standardization Sector. Introduction to the cloud ecosystem : definitions, taxonomies, use cases and high-level requirements. Technical Report Part 1, International Telecommunication Union, February 2012.

[USR02] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI) :239–254, December 2002.

[VDL10] Dirk Vogt, Björn Döbel, and Adam Lackorzynski. Stay strong, stay safe : Enhancing reliability of a secure operating system. In *Proceedings of the Workshop on Isolation and Integration for Dependable Systems*, IIDS '10, April 2010.

[VGLS12] Thibaut Vuillemin, François Goichon, Cédric Lauradoux, and Guillaume Salagnac. Entropy transfers in the Linux random number generator. Technical Report RR-8060, INRIA, September 2012.

[VGR98] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation : sharing and isolation in shared-memory multiprocessors. *ACM SIGPLAN Notices*, 33(11) :181–192, 1998.

[VKF+12] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-freeing attacks : improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[VMw13] VMware. vSphere ESX and ESXi info center [en ligne]. `http://www.vmware.com/products/esxi-and-esx/overview` (consulté le 20/10/2013), 2013.

[vR12] Rik van Riel. [PATCH -mm] remove swap token code [en ligne]. `https://lkml.org/lkml/2012/4/9/173` (consulté le 20/10/2013), April 2012.

[Wal02] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI) :181–194, December 2002.

[WB04] Joel C. Wu and Scott A. Brandt. Storage access support for soft real-time applications. In *Proceedings of the tenth IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '04, pages 164–, Washington, DC, USA, 2004. IEEE Computer Society.

[WGB11] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based resource allocation for Software as a Service provider (SaaS) in cloud computing environments. In *Proceedings of the eleventh IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 195–204, Washington, DC, USA, 2011. IEEE Computer Society.

[Wik13] Wikipedia. Fork bomb [en ligne]. `http://en.wikipedia.org/wiki/Fork_bomb` (consulté le 20/10/2013), 2013.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI) :195–209, 2002.

[WW94] Carl A Waldspurger and William E Weihl. Lottery scheduling : Flexible proportional-share resource management. In *Proceedings of the first USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[YSEY10] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. NCQ vs. I/O scheduler : Preventing unexpected misbehaviors. *ACM Transactions on Storage*, 6(1) :2 :1–2 :37, April 2010.

[ZGDS11] Fangfei Zhou, Manish Goel, Peter Desnoyers, and Ravi Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. In *Proceedings of the tenth IEEE International Symposium on Network Computing and Applications (NCA)*, pages 123–130, 2011.

[Zha90] Lixia Zhang. Virtual clock : a new traffic control algorithm for packet switching networks. In *Proceedings of the 1990 ACM symposium on Communications architectures & protocols*, SIGCOMM '90, pages 19–29, New York, NY, USA, 1990. ACM.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the eleventh international conference on Architectural support for programming languages and operating systems*, ASPLOS '04, pages 177–188, New York, NY, USA, 2004. ACM.

[ZW09] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 21–30, New York, NY, USA, 2009. ACM.