Conception et Architecture de Systèmes d'Exploitation Persistants

Ludovic Courtès
<ludovic.courtes@utbm.fr>

Université de Technologie de Belfort-Montbéliard

13 janvier 2003

Résumé

Un système d'exploitation persistant est un système mettant en œuvre des mécanismes permettant de rendre toutes les données et applications persistantes, même si la machine hôte a été arrêtée ou redémarrée. Cette propriété est intéressante pour les applications demandant de la haute-disponibilité et de la tolérance aux fautes, mais également pour n'importe quel utilisateur.

Cet document est avant tout un travail de synthèse présentant différents travaux conduits jusqu'à présent dans le domaine de la conception de systèmes d'exploitation persistants. Il présente les leçons tirées d'expériences passées et montre l'intérêt des architectures à base de micro-noyaux comme base pour la mise-en-œuvre des mécanismes de persistance de manière orthogonale. A ce titre, le système d'exploitation multi-serveurs GNU est étudié pour voir comment cette fonctionnalité pourrait lui être ajoutée.

Mots-clef : Système d'exploitation, persistance, tolérance aux fautes, micro-noyau, GNU/Hurd.

Copyright © 2003 Ludovic Courtès
Permission vous est donnée de distribuer des copies exactes de ce document tant que cette note de permission et le copyright apparaissent clairement.
Permission vous est donnée de distribuer des versions modifiées de ce document tant que la totalité du travail dérivé est distribuée sous les mêmes termes que celui-ci.

Table des Matières

1. Introduction .. 2
2. Mise-en-œuvre .. 2
   2.1. Concepts clef .. 2
   2.2. Approche générale .. 3
   2.3. Travaux étudiés .. 4
3. Leçons tirées .. 4
   3.1. Captures synchrones ou indépendantes 5
   3.2. Avantages d'un μ-noyau .. 5
4. Persistance au-dessus d'un micro-noyau .. 7
   4.1. Propriétés d'un micro-noyau candidat 7
   4.2. Observabilité des interactions inter-processus .. 9
   4.3. Gestion de la mémoire virtuelle .. 10
5. Etude de cas : rendre GNU persistant .. 11
   5.1. Principes du GNU Hurd .. 11
   5.2. Avantages du Hurd, limites de Mach 13
   5.3. Communication inter-processus persistante .. 13
   5.4. Encapsulation de processus .. 14
   5.5. Gestion de la mémoire virtuelle .. 15
   5.6. Mise-en-œuvre .. 15
6. Conclusion .. 15

Références .. 15

1. Introduction

L'idée d'avoir un système informatique persistant, c'est-à-dire capable de poursuivre de manière transparente son exécution après une faute matérielle ou logicielle, a longtemps été un sujet de recherche. A vrai dire, la notion de persistance de l'état du système est plutôt naturelle. A titre d'exemple, il est plus simple d'expliquer à un non-informaticien que « *pour d'obscures raisons techniques, une panne d'électricité détruira le travail que vous avez fait pendant les dernières minutes* » [17], plutôt que de lui exposer le caractère exceptionnel de cette prouesse technique. Pour des applications courantes telles que le traitement de texte, un système d'exploitation persistant peut s'avérer salvateur en cas de panne puisqu'il garantit à son utilisateur que seulement une petite partie de son travail risque d'être perdue. Pour des applications telles que des calculs scientifiques pouvant durer plusieurs jours, cela peut être d'autant plus salvateur. Du point de vue du programmeur, ce type de système supprime la frontière entre support de stockage à long terme (e.g. disque dur) et mémoire principale puisque les données manipulées par une application seront présentes pendant toute sa durée de vie.

Cependant, malgré la recherche effectuée dans ce domaine depuis des décennies, force est de constater que peu de systèmes d'exploitation sont persistants. Une des conséquences est que beaucoup d'applications (éditeurs, navigateurs internet, etc.) ont leur propre système de sauvegarde temporaire sur disque des données en cours de traitement. Des bibliothèques et outils génériques ou spécialisés pour un type d'application permettent de façon plus ou moins transparente de rendre des applications persistantes. Par rapport à ces solutions au niveau applicatif, la prise-en-charge de la persistance par le système d'exploitation a pour avantage d'offrir une solution générique et potentiellement plus performante.

En se basant sur différents travaux relatifs à la persistance et à sa mise-en-œuvre au niveau du système d'exploitation et après avoir défini un certain nombre de concepts et de techniques relatifs à ces travaux (section 2), ce document propose une synthèse des leçons tirées de chacune de ces expériences (section 3). En particulier, nous verrons en quoi les micro-noyaux de seconde génération tels que L4 se prêtent bien à l'implémentation d'un mécanisme de persistance dans l'espace utilisateur. Enfin, la section 5 propose une première approche à comment la persistance pourrait être mise-en-œuvre dans le cadre du système d'exploitation GNU Hurd [10].

2. Mise-en-œuvre

Beaucoup de points communs existent entre les approches et techniques adoptées par les différents systèmes d'exploitation persistants. Nous donnons ici un aperçu des principales idées et techniques mises-en-œuvre par des systèmes conçus pour fonctionner sans prise-en-charge de la persistance au niveau matériel, c'est-à-dire sur du matériel générique.

2.1. Concepts clef

Persistance transparente et orthogonale. Un système d'exploitation persistant est un système garantissant la stabilité à la fois de ses données et de ses processus. En d'autres termes, le système doit permettre la tolérance aux fautes en étant capable de restaurer, après une faute matérielle ou logicielle, un état aussi proche que possible de la faute : processus et données doivent être persistants, doivent pouvoir *survivre* aux fautes. Il s'agit dans ce cas de *persistance transparente* : les mécanismes de la persistance sont transparents aux applications (i.e. aucune action particulière, aucune modification n'est nécessaire pour qu'une application devienne persistante).

En outre, tolérance aux fautes et la persistance sont également des problèmes *orthogonaux* à la mise-en-œuvre des fonctionnalités de base d'un système et ne devraient pas influencer directement sa conception. Au contraire, il s'agit d'une fonctionnalité qui doit pouvoir être ajoutée au système d'exploitation sans que cela ne requiert de modifications du noyau. Certains systèmes mentionnés par la suite ont été développés dans le but d'être persistants et intègrent donc les mécanismes de persistance. Toutefois, on peut argumenter que les mécanismes de persistance, qui sont des mécanismes non-fonctionnels, n'ont pas à être intégrés au noyau du système mais peuvent au contraire lui être ajoutés [3]. Dans cette optique, un système devrait être assez générique et fournir suffisamment de mécanismes permettant de lui ajouter des fonctionnalités telles que la persistance, sans que cela ne nécessite de modifier son noyau. Cela permet de respecter le principe de « *séparation des soucis*[1] » [3] : le noyau ne fait qu'accomplir sa tâche principale en fournissant les moyens permettant par exemple de lui ajouter des mécanismes de tolérance. La figure 1 illustre ces principes de transparence et d'orthogonalité de la persistance.

Pour que les fonctionnalités de persistance puissent ainsi « s'interposer » entre l'application et la base du système, il doit être possible d'*observer* les données et le comportement du système (*introspection*) et même de les *modifier* (*ingérence*[2]). Un tel système est dit *réflexif* [26]. La section 3.2 montre en quoi un système à base de µ-noyau se prête bien à la mise-en-œuvre de cette notion de réflexivité. La section 4.1 présente l'approche adoptée par différents µ-noyaux pour mettre en œuvre une forme de réflexivité pour différents services du système.

Mémoire volatile, mémoire à long terme. Une différence marquée a toujours existé entre la façon de traiter la mémoire à long terme et la mémoire à court terme : les systèmes de fichiers sont une abstraction offerte par les systèmes d'exploitation conventionnels pour permettre l'accès à la mémoire à long terme, tandis que la mémoire à court terme est prise en charge par le langage de programmation [9]. Un des objectifs d'un système d'exploitation persistant est donc de masquer cette frontière entre mé-

[1] « *separation of concerns* » en anglais [3].

[2] « *ingérence* » est une traduction du mot anglais « *intercession* ».

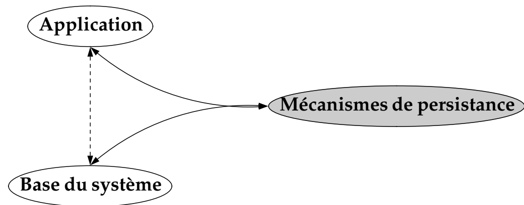


Figure 1. Mise-en-œuvre des mécanismes de persistance de manière orthogonale et transparente. L'application n'a aucune action particulière à faire pour devenir persistante et la base du système peut rester inchangée : les mécanismes de persistance s'*interposent* entre les deux.

moire à court terme et à long terme, en rendant la transition de l'une à l'autre totalement transparent. En d'autres termes, la mémoire principale (mémoire vive) est alors vue comme un simple cache de la mémoire secondaire (disque dur) [24]. Pour ce faire, le système doit sauvegarder régulièrement vers une mémoire à long terme (i.e. faire des *captures* de l'état du système, *checkpoint* ou *fixpoint* en Anglais), de manière transparente aux applications, l'ensemble des données manipulées par chaque processus ainsi que l'état de chacun de ces processus; en cas de panne, le système doit pouvoir utiliser la dernière *capture* réalisée pour restaurer un état *récent*. Par ailleurs, la mémoire principale servant de cache de la mémoire secondaire, les systèmes d'exploitation persistants offrent de meilleures performances pour les opérations d'entrée/sortie avec le système de fichiers que les systèmes d'exploitation traditionnels [17, 23, 15].

2.2. Approche générale

Captures. Une capture de l'état du système doit comporter au moins une image mémoire de chaque tâche et l'état d'exécution (i.e. pointeurs d'instruction, de pile, registres, etc.) de chaque brin d'exécution (*thread*), ce qui s'apparente à l'écriture d'un *core* (*core dump*). Selon l'architecture du système, il peut être nécessaire d'incorporer plus d'informations (voir section 3). Deux approches existent pour la prise de captures de l'état du système : la réalisation de captures de l'ensemble du système de manière synchrone (*synchronous/consistent checkpointing*), ou bien de manière indépendante (*asynchrnous/independent*). Cette deuxième approche, comme nous le verrons par la suite (section 3.1), a pour inconvénient majeur de rendre le travail de sauvegarde plus compliqué.

Recouvrement. Poursuivre son exécution après un arrêt du système doit rester totalement transparent aux applications. Pour que la capture réalisée puisse être utilisée pour recouvrir l'état du système, il faut qu'elle soit *cohérente*. En outre, aucune information nécessaire à la reconstitution de l'état du système ne doit manquer. Par exemple, l'état de la communication entre processus doit être strictement le même avant et après la panne (i.e. aucun message ne doit être perdu, voir section 4.1.2); dans le cas contraire, les applications utilisateurs devraient explicitement vérifier l'intégrité de leurs échanges en cours et la persistance ne serait donc plus *transparente*.

Tâches explicitement persistantes, tâches non-persistantes. Parmi l'ensemble des tâches exécutées par un système d'exploitation, la plupart peut être rendue *implicitement persistante* en utilisant les mécanismes décrits ci-dessus. Cependant, comme le souligne [25], certaines tâches ne peuvent pas être rendues persistantes car elles dépendent d'objets extérieurs au système qui ne sont pas persistants. Les pilotes de périphériques, lorsqu'ils sont exécutés dans l'espace utilisateur (comme c'est le cas pour les systèmes à base de µ-noyaux de seconde génération tels que L4), sont un exemple évident de tâches non-persistantes (*transient tasks*) : l'état d'un périphérique tel qu'une carte réseau ou carte graphique est perdu lors d'un arrêt du système. Par conséquent, les pilotes de périphériques ne peuvent être persistants et doivent être redémarrés avec le système.

Certaines applications doivent par ailleurs s'assurer d'être synchronisées avec les captures du système : on parle de *tâches explicitement persistantes* [25]. Les systèmes de fichiers, par exemple, doivent effectivement synchroniser leurs écritures sur disque avec celles des captures du système.

Interaction avec des entités non persistantes. D'une manière générale, des applications persistantes sont amenées à utiliser les services des tâches non-persistantes. Lors d'un recouvrement, toutes les connections avec ce type de tâches devront donc être invalidées puis éventuellement recréées. Dans ce cas, il est nécessaire que le système ait pu observer les interactions des applications persistantes avec des éléments non persistants; en d'autres termes, il doit pouvoir contrôler cette *frontière* (« *border control* » [4]). Il peut s'agir de communication entre tâches persistantes et tâches non persistantes, ou bien de communication à travers un réseau ou avec un périphérique non persistant. Des techniques de *journalisation* des échanges (voir la *Output Boundary Function* dans [6]) peuvent être utilisées pour permettre de « rejouer » ces transactions lors un recouvrement.

Support de stockage stable. Pour permettre le stockage de leurs données persistantes, les systèmes fournissent généralement un *support de stockage stable* (*stable store*) tel que celui décrit dans [25]. L'objectif est de permettre le recouvrement des dernières données *écrites et validées* (*commited* ou *checkpointed*), ce qui signifie que l'écriture d'un bloc de données ne doit pas détruire le bloc précédemment écrit et n'être effective qu'après validation. Sans ce mécanisme, si le système s'arrêtait alors qu'une capture était en cours d'écriture, la capture disponible sur disque serait incohérente et donc inutilisable. Les supports de stockage stable doivent fournir, en plus des opérations « lecture d'un bloc » et « écriture d'un bloc », des opérations « validation » (*checkpoint*) et « recouvrement » [25]. La validation des données écrites ne sera en l'occurence faite qu'après écriture de la totalité des données à capturer.

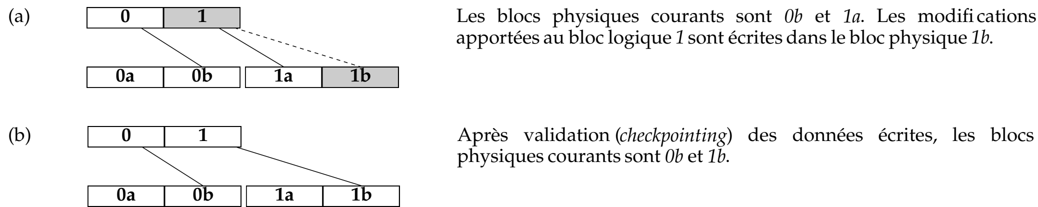


Figure 2. Technique de *shadowing* utilisée par les supports de stockage stables. Les boîtes représentent les blocs physiques et les blocs logiques sous-jacents du support et les droites représentent le lien entre un bloc logique et le bloc physique correspondant en cours d'utilisation.

La technique sous-jacente décrite dans [25] et [9], appellée *shadowing* ou *shadow paging*, consiste à disposer de deux blocs physiques sur disque, un bloc *a* et un bloc *b*, pour chaque bloc logique visible par l'utilisateur. Un tableau spécifie pour chaque bloc logique le bloc physique en cours d'utilisation ; un deuxième tableau permet de déterminer quels blocs logiques ont été écrits depuis le début de cet intervalle. La figure 2 illustre le principe de cet algorithme. Lorsque l'on écrit sur un bloc logique, les données sont écrites non pas sur le bloc physique courant mais sur l'autre ; lorsque l'écriture des données est validée, le bloc physique courant devient le bloc physique sur lequel on a écrit et un tableau mis-à-jour de blocs physiques courants est réécrit sur disque *de façon atomique*. Il est important de noter que cette opération doit être atomique de sorte qu'une panne ne puisse survenir qu'avant ou après l'écriture dudit tableau. Pour cette raison, le tableau de blocs physiques courants doit être suffisamment petit (voir [25]).

2.3. Travaux étudiés

De nombreux travaux se sont portés sur la tolérance aux fautes et la persistance *transparente* à différents niveaux. Au niveau applicatif, des bibliothèques génériques permettent de rendre des applications persistantes avec peu ou pas de modifications dont notamment *libckpt* [21], *libckp* [28] et *libft* [8] pour systèmes Unix. Toutes ces bibliothèques travaillent au niveau du processus, indépendamment du type d'application. *libooft* [12] propose un cadre de développement pour des applications orientées-objets (C++) à plusieurs brins d'exécution (*multithreaded*) offrant dans ce cas de meilleures performances que les bibliothèques génériques précédemment évoquées, et un projet similaire existe pour Java. Ce rapport s'intéresse par la suite aux solutions mettant en œuvre la persistance au niveau du système d'exploitation. Toutefois, un certain nombre de techniques utilisées par ces bibliothèques et notamment décrites dans [21] s'avèrent intéressantes, également au niveau du système d'exploitation.

Un certain nombre de systèmes d'exploitation ont été spécialement conçus avec l'objectif de fournir la persistance : Eumel (1979) et son descendant L3 (1988) [17], Grasshopper (1993) [18], KeyKOS (1983) [15] et son descendant EROS (1996) [23]. A l'exception de Grasshopper, ces systèmes sont tous architecturés autour d'un µ-noyau et réalisent des captures synchrones de l'état du système. Dans le cas de Grasshopper, une partie des mécanismes permettant la persistance est présente au niveau du noyau et un certain nombre d'abstractions spécialement conçues permet d'obtenir un fonctionnement proche de celui décrit précédemment. L3, KeyKOS, et EROS ont tous trois été conçus pour être persistants et leurs noyaux intègrent les mécanismes nécessaires.

Outre ces systèmes spécialement conçus dans le but d'atteindre la persistance, différentes expériences ont été menées pour mettre en œuvre la persistance *par-dessus* un système existant : citons notamment la bibliothèque *libckpt* [21] pour systèmes Unix, et *Fault-Tolerant Mach* (FTM) [19] et *Optimistically Recoverable Mach* (ORM) [6] qui sont construits au-dessus du µ-noyau Mach. La bibliothèque *libckpt* permet de rendre facilement une application Unix persistante. Elle a toutefois une efficacité limitée puisqu'elle dépend totalement de la stabilité du système et que son mécanisme est peu performant puisque mis-en-œuvre au niveau applicatif. Les deux autres systèmes conçus au-dessus du µ-noyau Mach ont tous deux montrés les limites de Mach. Comme nous le verrons dans la section 3, Mach intègre trop de mécanismes pour bien se prêter à l'implémentation de la persistance au niveau utilisateur.

Une dernière approche consiste en l'utilisation de µ-noyau dits de seconde génération. L'idée à la base de ces µ-noyaux est qu'une abstraction ne peut rester à l'intérieur du noyau que si la sortir empêcherait la mise-en-œuvre de fonctionnalités [25]. Une conséquence est que, par rapport aux premiers µ-noyaux comme Mach et Chorus, ces noyaux n'intègrent qu'un nombre très réduit de concepts. Dans le cadre des noyaux L4 [25] et Fluke [27], l'idée est donc de permettre ainsi l'implémentation de la persistance au niveau utilisateur. Cette approche permet de conserver une certaine flexibilité et en particulier, elle permet de développer les mécanismes nécessaires à la persistance de manière *orthogonale*, comme une fonctionnalité que l'on peut ajouter au système. La section 5 (page 11) s'intéresse de plus près à cette approche dans le cadre du GNU Hurd [10], un système d'exploitation libre multi-serveurs fonctionnant actuellement au-dessus de Mach mais destiné à être porté vers d'autres µ-noyaux dont L4.

3. Leçons tirées

Les systèmes d'exploitation persistants évoqués précédemment présentent un certain nombre de différences,

tout d'abord dans leur approche. Pour certains systèmes, la réalisation de captures de l'état est faite indépendemment par différents composants, alors que d'autres systèmes réalisent des captures synchrones de l'ensemble de l'état.

Outre cette différence d'approche, nous verrons que les principales différences entre ces systèmes sont architecturales. Force est de constater que la plupart de ces systèmes sont basés sur un μ-noyau au-dessus duquel un certain nombre de serveurs fournissent les fonctionnalités de base. Les sections suivantes s'attachent à montrer en quoi cette architecture modulaire facilite la mise-en-œuvre de la persistance.

3.1. Captures synchrones ou indépendantes

Comme l'explique la section 2, un système persistant a plusieurs possibilités pour envisager la sauvegarde de captures du système : il peut soit réaliser une capture de l'ensemble des espaces d'adressage des différentes tâches *en même temps* (on parle alors de *capture synchrone*), soit laisser chaque tâche ou espace d'adressage se sauvegarder *indépendemment*.

La bibliothèque *libooft* [12], conçue pour rendre persistants des programmes C++ à plusieurs brins d'exécution, opte pour des captures asynchrones des différents objets et brins d'exécutions du programme. L'objectif est *réduire la granularité* des points de contrôle sans dégradation de la performance par rapport à une bibliothèque de capture des données agissant au niveau du processus telle que *libckpt* [21]. Pour ce faire, *libooft* réalise des points de contrôles *sélectifs* en se basant sur l'observation des interactions entre objets du programme. En gardant trace des appels de méthodes et donc de la dépendance causale entre objets, elle parvient à réaliser des captures ne contenant que les données des objets en interaction. Lors d'un recouvrement après un arrêt du programme, il s'agit de déterminer quelle version de chaque objet utiliser pour parvenir à un état cohérent (*consistent recovery line*).

Le système d'exploitation Grasshopper [18] a une approche similaire principalement due aux abstractions utilisée pour représenter les données et flux d'exécution du système. Les *containers* sont des entités passives créant une abstraction pour l'accès à la mémoire persistante ; chaque *container* dispose d'un espace d'adressage séparé. Les *loci* représentent un flux d'exécution séquentiel (i.e. état d'exécution d'un brin : registres, pointeur d'instruction, etc.) contenu par un *container*. Chaque *container* possède un *point d'invocation* par lequel un *locus* peut « entrer ». Ces deux abstractions sont donc proches de celles d'objet et de brin d'exécution. En l'occurence, Grasshopper, tout comme *libooft*, garde une trace des interactions entre *containers* lui permettant de créer un arbre de dépendances. Ce sont des gestionnaires de *container* qui prennent en charge la sauvegarde des *containers* qu'ils contrôlent, *indépendamment* des autres. Un certain nombre de sauvegardes de chaque *container* sont gardées sur support stable. Lors d'un recouvrement après un arrêt du système, un algorithme permet de déterminer, à partir de l'arbre de dépendance des *containers*, quelle version de chaque *container* utiliser pour parvenir à un état cohérent (cela s'appelle un *consistent cut*).

La réalisation de capture synchrone consiste à sauvegarder à intervalles réguliers l'ensemble des données nécessaires au recouvrement de tous les processus. Dans Grasshopper, l'utilisation de cette technique aurait obligé d'arrêter l'ensemble des processus pendant toute la durée de la sauvegarde des données, ce qui aurait été désastreux [18]. Toutefois, une bonne utilisation du mécanisme de gestion de la mémoire virtuelle permet de ne pas arrêter tous les processus pendant la durée de la sauvegarde ou bien de les arrêter pendant une très courte durée (voir section 3.2). La sauvegarde des données peut alors se faire en parallèle de l'exécution des processus. Cela n'empêche pas pour autant de réaliser des points de contrôles à intervalles proches : dans le cadre de L3 [17], il est possible de réduire à 3 minutes cet intervalle[1]. De plus, dans L4 [25], une raison avancée pour ne pas utiliser un protocole de causalité pour produire une version cohérente de différents points de contrôle asynchrones est que cela nécessiterait de connaître tous les types de communication entre brins. Or, la communication entre brins peut se faire de différentes façons : en utilisant les primitives d'IPC mais aussi en partageant de la mémoire entre brins, par exemple. Il est donc impossible de connaître tous les moyens de communication entre brins et par conséquent d'établir un arbre de dépendance fiable.

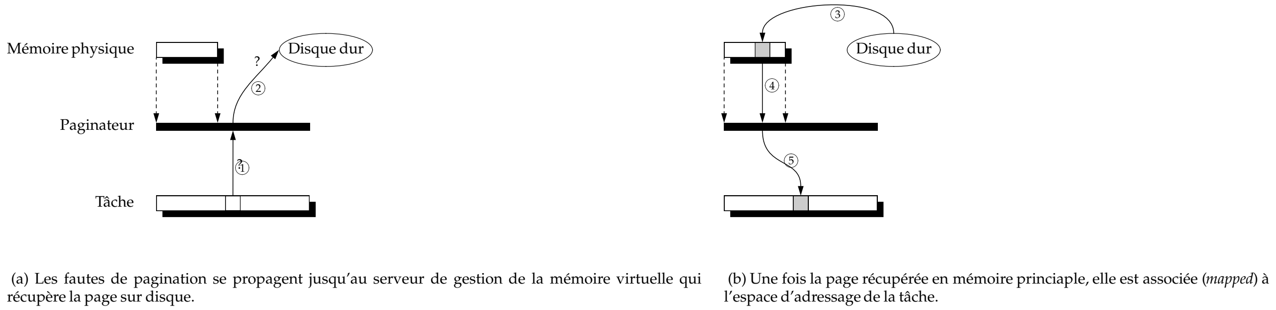
Par ailleurs, pour une capture synchrone utilisant la technique de *shadow paging* décrite en section 2.2 (voir figure 2), on ne conserve au total que deux versions de chaque page, alors que la réalisation de captures asynchrones nécessite beaucoup plus d'espace disque. Dans ces conditions, et vue la complexité ajoutée par la réalisation de captures indépendentes, le mécanisme de points de contrôles synchrones semble mieux adapté à un système d'exploitation monoprocesseur.

3.2. Avantages d'un μ-noyau

3.2.1. Flexibilité

Les systèmes basés sur des μ-noyaux présentent un avantage évident qui est leur *flexibilité*. Cette flexibilité permet de créer des systèmes d'exploitation *réflexifs* (voir section 2.2) et donc de développer les mécanismes de persistance de manière *orthogonale*, c'est-à-dire comme une fonctionnalité que l'on peut choisir d'ajouter au système. Trop souvent, les noyaux de systèmes d'exploitation encapsulent et cachent nombre d'informations susceptibles d'être nécessaires aux applications et en l'occurence nécessaires pour la mise-en-œuvre de la persistance. Par conséquent, pour rendre persistant un système basé sur un tel

[1] Cette durée est peu dépendante du type de machine puisqu'en règle générale, une machine ayant une mémoire physique plus large sera également plus rapide.



(a) Les fautes de pagination se propagent jusqu'au serveur de gestion de la mémoire virtuelle qui récupère la page sur disque.

(b) Une fois la page récupérée en mémoire princiaple, elle est associée (*mapped*) à l'espace d'adressage de la tâche.

Figure 3. Gestion des fautes de pagination. Les nombres portés sur les flèches indiquent l'ordre des étapes. Les flèches en pointillés indiquent les associations (*mappings*); les autres flèches indiquent des requêtes sur la figure (a)et des transferts de données et associations sur la figure (b).

noyau, l'application doit essayer de « recréer » les informations sur l'état du système qui lui sont nécessaires en « inférant » à partir des informations disponibles (par des appels systèmes par exemple, voir section 4.1). Si cela est impossible, le noyau doit être directement modifié pour intégrer la fonctionnalité souhaitée. En l'occurence, pour permettre la persistance, le noyau « idéal » est un noyau ne contentant pas d'informations sur l'état du système (*stateless kernel*) [6].

Un µ-noyau est supposé être suffisamment minimaliste et donc générique (voir section 4.1.3, page 8) pour permettre la mise-en-œuvre de n'importe quel type de service dans l'espace utilisateur, sous forme de serveurs. La persistance pourrait donc être, de la même façon, un service que l'on peut choisir d' ignorer. L4 et Fluke ne proposent par contre aucun mécanisme particulier permettant la mise-en-œuvre de la persistance mais offrent des fonctionnalités la rendant *possible*. Comme le montre la section 4.1, les µ-noyaux L4 et Fluke *exportent* les informations relatives à l'état du système qu'ils maintiennent. Cette technique permet une réelle flexibilité pour le système implémenté au-dessus de ces µ-noyaux[1].

3.2.2. Gestion de la mémoire virtuelle

La gestion de la mémoire est un élément clef qu'il est nécessaire de pouvoir *observer* dans le cas d'un système d'exploitation persistant. Les noyaux de systèmes fournissent généralement l'abstraction de *mémoire virtuelle* : le système « donne l'impression » aux processus d'une quantité de mémoire disponible infinie (limitée seulement par l'espace d'adressage du système). Pour y parvenir, le système s'arrange pour faire migrer, en cas de besoin, des parties de la mémoire principale non utilisées vers un support stable tel qu'un disque dur (ce qui s'appelle aussi *swapping*, en Anglais). Si une partie de mémoire (une *page*) demandée par un processus utilisateur n'est plus disponible en mémoire principale, cela déclenche une exception (*faute de pagination* ou *page fault*) et le système s'empresse de recharger la page manquante à partir du disque pour pouvoir poursuivre l'exécution du processus. Ce processus est illustré par la figure3 dont [25] donne une version plus spécifique à l'architecture L4.

Le principe de sauvegarde de la mémoire principale sur support stable se trouve donc être très proche de celui d'un gestionnaire de mémoire virtuelle. Lors du recouvrement de l'état du système après une panne, le mécanisme de faute de pagination pourrait être utilisé pour recharger les pages manquantes. Les µ-noyaux offrent la possibilité de mettre en place le mécanisme de gestion de la mémoire virtuelle à l'extérieur du noyau (voir section 4.3). Cette possibilité a en premier lieu été proposée par les noyaux Mach et Chorus qui permettent de mettre en œuvre des gestionnaires de mémoire ou paginateurs utilisateurs (appellés *external pagers* par Mach et *mappers* par Chorus) [9]. Il est donc possible d'étendre les fonctionnalités du gestionnaire de mémoire virtuelle pour lui permettre d'intégrer les mécanismes nécessaires à la sauvegarde de captures de la mémoire. Grasshopper, bien qu'il soit monolithique, externalise sa gestion mémoire pour également appliquer ce principe : celle-ci est prise en charge par un ensemble de « gestionnaires » (*container managers*) indépendants [18].

Choices [2] propose une représentation orientée-objet de la mémoire : chaque partie de la mémoire virtuelle est représentée par un *MemoryObject* dont les objets persistants, instances de la classe *PersistentMemoryObject*, sont une spécialisation. La capacité d'*introspection* du système facilite la mise-en-œuvre de la persistance : la classe des objets persistants est elle-même représentée par un objet contenant entre autres la liste de ses instances et peut elle-même être représentée par un objet persistant [2].

3.2.3. Performances

L'avantage d'intégrer les mécanismes de capture au noyau est que l'accès aux structures de données internes du noyau (telles que l'état d'exécution des tâches et brins) est rendu plus facile. Toutefois, il peut arriver que le mécanisme de capture ne puisse alors être fait *qu'à l'intérieur* du noyau : dans ce cas, cela implique souvent une interruption de l'exécution de tous les processus en cours. C'est une des raisons qui a motivé la mise en place d'une cap-

[1]Notons toutefois que les µ-noyaux de L3, KeyKOS et EROS, intègrent eux-mêmes les mécanismes nécessaires à la persistance, faisant ainsi une antorse à ce principe.

ture asynchrone des différentes parties de la mémoire dans Grasshopper [18] : chaque « gestionnaire de mémoire » prend régulièrement des captures de la zone mémoire qu'il contrôle, sans synchronisation avec les autres gestionnaires. Cette méthode évite d'avoir à arrêter l'exécution de tous les processus pendant que les modifications sont enregistrées ; en contrepartie, le fait de réaliser un ensemble de captures asynchrones complexifie grandement le processus de recouvrement comme l'explique la section 3.1.

Les différentes bibliothèques [21, 28, 8, 12] et systèmes tels que ORM [6] mettant en œuvre la persistance au niveau applicatif souffrent du même problème. Pour réaliser une sauvegarde de l'état d'un processus, ces systèmes dupliquent le processus (typiquement par un appel à `fork ()`) et arrêtent le processus original (le père) le temps de l'écriture des données sur disque. Outre le fait que cette technique ne permet pas de recouvrir d'un arrêt total du système (les données étant écrites sur un système de fichier susceptible de les mettre en cache) et est peu performante, il faut noter qu'elle requiert beaucoup d'espace mémoire.

Au contraire, l'écriture d'une capture du système peut, dans le cas d'un système basé sur un μ-noyau tel que L4, se faire *en parallèle* de l'exécution des tâches du système. L'utilisation de « paginateurs externes » permet de mettre en place n'importe quel type de politique de pagination et d'utiliser un certain nombre de techniques répartissant la charge de travail liée à l'enregistrement des modifications de la mémoire principale (ces techniques sont détaillées en section 4.3).

4. Persistance au-dessus d'un micro-noyau

Les expériences menées pour rendre persistants des systèmes à base de μ-noyaux décrites précédemments ont permis d'identifier un certain nombre d'aspects critiques dans la conception desdits μ-noyaux. En particulier, les implémentations de *Fault-Tolerant Mach* [20] et de *Optimistically Recoverable Mach* [6] ont montré les limites de Mach pour la construction d'un système persistant. Les μ-noyaux de seconde génération tels que L4 [25] ou Fluke [27] semblent avoir réussi à offrir un ensemble d' abstractions beaucoup plus réduit mais suffisant pour *permettre* la mise en place d'un système persistant. En d'autres termes, comme le montre la section 3.2, ces mécanismes donnent des propriétés *réflexives* aux noyaux [26].

La section 4.1 explique les propriétés de base que doit avoir un μ-noyaux pour qu'il se prête à la mise-en-œuvre d'un système d'exploitation persistant, rappellant les défauts des μ-noyaux de première génération et les comparant à des alternatives modernes. Les sections suivantes présentent différentes architectures ou concepts utiles à cette réalisation.

4.1. Propriétés d'un micro-noyau candidat

Cette section présente les propriétés de base qui sont indispensables à un μ-noyau susceptible de servir de base à un système mettant en œuvre la persistance de manière *orthogonale*, comme le présente la section 2.1. Les sections suivantes développent plus en détail différents aspects architecturaux qui peuvent servir de base à la création d'un système persistant au-dessus d'un tel μ-noyau.

4.1.1. Appels systèmes

Comme le souligne [27], un μ-noyau au-dessus duquel on souhaite mettre en œuvre un système d'exploitation persistant doit assurer que toutes ses opérations (les « appels systèmes ») soient *interruptibles et puissent être continuées* ou alors *atomiques*. Ces conditions sont indispensables pour garantir que le système est, à tout moment, dans un état cohérent. De cette façon, l'intégrité des données capturées est assurée. Mach, par exemple, ne permet pas de reprendre l'exécution d'un brin si celui-ci a été interrompu au milieu d'une opération non atomique. Son exécution peut par exemple être interrompue au milieu d'une communication avec un autre processus sans qu'il soit possible d'obtenir des informations permettant par la suite de poursuivre l'exécution de cette communication [27]. L3 et L4, au même titre que Fluke, ne fournissent que des opérations atomiques ou redémarrables.

4.1.2. Communication inter-processus

L'abstraction offerte par un μ-noyau pour permettre la communication inter-processus (IPC), c'est-à-dire l'échange de données entre espaces d'adressage, est déterminante pour la faisabilité d'un système persistant. Les projets ORM [6] et FTM [20] sont particulièrement intéressants puisqu'ils mettent en avant les problèmes rencontrés dans ce domaine avec le μ-noyau Mach. Une comparaison avec les μ-noyaux de seconde génération Fluke et L4 permet de comprendre comment ces problèmes ont pu être résolus.

L'IPC de Mach. Le principal reproche fait à Mach est sa lourdeur parfois contradictoire avec l'idée même de système à base de μ-noyau. En l'occurence, l'abstraction de communication inter-processus offerte par Mach s'avère être de trop haut-niveau pour offrir une réelle flexibilité aux applications. Mach ne propose que l'abstraction de *port* pour l'IPC. Un port Mach est un canal de communication entre deux tâches auquel sont attachés des droits d'émission/réception (*send rights*, *receive rights*). A chaque port est associée, au niveau du noyau, une file de messages. Chaque port est repéré par un identifiant qui est lui aussi attribué par le noyau.

L'intégration de ces deux derniers mécanismes au noyau s'avère problématique si l'on souhaite implémenter un système persistant au-dessus de Mach. L'intégration des files de messages au noyau implique qu'en cas d'arrêt du système, il se peut que des messages ayant été envoyés aient été ajoutés à une file sans avoir été effectivement re-

çus, auquel cas ils sont perdus (*missing messages*). Pour pallier ce problème, ORM [6] « intercepte » les envois et réception de messages par les appels système `msg_send ()` et `msg_receive ()` pour journaliser les échanges. Ceci permet lors d'un recouvrement de renvoyer les messages non reçus afin qu'aucun ne soit perdu.

Par ailleurs, les identifiants de port étant automatiquement pris en charge par le noyau, un port recréé lors d'un recouvrement risque d'avoir un identifiant différent de celui qu'il avait avant l'arrêt du système, rendant alors les connexions entre tâches invalides. ORM a donc dû mettre en place un mécanisme permettant de fournir des identifiants de canaux de communication uniques dans le temps appellés *reliable ports*. Pour ce faire, ORM « intercepte » les appels système d'IPC (création de ports, envoi ou réception de messages, etc.) pour permettre aux applications de n'utiliser plus que des *reliable ports* au lieu des ports Mach. Lorsqu'un *reliable port* est créé, un port Mach lui est associé et cette association est transmise à un serveur spécial appellé *Port Authority*. Par la suite tous les accès à un *reliable port* utilisent le port associé pour exécuter l'opération. Après un recouvrement, le port associé n'étant plus valable, toutes les opérations sur ce port (eg. envoi ou réception de messages) vont échouer. Dans ce cas, le programme va demander à la *Port Authority* quel est le nouveau port associé à ce *reliable port*. Si aucun port n'a encore été associé, ORM en recrée un et divulgue cette nouvelle association à la *Port Authority*. ORM parvient ainsi à créer par-dessus l'abstraction des ports un mécanisme de communication plus flexible et tolérant aux fautes. Par ailleurs, il est en réalité impossible d'intercepter des appels systèmes. Par conséquent, les applications persistantes utilisant ORM doivent être liées avec une bibliothèque particulière contenant les « appels systèmes » modifiés. Les applications ORM ne sont donc pas compatibles au niveau binaire avec les autres applications Mach.

Pour sa part, FTM ne cherche pas à rendre les applications persistantes de manière transparente. La mise-en-œuvre de serveurs fiables pour FTM doit utiliser un cadre de développement spécifique (*reliable server framework*) qui inclut des protocoles de communication permettant de pallier aux problèmes exposés précédemment [20, p.29]. Les protocoles mis en place intègrent notamment une procédure d'acquittement permettant de s'assurer qu'un message a effectivement été reçu.

Les µ-noyaux de seconde génération. L4 propose une abstraction de plus bas-niveau que Mach pour la communication inter-processus. Il ne prend pas en charge le concept de « canal de communication » que sont les ports de Mach avec les mécanismes de gestion des identifiants et des files d'attentes [7]. L4 ne dispose que d'une simple primitive de communication (IPC) synchrone. Cette primitive permet l'envoi d'un message à un brin (*thread*) donné du système. S'agissant de messages synchrones, le noyau ne gère aucune file de messages et clients et serveurs doivent donc avoir un « rendez-vous » pour communiquer. La notion de canal de communication, et notamment la gestion des identifiants, doit donc être prise en charge par l'utilisateur de L4 (l'OS) (voir la section 5.3). De même, une abstraction de communication asynchrone peut être construite, si nécessaire, dans l'espace utilisateur. Cette approche était également celle adoptée par L3 [17]. Les informations relatives aux canaux de communication se retrouvent ainsi stockées dans l'espace utilisateur. Par conséquent, lorsque les données des applications sont écrites sur disque lors d'un point de contrôle, ces informations sont également sauvegardées. L'exécution du système peut donc se poursuivre sans difficultés lors d'un recouvrement.

4.1.3. Accès aux données du noyau

Traditionnellement, un certain nombre d'informations, notamment sur l'état des brins d'exécution du système, sont maintenues par le noyau et trop souvent inacessibles aux tâches du système. Comme vu dans la section 3.2, un noyau idéal pour un SE persistant est un noyau qui n'encapsulerait aucune information sur l'état du système : toutes les informations nécessaires à la réalisation d'un point de contrôle seraient accessibles « de l'extérieur », c'est-à-dire par des processus utilisateurs, permettant donc de mettre en œuvre les mécanismes de persistance au dehors du noyau, de manière *orthogonale*. Le noyau doit donc fournir des moyens permettant permettant d'observer (pour pouvoir les sauvegarder lors d'un point de contrôle) et de modifier ces données (pour pouvoir les restorer à partir d'un point de contrôle).

Mach. Le noyau Mach, une des premières implémentations de µ-noyau, manque précisément de ce type de mécanisme réflexifs. De plus, Mach intègre déjà un nombre important d'abstractions de haut-niveau : les notions de brins, de tâche, de canal de communication (voir section 4.1.2), font partie intégrante du noyau. Le problème est qu'il est par conséquent difficile pour un processus utilisateur de contrôler toutes ces données. Comme le montre l'expérience ORM [6], Mach ne propose qu'une interface minimale composée de quelques appels systèmes pour récupérer ou modifier les informations qu'il maintient. Il est par exemple possible d'obtenir des informations sur l'état d'un brin par l'appel système `thread_get_state ()`, voire de le modifier en appellant `thread_set_state ()`. Toutefois, l'utilisation de cette interface est assez lourde et peu élégante : le brin chargé de réaliser les points de contrôle doit récupérer *explicitement* ces informations alors que l'on souhaiterait plutôt qu'il puisse *observer* ou *écouter* les changements. Enfin, Mach ne propose pas ce type d'interface pour chaque abstraction qu'il gère. Il est par exemple impossible d'obtenir de l'information sur l'état des canaux de communication (voir section 4.1.2). La mise-en-œuvre de la persistance au-dessus de Mach, comme dans ORM ou FTM, doit donc faire preuve d'imagination pour pallier à ce manque de transparence de Mach.

L4. L4 [7], en tant que µ-noyau de seconde génération, n'offre qu'un nombre très réduit d'abstractions. En l'occurence, il s'agit essentiellement des notions d'espace

d'adressage et d'association (*mapping*), de brin d'exécution, et de communication inter-processus (IPC), qui à elles-seules suffisent à construire les abstractions de plus haut niveau telles que les tâches, les canaux de communication, etc.. L4 encapsule donc beaucoup moins d'informations que Mach. La section 4.1.2 explique les avantages de L4 dans le cas de la communication inter-processus et la section 4.3 présente la façon dont est gérée la mémoire virtuelle dans un système basé sur L4. Nous nous intéressons donc ici aux autres informations maintenues par L4 c'est-à-dire essentiellement les informations sur l'état des brins d'exécution.

Comme l'explique [25], les files de brins utilisées par l'ordonnanceur (*ready queues* et *waiting queues*) n'ont pas besoin d'être sauvegardées puisque aucune garantie n'est donnée aux applications quant à leur ordonnancement. Les seules informations qu'il est nécessaire de sauvegarder sont donc les informations d'état de chaque brin (pointeur d'instruction, de pile, espace d'adressage, etc.). La structure de données contenant ces information est appellée *Thread Control Block* (TCB). Dans L4, les TCBs sont stockés dans un espace mémoire qui n'est pas propre au noyau et est soumis à la pagination de la même façon que n'importe quelle partie de la mémoire utilisateur. Par conséquent, le serveur en charge de la sauvegarde de captures peut sauvegarder les TCBs de manière transparente, au même titre que n'importe quelle autre partie de la mémoire.

Fluke. Dans l'architecture à base de µ-noyau Fluke, toutes les informations d'état utilisées par le noyau peuvent être *exportées* vers l'espace utilisateur [27]. Pour ce faire, un certain nombre d'objets encapsulant les données relatives à une instance d'une abstraction donnée (eg. brin, espace d'adressage, *mapping*, etc.) ont été définis. Ces objets, appelés *fbbs*, peuvent permettre de récupérer à n'importe quel moment, dans l'espace utilisateur, les données sur l'état d'un objet donné, et peuvent même être modifiés. Il s'agit là, clairement, d'une propriété *réflexive* du noyau Fluke. Les *fbbs* sont comparables à la notion de *méta-objets* qui représentent, dans une optique orientée-objet, l'état et le comportement d'un objet donné (voir [26] à ce sujet). Il existe en outre un appel système permettant de connaître l'ensemble des objets contenus dans une partie donnée de la mémoire [4]. Cette abstraction fait de Fluke un noyau sans information d'état (« *stateless kernel* »).

EROS. Le noyau de EROS utilise un mécanisme similaire à celui de *fbbs* de Fluke [22]. Chaque objet du système se voit attribuer un identifiant unique (OID) par le noyau lors de son allocation. L'accès aux objets du système se fait ensuite par des *capacités* ou « *capabilities* » fournies par le système et contenant l'OID de l'objet correspondant, son type et les droits d'accès correspondants. Il n'existe que deux types de capacité servant à représenter l'ensemble des objets du système :

- les *pages* qui contiennent les données utilisateurs ;
- les *nœuds* (ou *cgroups*) qui sont des tableaux de capacités.

Les objets pris en charge par le noyau, tels que les processus, sont représentés à l'aide de ces abstractions (voir [24]). Les capacités peuvent prendre deux formes différentes selon qu'elles soient sur disque (forme « non-préparée ») ou en mémoire (« préparée »). Lorsqu'une capacité est écrite sur disque, elle est d'abord convertie sous sa forme non-préparée qui ne contient aucun pointeur vers des données du système.

4.2. Observabilité des interactions inter-processus

Comme l'explique la section 2.1, il est généralement souhaitable de ne pas mélanger la mise-en-œuvre des mécanismes de tolérance aux fautes (notamment de persistance) avec la mise-en-œuvre des fonctionnalités de base du système. Cette optique implique que le système fournisse des mécanismes permettant d'*observer* et de *modifier* l'état d'un processus. Si on désire qu'une tâche (un *checkpointer*) puisse réaliser des « captures » de l'état d'une autre tâche, il est évident qu'elle va devoir être dotée de ces capacités d'observation et d'ingérence. Ces capacités sont décrites avec plus de précision dans la description de l'architecture Fluke [4] :

- **Encapsulation de l'état** : le *checkpointer* doit avoir le contrôle sur l'état de la tâche qu'il souhaite rendre persistante. En particulier, son état doit être *visible* et *modifiable*, ce qui est rendu possible par les techniques présentées dans la section 4.1.3.
- **Interaction avec l'extérieur** : toutes les interactions d'une tâche avec l'extérieur (que ce soit avec une autre tâche par *IPC*, avec un système de fichiers, avec un terminal, avec le gestionnaire de mémoire virtuelle, ou autre) doivent pouvoir être observées par le *checkpointer*.

Dans un système à base de µ-noyaux, les fonctionnalités du système sont réparties entre plusieurs tâches (des serveurs) fournissant différents services : gestion des processus, gestion des terminaux, systèmes de fichiers, gestion de la mémoire virtuelle. Toute interaction d'une application avec le système se résume donc à de la communication (*IPC*) avec ces serveurs. Les interactions d'une tâche avec l'extérieur peuvent donc être contrôlées en *interceptant* ses communications. L3 [16], L4 [11] et Fluke [4] adoptent des approches semblables permettant l'observation des interactions d'une tâche avec l'extérieur. Le concept avancé dans L3 et L4 est limité à l'observation de la communication inter-processus mais est par ailleurs complété par d'autres concepts (voir section 4.3) alors que Fluke propose une conception plus générale destinée à permettre d'intercepter tout type d'interaction. Dans les deux cas, il s'agit de créer une *hiérarchie* entre les tâches où une tâche parente a la possibilité d'observer sa ou ses tâches filles.

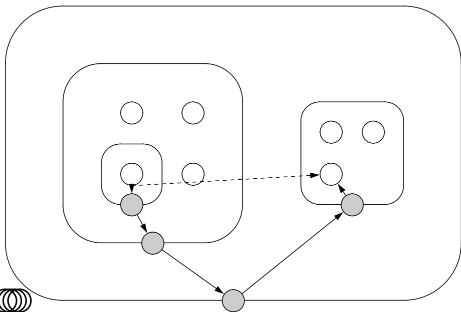


Figure 4. Vue abstraite du principe des « *clans et chefs* » de L3 [16]. Chaque cercle représente une tâche ; les cercles gris représentent les « chefs ». Les flèches représentent des messages ou autre type d'interaction entre tâches, la flèche en pointillés représentant l'interaction voulue par la tâche.

L3 et L4. Le modèle proposé par L3 est celui des *clans et des chefs*. Chaque tâche du système fait partie d'un « clan » : tous les échanges entre tâches d'un même clan sont directs mais par contre les échanges entre tâches de clan différents sont susceptibles d'être inspectés par les différents « chefs » de clan. La figure 4 illustre ce principe. Dans L3, ce organisation hiérarchique de la communication inter-processus faisait partie intégrante du µ-noyau lui-même. Toutefois, ce mécanisme s'avèrant parfois trop lourd, L4, le successeur de L3, se contente de permettre la mise-en-œuvre de ce mécanisme sans le contraindre. Ce mécanisme se base sur le principe de *redirection d'IPC* vers des *superviseurs*[1] [11] et rend possible certaines optimisations, telles que le fait de « court-circuiter » une partie du chemin qu'aurait normalement pris un message avec le modèle des « clans & chefs ».

L'architecture proposée par Fluke [4] se veut plus générale. L'idée est de créer une hiérarchie entre les tâches du système avec des liens de parenté forts, contrairement au modèle d'Unix (un processus père perd pratiquement tout lien avec son fils). L'idée est de pouvoir *virtualiser* les processus : chaque processus père, ou *nester*, s'interpose entre ses applications filles et son propre parent, à la manière d'une machine virtuelle, en fournissant une interface donnée faisant parti d'un jeu de « *protocoles communs* ». Ces protocoles communs intègrent des interfaces telles qu'une interface de gestionnaire de mémoire, de système de fichier, ou encore de gestionnaire de processus, à la manière des interfaces de base du GNU Hurd décrites en section 5. Il existe en outre l'interface *Parent* qui permet à un processus d'obtenir des informations de son *nester* telles qu'un référence sur son gestionnaire de mémoire ou de processus. Il est donc possible d'empiler les fonctionnalités du système ; en d'autres termes, ce modèle permet une « *décomposition verticale* » des fonctions du système qui vient s'ajouter à la « *décomposition horizontale* » offerte par un système multi-serveurs.

L'avantage de cette architecture par rapport au modèle des clans & chefs est qu'un *nester* peut choisir de ne

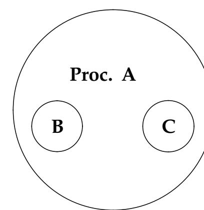


Figure 5. Le modèle de processus imbriqués de Fluke [4]. Ici, le processus A le parent ou « *nester* » des processus B et C.

s'interposer que sur un type de communication (eg. accès au système de fichiers, accès à la gestion de mémoire virtuelle) tandis qu'un chef intercepte toute les communications effectuées par une tâche. Le serveur réalisant les points de contrôle est donc mis-en-œuvre par un *nester* particulier qui fournit les fonctionnalités de tolérance aux fautes. Cette architecture arborescente donne à Fluke l'avantage de pouvoir rendre tout ou partie du système persistant, en fonction de la position du *nester* réalisant les points de contrôle : s'il s'agit du *nester* racine, alors tous les processus du système sont persistants.

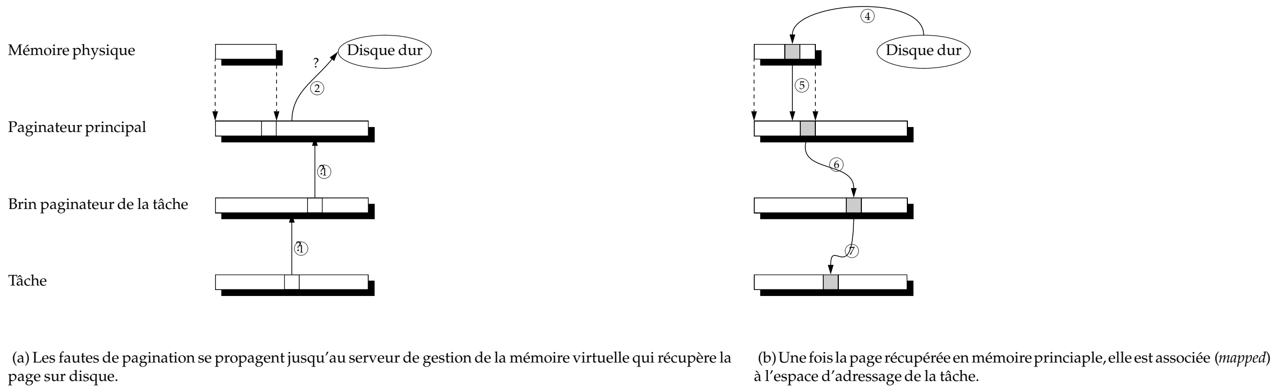
De plus, Fluke utilise des *capacités* telles que celles utilisées par EROS et décrites en section 4.1.3 pour référencer la plupart des objets du système [4]. Un processus père a la possibilité de connaître toutes les capacités que détient son fils et peut lui-même y accéder et les modifier de manière transparente. Un *nester* peut donc intercepter les modifications faites aux objets sous-jacents, contrôlant ainsi toutes les interactions du processus avec l'extérieur. Lors de la réalisation d'un point de contrôle, les capacités détenues par les processus contrôlés sont repérées et se voient assigner un identifiant unique dans l'image sauvegardée sur disque. Le *checkpointer* sauvegarde également un catalogue des références (des identifiants) internes aux processus ; les références externes n'apparaissent pas dans ce catalogue et lors d'un recouvrement, le *checkpointer* pourra éventuellement les prendre en charge. On peut par exemple imaginer que le *checkpointer* journalisera les échanges avec l'extérieur de manière à pouvoir les rejouer lors d'un recouvrement.

Le problème de l'observation des données des processus utilisateurs, présenté dans la section 4.3, s'avère n'être qu'un cas particulier de ce principe : il s'agit d'observer les interactions d'applications avec le gestionnaire de mémoire virtuelle. La section 4.1.3 montre que l'on peut, dans une certaine mesure, encapsuler l'état du noyau lui-même de la même façon que pour une simple tâche (approche L4) et se ramener alors à l'observation de ses interactions avec le gestionnaire de mémoire.

4.3. Gestion de la mémoire virtuelle

La section 3.2 a montré qu'un système d'exploitation actuel fournit toujours un mécanisme de gestion de la mémoire virtuelle. Dans un µ-noyau, ce mécanisme est un service du système, mis-en-œuvre par un serveur particulier qui propose une interface de gestion de la mémoire. Les noyaux Mach et Chorus sont les premiers à avoir proposer une gestion externe de la mémoire. Le rôle des serveurs de

[1] Des « *monitors* » en Anglais.



(a) Les fautes de pagination se propagent jusqu'au serveur de gestion de la mémoire virtuelle qui récupère la page sur disque.

(b) Une fois la page récupérée en mémoire princiaple, elle est associée (*mapped*) à l'espace d'adressage de la tâche.

Figure 6. Gestion des fautes de pagination avec L4 [25].

pagination au-dessus des noyaux Mach et Chorus reste limité puisque le noyau fait toujours une partie du travail de pagination. C'est le noyau qui décide des pages à retirer de la mémoire avant d'en informer le serveur de pagination, alors qu'il eût été plus utile que le noyau demande directement au serveur de retirer une page de son choix [9]. Les µ-noyaux de seconde génération retirent cette limitation et permettent l'implémentation de toute la gestion mémoire dans l'espace utilisateur.

L4. Dans L4 [25], les tâches peuvent elles-mêmes gérer leur pagination (elles sont « auto-paginées » ou « *self-paged* ») : chaque tâche a un brin d'exécution dédié à sa pagination. Lorsqu'un brin d'exécution de la tâche accède à une zone mémoire non-actuellement disponible, il créé une « faute de pagination » qui est détectée par le matériel (*Memory Management Unit* ou MMU) qui génère alors une interruption prise en charge par le noyau ; le noyau génère une IPC vers le brin de pagination de la tâche pour l'informer de l'endroit de la faute. Celui-ci peut alors éventuellement en informer son « paginateur père », c'est-à-dire un serveur responsable de la pagination de cette tâche, et ainsi de suite (voir figure 6). Cette possibilité d'empiler les gestionnaires de mémoire virtuelle et de propager les fautes de pagination est rendue possible par le concept d'*espaces d'adressage récursifs* proposé par L4 : une tâche peut associer une partie de son espace d'adressage à celui d'une autre tâche (*address space mapping*). Le paginateur de base de L4, appellé σ_0, a son espace d'adressage associé directement à la mémoire physique du système; les autres applications peuvent obtenir une partie de cette mémoire en associant leur espace d'adressage à une partie de celui de σ_0. Si l'on souhaite rendre la totalité des applications persistantes, on peut donc implémenter le *checkpointer* du système comme un paginateur « racine ».

Fluke. La section 4.2 présente le concept de processus parent, appellé *nester* dans Fluke. Dans l'architecture de Fluke, un gestionnaire de mémoire n'est autre qu'un *nester* proposant l'interface de gestionnaire de mémoire et s'interposant entre l'application et le noyau [4]. Il est en outre possible pour un processus père d'associer une partie de son espace d'adressage à celui d'un processus fils, ce qui produit une hiérarchisation des paginateurs équivalente à celle présentée précédemment. Un *nester* peut donc « encapsuler » les données de ses fils et donc accéder à ses données, comme dans l'architecture à base de L4 présentée ci-dessus.

5. Etude de cas : rendre GNU persistant

Le Hurd [10] est le « noyau » du système d'exploitation libre GNU, également appellé GNU/Hurd. Le Hurd n'est pas à proprement parler un « noyau » mais plutôt un ensemble de serveurs fonctionnant au-dessus d'un µ-noyau et fournissant les fonctionnalités de base du système d'exploitation. Après un présentation rapide de la conception du GNU Hurd, cette section tente de dégager certains de ses aspects qui permettraient d'en faire la base d'un système persistant, au-dessus de L4.

5.1. Principes du GNU Hurd

5.1.1. Architecture

Le GNU Hurd répartit les fonctionnalités de base du système d'exploitation dans plusieurs *serveurs*. Chaque serveur est un programme exécuté dans l'espace utilisateur, au-dessus du µ-noyau, avec le moins de privilèges possibles, et remplissant une fonctionnalité bien précise. L'objectif est que les fonctionnalités du µ-noyau soient réduites au strict minimum et que l'essentiel des services soient fournis par ces serveurs. Cette approche permet de rendre le système beaucoup plus flexible : il est possible d'ajouter ou d'enlever des fonctionnalités du système pendant son exécution simplement en démarrant ou en arrêtant un serveur. La communication avec ces serveurs se fait par appel de procédures distantes (RPC pour *Remote Procedure Calls*). Les procédures pouvant être appellées sur un serveur sont regroupées dans des *interfaces ;* un serveur peut implémenter une ou plusieurs interfaces.

En pratique, les fonctionnalités implémentées sous forme de serveur dépendent de celles fournies par le µ-noyau sous-jacent. Le µ-noyau GNU Mach, par exemple,

intègre les pilotes de périphériques, alors que L4 délègue la gestion des périphériques à des serveurs utilisateur. En dehors du cas particulier de la gestion des périphériques matériels, la plupart des fonctions traditionnelles de système d'exploitation sont fournies par des serveurs : systèmes de fichiers, protocoles réseau, gestion des processus, gestion de la mémoire virtuelle, etc. A chacun de ces services correspond une interface : interface de protocole réseau, interface d'entrées/sorties, etc.

Enfin, la bibliothèque du C (*glibc*) fournit une interface POSIX complète qui permet d'y faire fonctionner de manière transparente n'importe quelle application écrite pour un système POSIX.

5.1.2. Nommage des serveurs

Pour pouvoir établir une communication avec un programme donnée (un serveur) de pouvoir y faire référence, de pouvoir le nommer. La fonction de nommage de serveur, c'est-à-dire d'association d'un « nom » à une tâche, n'étant pas prise en compte par le µ-noyau, il est nécessaire de mettre en place un *serveur de nom* à qui tout programme peut demander l'accès à un serveur à partir de ce nom. Le serveur de nom est ainsi le seul serveur dont une tâche devra avoir connaissance à sa création ; un tâche peut ensuite se référer à tout autre serveur dont elle connaît le nom de manière *dynamique*, en demandant au serveur de nom d'établir un canal de communication entre elle et ledit serveur.

Dans le Hurd, l'espace de nommage des serveurs est le système de fichiers. Le nommage des serveurs est donc hiérarchisé. Le *serveur racine*, appellé `"/"`, a connaissance des serveurs qui se sont enregistrés auprès de lui, mais chacun de ses serveurs peut lui-même servir de serveur de nom pour d'autres serveurs ; il est alors apparenté à un *répertoire* du système de fichiers. Chaque serveur du Hurd a donc un nom et un *chemin* dans l'arborescence des serveurs. De la même façon que pour un fichier, on peut se référer à un serveur par un chemin absolu (à partir de la racine du système de fichiers) ou par un chemin relatif (par rapport au « répertoire courant »). Ce mécanisme est schématisé par la figure 7. Un serveur du Hurd est donc res-

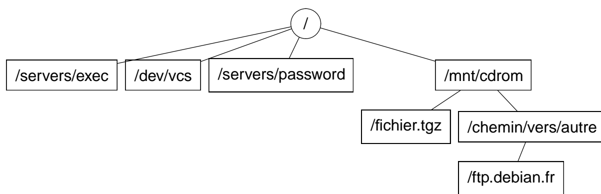


Figure 7. Le système de fichiers comme espace de noms des serveurs du GNU Hurd.

ponsable « d'interpréter » les requêtes qui sont faites au nœud auquel il est rattaché dans l'arborescence du système de fichiers; pour cette raison, les serveurs du Hurd sont appellés des *traducteurs* (ou *translators* en Anglais).

5.1.3. Etablissement d'une connexion avec un serveur

Pour que des clients puissent accéder à un serveur du système, ce dernier doit avoir une entrée dans le système de fichiers qui constitue « l'annuaire » des serveurs du système. Pour qu'un serveur puisse avoir une entrée dans le système de fichiers, il doit implémenter au moins l'interface de système de fichiers. Cette interface permet aux autres applications de le voir (`fsys_getroot ()`), de lui demander la liste des « fichiers »[1] qu'il contient (`dir_readdir ()`) ou encore d'établir une connexion avec l'un d'eux (`dir_lookup ()`).

Lorsqu'une application souhaite établir une connexion avec un serveur dont elle connaît le chemin absolu, de la forme `/chemin/vers/serveur`, elle commence par interroger le serveur racine (avec lequel elle a déjà établi une connexion héritée de son père) en appellant sa méthode (RPC) `dir_lookup ()` avec pour argument ce chemin. Si, par exemple, `/chemin` désigne un autre serveur, le serveur racine lui renverra un *handle* qui lui permettra de dialoguer avec ce serveur. En appellant `dir_lookup ()` sur ce serveur avec le chemin `vers/serveur` en argument, elle établira alors une connexion avec le serveur voulu. Si le chemin vers le serveur est relatif, cela se passe de la même manière mais en commençant par interroger le « répertoire courant » au lieu du serveur racine.

5.1.4. Serveurs de base

Les serveurs de base du Hurd sont partiellement décrits dans [1]. Voici les principaux et une brève explication sur leur fonction dans le système :

- **Le serveur d'authentification.** Le serveur d'authentification garde traces des utilisateurs du système et de leurs identifiants. Il permet également d'ajouter un nouvel utilisateur au système.

- **Le serveur de mots de passe.** Ce serveur permet à un utilisateur ayant saisi le bon mot de passe de s'authentifier auprès du serveur d'authentification. Il s'agit d'un serveur « de confiance » par rapport au serveur d'authentification, c'est-à-dire ayant les privilèges d'administrateur (« *root* »).

- **Le serveur de processus** s'assure d'attribuer un identifiant à chaque tâche du système et maintient une correspondance entre cet identifiant et la tâche correspondante. C'est également lui qui a connaissance des arguments passés à une tâche (« `argv` ») et de ses variables d'environnement (« `envp` »).

- **Le serveur d'exécution** prend en charge l'exécution de tâches. Il prend donc principalement pour arguments un fichier à exécuter, les arguments à lui passer,

[1] Il peut s'agir de « fichiers » gérés par le serveur lui-même ou de serveurs enregistrés auprès de ce serveur.

ses variables d'environnement, et les canaux de communication dont il doit hériter.

- **Le serveur de terminaux virtuels.** Il s'agit du programme mettant en œuvre la notion de « terminaux virtuels » : une même machine propose plusieurs terminaux et il est possible de passer de l'un à l'autre (de le même façon que sous Linux). Le serveur lui-même n'est pas responsable de la gestion de l'affichage ni de la gestion des entrées (clavier, souris, etc.) qui sont prises en charges par des programmes clients ; il se contente donc de maintenir le contenu de chacun des terminaux, de prendre en compte les changements demandés par les clients et d'en informer les clients d'affichage.

Comme nous l'avons vu plus haut, chaque tâche du système doit avoir établi un canal de communication avec le serveur racine, sans quoi il lui sera impossible d'accéder aux autres serveurs du système. Ce canal est donc créé à la création de la tâche par sa tâche parente. De la même façon, chaque tâche du système « hérite » d'un canal de communication vers les serveurs d'authentification (*auth*) et de processus (*proc*) pour des raisons de sécurité. Ce sont les trois seuls serveurs n'ayant pas d'entrée dans le système de fichiers.

5.2. Avantages du Hurd, limites de Mach

L'architecture du GNU Hurd présente un certain nombres d'avantages qui pourraient en faire une bonne base pour un système d'exploitation persistant. Il s'agit d'une application concrète des principes de modularité et de flexibilité qui ont présidé à la recherche sur les µ-noyaux. Toutes les fonctionnalités de base du système ont été réparties entre différents serveurs, chacun rendant un service bien délimité. En outre, il est possible de réutiliser les interfaces des différents services de base pour écrire un serveur qui pourraient *s'interposer* entre une application et le serveur de base du système remplissant la fonctionnalité voulue, comme décrit dans la section 5.4, à la manière des *nesters* de Fluke [4]. Le GNU Hurd se prête donc *a priori* bien à la mise-en-œuvre de mécanismes de persistance de manière orthogonale.

Toutefois, le Hurd est pour l'instant limité par le µ-noyau sous-jacent, GNU Mach. Il s'agit d'une variante de Mach, avec toutes les limitations déjà évoquées (section 4.1, page 7). Un port du Hurd sur le µ-noyau L4 est en cours [13] et constituerait une bonne base pour la mise-en-œuvre des mécanismes de persistance de manière orthogonale. Nous ne considérerons donc par la suite que le port du Hurd sur L4. Beaucoup d'idées sur la façon dont pourrait être mis-en-œuvre un système persistant au-dessus de L4 ont déjà été décrite par Espen Skoglund *et al.* dans [25]. Nous essaierons donc ici de mettre l'accent sur certains aspects spécifiques à GNU.

5.3. Communication inter-processus persistante

Le µ-noyau Mach intègre lui-même la notion de canal de communication, représentée par des *ports*, ce qui rend difficile la mise-en-œuvre d'IPC persistante (voir section 4.1). Cette abstraction ne fait par contre pas partie de L4 et peut donc être implémentée de la façon voulue au-dessus de celui-ci. Rappellons que dans L4, l'IPC est *synchrone* et le destinataire est repéré par son identifiant de brin (« *thread id* »). En identifiant bien les informations nécessaires à l'identification d'un canal de communication entre client et serveur, on se rend compte qu'elle peut être prise en charge directement par ceux-ci[1] :

- **côté serveur**, un triplet comprenant l'identifiant de la tâche cliente, l'identifiant du brin du serveur prenant en charge la communication, et un identifiant de canal suffit à identifier un canal de communication avec une tâche ; l'identifiant du canal (« *handle id* ») peut être attribué par le serveur et doit être unique pour un client donné ;

- **côté client**, le couple formé de l'identifiant du brin du serveur et du *handle id* suffit à identifier un canal de communication.

A condition que les identifiants des brins d'exécution et que les identifiants de processus (attribués par le serveur de processus du Hurd) soient persistants, les identifiants de canaux de communication le sont aussi. De plus, si l'on considère la possibilité de pouvoir rendre persistantes seulement certaines tâches du système, il faut que ces identifiants soient *relatifs* à l'ensemble de ces tâches comme expliqué par Bryan Ford *et al.* dans [4]. Si les identifiants de canaux de communication sont globaux (chacun est unique dans le système), il est peu probable qu'il puissent être réutilisés lorsque l'on souhaitera reprendre l'exécution des tâches après les avoir arrêtées.

Les identifiants de processus (PID) d'un ensemble de tâches peuvent être rendus locaux assez facilement. Il suffit pour cela de créer un serveur *proc* qui soit propre à ces tâches et leur délivre donc des identifiants locaux auxquels correpondent des PIDs globaux (délivrés par le serveur *proc* principal) qui seront cachés aux tâches en question (voir section 5.4). Toutefois, les identifiants de brins d'exécutions (*ThreadIds*) sont globaux et délivrés par un serveur privilégié du système en charge de la création et destruction des tâches [14, section 2.4], le « serveur de tâches ». Par conséquent, l'utilisation de *ThreadId* comme élément formant les identifiants de canaux de communication en fait des références *absolues* dans le système ce qui empêche *a priori* la sauvegarde de seulement certaines tâches du système.

Il s'agit donc de trouver un moyen de rendre les identifiants de canaux de communication (ou « *object handles* »)

[1] Voir à ce sujet la discussion menée par Neal H. Walfield, principal développeur impliqué dans L4-Hurd : *<http://mail.gnu.org/archive/html/l4-hurd/2002-10/msg00016.html>*.

locaux à un ensemble de tâches persistantes. Au lieu d'utiliser *directement* les identifiants de brin, les applications utiliserait ce type d'identifiant. Lorsqu'une application veut faire une IPC, l'application (en fait, une bibliothèque de communication sous-jacente) traduit cet identifiant local en le *ThreadId* du serveur correspondant. Pour maintenir cette association entre identifiants de canaux de communication et *ThreadId*s, il est donc nécessaire d'avoir un serveur de nom particulier à la manière de la *Port Authority* de ORM (section 4.1.2) : lorsqu'un canal de communication est créé, ce serveur en est averti ; lorsqu'une application souhaite connaître le *ThreadId* du serveur correspondant à un canal, elle peut le demander à ce serveur.

Toutefois, comme dans le cas d'ORM, les applications n'ont pas besoin d'interroger ce serveur de nom à chaque fois qu'elles veulent faire une IPC. Elles peuvent simplement garder une copie locale du *ThreadId* du serveur (en fait, la bibliothèque de communication s'en chargerait) et le réutiliser directement pour les prochaines IPC, jusqu'à ce qu'une IPC échoue. Quand une IPC échoue, cela signifie que soit le *ThreadId* ne désigne aucun brin, soit que le brin désigné par le *ThreadId* n'a pas pu prendre en charge la requête. Dans les deux cas, cela signifie probablement que l'exécution de la tâche a été interrompue et vient de reprendre (recouvrement) et que donc les identifiants de brin ont changé. Dans ce cas, la tâche client peut réinterroger le serveur de nommage des canaux pour connaître le nouveau *ThreadId* du serveur.

En regardant encore l'architecture du Hurd, on se rend compte que les PIDs fournis par le serveur *proc* répondent aux exigences de nos identifiants de canaux : il peuvent être rendus locaux (en démarrant un nouveau serveur *proc* – section 5.4) et persistants (en stoquant dans l'image sauvegardée l'association entre PID et tâche). Par conséquent, un serveur de nommage des canaux est inutile pour le Hurd sur L4. Les canaux de communication peuvent alors être représentés de la façon suivante :

- **côté client**, l'application stocke un couple (PID du serveur, identifiant de l'objet (*handle id*)) où le PID du serveur aura été attribué par un serveur *proc* local à l'ensemble des tâches persistantes ;
- **côté serveur**, le triplet présenté ci-dessus est toujours valide car L4 permet d'utiliser des identifiants de brins locaux entre brins exécutant dans un même espace d'adressage [14, section 2.1] ; les identifiants de brins (globaux et locaux) font tous deux partie des *Thread Control Blocks* qui sont sauvegardés dans l'image (section 4.1.3 et [14, section 2.2]).

A ce stade, une question reste toutefois ouverte : comment les applications persistantes se réfèrent-elles à leur serveur *proc* ? Il en effet impossible d'utiliser le PID dudit serveur pour s'y référer puisque c'est à ce serveur que les applications demandent le *ThreadId* correspondant à un PID. Lors d'un recouvrement, si le *ThreadId* du serveur a changé (ce qui est très probable), les application persistantes seraient isolées car incapables de communiquer. On se retrouve donc avec le problème posé par les identifiants globaux persistants. Heureusement, L4 permet à l'utilisateur d'intégrer des « numéros de version » aux identifiants de brin : sur un *ThreadId* de 32 bits, 14 peuvent être utilisés par l'utilisateur en fonction de ses besoins [14, section 2.1]. Il serait donc possible d'attribuer aux serveurs *proc* persistants un *ThreadId* avec un « numéro de version » particulier (différent de celui des autres brins) et de s'assurer que seul un serveur *proc* peut se faire attribuer cet identifiant.

5.4. Encapsulation de processus

Comme nous l'avons vu précédemment, chaque processus du Hurd « hérite » de son père des canaux de communication avec le serveur racine, le serveur d'authentification, et le serveur de processus et l'accès à tous les autres serveurs du système se fait à partir du chemin du système. Il est donc possible de créer des tâches avec des canaux de communication ouverts vers des serveurs racine, *auth* et *proc* secondaires qui seraient en mesure d'intercepter et éventuellement de faire suivre toutes les requêtes faites par ladite tâche. Il est donc possible par ce biais de contrôler les interactions d'une tâche avec l'extérieur à la manière des *nesters* de Fluke (section 4.2)[1].

Plus précisément, la section 5.3 montre l'intérêt d'avoir un serveur *proc* propre à l'environnement des tâches persistantes du système. Démarrer des tâches en leur fournissant un canal de communication vers un *proc* secondaire est tout-à-fait possible. Celui-ci aurait lui-même pour serveur *proc* sont *proc* « parent » (éventuellement le serveur *proc* « racine »). Il peut donc se comporter comme un « *proxy* » de son serveur *proc* parent : son rôle se réduirait à établir une correspondance entre les « vrais » PIDs des tâches attribués par son *proc* et les PIDs persistants qu'il attribue aux tâches visibles dans son sous-environnement. Notons qu'il devra également attribuer des PIDs aux tâches extérieures à l'environnement persistant afin que les tâches persistantes puissent s'y référer. Toutefois, sachant que tel PID désigne une tâche extérieure, il peut choisir lors d'un recouvrement d'invalider les canaux de communication repérés par ce PID.

Créer un environnement persistant pour des tâches consisterait donc à lancer ces tâches en leur passant un serveur *proc* particulier. On peut même imaginer aller plus loin en observant et en journalisant les communications des tâches persistantes avec l'extérieur. Pour ceci, il suffit de les lancer avec pour serveur racine un serveur particulier. Ce serveur se comporterait essentiellement comme un *proxy* du « vrai » serveur racine. Puisque toutes les communications initiées par les tâches persistantes passeraient par lui (au moins à l'établissement), il lui serait possible de

[1]Un certain nombre d'outils du Hurd existent déjà pour exécuter une application avec un autre serveur racine (option `-chroot` de la commande `settrans`) ou avec un autre serveur *auth* (`fakeauth`). Il est également possible de démarrer un Hurd entier à l'intérieur d'un Hurd (`boot`).

connaître tous les canaux de communication ouverts par ces tâches. Le serveur racine pourrait donc, par exemple, journaliser toutes les communications de l'interface d'entrée/sortie. De cette façon, lors d'un recouvrement, le serveur racine pourrait s'attacher à essayer de *recréer* l'état des serveurs d'entrée/sortie non-persistants : il peut par exemple réouvrir les fichiers qui étaient resté ouverts, repositionner les pointeurs de fichier au bon endroit, etc. C'est un travail assez fastidieux qui ne pourrait pas être généralisé à toutes les interfaces de communication. On peut toutefois penser qu'il serait possible de l'implémenter en se limitant à un type d'interface telle que l'interface d'entrée/sortie.

Enfin, pour les applications POSIX effectuant toutes leurs opérations d'entrée/sortie en utilisant le mécanisme des descripteurs de fichiers, une interface du Hurd (l'interface de *messages*) permet d'obtenir et de modifier la table des descripteurs de fichiers d'un processus. Cette possibilité se rapproche de la celle offerte aux *nesters* de Fluke de pouvoir connaître toutes les capacités détenues par leurs fils (section 4.2).

5.5. Gestion de la mémoire virtuelle

Le Hurd sur L4 disposera d'un type de serveur supplémentaire que n'a pas le Hurd sur Mach : les serveurs mémoires. Utilisant la possibilité offerte par L4 d'associer « récursivement » des espaces d'adressages (section 4.3), il sera possible de créer une hiérarchie de serveurs mémoires[1]. L'environnement des tâches persistantes pourra donc avoir son propre serveur mémoire mettant en œuvre des mécanismes tels que ceux décrits par Espen Skoglund *et al.* [25]. Il pourra notamment réaliser la pagination en utilisant la technique du *shadow paging* décrite en section 2.2.

Par ailleurs, une première approche pourrait consister à utiliser les services offerts par le serveur *crash* du Hurd. Ce serveur est responsable de la gestion des fautes des tâches du système : il peut soit suspendre l'exécution d'un tâche fautive, soit la « tuer », soit sauvegarder ses données sur disque (ce que l'on appelle un *core dump*) avant de la tuer. Lorsque l'on souhaite arrêter l'exécution d'une tâche, on pourrait donc d'abord demander à ce serveur de générer un fichier *core* ; lors du recouvrement, il s'agirait de relancer l'application en utilisant les données contenues dans ce fichier, à la manière de ce que font les débogueurs tels que *gdb*.

5.6. Mise-en-œuvre

Il ne s'agit ici que d'une proposition sur certains aspects de la mise-en-œuvre de la persistance pour GNU sur L4. Le port du Hurd sur L4 en étant encore à ses débuts, cette proposition n'a pas été implémentée. En outre, les détails de conception du Hurd sur L4 n'étant pas encore tous été fixés (ou étant tout simplement inconnus), elle aura sans doute à être adaptée. En particulier, les nouveaux serveurs dont disposera le Hurd sur L4 tels que le serveur de tâches, ou encore le serveur de mémoire, n'ont que peu été pris en compte. Notons toutefois que l'idée d'encapsulation des tâches persistantes, à la manière des *nesters* de Fluke, ne dépend que de fonctionnalités du Hurd existantes et indépendantes du µ-noyau sous-jacent (à l'exception du mécanisme d'identification des canaux de communication). Par conséquent, certaines expériences pourraient être menées dans un premier temps sur Hurd/Mach pour tester, par exemple, l'idée de journalisation des opérations d' entrée/sortie faites par une application.

[1] La présentation faite par Neal H. Walfield aux *Rencontres Mondiales du Logiciel Libre 2002* propose une architecture pour la gestion de la mémoire virtuelle dans Hurd/L4 [5].

6. Conclusion

La recherche sur les systèmes d'exploitation persistants n'est pas un sujet récent. Toutefois, malgré tous les travaux faits dans ce domaine, il s'avère que peu d'implémentations sont disponibles. Des projets pourtant récents tels que Fluke ou EROS mettent effectivement en œuvre la persistance mais ne sont malheureusement pas réellement utilisables, et des projets plus anciens tels que KeyKOS ou L3 ne semblent plus être disponibles. Même si le port du GNU Hurd sur L4 est encore loin d'être réellement fonctionnel, il devrait tout de même pouvoir constituer une bonne base pour un système persistant « utilisable dans la vie de tous les jours » puisqu'il propose une interface POSIX. Il a de plus l'avantage de proposer une architecture permettant de mettre en œuvre les mécanismes de tolérance aux fautes de manière indépendante du reste du système.

Références

[1] Thomas Bushnell, BSG. Towards a New Strategy of OS Design. *GNU's Bulletin, Cambridge, MA (USA)* (janvier 1994). URL *http://www.gnu.org/software/hurd/hurd-paper.html*.

[2] Roy H. Campbell, Nayeem Islam, David Raila, Peter Madany. Designing and Implementing Choices : An Object-Oriented System in C++. *Communications of the ACM* **36** (9) (septembre 1993). URL *http ://choices.cs.uiuc.edu/Papers/Journal/cacm93.pdf*.

[3] Juan Carlos Ruiz, Marc-Olivier Killijian, Jean-Charles Fabre, Pascale Thévenod-Fosse. Reflective fault-tolerant systems : from experience to challenges. Rapport technique (février 2002), LAAS (TSF).

[4] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Stephen Clawson. Microkernels Meet Recursive Virtual Machines. dans *Proceedings of OSDI '96*. University of Utah, Salt Lake City, UT 84112 (USA), octobre 1996. URL *http://www.cs.utah.edu/flux/fluke/html/* . Présentation de l'architecture à base de µ-noyau *Fluke*.

[5] Neal H. Walfield. Virtual Memory Management, a New Approach for the Hurd on the L4 Microkernel. URL *http ://web.walfield.org/papers/gnu-virtual-memory-management-system-lsm-2002-07-14/*. Transparents de la présentations faite aux *Rencontres Mondiales du Logiciel Libre 2002*.

[6] Arthur Goldberg, Ajei Gopal, Kong Li, Rob Strom, David F. Bacon. Transparent Recovery of Mach Applications. Research Paper (1990). A propos de *Optimistically Recoverable Mach*.

[7] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, Jean Wolter. The Performance of µ-Kernel-Based Systems. dans *16th ACM Symposium on Operating Systems Principles (SOSP '97), Saint-Malo, France*. Dresden University of Technology, D-01062 Dresden, Germany, octobre 1997.

[8] Yennun Huang, Chandra Kintala. Software Fault Tolerance in the Application Layer. dans *23rd Intl. Symposium on Fault Tolerant Computing (FTCS-23), Toulouse, France*. AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974., 1995. Etudes de composants applicatifs Unix pour la tolérance aux fautes (*libft*, *watchd*, *REPL*).

[9] David Hulse, Alan Dearle. RT1R1/2 : Report on the efficacy of Persistent Operating Systems in supporting Persistent Application Systems. Research Paper, University of Stirling, Scotland.

[10] Le GNU Hurd, Free Software Foundation. URL *http ://www.gnu.org/software/hurd/hurd.html*.

[11] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, Yoonho Park. Flexible Access Control Using IPC Redirection. dans *7th Workshop on Hot Topics in Operating Systems, Rio Rico, Arizona (USA)*, mars 1999. URL *http://www.ira.uka.de/research/documents/l4ka/*.

[12] Mangesh Kasbekar, Chandramouli Narayanan, Chita R. Das. Selective Checkpointing and Rollbacks in Multi-Threaded Object-Oriented Environment. dans *IEEE Transactions on Reliability*. Pennsylvania State Universisty (USA), décembre 1999. Implémentation de *libooft*.

[13] Ian Duggan. Site internet du port du GNU Hurd sur L4 (L4-Hurd).. URL *http://www.nongnu.org/l4hurd/*.

[14] L4Ka Team. L4 eXperimental Kernel Reference Manual, Version X.2. Manuel de référence (juin 2002), Universität Karlsruhe. URL *http://www.l4ka.org/projects/version4/*.

[15] Charles R. Landau. The Checkpoint Mechanism in KeyKOS. dans *Second International Workshop on Object Orientation in Operating Systems (IWOOOS'92)*. Key Logic, California (USA), septembre 1992. URL *http://www.cis.upenn.edu/~KeyKOS/*.

[16] Jochen Liedtke. Clans & Chiefs. dans *GI/ITG-Fachtagung Architektur von Rechensystem*, mars 1992. URL *http://os.inf.tu-dresden.de/L4/bib.html*.

[17] Jochen Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. dans *International Workshop on Object-Orientation in Operating Systems (I-WOOOS'93), Asheville, North Carolina*. German National Research Center for Computer Science, décembre 1993. URL *http://os.inf.tu-dresden.de/L4/doc.html*. Expérience des systèmes Eumel et L3.

[18] Anders Lindström, Rex di Bona, Alan Dearle, Stephen Norris, John Rosenberg, Francis Vaughan. Persistence in the Grasshopper Kernel. dans *Proceedings of the Eighteenth Australasian Computer Science Conference, ACSC-18, ed. Ramamohanarao Kotagiri, Glenelg, South Australia*, pages 329-338. University of Sydney, University of Adelaide, février 1995. URL *http://os.dcs.st-and.ac.uk/GH/*.

[19] Gilles Muller, Mireille Hue, Nadine Peyrouze. Performance of Consistent Checkpointing in a Modular Operating System : Results of the FTM Experiment. dans *First Dependable Computing Conference, Berlin, Germany*. IRISA/INRIA, Bull Research, octobre 1994.

[20] Gilles Muller, Michel Banâtre, Mireille Hue, Nadine Peyrouze et Bruno Rochat. Lessons from FTM : an Experiment in the Design and Implementation of a Low Cost Fault Tolerant System. Publication Interne n°913, projet Solidor (février 1995), IRISA/INRIA (Rennes, France), Bull Research.

[21] James S. Plank, Micah Beck, Gerry Kingsley. Libckpt : Transparent Checkpointing Under Unix. dans *Usenix Conference Proceedings*. University of Tennessee, Knoxville (USA), janvier 1995.

[22] Jonathan S. Shapiro, Jonathan Adams. Design Evolution of the EROS Single-Level Store. dans *2002 USENIX Annual Technical Conference*. Johns Hopkins University, University of Pennsylvania, mai 2002. URL *http://www.eros-os.org/devel/00Devel.html*.

[23] Jonathan S. Shapiro, David J. Farber, Jonathan M. Smith. Consistency Management in the EROS Kernel. Research Paper (février 1996), University of Pennsylvania. URL *http://www.eros-os.org/devel/00Devel.html*.

[24] Jonathan S. Shapiro, David J. Farber, Jonathan M. Smith. State Caching in the EROS Kernel. Research Paper (septembre 1996), University of Pennsylvania. URL *http://www.eros-os.org/devel/00Devel.html*.

[25] Espen Skoglund, Christian Ceelen, Jochen Liedtke. Transparent Orthogonal Checkpointing Through User-Level Pagers. dans *9th International Workshop on Persistent Object Systems (POS9), Lillehammer, Norway*. System Architecture Group, University of Karlsruhe, septembre 2000. URL *http://www.l4ka.org/publications/*.

[26] François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian. Principles of Multi-Level Reflection for Fault-Tolerant Architectures. dans *Pacific Rim International Symposium on Dependable Computing 2002, Tsukuba (Japon)*. LAAS-CNRS, 31077 Toulouse, France, décembre 2002.

[27] Patrick Tullmann, Jay Lepreau, Bryan Ford, Mike Hibler. User-level Checkpointing Through Exportable Kernel State. Research Paper (1996), University of Utah (Etats-Unis). URL *http ://www.cs.utah.edu/projects/flux/* . A propos du µ-noyau Fluke.

[28] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, Chandra Kintala. Checkpointing and Its Applications. dans *IEEE Fault-Tolerant Computing Symposium (FTCS-25)*. AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974., June 1995. Décrit l'expérience acquise avec la mise-en-œuvre de *libckp* et ses différentes applications.