Gestion de mémoire informatique

Michelland Sébastien, TIPE 2017

Problématique

Les systèmes d'exploitation modernes sont tous multiprogrammés : si les processus géraient chacun leur mémoire, ils devraient communiquer pour s'entendre sur le partage et de nombreux problèmes seraient à prévoir. À la place, les systèmes se chargent d'implémenter la mémoire virtuelle, le va-et-vient et les algorithmes d'allocation. Cependant ces procédures sont cruciales pour garantir la performance et la complexité de toute structure de données ; les processus doivent donc connaître en détail les algorithmes sous-jacents et quand les utiliser.

Travail réalisé

Pour répondre à ce problème, j'ai développé un programme de simulation avec lequel j'ai comparé le comportement de plusieurs algorithmes. Le programme simule ainsi un ordinateur paginé exécutant parallèlement plusieurs tâches qui allouent et libèrent de la mémoire selon des schémas prédéterminés. Les algorithmes sont évalués sur des critères pratiques pour dégager les cas dans lesquels ils sont les plus performants. Un algorithme composite est proposé en conclusion : on cherche par son biais à améliorer la performance et la robustesse du système d'allocation.

Schémas d'exécution des processus

Le simulateur fonctionne en utilisant des schémas d'allocation/libération prédéterminés. Pour assurer une simulation réaliste, il était souhaitable d'enregistrer le comportement de programmes réels. J'ai essayé sans succès plusieurs outils de Linux pour cela :

- `ps` et les fichiers de `/proc` fournissent l'utilisation totale de mémoire d'un processus à un instant donné, mais cela est loin d'être suffisant ;
- `valgrind` prend des captures régulières de la mémoire totale, mais seulement une pour cent allocations en moyenne ;
- `pmap` décrit en détail l'espace virtuel d'un processus défini, mais de manière encore trop grossière pour remonter aux allocations.

La dernière solution aurait été de recompiler les programmes à tester pour recompiler `malloc()` et affiliées, mais cela aurait été très fastidieux (compter déjà deux heures pour compiler `gcc`). À défaut, je me suis tourné vers une génération aléatoire à deux paramètres :

- La probabilité p qu'un processus réalise une allocation à chaque instant d'exécution ;
- Une taille moyenne μ de blocs utilisée pour définir une loi normale (σ, μ) d'écart-type élevé.

La durée de vie de chaque bloc est en moyenne proportionnelle à la taille du bloc, et de suit aussi une loi normale, implémentée *via* la transformation de Box-Muller [1].

Écriture du simulateur

Le logiciel que j'ai écrit simule un ordinateur paginé (avec 64 pages de 1024 octets) qui répond aux requêtes de plusieurs processus s'exécutant en parallèle. Chaque processus dispose d'un espace virtuel dans lequel un algorithme (unique pour toute l'exécution) réalise les opérations. Le simulateur en lui-même est en ligne de commande et génère un log de la simulation dans un « langage » facile à parser (voir la figure 1). Les sources commentée du simulateur (rédigées en C) sont jointes en annexe.

1. A note on the generation of normal random deviates, G.E.P. Box et Mervin E. Muller

J'ai indépendamment développé une interface graphique (framework Qt) pour analyser le compte-rendu, afficher graphiquement l'état de la mémoire (c'est en effet plus parlant), calculer des statistiques et construire des graphes d'évolution (figure 1). L'implémentation de cette interface, volumineuse et peu intéressante, n'a pas été jointe au rapport.

Sur la figure 1, la mémoire de chaque processus est représentée en ligne (les adresses extrêmes de la zone sont indiquées de chaque côté). Les carrés en fond représentent les pages mémoires réservées auprès du système, et les rectangles colorés indiquent la position les blocs de données.

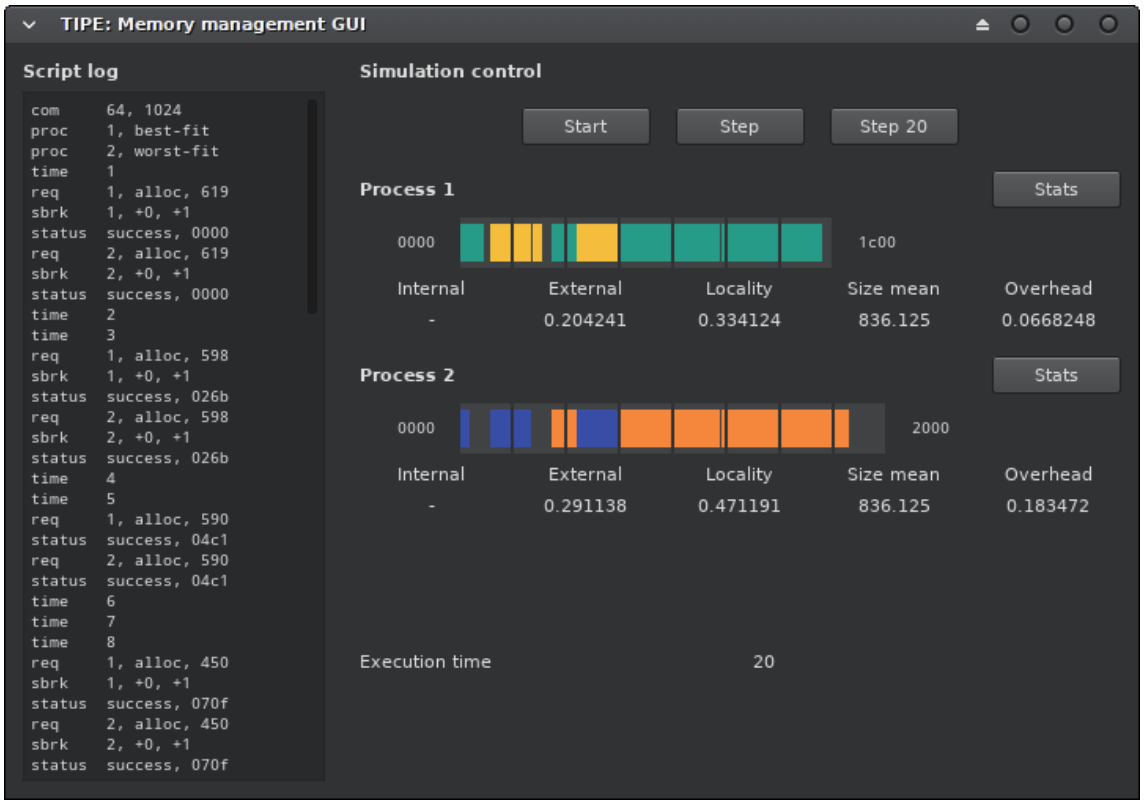

FIGURE 1 – Un best-fit (processus 1) et un worst-fit (processus 2) en concurrence sur le même programme

Modélisation du système

Chaque processus est évalué sur la seule donnée de la structure de son tas (*heap*). Celui-ci contient un certain nombre de pages, allouées au fur et à mesure de l'exécution, dans lesquels des blocs de mémoire (colorés) sont alloués et libérés.

— On choisit un processus à évaluer et on note n la taille de son tas (qui est un multiple de la taille d'une page mémoire).

On s'intéresse ensuite à des ensembles de *blocs*, chaque bloc correspondant à une zone de mémoire libre ou allouée, et susceptible de contenir des données.

— On note $\mathcal{L}$ l'ensemble des blocs libres du tas et $\mathcal{A}$ l'ensemble des blocs alloués.

Quelques opérateurs sont nécessaires pour travailler avec ces blocs. Ceux que j'ai utilisés sont :

— Si $b \in \mathcal{A}$, $|b|$ désigne la quantité de données stockée dans b ;

— Si b est un bloc, $\|b\|$ désigne sa taille en octets (égale à $|b|$ pour les blocs alloués, sauf en cas de fragmentation interne) ;

— Si $\mathcal{B}$ est un ensemble de blocs, $\operatorname{sum}\mathcal{B} = \sum_{b\in\mathcal{B}}\|b\|$ et $\operatorname{mean}\mathcal{B} = \frac{1}{n}\operatorname{sum}\mathcal{B}$.

Finalement, j'ai séparé les blocs libres en deux catégories qui décomptent les espaces dans lesquels on a en moyenne une probabilité faible, ou élevée, de pouvoir réaliser une allocation :

— $\mathcal{L}_< = \{b \in \mathcal{L}, \|b\| < \operatorname{mean}\mathcal{L}\}$ (interstices libres) ;

— $\mathcal{L}_> = \{b \in \mathcal{L}, \|b\| > \operatorname{mean}\mathcal{L}\}$ (espacements conséquents),

Critères d'évaluation du système

J'ai défini quatre critères pour évaluer la performance de mes algorithmes. Ce sont des grandeurs issues de l'expérience :

— Le taux de fragmentation externe quantifie la tendance de l'algorithme à laisser derrière lui des miettes (petits interstices de mémoire) trop petites pour être facilement utilisées ;
— Le taux de fragmentation interne représente la tendance à allouer plus de mémoire que ce qui était réellement nécessaire ;
— Le taux de localité indique la capacité à « serrer » les données dans la mémoire (ce qui est un bonus de performance, car cela évite les défauts de page et limite le va-et-vient) ;
— Le taux de « surcharge » quantifie la proportion de mémoire réservée par le processus auprès du système d'exploitation (comptée en pages) mais restée non allouée.

Critère	*Formule*
Taux de fragmentation externe	$f_e = \frac{\operatorname{sum} \mathcal{L}_{\leq}}{n}$
Taux de fragmentation interne	$f_i = \frac{1}{n} \sum_{b \in \mathcal{A}} (\|\|b\|\| - \|b\|)$
Taux de localité des données	$l = 1 - \frac{\operatorname{sum} \mathcal{L}_{>}}{n}$
Taux de surcharge (mémoire inutilisée)	$r = 1 - \frac{1}{n} \sum_{b \in \mathcal{A}} \|\|b\|\|$

L'interface graphique permet de consulter l'évolution de ces grandeurs au cours du temps. L'idée était de faire apparaître les spécificités de chaque algorithme par comparaison de ces graphes.

Algorithmes étudiés : les listes chaînées

J'ai réalisé des tests sur quatre algorithmes. Les trois premiers sont des variantes d'une technique consistant à représenter la mémoire comme une liste chaînée de blocs alternativement libres et occupés, triée par adresses croissantes (car en pratique les noeuds sont installés dans un *overhead* au début de chaque bloc). Pour allouer un bloc, la liste est parcourue jusqu'à trouver au moins une zone libre assez grande ; la libération est (en pratique, mais pas dans le simulateur) en temps constant si la liste est doublement chaînée.

Les algorithmes dont il est question comparent la taille des blocs libres de la liste avec la quantité de mémoire demandée pour décider quand réaliser l'allocation :

— L'algorithme du *first-fit* se contente du premier bloc qu'il trouve ;
— L'algorithme du *best-fit* sélectionne toujours le plus petit bloc parmi ceux qui conviennent ;
— L'algorithme du *worst-fit*, au contraire, sélectionne le plus grand bloc possible.

La libération d'un bloc dans une telle liste enclenche la fusion des blocs libres adjacents, lorsqu'il y en a, pour recombiner la mémoire.

Algorithmes étudiés : l'arbre binaire

Le quatrième algorithme dont il est question est nommé *buddy tree*. Sa représentation de la mémoire est celle d'un arbre binaire qui associe à chaque noeud une subdivision de taille 2^n de la mémoire. Un noeud entièrement libre, ou alloué pour stockage, n'a pas d'enfants ; sinon, sa zone mémoire est divisée en deux et chaque moitié est gérée par un de ses enfants. Une taille amotique est fixée (dans la simulation, 16 octets) pour éviter de subdiviser à l'infini.

L'algorithme d'allocation du *buddy tree* consiste à arrondir la taille de la zone demandée à la puissance de 2 supérieure et à chercher un noeud libre gérant une zone de cette taille. Si aucun n'existe, un autre noeud, choisi de taille minimale, est subdivisé pour créer la zone recherchée.

Lors d'une libération, les blocs libres adjacents ne sont pas systématiquement fusionnés : seuls les noeuds libres de même parent immédiat (qualifiés de *buddies*) sont recombinés. Le *buddy tree* conserve ainsi toujours les blocs alignés sur une adresse multiple de leur taille, et limite en pratique la fragmentation externe sur le long terme grâce à ce principe.

Résultats observés et discussion

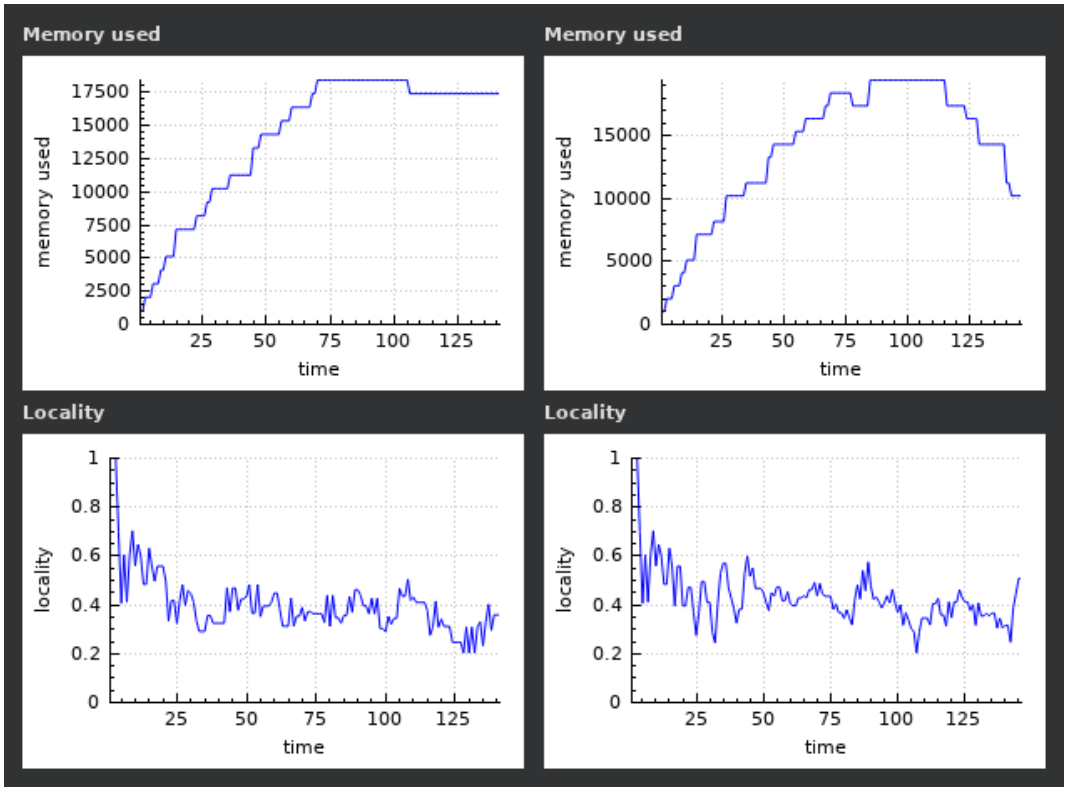

FIGURE 2 – Le best-fit (à gauche) et le worst-fit (à droite) de tout à l'heure, engagés dans un duel intense sur l'exécution du même processus

Après de nombreux tests, il apparaît que les simulations ne démarquent pas extrêmement bien les algorithmes, surtout les *fits* sur les listes. La génération aléatoire des schémas d'allocation/libération est sans doute l'approximation responsable de ce résultat : le modèle choisi, fort simple, présente trop de régularité. Un programme réel alloue par exemple au début de son exécution des blocs qu'il gardera probablement jusqu'à la fin. Ces blocs vont mettre en défaut certains algorithmes, mais une loi normale ne peut pas rendre compte de ce phénomène peu importe la largeur de l'écart-type que l'on choisit.

Malgré ces quelques difficultés, plusieurs résultats attendus formellement ont été observés, notamment :

— Le renouvellement régulier des pages utilisées par l'algorithme du *worst-fit* (roulement d'allocation/libération de pages) ;
— La compacité remarquable du *worst-fit* (le plus gros bloc le restant souvent après une requêtes, plusieurs allocations d'affilée sont effectuées dans la même zone) ;
— L'efficacité du *first-fit* quand les blocs ont une durée de vie faible (meilleures performances en vitesse).

À l'inverse, certaines prédictions n'ont pas été vérifiées :

— La fragmentation externe propre au *best-fit* (qui laisse derrière lui des restes de taille minimale dont difficile à employer) ; ce phénomène étant largement reconnu par ailleurs, ou la génération aléatoire ou la définition du taux de fragmentation externe est sans doute à revoir. Dans les simulations, ce taux a rarement dépassé celui du *worst-fit* ou la barre de 40 % ;
— Le taux de fragmentation interne du *buddy tree* s'est révélé faible (rarement supérieur à 20 %) ;
— Le taux de localité du *buddy tree* atteignait souvent 60 % alors que les autres algorithmes plafonnaient à 40 %, alors que la fragmentation interne laissait penser que la localité serait plutôt mauvaise.

Un algorithme composite

L'algorithme composite que je voulais évaluer consistait à séparer la mémoire des processus en trois segments indépendants, chacun servant à répondre à un type de requêtes pour maximiser l'efficacité de chaque algorithme. La décomposition était la suivante :

— Un segment géré par un *first-fit* pour tous les blocs de durée de vie faible (on se moque de la fragmentation grâce au roulement des données dans la mémoire, et c'est plus rapide) ;
— Une liste de type *worst-fit* pour les blocs larges, qui créent le maximum de fragmentation externe (afin d'éviter de laisser de grands espaces libres inutilisés) ;
— Un *buddy tree* pour les blocs dont la taille est une puissance de deux, et utilisé par défaut pour sa robustesse (il tient en effet bien dans le temps car sa structure alignée ne peut pas se déformer).

Cependant les schémas aléatoires n'ont pas permis de décrire efficacement des situations distinctes (lecteur multimédia, navigateur web, compilateur, etc). À cause de la régularité des allocations, tous les algorithmes se sont comportés de manière correcte en le composite n'a pas pu se démarquer réellement (il était attendu dans les situations où les algorithmes individuels ont du mal à tenir la charge).

Conclusion

Malgré l'utilisation de schémas d'allocation trop éloignés de la réalité car fondés sur une génération aléatoire trop régulière, les simulations réalisées ont révélé des variations de comportement pour trois *fits* finalement assez proches, et ont démarqué significativement le *buddy tree*. Cela justifie l'approche composite qui classifie les allocations dans différentes catégories pour gérer chaque bloc avec l'algorithme le plus adapté, à défaut de permettre une évaluation précise du modèle suggéré.