

Compilateurs : Sujets Choisis

Matthieu Amiguet

2008 – 2009



1 Édition des liens

2 Gestion de la mémoire

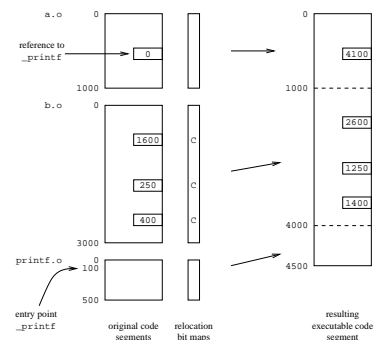
Édition des liens

3

- La plupart des langages de programmation autorisent la compilation séparée de fragments de programmes
- Ces fragments une fois compilés doivent être reliés entre eux pour former un exécutable
- Ceci pose deux problèmes
 - On ne sait pas au moment de la compilation quel sera l'emplacement du code en mémoire
 - On fait parfois référence à des adresses situées en dehors du fragment concerné
- Le fragment compilé devra donc contenir des informations de translation, et une table des symboles "importés" et "exportés".

Exemple

4



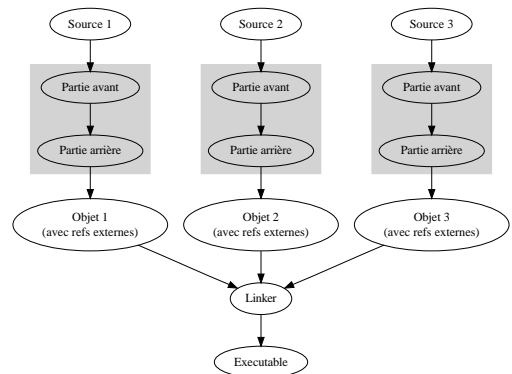
Liaison dynamique

5

- La liaison peut se faire juste après la compilation : "liaison statique"
- L'utilisation croissante de bibliothèques de sous-programmes a amené un nouveau type de liaison : la liaison dynamique
 - ↳ Permet d'économiser des ressources en ne chargeant en mémoire qu'une seule copie de la bibliothèque partagée
- La liaison dynamique peut se faire au moment du chargement. . .
- . . . ou "à la demande" en cours d'exécution ("liaison paresseuse")
 - ↳ Économise potentiellement beaucoup de ressources, au prix d'un *overhead* un peu plus grand. . .

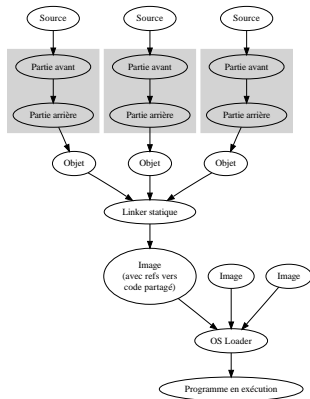
Version statique

6



Version statique + dynamique

7



1 Édition des liens

- 2 Gestion de la mémoire
 - Structure de la mémoire
 - Allocation
 - Désallocation

Allocation de mémoire

9

- La plupart des langages de programmation permettent de traiter des données d'une taille non connue au moment de la compilation
- Ceci nécessite donc une allocation dynamique de la mémoire en cours d'exécution
- Pour éviter d'être à court, il faudra aussi désallouer cette mémoire
 - explicitement, ou
 - implicitement.

La structure de la mémoire

Le segment de code

10

La plupart des systèmes d'exploitation allouent au moins les trois segments de mémoire suivants :

Le segment de code contient le code du programme.

- pointé par le pointeur d'instructions
- on ne s'y réfère presque jamais depuis le code du programme.

La structure de la mémoire

Le segment de pile (*Stack*)

11

Le segment de pile contient la pile d'exécution

- pointée par un ou plusieurs pointeurs de pile...
- généralement manipulés automatiquement par les instructions machine
- Contient (notamment) les données dont la taille est connue à la compilation (sauf celles qui sont attribuées dans les registres !)

La structure de la mémoire

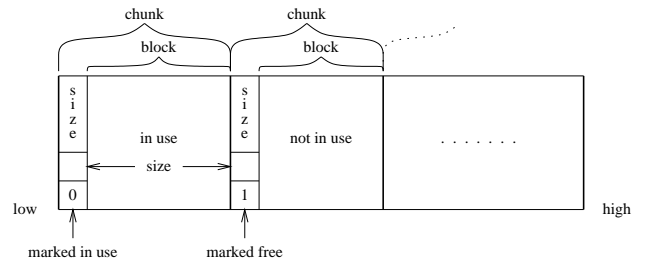
Le segment de données (*Heap, Free Store*)

12

Le segment de données est une zone mémoire à la disposition du programme pour ranger ses données

- utilisation entièrement gérée par le processus
- aussi appelé *tas* lorsqu'on veut mettre l'accent sur la gestion de la mémoire
- il faut donc gérer l'espace libre, l'allocation, la désallocation d'espace, etc.
- En théorie, trois possibilités :
 - tout gérer à la main (Assembleur...)
 - gérer les (dés)allocations à la main, mais pas la "géographie" (C[++]
 - ne rien gérer explicitement (Java, Python, ...)

- Une demande d'allocation de N octets de mémoire fourni à l'utilisateur un pointeur sur le premier octet d'un bloc libre de N octets
- On doit donc garder une information sur l'emplacement et la longueur des blocs, et s'ils sont utilisés
- On garde généralement cette information dans un *en-tête de bloc*
- Pour la désallocation, il suffit donc de modifier cet en-tête en conséquence.



- Pour allouer un bloc B de taille N , on parcourt le tas à la recherche d'un bloc disponible et assez grand
 - Si on le trouve, on le sépare en deux (s'il est assez grand) et on en alloue une partie à B
 - Sinon, on peut essayer de fusionner deux blocs libres adjacents
 - Si ça ne marche pas, on peut avoir un problème de fractionnement de la mémoire. Il faudra alors éventuellement la compacter
 - Si ça ne marche toujours pas, on va alerter le système d'exploitation (augmentation de la mémoire ou erreur).

- La solution ci-dessus est bien entendu très peu efficace
- On peut ranger les blocs disponibles dans une liste chaînée, ce qui évite un parcours complet de la mémoire
- On peut trier cette liste chaînée dans l'ordre croissant de la grandeur des blocs
- On peut fusionner au vol les blocs libres adjacents (test à chaque libération)
- ...

- La désallocation manuelle de la mémoire est difficile à maîtriser et source de nombreux bugs
- Un oubli de désallouer peut amener à une fuite de mémoire (*memory leak*)
- Une désallocation précoce peut produire un pointeur fantôme (*dangling pointer*)
 - particulièrement difficile à détecter et corriger...
- On peut donc se demander si cette désallocation peut être automatisée.

- La première idée donne la technique du *comptage des références*
- La deuxième engendre deux techniques
 - *marquage et balayage*
 - *copie entre deux espaces*

- La désallocation implicite est aussi appelée récupération de mémoire
- Le récupérateur de mémoire est parfois appelé *ramasse-miettes (garbage collector)*
- L'idéal serait de récupérer tous les blocs qui ne seront plus utilisés par le programme
- Cet idéal est inatteignable en général. On peut l'approximer de deux manières, en trouvant
 - l'ensemble des blocs sur lesquels ne pointe aucun pointeur
 - l'ensemble des blocs non accessibles depuis des données non allouées dans le tas.

Il existe trois sortes d'algorithmes de récupération de mémoire

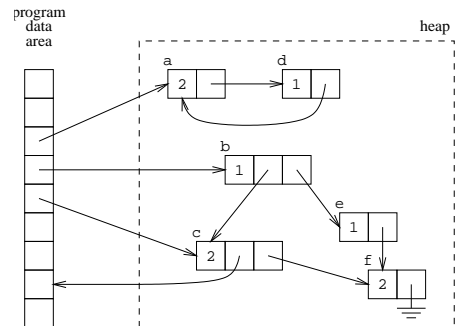
Une passe Quand le récupérateur est lancé, il s'exécute en une fois avec un contrôle sur l'ensemble des blocs de mémoire. L'exécution du programme est suspendue

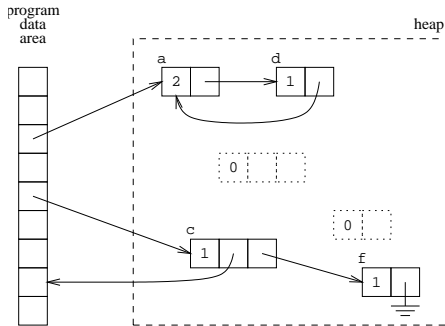
Au vol (ou incrémental) Les actions de récupération de la mémoire sont effectuées à chaque (dés)allocation

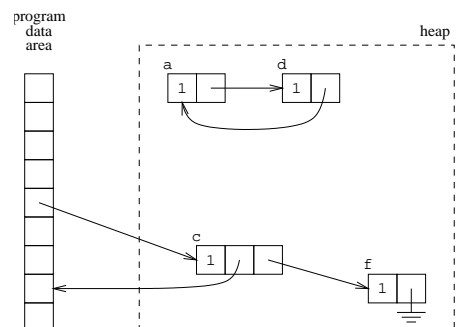
- Plus difficile à écrire
- Pas d'effet "stop the world"

En parallèle Le récupérateur de mémoire tourne sur un second processeur.

- Algorithme de récupération au vol
- L'idée est de garder en permanence dans l'en-tête des blocs mémoire un compte du nombre de pointeurs s'y référant
- Lorsque ce nombre tombe à zéro, le bloc n'est plus accessible et peut donc être libéré
- Nécessite un suivi de toutes les opérations sur les pointeurs.

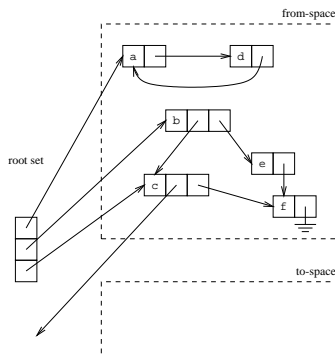


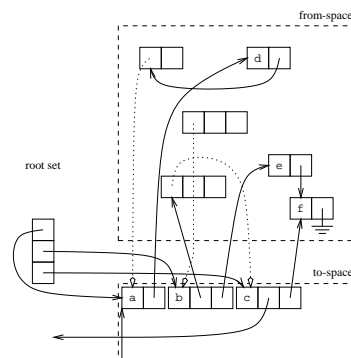


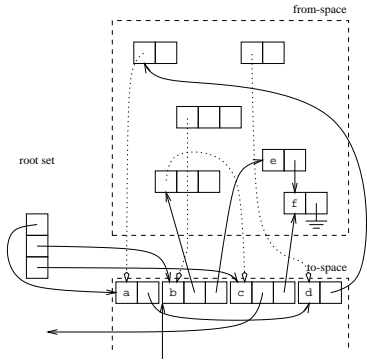


- Algorithme de récupération en une passe (ou en parallèle)
- On ajoute à l'en-tête de bloc un bit indiquant si le bloc est accessible
- Lors du déclenchement de l'algorithme, on commence par marquer tous les blocs du tas comme inaccessibles
- On parcourt ensuite tous les "chemins" de pointeurs ayant leur origine hors du tas, en marquant comme accessibles tous les blocs rencontrés
- Il suffit ensuite de "balayer" le tas pour désallouer tous les blocs encore marqués comme inaccessibles.

- Optimisation en temps de l'algorithme précédent (au détriment de la mémoire !)
- Le tas est divisé en deux moitiés : l'espace de départ et l'espace d'arrivée
- On alloue les blocs dans l'espace de départ jusqu'à ce qu'il soit presque plein
- On copie alors tous les blocs accessibles dans l'espace d'arrivée et on inverse les rôles des deux moitiés
- Ne pas oublier de mettre à jour les pointeurs !







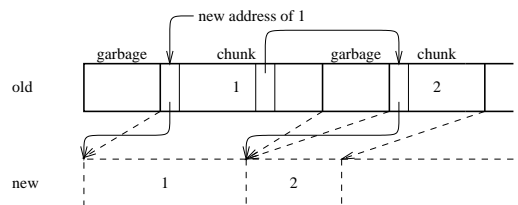
Attention !

Pour tous ces algorithmes, il est important de ne pas garder de pointeur sur des structures inutilisées !

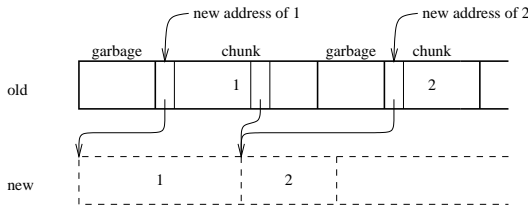
- Cela devient l'équivalent d'un memory leak par oubli d'un `free` en C. . .
- Donc gestion *implicite* de la mémoire ne veut pas dire *aucune* gestion de la mémoire. . .

- La récupération par copie laisse derrière elle un tas "sans trou"
- Les deux autres algorithmes nécessitent un *compactage* pour éviter une trop grande fragmentation
- Celui-ci peut se faire en trois passes simples
 - Calcul d'adresses
 - Mise à jour des pointeurs
 - Déplacement des blocs.

- On calcule les nouvelles adresses en supprimant tous les blocs disponibles



- On met à jour tous les pointeurs vers des blocs déplacés



- Il ne reste plus qu'à déplacer les blocs

