Programmation – TD10 : Djinn gueule “bêle !”

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

15 décembre 2004

Pour ce dernier TD de l’année civile, vous pouvez au choix finir le TD précédent (la partie `malloc`) ou vous essayer à l’un des trois problèmes suivants.

1 Le père Noël

1.1 Position du problème

Il a tant été dit sur le père Noël qu’il convient avant tout de fixer certains points :

1. le père Noël existe ;
2. le père Noël distribue des cadeaux ;
3. seuls ceux qui sont gentils peuvent avoir des cadeaux[1].

Contrairement à ce qui a été raconté, la hotte du père Noël n’est pas d’une contenance infinie. En fait, le père Noël doit, pendant la nuit de Noël, effectuer d’incessants aller-retour entre l’Australie[2] et le reste du globe pour faire sa livraison. Il y a en fait un grand entrepôt quelque part en Australie où il vient se réapprovisionner. Cet entrepôt est, parallèlement aux visites du père Noël, rempli de cadeaux par des petits gremlins. On s’intéresse ici à comment remplir cet entrepôt.

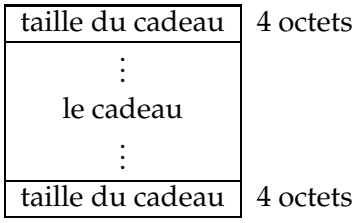
1.2 Résolution

On représentera l’entrepôt en C par la structure suivante :

```
void *entrepôt = malloc(1000000);
```

On représente un cadeau par un emplacement consécutif de l’entrepôt. Comme il faut que le père Noël puisse se déplacer autour du cadeau (afin de le mettre sur son buggy), il y a beaucoup de place libre entre les cadeaux dans laquelle on peut stocker de l’information.

Une place occupée par un cadeau sera représentée de la manière suivante :



[1] Un rapide raisonnement montre alors qu’accompagner ses élèves au ski n’est pas un cadeau.

[2] Le père Noël n’habite pas en Laponie, contrairement aux croyances populaires. Il fait bien plus chaud en Australie, et le père Noël est frileux.

Une place libre, qu'on peut occuper par un cadeau ou plusieurs cadeaux, sera définie par :

taille disponible	4 octets
emplacement libre précédent	4 octets
emplacement libre suivant	4 octets
⋮ n'importe quoi ⋮	
taille disponible	4 octets

L'information sur la taille des emplacements est ainsi redondante, puisqu'on l'écrit au début et à la fin. Ceci permet en fait de parcourir l'emplacement rapidement d'un sens à l'autre. On peut aussi s'en servir pour “fusionner” deux places libres.

Les deux emplacements indiqués dans les places libres permettent en fait, d'un point de vue programmation et donc plus terre à terre, de maintenir en fait une liste (doublement) chaînée des places libres. Lorsque les gremlins viendront apporter leurs cadeaux, il suffira donc de parcourir cette liste afin de trouver rapidement une place libre.

Dans la vraie vie, la gérante de l'entrepôt[3] peut repérer assez facilement si une place est libre. Ce n'est pas le cas dans notre modèle. On va en fait utiliser une petite astuce. Comme la taille des cadeaux est un multiple de 4 [4], on peut coder de l'information supplémentaire de la façon suivante : si un emplacement est pris, on ajoutera artificiellement 1 à sa taille. De sorte qu'un emplacement de taille $4n + 1$ est en fait un emplacement occupé de taille $4n$, et qu'un emplacement de taille $4n$ est en fait un emplacement libre de taille $4n$.

1.3 À vous maintenant

1. Écrivez une première version des fonctions `void *nouvelle_place(size_t taille)` et `void libère(void *place)`. `nouvelle_place` recherche un emplacement libre de taille suffisante pour acceuillir le cadeau, le découpe s'il est trop gros ; `libère` se contente de libérer l'emplacement (et donc de le placer dans la liste des emplacements libres).

Pour tester votre code, vos TDmen adorés ont écrit un petit programme qui se charge de fournir un joli test.

2. Améliorez vos fonctions de sorte que `libère` fusionne maintenant des emplacements libres adjacents.

3. Faites en sorte de maintenir la liste des emplacements libres triée par taille croissante. Ceci vous permettra, au moment où l'on cherche un emplacement libre, de prendre celui qui convient le mieux.

4. Améliorez encore et toujours.

[3] La très jolie Marie Noël, mariée à l'ignoble Jean Balthazar.

[4] De sorte que la hotte du père Noël, qui fait 64 en unité de longueur, puisse contenir 16 cadeaux. Le père Noël pensait que ca lui permettrait de résoudre plus rapidement ses problèmes de Bin-Packing, mais il n'en est rien, le problème est toujours NP-complet.

2 La crise de foie

Noël, les fêtes tout ça, c'est avant tout l'occasion de manger et de boire plus que de raison : champagne, foie gras, champagne, confit, champagne, saumon, champagne, champagne, chapon, champ', champ', champ', **hips**, champ', bûche, calva, ... Dans ces conditions, dur d'échapper à la crise de foie si l'on n'a pas pris ses précautions. Heureusement pour vous, vos TDmen favoris ont pensé à tout, et vous proposent ici même une méthode révolutionnaire pour partitionner, optimiser et gérer votre foie avec prudence, histoire de tenir à douze grammes fixes jusqu'à l'Épiphanie[5].

2.1 Formalisation

On représentera votre foie[6] en C par la structure suivante :

```
void *foie = malloc(1000000);
```

À chaque plat que vous engloutissez et à chaque verre que vous descendez, vous allez devoir réserver une partie de votre foie. Selon la quantité ingérée, la taille de ce bout de foie pourra varier. Après un temps dépendant de la nature de ce que vous venez de manger / boire, le foie aura enfin fini son affaire, et les cellules seront libérées et pourront à nouveau resservir.

Ainsi, on représente votre foie par une suite de cellules consécutives. Comme il vaut mieux isoler les zones actives, il y a de la place libre entre ces zones dans laquelle on peut stocker de l'information.

Un bout de foie occupé à assimiler votre orgie de la veille sera donc représenté de la manière suivante :

nombre de cellules	4 octets
⋮ les cellules ⋮	
nombre de cellules	4 octets

Des cellules libres qui pourront resservir, seront définies par :

nombre de cellules disponibles	4 octets
zone libre précédente	4 octets
zone libre suivante	4 octets
⋮ n'importe quoi ⋮	
nombre de cellules disponibles	4 octets

L'information sur le nombre de cellules est ainsi redondante, puisqu'on l'écrit au début et à la fin. Ceci permet en fait de parcourir le foie rapidement d'un sens à l'autre. On peut aussi s'en servir pour “fusionner” deux zones libres.

[5] Sous le baby.

[6] Un foie moyen contient un bon million de cellules. Pour celles-eux présent-e-s à la soirée DMI, on n'en comptera qu'une petite dizaine de milliers. Z'aviez qu'à pas vous entamer comme ça.

Les deux zones indiquées dans les cellules libres permettent en fait, d'un point de vue programmation et donc plus terre à terre, de maintenir en fait une liste (doublement) chaînée des bouts de foie libres. Lorsque vous attaquez le plat / verre suivant, il suffira donc de parcourir cette liste afin de trouver rapidement des cellules libres pour digérer tout ça.

Cependant, ce n'est pas si facile de repérer si un bout de foie ne travaille pas. On va en fait utiliser une petite astuce. On va supposer que le nombre de cellules est toujours un multiple de 4. Ainsi, on peut coder de l'information supplémentaire de la façon suivante : si une zone est prise, on ajoutera artificiellement 1 à sa taille. De sorte qu'un bout de foie de taille $4n + 1$ est en fait $4n$ cellules occupées, et qu'un bout de foie de taille $4n$ est en fait $4n$ cellules libres.

2.2 À vous maintenant

1. Écrivez une première version des fonctions `void *nouvelles_cellules(size_t nombre)` et `void digère(void *cellules)`. `nouvelles_cellules` recherche un bout de foie libre de taille suffisante pour digérer le plat, le découpe s'il est trop gros ; `digère` se contente de libérer ces cellules (et donc de le placer dans la liste des cellules libres).

Pour tester votre code, vos TDmen adorés ont écrit un petit programme qui se charge de fournir un joli test.

2. Améliorez vos fonctions de sorte que `digère` fusionne maintenant des zones de foie libres adjacentes.
3. Faites en sorte de maintenir la liste des zones libres triée par taille croissante. Ceci vous permettra, au moment où l'on cherche des cellules libres, de prendre celles qui conviennent le mieux.
4. Améliorez encore et toujours.

3 Gérer un parking

Comme vous prévoyez un gros repas pour le nouvel an, et que votre chef vous a pris pour un imbécile lorsque vous lui avez demandé une prime de $100.000 pourtant dûment méritée suite à votre débuggage de son code Excel™, vous voilà dans l'obligation de faire des petits boulots afin de pouvoir payer la dinde.

Votre oncle vous a donc confié la gestion du parking du célèbre hôtel Podetex&Podefex[7]. Votre rôle est simple : à chaque fois que le groom de service vous amène une voiture, vous devez trouver un emplacement pour la garer.

3.1 Résolution

On représentera le parking en C par la structure suivante :

```
void *parking = malloc(1000000);
```

On représente une voiture par une suite d'éléments consécutifs du parking. Comme il faut que le groom puisse se déplacer autour des voitures, ne serait-ce que pour en descendre, il y a de la place libre entre les voitures dans laquelle on peut stocker de l'information.

Une place occupée par une voiture sera représentée de la manière suivante :

taille de la voiture	4 octets
⋮ la voiture ⋮	
taille de la voiture	4 octets

Une place libre, qui pourra être occupée par une ou plusieurs voitures, sera définie par :

taille disponible	4 octets
emplacement libre précédent	4 octets
emplacement libre suivant	4 octets
⋮ n'importe quoi ⋮	
taille disponible	4 octets

L'information sur la taille des emplacements est ainsi redondante, puisqu'on l'écrit au début et à la fin. Ceci permet en fait de parcourir l'emplacement rapidement d'un sens à l'autre. On peut aussi s'en servir pour “fusionner” deux places libres.

Les deux emplacements indiqués dans les places libres permettent en fait, d'un point de vue programmation et donc plus terre à terre, de maintenir en fait une liste (doublement) chaînée des places libres. Lorsque le groom vient avec une nouvelle voiture, il suffira donc de parcourir cette liste afin de trouver rapidement une place libre.

[7] À prononcer comme une célèbre fable de la Fontaine.

Dans la vraie vie, le gérant du parking[8] peut repérer assez facilement si une place est libre. Ce n'est pas le cas dans notre modèle. On va en fait utiliser une petite astuce. On va supposer que la taille des voitures est un multiple de 4. Ainsi, on peut coder de l'information supplémentaire de la façon suivante : si un emplacement est pris, on ajoutera artificiellement 1 à sa taille. De sorte qu'un emplacement de taille $4n + 1$ est en fait un emplacement occupé de taille $4n$, et qu'un emplacement de taille $4n$ est en fait un emplacement libre de taille $4n$.

3.2 À vous maintenant

1. Écrivez une première version des fonctions `void *nouvelle_place(size_t taille)` et `void libère(void *place)`. `nouvelle_place` recherche un emplacement libre de taille suffisante pour acceuillir une voiture, le découpe s'il est trop gros ; `libère` se contente de libérer l'emplacement (et donc de le placer dans la liste des emplacements libres).

Pour tester votre code, vos TDmen adorés ont écrit un petit programme qui se charge de fournir un joli test.

2. Améliorez vos fonctions de sorte que `libère` fusionne maintenant des emplacements libres adjacents.

3. Faites en sorte de maintenir la liste des emplacements libres triée par taille croissante. Ceci vous permettra, au moment où l'on cherche un emplacement libre, de prendre celui qui convient le mieux.

4. Améliorez encore et toujours.

[8] C'est-à-dire vous.

4 *Bonus* : Recette de la dinde au whisky

1. Acheter une dinde d’environs 5 kg pour 6 personnes et une bouteille de whisky, du sel, du poivre, de l’huile d’olive, des bardes de lard.
2. La barder de lard, la ficeler, la saler, la poivrer et ajouter un filet d’huile d’olive.
3. Faire préchauffer le four (thermostat 7) pendant dix minutes.
4. Se verser un verre de whisky pendant ce temps-là.
5. Mettre la dinde au four dans un plat à cuisson.
6. Se verser ensuite 2 verres de whisky et les boire.
7. Mettre le therpostat à 8 après 20 binutes pour la saisir.
8. Se bercer 3 berres de whisky.
9. Apres une debi-beurre, fourrer l’ouvrir et surveiller la buisson de la pinde.
10. Brendre la vouteille de biscuit et s’enfiler une bonne rasade derrière la bravate - non - la cravate.
11. Apres une demi-heure de blus, tituber jusqu’au bour. Oubrir la putain de borte du bour et reburner - non - revourner - non - recourner - non - enfin, mettre la guinde dans l’autre sens.
12. Se prûler la main avec la putain de borte du bour en la refermant - bordel de merde.
13. Essayer de s’asseoir sur une putain de chaise et se reverdir 5 ou 6 whisky de verres ou le gontraire, je sais blus.
14. Buire - non - luire - non - cuire - non - ah ben si - cuire la bringue bandant 4 heures.
15. Et hop, 5 berres de plus. Ça fait du bien par oû que ça passe.
16. R’tirer le four de la dinde.
17. Se rebercer une bonne goulée de whisky.
18. Essayer de sortir le bour de la saloperie de pinde de nouveau parce que ça a raté la bremière fois.
19. Rabasser la dinde qui est tombée bar terre. L’ettuyer avec une saleté de chiffon et la foutre sur un blat, ou sur un clat, ou sur une assiette. Enfin, on s’en fout...
20. Se péter la gueule à cause du gras sur le barrelage, ou le carrelage, de la buisine et essayer de se relever.
21. Décider que l’on est aussi bien par terre et binir la mouteille de rhisky.
22. Ramper jusqu’au lit, dorbir toute la nuit.
23. Manger la dinde froide avec une bonne mayonnaise, le lendemain matin et nettoyer le bordel que tu as mis dans la cuisine la veille, pendant le reste de la journée.

NOYEUX JOEL

&

BONNE ANNEE !

C’est fou comme on peut aussi faire des trucs super-moches avec LaTeX !