TP 3

Principes des Systèmes d'Exploitation

IUT Montpellier

Introduction/ Objectifs et rappels

Objectifs de la séance

- Comprendre l'organisation de la mémoire pour un processus
- Maitriser l'allocation dynamique de la mémoire sur le tas

Rappels

- La mémoire accessible par un processus est organisée en plusieurs 'segments'
  - data
  - bss
  - tas
  - pile
  - code
- Pour cette séance nous allons nous intéresser au segment 'tas' pour allouer/libérer de l'espace mémoire durant l'exécution du programme

Partie I/ Exo 1

Objectif

- Introduction à l'analyse des performances
- Nous avons deux programmes: dynamicmemory.c et staticmemroy.c
- Notre objectif est d'étudier la différence de comportement entre ces deux programmes
- Pour cela nous allons faire varier un paramètre qui représente la taille en octet que va occuper un tableau
- Ce tableau sera déclaré comme une variable globale initialisée dans le programme 'staticmemory.c'
- Il sera déclaré comme une variable locale initialisée avec une allocation de mémoire dynamique dans le cas de 'dynamicmemory.c'

staticmemory.c

```
clock_t start, end;
double cpu_time_used;
int tab[DATASIZE] = { 'X', };
int main(int argc, char** argv)
{
    start = clock();
    int i ;
    for(i = 0 ; i < DATASIZE; i++)
    {
        usleep(5);
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("%f",cpu_time_used);
    return 0;
}
```

dynamicmemory.c

```
clock_t start, end;
double cpu_time_used;
int main(int argc, char** argv)
{
    start = clock();
    int* tab = malloc(DATASIZE);
    int i ;
    for(i = 0 ; i < DATASIZE; i++)
    {
        usleep(5);
    }
    free(tab);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("%f",cpu_time_used);
    return 0;
}
```

Les données

- Les données de notre expérimentation sont dans le fichier: 'experimentation/data.csv'
- Nous avons fait varier la valeur du paramètre DATASIZE et observer la taille de l'exécutable ainsi que le temps d'exécution dans deux cas:
  - Avec une variable globale
  - Avec une allocation dynamique de mémoire
- Il est à noter que pour chaque valeur de DATASIZE nous avons effectué 10 observations indépendantes

Exercice

- A faire
  - Pour chaque valeur de DATASIZE, calculer la moyenne et l'écart type
  - Réaliser des courbes pour interpréter facilement les données
  - Analyser les courbes
  - Formuler vos conclusions sur la difference entre staticmemory.c et dynamicmemory.c

Partie II: Projet Game Engine/ Introduction

Présentation du projet

- Vous allez implémenter vos propres fonctions d'allocation dynamique de la mémoire malloc/free.
- Ces fonctions seront des versions simplifiées mais fonctionnelles de malloc/free.
- Comme malloc/free nous allons nous baser sur les fonctions brk/sbrk pour gérer le pointeur program break et ainsi modifier la taille du tas (heap)

Partie II: Projet Game Engine/ Etape 1

Objectif

- Pour notre première implémentation nous proposons de réaliser une implémentation naïve: nous allons allouer simplement un bloc mémoire en augmentant la taille du tas.

ALIGN

- Attention la taille réelle du bloc alloué doit être alignée c'est à dire elle doit commencer à une adresse multiple d'une constante donnée (puissance de 2).
- Pour vous aider à calculer une adresse alignée vous pouvez utiliser la macro ALIGN définie dans 'memalloc.h'

Méta-données du bloc mémoire

- Pour gérer les blocs mémoires de l'utilisateur, il nous faut sauvegarder certaines données.
- Par exemple, la taille du bloc mémoire de l'utilisateur est une information importante à sauvegarder.
- Ces données sur les données sont appelées des *méta-données*.

Méta-données du bloc mémoire

- Un bloc mémoire réel va toujours être composé d'une partie de méta-données rangées dans un header et d'une partie pour ranger les données utiles (payload)

entete

- Voici une première structure C qui permet de réaliser ce schéma:
  - ```
    typedef struct bloc_entete
    {
     size_t taille; //taille du bloc utilisateur
     unsigned short libre : 1 ; //drapeau de 1 bit qui indic
    } bloc_entete ;
    ```
- Pour avoir la taille que prend notre entête, nous allons définir la macro ENTETE_SIZE:
  - ```
    #define ENTETE_SIZE (ALIGN(sizeof(bloc_entete)))
    ```

A faire: memalloc.0.c

- Implémentez les fonctions manquantes du fichier 'memalloc.0.c'
  - ```
    void blocinfo0(void* ptr);
    void* myalloc0(size_t t);
    void myfree0(void* ptr);
    ```
- Tester votre implémentation en intégrant ces fonctions au moteur de jeu
    ```

Partie II: Projet Game Engine/ Etape 2: : Recyclage des blocs

Objectif

- Notre solution alloue bien de la mémoire dynamiquement.
- Cependant, pour l'instant pour libérer de la mémoire nous avons simplement mis le flag à 1.
- Les blocs libérés ne sont jamais recyclés pour les prochaines demandes d'allocation.
- C'est l'objectif de cette étape: avant d'augmenter le programme break nous allons vérifier si un bloc libre ne peut pas être réutilisé.

Stratégie

- Pour l'instant nous allons adopter une stratégie de recyclage simple:
  - Nous allons allouer le premier bloc qui convient (first fit)
  - Si un bloc convient il est entièrement réutilisé (sans split)
- Afin de parcourir tous les blocs alloués, il nous faut sauvegarder l'adresse du premier bloc.
- Nous pouvons ensuite parcourir l'ensemble des blocs connaissant la taille de chaque bloc. Évidemment, nous ne devons pas dépasser le program break courant (obtenu par l'astuce: sbrk(0))

A faire: memalloc.1.c

- Implémentez les fonctions manquantes du fichier 'memalloc.1.c'
  - `void blocinfo1(void* ptr);`
    `void* myalloc1(size_t t);`
    `void myfree1(void* ptr);`
- Testez votre implémentation en intégrant ces fonctions au moteur de jeu

Partie II: Projet Game Engine/ Étape 3: Recyclage des blocs avec une liste

Objectif

- L'objectif de cette étape est d'améliorer les performances de malloc/free. Pour cela, nous proposons de gérer uniquement les blocs libres avec une liste. Cette liste sera parcourue pour trouver un bloc à recycler.

Spécifications

- La première modification donc à faire est au niveau de l'entête des blocs:

```
typedef struct bloc_entete
{
  size_t taille; //taille du bloc utilisateur
  struct bloc_entete* suivant_ptr ;    //bloc suivant
  struct bloc_entete* precedent_ptr ;    //bloc preceder
} bloc_entete ;
```

Nous n'avons plus besoin du flag qui indique si le bloc est libre et nous rajoutons deux pointeurs pour gérer une liste doublement chainée.

Stratégie

- Nous allons conserver la même stratégie de recyclage i.e:
  - allocation du premier bloc qui convient (first fit)
  - si un bloc convient, alors il est entièrement réutilisé (pas de split)

A faire

- Modifiez la fonction myalloc dans 'memalloc.2.c' pour:
  - parcourir la liste des blocs libres pour rechercher un bloc à recycler
  - si un bloc à recycler est trouvé, alors le retirer de la liste des blocs libres
  - si aucun bloc à recycler n'est disponible, alors proposer un nouveau bloc.
- Modifier la fonction myfree pour ajouter le bloc libéré dans la liste des blocs libres.