Pràctica 4: **Gestió de memòria dinàmica per a processos**

Maig 2018

Índex

1 Introducció: la crida a sistema `sbrk`

L'objectiu d'aquesta pràctica és entendre com funciona la gestió de memòria dinàmica en un procés a través de la implementació de les funcions de la llibreria estàndard `malloc` i `free` (no són crides a sistema), funcions que s'utilitzen habitualment en C per gestionar la memòria dinàmica.

Concretament repassarem els conceptes de:

- Apuntadors i llistes encadenades en C
- Reservar memòria de forma dinàmica fent servir malloc.

La signatura de la funció `malloc` (memory allocation) de C és la següent: `void *malloc(size_t size)`. Com a paràmetre d'entrada rep un nombre el bytes a reservar i retorna un apuntador al bloc de dades que s'ha reservat.

Una forma d'implementar el `malloc` és fent servir la crida a sistema `sbrk`, veure aquest enllaç: http://man7.org/linux/man-pages/man2/sbrk.2.html. Amb aquesta crida podem manipular l'espai de *heap* del procés (el *heap* gestiona la reserva de memòria dinàmica en temps d'execució)[1].

La funció `sbrk` es pot interpretar intuïtivament com una funció que permet augmentar o disminuir la quantitat d'aigua (i.e. memòria dinàmica) associada al un pantà (i.e. procès). La crida `srbk(0)` retorna el nivell de l'aigua actual del pantà (i.e. un apuntador al nivell actual del *heap*). Si hi especifiquem qualsevol altre quantitat com a paràmetre podem augmentar o disminuir el nivell de l'aigua del pantà, i.e. el *heap* s'incrementa o disminueix en aquest valor i la funció retorna un apuntador al valor antic abans de fer la crida.

Així, per exemple, la crida `sbrk(1000)` augmenta en 1000 bytes el heap i retorna un apuntador a l'inici d'aquests 1000 bytes de forma que es puguin fer servir els 1000 bytes per l'aplicació. Observar, en canvi, que si es fa la crida `sbrk(-1000)` es disminueix en 1000 bytes l'espai de memòria associat a la *heap*, el valor retornat és un apuntador al nivell de heap abans de fer la crida. La funció `sbrk(size)` retorna un -1 en cas que no s'hagi pogut realitzar l'operació desitjada.

2 Una versió "dummy" de `malloc` i `free`

Es proposa a continuació una implementació ben senzilla de les funcions `malloc` i `free`. El fitxer associat es diu `malloc_dummy.c`

[1] La funció `sbrk` no és la única funció que permet obtenir memòria dinàmica: també existeix la crida a sistema `mmap`. La funció `malloc`, però, utilitza la funció `sbrk`.

```c
#include <assert.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void *malloc(size_t mida) {
    void *p = sbrk(0);

    printf("Soc a la crida de malloc\n");

    if (mida <= 0)
        return NULL;

    if (sbrk(mida) == (void*) -1)
        return NULL; // sbrk failed.

    return p;
}

void free(void *p)
{
    printf("Soc a la crida de free\n");
}
```

Observar la implementació d'aquest `malloc`. Aquesta implementació del `malloc` té l'inconvenient que no podem fer un `free` de la memòria ocupada un cop no la necessitem atès que la funció `sbrk` només permet augmentar o disminuir el nivell del `heap`, però la funció `sbrk` no permet alliberar "un tros" del mig de la `heap`. Això passa sovint en una aplicació atès que anirem reservant i alliberant memòria dinàmica.

Amb la implementació del fitxer `malloc_dummy.c` la memòria s'acabaria omplint ràpidament amb aplicacions com el `firefox` atès que només anem augmentant el nivell de heap (l'aigua del pantà), però podem provar si el codi funciona amb aplicacions senzilles. Aquí teniu un petit codi en C que provarem amb la implementació del `malloc` que hem fet, codi `exemple.c`

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int i;
    int *p;

    p = malloc(10 * sizeof(int));

    for(i = 0; i < 10; i++)
        p[i] = i;

    for(i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    free(p);

    return 0;
}
```

Aquesta funció crida a la funció `malloc`. L'objectiu és generar un executable de forma que es

faci servir la funció `malloc` que acabem de definir en comptes de la funció `malloc` de la llibreria estàndard. Per això cal executar les següents instruccions a un mateix terminal. *Atenció! Les següents instruccions funcionen a Linux però no pas a Mac.*

1. Generem l'executable associat al fitxer `exemple.c`

   ```
   $ gcc exemple.c -o exemple
   ```

2. Generem una llibreria dinàmica associada al fitxer `malloc_dummy.c`

   ```
   $ gcc -O -shared -fPIC malloc_dummy.c -o malloc_dummy.so
   ```

3. Indiquem, a través d'una variable d'entorn, que cal carregar aquesta llibreria dinàmica abans que qualsevol altre llibreria

   ```
   export LD_PRELOAD=/$PWD/malloc_dummy.so
   ```

   Perquè aquesta instrucció funcioni correctament assegureu-vos que el directori on esteu no conté espais.

4. Finalment executem la nostra aplicació de forma habitual

   ```
   $ ./exemple
   ```

   En executar d'aquesta forma es farà servir la nostra implementació de `malloc`. Podeu provar d'executar altres aplicacions senzilles com les comandes `ls` o `cp`. Totes faran servir la nostra implementació del `malloc`! Tingueu en compte també que altres aplicacions habituals poden no funcionar. Encara no està tot preparat...

A la nostra implementació de `malloc` no s'allibera de forma explícita la memòria dinàmica. En sortir del procés el sistema operatiu s'encarregarà d'alliberar aquesta memòria dinàmica. En tot cas, per tenir un codi net cal alliberar la memòria dinàmica per assegurar que l'aplicació només fa servir la memòria dinàmica que li fa falta.

3 Una versió de `malloc` més "adient"

Es presenta a continuació una implementació de `malloc` més adient, en particular un `malloc` en què després es pugui alliberar la memòria reservada fent servir un `free`.

3.1 Associar una estructura a cada bloc

Per tal de implementar-ho s'associa, per a cada bloc reservat amb `malloc`, una estructura amb la informació sobre el bloc reservat, veure la figura 1. Una manera de fer-ho és guardar al començament de cada bloc de memòria la següent informació:

- La mida del bloc demanat per l'usuari, en bytes.

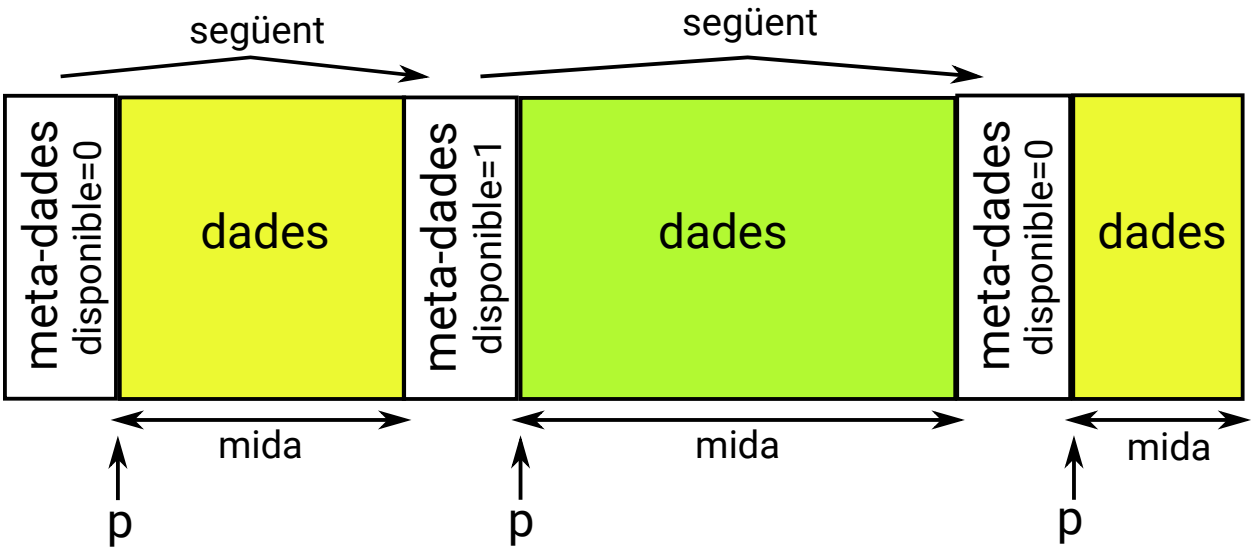

Figura 1: Cada bloc té associat unes meta-dades que contenen informació associat al bloc.

- Indicar si el bloc està disponible o no. Si no està disponible s'està fent servir en aquell moment per emmagatzemar-hi coses. Si està disponible, és indicació que s'ha alliberat perquè es pugui fer servir.
- Un apuntador al següent bloc demanat per l'usuari.

Aquest tipus d'informació s'anomena *meta-data* i en aquesta implementació s'emmagatzema abans de l'adreça de memòria associada a l'espai de memòria que podrà fer servir l'usuari. Aquí tenim l'estructura a utilitzar

```
#include <stddef.h>
#include <unistd.h>

#define MIDA_META_DADES  sizeof(struct m_meta_dades)
typedef struct m_meta_dades *p_meta_dades;

struct m_meta_dades {
  size_t  mida;
  int     disponible;
  int     magic;
  p_meta_dades seguent;
};
```

Fixeu-vos que dins del `struct` estem definint tres atributs: `mida`, `disponible`, `magic` i `següent`. Gràcies a aquests tres atributs podem implementar una versió més adient del `malloc`. L'atribut `magic` es un "valor màgic" que s'assigna a les metadades i que podreu fer servir per assegurar que tot funciona correctament.

3.2 First fit-malloc

La nostra nova funció `malloc` és aquesta, veure codi `malloc.c`

```c
#include "struct.h"

p_meta_dades primer_element = NULL;
p_meta_dades darrer_element = NULL;

#define ALIGN8(x) (((((x)-1)>>3)<<3)+8)

void *malloc(size_t mida)
{
  void *p;
  p_meta_dades meta_dades;

  if (mida <= 0) {
    return NULL;
  }

  mida = ALIGN8(mida);
  fprintf(stderr, "Malloc %zu bytes\n", mida);

  if (!primer_element) // Es el primer cop que es crida a malloc?
  {
    meta_dades = demanar_espai(mida);
    if (!meta_dades)
      return(NULL);
    primer_element = meta_dades;
  }
  else {  // Hem cridat abans al malloc
    meta_dades = cercar_bloc_lliure(mida);
    if (meta_dades) { // meta_dades trobat
      meta_dades->disponible = 0;
    } else {     // no s'ha trobat meta_dades
      meta_dades = demanar_espai(mida);
      if (!meta_dades)
        return (NULL);
    }
  }

  p = (void *) meta_dades;

  // Es retorna a l'usuari el punter a l'espai
  // de memoria que pot fer servir per a les dades

  return (p + MIDA_META_DADES);
}
```

La variable `primer_element` apunta al primer element de la llista de blocs reservat. La variable `darrer_element` apunta al darrer element de la llista. Observar que el valor de `mida` s'alinea amb un múltiple de 8 bytes. Això és perquè el punter retornat per malloc ha d'estar alienat amb 8 bytes ja que els processadors d'avui en dia utilitzen instruccions (com les SSE) que requereixen aquest alineament[2].

El primer cop que es crida a la funció `malloc` es cridarà a la funció `demanar_espai`. Aquesta funció és equivalent a la funció dummy que hem definit fa un moment amb la diferència que es reserva també espai per a les meta-dades que emmagatzemaran informació associat al bloc reservat. Aquest és el codi, veure fitxer `demanar_espai.c`

[2]Observeu que la estructura de `meta_dades` té una mida múltiple de 8 bytes!

```
#define MAGIC     0x01102017

p_meta_dades demanar_espai(size_t mida)
{
  p_meta_dades meta_dades;

  meta_dades = (void *) sbrk(0);

  if (sbrk(MIDA_META_DADES + mida) == (void *) -1)
    return (NULL);

  meta_dades->mida = mida;
  meta_dades->disponible = 0;
  meta_dades->magic = MAGIC;
  meta_dades->seguent = NULL;

  if (darrer_element)
    darrer_element->seguent = meta_dades;

  darrer_element = meta_dades;

  return meta_dades;
}
```

La funció `free`, que no es dóna aquí, només ha de posar el valor '`disponible`' a 1 (l'haureu d'implementar vosaltres!). D'aquesta forma s'indica que el bloc és lliure per a futurs `malloc`. A l'hora de fer un `malloc` caldrà doncs mirar primer si a la llista de blocs disponibles n'hi ha algun que sigui prou gran. Aquí teniu la funció `cercar_bloc_lliure` que ho permet fer

```
p_meta_dades cercar_bloc_lliure(size_t mida) {
  p_meta_dades current = primer_element;

  while (current && !(current->disponible && current->mida >= mida))
    current = current->seguent;

  return current;
}
```

Analitzeu bé el codi de la funció `malloc` que acabem de definir: en cas que es trobi un bloc lliure suficientment gran, s'indicarà que ja no està disponible. En cas que no se'n trobi cap bloc lliure suficientment gran, es demanarà nou espai amb la crida a sistema `sbrk`. Observeu que la funció `malloc` retorna un punter p a l'espai de memòria que l'usuari farà servir (veure dibuix). Les meta-dades es troben a memòria, "just a sota", però l'usuari no s'ha d'encarregar de manipular-les.

4 Feina a realitzar

Amb la implementació proposada a la secció anterior es demana realitzar els següents passos. Per facilitar la implementació del codi associat a la implementació del `malloc`, s'han unificat tots els fitxers comentats a la secció 3.2 en un de sol anomenat `malloc_first_fit.c`. Comproveu que el codi compila i feu-lo anar amb `exemple.c`. Proveu també la solució que s'adjunta amb el codi amb aplicacions com el `grep`, el `find`, o inclús `firefox`[3]. Aquesta solució implica la implementació dels

[3] Amb el `firefox` (o altres aplicacions gràfiques) es poden produir problemes ja que aquestes aplicacions fan servir múltiples fils d'execució, i el codi que fem servir no està protegit pels problemes que es poden produir si múltiples fils criden a la vegada al malloc.

punts 1 a 4.

1. Implementació de la funció `free(void *ptr)` que faci servir l'estructura proposada. Bàsicament el que ha de fer és posar l'atribut `disponible` a 1. Comproveu que l'atribut `magic` té el valor que ha de tenir (en cas contrari, imprimiu un missatge d'error). Per implementar aquesta funció tingueu en compte que es passa com a paràmetre a la funció `free` un punter a les dades (la variable `p` del dibuix), però l'estructura es troba "just a sota". S'ha de tenir en compte que es pot cridar a `free` amb un apuntador a NULL. En aquest cas s'ha d'ignorar la crida.

2. Un cop implementada la funció `free` assegureu-vos que tot funciona correctament. Proveu la vostra implementació fent servir l'exemple que es mostra a l'inici de la pràctica.

3. Implementeu la funció `void calloc(size_t nelem, size_t elsize)`. La funció `calloc` permet reservar varis elements de memòria i els deixa inicialitzats a zero. S'aconsella fer servir la funció `memset` per inicialitzar el bloc de memòria a zero.

4. Implementeu la funció `void *realloc(void *ptr, size_t mida)`. La funció `realloc` reajusta la mida d'un bloc de memòria obtingut amb `malloc` a una nova mida. El funcionament és aquest: a) si li passem un NULL pointer, es suposa que actua com un `malloc` normal i corrent. b) si li passem un apuntador que hem creat amb el nostre `malloc` i la mida que demanem és suficient amb el bloc que ja té reservat, no cal fer res, el retornem el punter tal qual. c) en cas contrari, haurem de reservar un bloc amb més espai i copiar les dades de l'antic bloc en aquest nou. Per això es usar la funció `memcpy` per a copiar el contingut d'un bloc en un altre.

5. Proveu ara de nou la implementació del `malloc` que teniu. Proveu d'executar aplicacions com el `grep`, el `find`, o inclús `firefox` (tot i que amb aquest darrer podeu tenir problemes ja és un programa multifil). Tingueu en compte que caldrà executar aquestes aplicacions des del terminal on hagueu definit el LD_PRELOAD.

6. Modifiqueu el codi per tal que el `malloc` faci un *best fit* en comptes d'un *first fit*. És a dir que busqui el bloc de mida més adient.

7. Modifiqueu el codi del `free` de tal forma que quan alliberem un bloc pugui ajuntar varis blocs contigus si estan buits també.

8. De forma opcional, modifiqueu el codi del `malloc` i de `realloc` de tal forma que quan reutilitzem blocs aquests es puguin dividir a la mida necessària.

5 Entrega

Entregueu el següents directoris

1. Una implementació del `malloc` en què hi hagi el `free`, `calloc` i `realloc` fent servir el *first fit*. El codi `exemple.c` ha de contenir exemples de crides a aquestes funcions que demostrin que les funcions funcionen correctament. Inclogueu també un script que compili i executi l'exemple fent servir la llibreria `malloc` amb la variable d'entorn LD_PRELOAD.

2. Una implementació del malloc en què hi hagi el `free`, `calloc` i `realloc` fent i que implementi els punts 6 i 7 especificats en aquesta pràctica. El codi `exemple.c` ha de contenir exemples de crides a aquestes funcions que demostrin que les funcions funcionen correctament. Inclogueu també un script que compili els fitxers i executi l'exemple fent servir la llibreria `malloc` amb la variable d'entorn LD_PRELOAD.