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

Maig 2017

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 `malloc` i `free`, 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 de bytes i retorna un apuntador a un bloc de memòria d'igual mida.

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 en temps d'execució).

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)` reserva 1000 bytes al 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`

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

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

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

    if (size <= 0)
        return NULL;

    if (sbrk(size) == (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`.

Amb aquesta implementació la memòria s'acabaria omplint ràpidament, però podem provar si el codi funciona. 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

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 -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 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...

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. Una manera de fer-ho és guardar al començament de cada bloc de memòria la següent informació:

- la mida del bloc en bytes.
- indicar si el bloc està ocupat o no.
- l'adreça del següent bloc.

Aquest tipus d'informació s'anomena *meta-data* i en aquesta implementació s'emmagatzema abans de l'adreça de memòria que apunta l'apuntador que retornem amb el `malloc`. Aquí tenim l'estructura a utilitzar

```
typedef struct m_block *p_block;

struct m_block {
    size_t  size;
    p_block next;
    int     free;
};

#define META_SIZE  sizeof(struct m_block)
```

Fixeu-vos que dins del `struct` estem definint tres atributs, 'size', 'next' i 'free'. Gràcies a aquests tres atributs podem implementar una versió més adient del `malloc`. L'atribut 'next' ens permet implementar una llista encadenada de blocs.

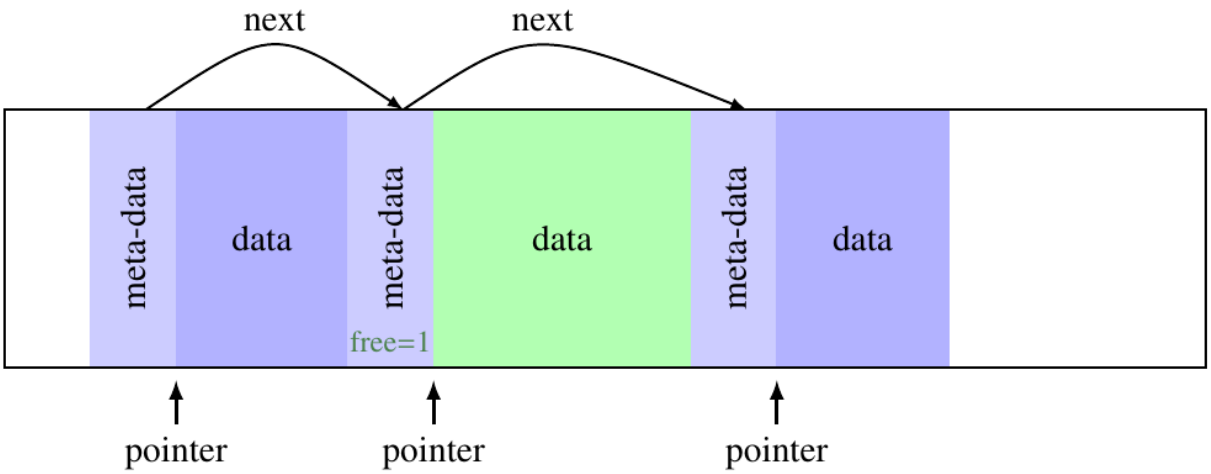

Figura 1: Cada bloc té associada una estructura que conté informació associat al bloc.

3.2 First fit-malloc

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

```
p_block first_element = NULL;
p_block last_element  = NULL;

void *malloc(size_t size)
{
    void *p;
    p_block block;

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

    if (first_element){
        block = cercar_bloc_lliure(size);
        if (block) { // block found
            block->free = 0;
        } else {     // no block found
            block = demanar_espai(size);
            if (!block)
                return (NULL);
        }
    }
    else  // This is done the first time malloc is called
    {
        block = demanar_espai(size);
        if (!block)
            return(NULL);
        first_element = block;
    }

    p = (void *) block;
    return (p + META_SIZE);
}
```

La variable `first_element` apunta al primer element de la llista de blocs reservat. La variable `last_element` apunta al darrer element de la llista.

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 la l'estructura que emmagatzemarà informació associat al bloc reservat. Aquest és el codi, veure fitxer `demanar_espai.c`

```
p_block demanar_espai(size_t size)
{
    p_block block;

    block = sbrk(0);

    if (sbrk(META_SIZE + size) == (void *) -1)
        return (NULL);

    block->size = size;
    block->next = NULL;
    block->free = 0;

    if (last_element)
      last_element->next = block;

    last_element = block;

    return block;
}
```

La funció `free`, que no es dóna aquí, només ha de posar el valor 'free' a 1 (l'haureu d'implementar vosaltres!). D'aquesta forma s'indica que el bloc és lliure per a futurs `malloc`. Observar també que es poden alliberar múltiples blocs posant el valor de `free` a 1. 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_block cercar_bloc_lliure(size_t size) {
    p_block current = first_element;

    while (current && !(current->free && current->size >= size))
        current = current->next;

    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.

4 Feina a realitzar

Amb la implementació proposada a la secció anterior es demana realitzar els següents passos

1. Implementació de la funció `free(void *ptr)` que faci servir l'estructura proposada. Bàsicament el que ha de fer és posar l'atribut `free` a 1. Per implementar aquesta funció tingueu en compte que es passa com a paràmetre a la funció free un punter a les dades, però l'estructura es troba "just a sota" (observeu què és el que retorna la funció `malloc`). 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 size)`. 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 `kate` i `firefox`! És molt possible que funcionin tot i que ho faran molt lentament. 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. Inclogueu també un script que compili els fitxers 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. Inclogueu també un script que compili els fitxers i executi l'exemple fent servir la llibreria `malloc` amb la variable d'entorn LD_PRELOAD.