Pràctica 4

Gestió de memòria dinàmica per a processos

Maig 2019

Índex

1 Introducció: la crida a sistema `sbrk`

L'objectiu d'aquesta pràctica és fer una implementació pròpia de la gestió de memòria dinàmica en un procés a través de la implementació de les funcions de la llibreria estàndard `malloc` i `free`. Aquestes dues funcions no són crides a sistema, sinó que són crides que formen part de la llibreria d'usuari. Internament utilitzen crides a sistema.

Atenció! Aquesta pràctica no funcionarà a sistemes Mac ni a màquines virtuals.

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`, una funció que permet manipular l'espai de *heap* del procés.

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`

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

  fprintf(stderr, "Malloc\n");

  if (mida <= 0)
    return NULL;

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

  return p;
}

void free(void *p)
{
  fprintf(stderr, "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
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 -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 quan el procés no la necessita per assegurar que l'aplicació només fa servir la memòria dinàmica que li fa falta.

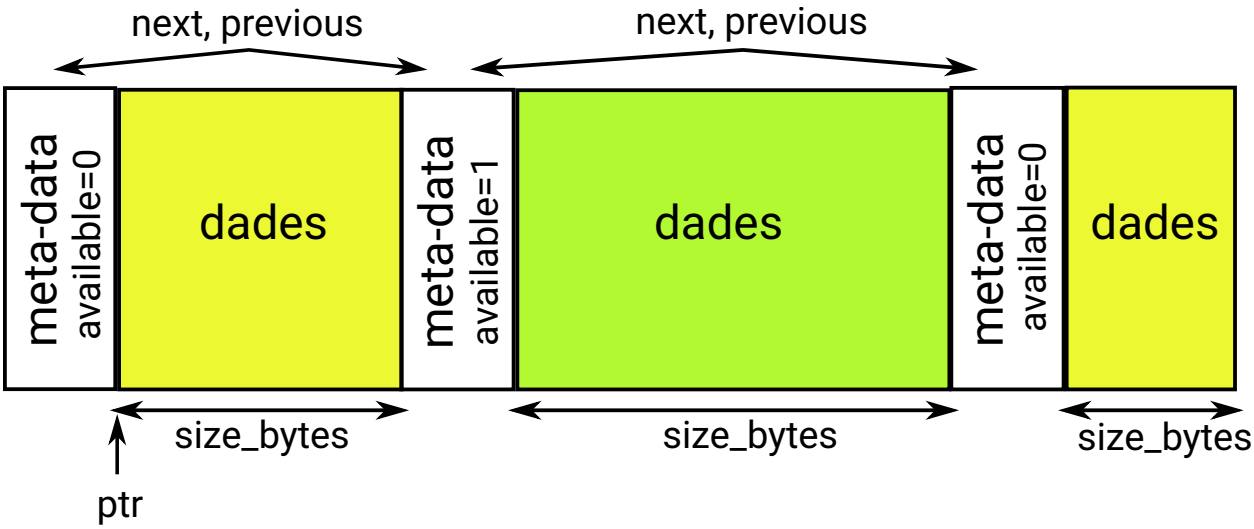

Figura 1: Cada bloc de dades té associat unes meta-dades que contenen informació associat al bloc.

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 informació, *meta-dades*, associat al bloc. Aquí tenim l'estructura a utilitzar, fitxer `struct.h`.

```
#define SIZE_META_DATA  sizeof(struct m_meta_data)
typedef struct m_meta_data *p_meta_data;

/* This structure has a size multiple of 8 bytes */

struct m_meta_data {
    size_t  size_bytes;
    int     available;
    int     magic;
    p_meta_data next;
    p_meta_data previous;
};
```

Fixeu-vos que dins del `struct` estem definint tres atributs: size_bytes, available, `magic`, next i `previous`.

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

- L'atribut `magic` es un "valor màgic" que s'assigna a les metadades i que podreu fer servir per assegurar que tot funciona correctament.
- Un apuntador la següent i anterior estructura de meta-dades.

3.2 First fit-malloc

Tot el codi que es mostra a continuació es troba al codi `malloc_first_fit.c`. La nostra nova funció `malloc` és aquesta.

```
p_meta_data first_element = NULL;
p_meta_data last_element  = NULL;

#define ALIGN8(x) (((((x)-1)>>3)<<3)+8)
#define MAGIC     0x12345678

void *malloc(size_t size_bytes)
{
    void *p;
    p_meta_data meta_data;

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

    // We allocate a size of bytes multiple of 8
    size_bytes = ALIGN8(size_bytes);
    fprintf(stderr, "Malloc %zu bytes\n", size_bytes);

    meta_data = search_available_space(size_bytes);

    if (meta_data) { // free block found
      meta_data->available = 0;
    } else {     // no free block found
      meta_data = request_space(size_bytes);
      if (!meta_data)
        return (NULL);

      meta_data->previous = last_element;
      last_element = meta_data;

      if (first_element == NULL) // Is this the first element ?
        first_element = meta_data;
    }

    p = (void *) meta_data;

    // We return the user a pointer to the space
    // that can be used to store data

    return (p + SIZE_META_DATA);
}
```

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. Observar que el valor de `size_bytes` s'alinea amb un múltiple de 8 bytes. Això és perquè el punter retornat per malloc ha d'estar alineat amb

8 bytes ja que els processadors d'avui en dia utilitzen instruccions (com les SSE) que requereixen aquest alineament[1].

La funció malloc utilitza les funcions `search_available_space` i `request_space` per gestionar la memòria. La funció `request_space` és la que fa la crida a sistema per demanar espai al sistema operatiu mitjançant la crida a sistema `sbrk`. Aquest és el codi:

```
p_meta_data request_space(size_t size_bytes)
{
    p_meta_data meta_data;

    meta_data = (void *) sbrk(0);

    if (sbrk(SIZE_META_DATA + size_bytes) == (void *) -1)
        return (NULL);

    meta_data->size_bytes = size_bytes;
    meta_data->available = 0;
    meta_data->magic = MAGIC;
    meta_data->next = NULL;
    meta_data->previous = NULL;

    return meta_data;
}
```

La funció `free`, que haureu d'implementar vosaltres, és la que haurà d'alliberar el bloc de dades corresponent (veure secció 4). Per fer-ho només cal posar el valor de l'atribut `available` a 1. D'aquesta forma s'indica que el bloc és lliure per a futurs `malloc`. A l'hora de fer un `malloc` es comprova primer, mitjançant la funció `search_available_space`, si hi ha un bloc disponible prou gran. Si és així, es retorna a l'usuari aquest bloc i s'evita fer una crida a sistema. Aquest és el codi corresponent:

```
p_meta_data search_available_space(size_t size_bytes) {
    p_meta_data current = first_element;

    while (current && !(current->available && current->size_bytes >= size_bytes))
        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 amb la crida a sistema `sbrk`. Observeu que la funció `malloc` retorna un punter a l'espai de memòria que l'usuari pot fer servir (el bloc de dades, veure Figura 1). 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

Totes les funcions comentades a la secció anterior es troben al fitxer `malloc_first_fit.c`. Comproveu que el codi compila i feu-lo anar amb `exemple.c`. A continuació es proposa afegir funcions

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

addicionals a les funcions que ja teniu perquè pugui funcionar amb altres aplicacions. Això implica la realització dels punts 1 a 5.

1. Implementació de la funció `free(void *ptr)` que faci servir l'estructura proposada. Per implementar aquesta funció tingueu en compte que es passa com a paràmetre a la funció `free` un punter a les dades (la variable `ptr` del dibuix), però l'estructura es troba "just a sota". Bàsicament la funció `free` ha de posar l'atribut `available` a 1. Per assegurar que la funció fa la feina correctament, comproveu que l'atribut `magic` té el valor que ha de tenir (en cas contrari, imprimiu un missatge d'error). 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. Imprimiu per pantalla el valor de `size_bytes` que s'allibera.

2. Un cop implementada la funció `free` assegureu-vos que funciona correctament. Proveu la vostra implementació fent servir el codi `exemple.c` 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, en concret `nelem` elements de mida `elsize` bytes, i els deixa inicialitzats a zero. S'aconsella fer servir la funció `memset` per inicialitzar el bloc de dades a zero. La funció retorna un punter a la memòria reservada.

4. Implementeu la funció `void *realloc(void *ptr, size_t size_bytes)`. La funció `realloc` reajusta la mida d'un bloc de memòria obtingut amb `malloc` a una nova mida. Es proposa que la implementació de la funció `realloc` sigui la següent: a) si li passem un punter NULL a `ptr`, se suposa que la funció `realloc` actua com un `malloc` normal i corrent. b) si li passem a `ptr` 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 nou bloc amb més espai i copiar les dades de l'antic bloc en aquest nou. Per això es pot 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. Assegureu-vos que la llibreria funciona amb aplicacions com el `grep`, el `find`. Podeu provar inclús d'executar el `kate` o el `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.

Es proposa fer una segona versió de la implementació del malloc, `malloc_best_fit_first_fit.c`.

6. Primer de tot, modifiqueu el codi per tal que el es 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. Aprofiteu el fet que disposeu dels atributs `next` i `previous` per evitar haver de recórrer tota la llista de blocs.

8. Assegureu-vos que la llibreria funciona amb aplicacions com el `grep`, el `find`.

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. Els punts 1 i 2 permeten obtenir una qualificació màxima de 10 a la pràctica, mentre que la part opcional permet afegir un punt addicional (qualificació màxima d'11).

1. Una implementació del `malloc` en què hi hagi el `free`, `calloc` i `realloc` fent servir el *first fit*. Inclogueu també un script que compili i executi l'exemple fent servir la llibreria `malloc` amb la variable d'entorn LD_PRELOAD. Assegureu-vos que la llibreria funciona amb l'aplicació `find` o `grep` (a l'hora de revisar la pràctica els executarem sense paràmetres).

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. Inclogueu també un script que compili els fitxers i executi l'exemple fent servir la llibreria `malloc` amb la variable d'entorn LD_PRELOAD. Assegureu-vos que la llibreria funciona amb l'aplicació `find` o `grep` (a l'hora de revisar la pràctica els executarem sense paràmetres).

3. En cas que vulgueu entregar la part opcional, entregueu-la en un tercer directori, integrat amb la funcionalitat descrita.