Pràctica 4

Gestió de memòria dinàmica per a processos

Maig 2020

Índex

1 Introducció: la crida a sistema sbrk

L'objectiu d'aquesta pràctica és fer una implementació senzilla i 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.

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 (un sencer de 64 bytes sense signe) 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 of size %zu\n", mida);

  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 `kate` 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`

```
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. Per a sistemes Mac OS consulteu la secció 6.

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.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.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[1]. Se us mostrarà per pantalla un missatge cada cop que l'aplicació faci un malloc o un free. Podeu provar d'executar altres aplicacions senzilles com les comandes `ls` o `cp`. Faran servir la nostra implementació del malloc! Tingueu en compte que altres aplicacions habituals poden no funcionar. Encara no està tot preparat!

[1] Es pot evitar definir la variable d'entorn `LD_PRELOAD` executant l'aplicació fent "`LD_PRELOAD=$PWD/malloc.so ./exemple`".

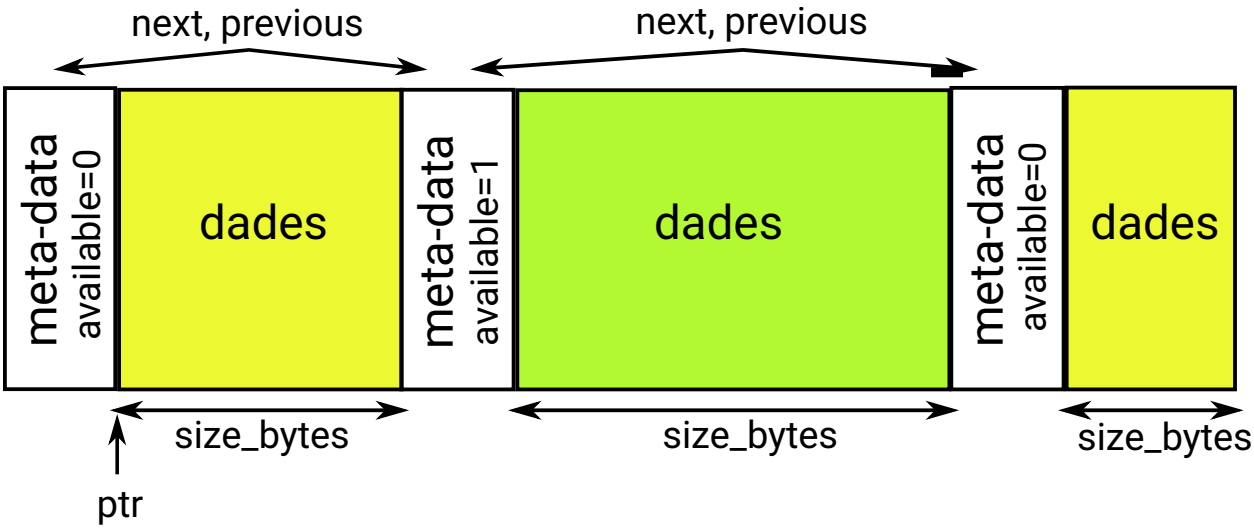

Figura 1: Cada bloc de dades té associat unes meta-dades que contenen informació associat al bloc.

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 ja que aquest sap quina memòria ha estat assignada al procès. 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.

3 Una versió de malloc senzilla

Es presenta a continuació un prototip d'una implementació del malloc més adient, en particular un malloc en què després es pugui alliberar la memòria reservada fent servir un free. La implementació que es mostra a continuació es pot considerar una versió simplificada del malloc que teniu disponible a la llibreria d'usuari del sistema operatiu[2].

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`.

[2]Atenció! La versió que es mostra en aquesta secció no funcionarà amb les comandes `ls` o `cp` ja que en aquest prototip es realitzen comprovacions addicionals a l'hora d'alliberar memòria. Part del vostre treball a la pràctica és afegir funcions que permetin que aquest prototip funcioni correctament.

```c
#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 */
struct m_meta_data {
    size_t  size_bytes;
    int     available; // has it been free'd?
    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.

- L'atribut size_bytes emmagatzema la mida del bloc associada que ha demanat l'usuari, en bytes.
- L'atribut available permet indicar si el bloc està disponible o no perquè es pugui fer servir. Si no està disponible el valor d'aquest atribut serà 0 i indicarà que s'està fent servir en aquell moment. Si està disponible el valor de l'atribut és 1 i indica que s'ha alliberat perquè es pugui fer servir per futurs mallocs.
- L'atribut magic és un "valor màgic" que s'assigna a les metadades i que es pot fer servir per assegurar que tot funciona correctament.
- Els atributs next i previous són apuntadors que es fan servir per construir una llista. Són apuntadors la següent i anterior estructura de meta-dades.

3.2 Simple malloc

Tot el codi que es mostra a continuació es troba al codi `malloc_simple.c`. A continuació es escriuen algunes de les funcions. La nostra nova funció malloc és mostra a la Figura 2

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[3].

La funció malloc utilitza les funcions search_available_space i request_space per gestionar la memòria. 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. La funció request_space és la que fa la crida a sistema per demanar espai al sistema operatiu mitjançant la crida a sistema sbrk. El codi es mostra a la Figura 3.

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.

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

```c
void *malloc(size_t size_bytes)
{
  void *p, *ptr;
  p_meta_data meta_data;

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

  // Reservem una mida de memoria multiple de 8. Es
  // cosa de l'electronica de l'ordinador que requereix
  // que les dades estiguin alineades amb multiples de 8.
  size_bytes = ALIGN8(size_bytes);
  fprintf(stderr, "Malloc %zu bytes\n", size_bytes);

  // Bloquegem perque nomes hi pugui entrar un fil
  pthread_mutex_lock(&mutex);

  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);

    if (last_element) // we add to the last element of the list
      last_element->next = meta_data;
    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;

  // Desbloquegem aqui perque altres fils puguin entrar
  // a la funcio
  pthread_mutex_unlock(&mutex);

  // Retornem a l'usuari l'espai que podra fer servir.
  ptr = p + SIZE_META_DATA;
  return ptr;
}
```

Figura 2: Codi de la funció malloc.

```c
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;
}
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;
}
void free(void *ptr)
{
    p_meta_data meta_data;

    if (!ptr)
        return;

    // Bloquegem perque nomes hi pugui entrar un fil
    pthread_mutex_lock(&mutex);

    meta_data = (p_meta_data) (ptr - SIZE_META_DATA);

    if (meta_data->magic != MAGIC) {
      fprintf(stderr, "ERROR free: value of magic not valid\n");
      exit(1);
    }

    meta_data->available = 1;
    fprintf(stderr, "Free at %x: %zu bytes\n", meta_data, meta_data->size_bytes);

    // Desbloquegem aqui perque altres fils puguin entrar
    // a la funcio
    pthread_mutex_unlock(&mutex);
}
```

Figura 3: Codi de les funcions request_space, search_available_space i free.

Finalment, a la Figura 3 també es mostra la implementació de la funció free. Aquesta funció allibera el bloc posant el valor de l'atribut available a 1. D'aquesta forma es podrà fer servir el bloc alliberat en futures crides a malloc. No es retorna doncs el bloc al sistema operatiu en fer una crida a free.

A les funcions malloc i free es fan servir funcions específiques per tal d'assegurar que l'execució es realitzi correctament en cas que hi hagi múltiples fils (per a més informació consulteu el tema de "Introducció a la concurrència"). Les aplicacions com `cp`, `ls` o `find` només tenen un únic fil d'execució i, per tant, aquest bloqueig és realment innecessari i no afecta al funcionament del codi. Sí que afecta en cas que vulgueu executar aplicacions més complexes que incorporen una interfície gràfica com un editor de text, `kate` o `kwrite`, o alguna aplicació equivalent. Aplicacions com el Firefox tenen el seu propi gestor de memòria. És a dir, el Firefox no fa servir el malloc sinó que fa servir una implementació pròpia per gestionar la memòria dinàmica que li falta! Respecte la pràctica, no toqueu les funcions de bloqueig de fils. Tampoc farà falta que afegiu cap funció de bloqueig en el vostre codi.

4 Feina a realitzar

Totes les funcions comentades a la secció anterior es troben al fitxer `malloc_simple.c`. Comproveu que el codi compila i feu-lo anar amb `exemple.c`. A continuació es proposa afegir funcions addicionals a les funcions que ja teniu perquè pugui funcionar amb altres aplicacions. En cas que vulgueu imprimir alguna cosa per pantalla se us recomana fermament imprimir els missatges per la sortida estàndard d'error (stderr) i no pas la sortida estàndard (stdout)[4].

1. 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.

2. 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.

3. 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`. Tingueu en compte que caldrà executar aquestes aplicacions des del terminal on hagueu definit el `LD_PRELOAD`. Veureu que l'aplicació funciona molt lentament ja que es fa una crida a sistema cada cop fa falta memòria. I no patiu si l'aplicació "peta" en intentar executar el `kate`.

[4] La raó això es deu al fet que els missatges que s'imprimeixen per la sortida estàndard s'emmagatzemen a un buffer abans d'imprimir-se per pantalla, mentre que els missatges impresos per la sortida estàndard d'error no s'emmagatzemen a cap buffer, sinó que s'imprimeixen de forma immediata per pantalla.

Es proposa fer una segona versió de la implementació del malloc anterior per tal de millorar la gestió de la memòria. L'anomenarem `malloc_split_and_merge.c`. L'objectiu final és fer una implementació que gestioni millor la memòria alliberada així com fer l'execució sigui més eficient. Les modificacions que es proposen a continuació únicament impliquen afegir funcionalitat al malloc i al free. No cal modificar el calloc ni el realloc.

5. En alliberar un bloc de memòria (amb el free) fusioneu els blocs contigus que estiguin lliures. Se us recomana implementar aquesta funcionalitat en una funció independent que es cridarà des del free. D'aquesta forma, a mesura que es alliberant blocs s'aniran fusionant els blocs contigus lliures. Si, per exemple, es fa un malloc gran, es podrà aprofitar algun bloc lliure gran que hi hagi disponible i no caldrà fer cap crida a sistema.

6. La funció malloc comprova primer si hi ha algun bloc lliure a la seva llista. En cas que no n'hi hagi cap, demana espai mitjançant una crida a sistema per demanar l'espai necessari al sistema operatiu. Implementeu una funció que permeti dividir un bloc en dos trossos en cas que al bloc retornat tingui un espai en què "sobri" espai per futures crides a malloc. En particular, se us proposa dividir el bloc en dos trossos en cas que sobrin $3 \cdot 4096 = 12288$ bytes. Crideu aquesta funció en els dos casos descrits anteriorment; al següent punt veureu perquè. Aneu amb compte a l'hora de dividir el bloc: la mida en bytes d'un bloc és un sencer sense signe (i.e. size_t) i, per tant, fer l'operació d'una resta pot portar-vos a maldecaps! En cas que vulgueu fer una resta, comproveu abans que el número no pugui ser negatiu.

7. Finalment, per tal de fer més eficient la gestió de memòria es demana modificar la funció request_space, que fa la crida a sistema per demanar espai al sistema operatiu. L'espai que aquesta funció demanarà com a mínim serà de $30 \cdot 4096 = 122880$ bytes encara que l'usuari n'hagi demanat menys. En cas que l'usuari n'hagi demanat menys, la funció implementada al punt anterior ja s'encarregarà de dividir el bloc en dos en cas que sobri espai.

8. Ja hem acabat! Ara podeu provar la nova implementació amb funcions com `ls`, `cp` o `find`. Inclús també podeu tornar a provar amb el `kate` i hauríeu de notar que l'eficiència d'execució és molt superior a la versió anterior. Proveu d'executar el codi i desplegar el menú, per exemple. No patiu si el `kate` us peti en intentar, per exemple, escriure alguna cosa... En tot cas, ja teniu una primera versió del malloc!

5 Entrega

Per a l'entrega de la pràctica es demana entregar el codi font (80%) i un informe (20%).

5.1 Codi font

Entregueu el codi font desenvolupat fent servir dos directoris: en el primer directori inclogueu el codi font associat a la implementació dels punts 1 a 3. En el segon directori inclogueu l'extensió proposada als punts 5, 6 i 7. Els dos directoris han d'incloure un `Makefile` que compili el codi i generi el fitxer `malloc.so` corresponent a partir del fitxer C corresponent. Es comprovarà el bon funcionament del codi amb aplicacions com el `cp`, `ls` o `find`. En cap cas es farà servir una aplicació com el `kate` o similar per provar el codi.

5.2 Informe

A més dels scripts caldrà entregar un informe sobre la feina feta. L'informe a entregar ha d'estar en format `PDF` o equivalent (no s'admeten formats com `odt`, `docx`, ...). Aquest informe ha de mostrar les proves que s'han realitzat per assegurar el bon funcionament del codi. No inclogueu una descripció de l'algorisme implementat llevat que cregueu rellevant comentar algun detall. Inclogueu, preferentment en format text, la comanda que heu executat i algun comentari breu descrivint el resultat si ho creieu necessari. En cas que no us funcioni el codi indiqueu també el problema detectat així com possibles sospites de quin pot ser la font del problema. Sigueu breus i clars a les vostres respostes, no fa falta que us esteneu en el text escrit.

En cas que preferiu incloure captures de pantalla en comptes d'incloure el resultat en format text, assegureu-vos que el text de la captura es pot llegir bé (és a dir, que tingui una mida similar a la resta del text del document) i que totes les captures siguin uniformes (és a dir, que totes les captures tinguin la mateixa mida de text).

El document ha de tenir una llargada màxima de 3 pàgines (sense incloure la portada). El document s'avaluarà amb els següents pesos: proves realitzades i comentaris associats, un 60%; escriptura sense faltes d'ortografia i/o expressió, un 20%; paginació del document feta de forma neta i uniforme, un 20%.

6 Sistemes Mac OS

A sistemes Mac OS es pot fer l'equivalent al que es fa als sistemes Linux amb la variable d'entorn `LD_PRELOAD`[5]. Al Mac OS s'ha de fer servir la variable d'entorn `DYLD_INSERT_LIBRARIES`. La informació s'ha obtingut d'aquest enllaç. Els passos a realitzar són:

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`. A Mac OS les llibreries dinàmiques acostumen a tenir l'extensió dylib. És molt possible que en executar aquesta instrucció us doni un "Warning" indicant que "'sbrk' is deprecated". Feu cas omís d'aquest missatge.

   ```
   $ gcc -O -shared -fPIC malloc_dummy.c -o malloc.dylib
   ```

3. Finalment executem la nostra aplicació de forma habitual indicant que volem que es carregui la llibreria dinàmica indicada anteriorment

   ```
   $ DYLD_INSERT_LIBRARIES=$PWD/malloc.dylib DYLD_FORCE_FLAT_NAMESPACE=1 ./exemple
   ```

[5] El que es comenta a la secció no ha pogut ser validat a data del 30 d'abril del 2020.