Pràctica 5. Gestió de memòria dinàmica per a processos

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 amb c.

Concretament repassarem els conceptes de:

- Apuntadors.
- Reservar memòria dinàmicament usant malloc.
- Llistes encadenades.

La signatura de la funció malloc (memory allocation) de c és la següent:

```
void *malloc(size_t size);
```

Com a paràmtrre d'entrada rep un nombre de bytes o retorna un apuntador a un bloc de memòria d'igual mida.

Crida a sistema per a reservar memòria.

Una forma d'implementar el malloc és usant la crida a sistema sbrk. Amb aquesta crida podem manipular l'espai de *heap* (el *heap* gestiona la reserva de memòria en temps d'execució.) que el sistema ha reservat per al procés.

`srbk(0)` retorna un apuntador al començament del *heap*. Si hi especifiquem qualsevol altre quantitat el *heap* s'incrementa amb aquest valor i retorna un apuntador a l'anterior començament del *heap*.

Dummy Malloc

Si volem implemntar un malloc molt senzill seria una cosa com aquesta:

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

void *malloc(size_t size) {
  void *p = sbrk(0);
  if (sbrk(size) == (void*) -1)
    return NULL; // sbrk failed.
  return p;
}
```

Amb aquest codi estem ampliant l'espai en el heap tant com `size` indica. En cas que la crida a `srbk(size)` retornés -1 vol dir que no s'ha pogut extendre.

Aquest malloc te l'inconvenient que no podem fer un free de la memòria ocupada un cop ja no la necessitem. Per tant la memòria s'acabaria omplint ràpidament.

La funció **free** de c és capaç d'alliberar la memòria dinàmica. El seu prototipus és:

`void free(void *ptr);`

Quan se li passa un punter que havia sigut creat a partir d'un malloc, allibera el seu espai.

Introduint els Structs en C

Per poder implementar un *malloc* que després es pugui alliberar la memòria que ha reservat amb un free, necessitem guardar en algun lloc la informació sobre el bloc que hem reservat.

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

- la mida del bloc
- l'adreça del següent bloc
- i si el bloc està o no està reservat.

Aquest tipus d'informació s'anomena *meta-data* i s'ha de trobar abans de l'adreça de memòria que apunta l'apuntador que retornem amb el *malloc*.

Com es pot traduir això en C? El fet que estiguem apuntant al següent bloc ens dóna la clau de que el que necessitem implementar és una llista encadenada.

Per fer una llista encadenada (una llista de nodes iguals que cada un apunta al següent) primer hem de definir aqquest node. Per definir un tipus de dades que contingui varis atributs, en C s'usen els `struct`.

```
struct m_block {
  size_t          size;
  struct m_block  *next;
  int             free;
};
```

Fixeu-vos que dins del **struct** estem definint tres atributs, `size` , `next` i `free` de tipus **size_t**, un apuntador al propi tipus que estem definint **struct m_block ** i un **int**. La mida en memòria d'un **struct** és la suma de les mides de cada un dels atributs, estan col·locats de forma contigua. En el nostre cas si fem un `sizeof(struct m_block)` el resultat serà de 3*4 bytes, un int de 4 bytes per cada atribut.

Per accedir al contingut d'uns dels atributs es fa mitjançant l'operador punt ".".

Com que el tipus apuntador a struct m*block * el farem servir molt, podem crear un nou tipus de dada anomenat **p*block**. Això ho podem fer amb la instrucció `typedef` :

```
typedef struct m_block * p_block;

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

#define META_SIZE sizeof( struct m_block)
```

First Fit Malloc

Un *first fit* malloc consiteix en fer un malloc que busqui un bloc lliure amb suficient espai per satisfer la reserva que li han demanat. La funció per a buscar un bloc lliure seria algo així:

```
 p_block find_free_block(p_block *last, size_t size) {

  p_block current = global_base;
  while (current && !(current->free && current->size >= size)) {
    *last = current;
    current = current->next;
  }
  return current;
}
```

Fixeu-vos que per accedir accedir al contingut d'un atribut d'un apuntador a **struct** es fa mitjançant

l'operador de la fletxa (->):

`current->free`

Això és equivalent a fer:

`(*current).free`

`global_base` és una variable global amb l'inici de la llista encadenada, és a dir el primer bloc de meta data que s'ha creat. En cas de no trobar cap lloc lliure, és a dir, encara no s'ha fet cap free, s'ha de crear un nou espai. Per fer això utilitzarem la crida **sbrk** com fèiem abans.

```
p_block request_space(p_block last, size_t size) {
  p_block block;
  block = sbrk(0);
  if (sbrk(META_SIZE + size) == (void*) -1)
      return (NULL);
  block->size = size;
  block->next = NULL;
  if (last) { // NULL on first request.
    last->next = block;
  }
  block->free = 0;
  return block;
}
```

Ara que ja tenim fetes les funcions que usarem per manipular la llista encadenada de blocs, podem definir de nou la nostre funció malloc:

```c
void *global_base=NULL;

void *malloc(size_t size) {
   p_block block,last ;

  if (size <= 0) {
    return NULL;
  }
 if (global_base){
  last = global_base;
  block = find_free_block(&last,size);
  if (global_base) { // block found!
      block->free = 0;
  } else { // Not block found!
    block = request_space(last,size);
    if (!block)
      return (NULL);
  }
 } else{
  block = request_space(NULL,size);
  if (!block)
    return(NULL);
  global_base = block;
 }
 return(block+1); // since block is a pointer to a m_block structure, +1 increments t
}
```

Fixeu-vos que el que fem aquí és, primer de tot, mirar si és el primer cop que s'executa el malloc (la variable global `global_base` no està definida). Sí és així, es demana un bloc nou de mida *size* i es defineix global_base. En cas contrari es busca si hi ha algun espai que s'hagués alliberat. En cas contrari, es crearà un nou bloc.

Fixeu-vos que el que retornem és `block+1` això és perqué hem de retornar l'apuntador apuntant a la zona de memòria contigua al nostre meta dada. Com que block és un apuntador a la nostre estructura m_block, afegir un és igual a deplaçar-se amb una distància igual a la mida del bloc de meta-data.

Per provar que el vostre malloc funciona, creeu una llibreria dinàmica de nom malloc.so amb el fitxer malloc.c

Compileu la llibreria dinàmica de la següent forma:

```
gcc -shared -fPIC malloc.c -o malloc.so
```

Podeu feu servir aquest codi senzill de c per provar la llibreria que heu fet.

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{

  char *ptr_one;
  ptr_one = (char *)malloc(sizeof(char));
  if (ptr_one == NULL){
    printf("ERROR in MALLOC");
    return 1;
  }
  *ptr_one='C';
  printf("%c\n", *ptr_one);


  /******  Uncomment when exercise 1 it is done.

  free(ptr_one);

  ptr_five = (char *)malloc(sizeof(char)*5);
  if (ptr_five == NULL){
    printf("ERROR in MALLOC");
    return 1;
  }
  *ptr_five='C';
  ptr_five[1] = 'A';
  ptr_five[2] = 'S';
  ptr_five[3] = 'A';
  ptr_five[4] = '\0';
  printf("%s\n", ptr_five);
  free(ptr_five);
  ******/
  retun 0;
}
```

Per executar usant el vostre malloc en comptes del del sistema carregueu aquesta variable d'entorn abans d'executar el codi anterior.

```
gcc malloctest.c -o malloctest
export LD_PRELOAD=/$PWD/malloc.so
./malloctest
```

Un cop teniu això i heu exportat el LD_PRELOAD, si el vostre codi retorna un *Segmentation Fault*, vol dir que teniu algun error. És hora doncs de debuggar el vostre codi.

Problema 1a. part

Sobre aquesta implementació senzilla de malloc:

1. Implementeu la funció `void free(void *ptr)` La funció `free` allibera un bloc de memòria obtingut amb malloc. Consells:
    - Bàsicament el que ha de fer és posar l'atribut free a 1.
    - Definiu una funció que us retorni l'adreça del bloc de meta informació, donat un apuntador al bloc de dades.
    - S'ha de tenir en compte que es pot cridar a free amb un apuntador a NULL. Heu de controlar-ho.

Un cop fet aquest exercici descomenteu les instruccions *free** del fitxer de test que us hem passet.

Debugging

Per trobar errors en el vostre codi el més important a fer és poder debuggar el codi, és a dir, poder executar el codi i observar pas a a pas o en un punt concret com està evolucionant aquest, mirant el contingut de les variables. En C hi ha un debugger per defecte que és el `gdb` . Per poder debuggar un codi heu d'especificar-ho a l'hora de compilar el programa o llibreria amb l'opció `-g` . Es pot usar amb l'opció `O0` d'optimització de codi tal de no fer servir l'optimitzador de codi del compilador. En el notre cas hauríem de generar la llibreria dinàmica de la següent forma:

```
gcc -O0 -g -W -Wall -Wextra -shared -fPIC malloc.c -o malloc.so
```

Un cop teniu la llibreria preparada per ser debuggada, hem d'executar el codi que volem provar usant el debugger `gdb` .

```
$ gdb ./malloctest
(gdb) set environment LD_PRELOAD=./malloc.so
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7bd7c11 in malloc (size=5) at malloc.c:96
96          block->free = 0;
```

Un cop hem entrat el `gdb` , hem de dir que volem precargar la nostre llibreria malloc.c. Per fer això cridem `set environment LD_PRELOAD=./malloc.so` . Ara podem executar la comanda `malloctest` , fent un `run` . Si tot ha anat bé el programa acabarà normalment si hi ha algun error es pararà en algun punt i podrem explorar amb el `gdb` què està passant.

En el nostre cas ha petat perquè la implementació de la funció malloc té un bug. Per a trobar-lo poseu `list` al `gdb` . Us mostrarà la zona on ha petat i la línia. Podeu mirar el contingut d'una variable posant `p` i seguidament el nom de la variable:

```
(gdb) list
91      }
92      if (global_base){
93        last = global_base;
94        block = find_free_block(&last,size);
95        if (global_base) { // block found!
96          block->free = 0;
97        } else { // Not block found!
98          block = request_space(last,size);
99          if (!block)
100         return (NULL);
(gdb) p last
$1 = (p_block) 0x602000
```

Problema 2a. part

Sobre aquesta implementació senzilla de malloc:

1. Trobeu el "bug" i arregleu-ho.
2. Implementeu la funció `void* calloc(size_t nelem, size_t elsize)` La funció `calloc` permet reservar varis elements de memòria alhora i els deixa inicialitzats a zero. Consells:
    - Podeu usar la funció `memset` per deixar el bloc de memòria a zero.
3. Implementeu la funció `void* realloc(void *ptr, size_t size)` La funció realloc reajusta la mida d'un bloc de memoria obtingut amb malloc a una nova mida. Consells:
    - Si li passem un NULL pointer, es suposa que actua com un malloc normal i corrent.
    - Si li passem un apuntador que hem creat amb el nostre malloc i si la mida que demanem és suficient amb el bloc que ja té reservat el retornem tal qual. En cas contrari, haurem de reservar un bloc amb més espai i copiar les dades de l'antic bloc en aquest nou.
    - Podeu usar la funció `memcpy` per a copiar el contingut d'un bloc en un altre.

Un cop hàgiu implementat totes les funcions, torneu a crear la llibreria dinàmica malloc.so i proveu-la substituint la malloc.so del sistema per la vostre i proveu de fer un **ls** `bash $ ls` Si tot ha anat bé, es llistarà el contingut del directori de forma normal, però usant la vostra llibreria en comptes de la del sistema (serà més lent).

En cas contarari us donarà un **Segmentation fault.**

Millores en el codi que podeu fer:

1. Modifiqueu el codi del `malloc` i de `realloc` de tal forma que quan reutilitzem blocs aquests es puguin dividir a la mida necessària.

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

3. 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.

Entrega

Testegeu el vostre codi a fons i anoteu qualsevol bug que trobeu usant `gdb` en aquesta implementació senzilla de `malloc` .

1. Entregueu en un carpeta amb el vostre nom i cognom el següent contingut:
    1. El codi font de totes les funcions dins del fitxer malloc.c
    2. Fitxer README amb el vostre nom i cognoms.
    3. Fixer KNOWING_BUGS amb els bugs que heu trobat testajant el malloc. Apunteu amb quin programa us ha fallat i la causa del problema.
    4. Fitxer MILLORES enumerant quines són les millores que heu fet. Aquestes millores valen 3 punts de la nota final.