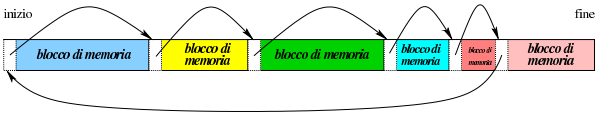
`free.c` 1950 `malloc.c` 1950 `mm.h` 1948 `mm_init.c` 1948 `mm_list.c` 1948 `realloc.c` 1950

Nel sistema in corso di realizzazione non si intende gestire la memoria in modo sofisticato; in particolare non si vogliono usare né segmenti, né pagine. In pratica, lo spazio che rimane dopo l'intervallo usato dal kernel viene gestito con una lista e sulla base di questa impostazione vengono realizzate le funzioni '...**`alloc()`**' e ***free()***.

Gestione della memoria attraverso una lista

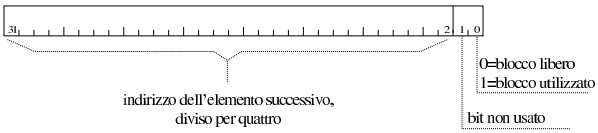
Si intende gestire l'allocazione di memoria attraverso una lista in cui l'inizio di un blocco di memoria contenga il riferimento al blocco successivo e l'indicazione se il proprio sia un blocco libero o utilizzato; pertanto, come si vede nella figura, l'ultimo blocco punta al primo.

Figura u170.1. Lista di blocchi di memoria.



L'intestazione dei blocchi di memoria, con la quale si fa riferimento al blocco successivo e si annota se il blocco (proprio) è impegnato o meno, utilizza solo 32 bit, partendo dal presupposto che i blocchi di memoria debbano essere multipli di tale valore. A tale proposito si osservi che se i blocchi di memoria sono da 4 byte, gli indirizzi sono sempre multipli di quattro, ovvero di 100_2. Pertanto, i due bit meno significativi possono essere utilizzati per altri fini.

Figura u170.2. Struttura dell'intestazione dei blocchi di memoria.



Sulla base del principio affermato e di quanto si vede nella figura, l'indirizzo effettivo del blocco successivo si determina moltiplicando per quattro il valore annotato tra il bit 2 e il bit 31 dell'intestazione. Si osservi che l'indirizzo in questione è quello dell'inizio dell'intestazione del blocco successivo, pertanto il blocco di memoria successivo inizia effettivamente dopo altri quattro byte.

Quando la memoria viene inizializzata si crea un blocco solo, la cui intestazione punta a se stessa, come si vede nella figura successiva. Da questo si comprende anche che il blocco che punta a se stesso è lungo fino alla fine dello spazio di memoria disponibile complessivamente; inoltre si intende che con questo meccanismo, molto semplice, la memoria possa essere gestita solo se è presente in modo continuo. Questa semplificazione è stata fatta volutamente per non complicare inutilmente il codice; d'altra parte si osserva che così la «memoria bassa» (quella dei primi 640 Kibyte) non venga usata affatto.

Figura u170.3. La lista al momento iniziale.

Libreria «mm.h»

Il file di intestazione '`mm.h`' descrive la struttura usata per interpretare i primi 32 bit dei blocchi di memoria (per distinguere l'indirizzo successivo dall'indicazione dello stato del blocco attuale) e dichiara due funzioni per inizializzare la memoria e per leggerne la mappa.

Listato u170.4. '`./05/include/kernel/mm.h`'

```
#ifndef _MM_H
#define _MM_H 1

#include <restrict.h>
#include <stdint.h>
#include <inttypes.h>
#include <stddef.h>
#include <stdarg.h>
#include <kernel/os.h>

//
// La dimensione di «uintptr_t» condiziona la struttura
// «mm_head_t» e la dimensione delle unità minime di memoria
// allocata. «uintptr_t» è da 32 bit, così l'immagine del
// kernel è allineata a blocchi da 32 bit e così deve essere
// anche per gli altri blocchi di memoria. Essendo i blocchi
// di memoria multipli di 32 bit, gli indirizzi sono sempre
// multipli di 4 (4 byte); pertanto, servono solo 30 bit
// per rappresentare l'indirizzo, che poi viene ottenuto
// moltiplicandolo per quattro.
// Di conseguenza, il bit meno significativo viene usato
// per annotare se il blocco di memoria è libero e il bit
// successivo non viene usato. Questo meccanismo potrebbe
// essere usato anche con un indirizzamento a 16 bit, dove
// servirebbero 15 bit per indirizzi multipli di due byte.
//
typedef struct {
    uintptr_t allocated : 1,
              filler    : 1,
              next      : 30;
} mm_head_t;

void mm_init (void);
void mm_list (void);

#endif
```

La funzione ***mm_init()*** inizializza la memoria, creando un blocco libero che la descrive completamente. Per sapere dove inizia e dove finisce la memoria disponibile, si avvale delle informazioni contenute nella variabile strutturata ***os.mem_ph***, le quali sono state inserite precedentemente dalla funzione ***kernel_memory()*** (listato u168.4)

Listato u170.5. '`./05/lib/mm/mm_init.c`'

```
#include <kernel/mm.h>
#include <stdio.h>
void
mm_init (void)
{
    uintptr_t  start = os.mem_ph.available_s;
    mm_head_t *head;
       size_t  available = os.mem_ph.available_e - os.mem_ph.available_s;
    //
    // La memoria disponibile deve essere di almeno 8 byte!
    //
    if (available < ((sizeof (mm_head_t)) * 2))
      {
        //
        // Il sistema viene fermato!
        //
        printf ("[%s] ERROR: not enough memory: %zu byte!\n",
                __func__, available);
        _Exit (0);
      }
    //
    // Predispone il nodo principale della lista.
    //
    head = (mm_head_t *) start;
    //
    // Inizializza il primo blocco, libero, che punta a se stesso,
    // essendo l'unico.
    //
    head->allocated = 0;
    head->next      = (start / (sizeof (mm_head_t)));
    //
    // Mostra come è andata.
    //
    printf ("[%s] available memory: %zu byte\n",
                __func__, available - (sizeof (mm_head_t)));
    //
    return;
}
```

La funzione ***mm_list()*** mostra la mappa della memoria gestita attraverso le funzioni '…**`alloc()`**'. Gli indirizzi che vengono forniti sono quelli di inizio dei blocchi, escludendo lo spazio utilizzato dalle intestazioni (pertanto, se l'intestazione inizia all'indirizzo ***n***, viene mostrato l'indirizzo ***n***+4).

Listato u170.6. '`./05/lib/mm/mm_list.c`'

```
#include <kernel/mm.h>
#include <stdio.h>
void mm_list (void)
{
    uintptr_t  start = os.mem_ph.available_s;
    mm_head_t *head  = (void *) start;
       size_t  actual_size;
    uintptr_t  current;
    uintptr_t  next;
    uintptr_t  up_to;
          int  counter;
    //
    // Scandisce la lista di blocchi di memoria.
    //
    counter = 2;
    while (counter)
      {
        //
        // Annota la posizione attuale e quella successiva.
        //
        current = (uintptr_t) head;
           next = head->next * (sizeof (mm_head_t));
        if (next == start)
          {
            up_to = os.mem_ph.available_e;
          }
        else
          {
            up_to = next;
          }
        //
        // Se è stato raggiunto il primo elemento, decrementa il
        // contatore di una unità. Se è già a zero, esce.
        //
        if (current == start)
          {
            counter--;
            if (counter == 0) break;
          }
        //
        // Determina la dimensione del blocco attuale.
        //
        if (current == start && next == start)
          {
            //
            // Si tratta del primo e unico elemento della lista.
            //
            actual_size = os.mem_ph.available_e - (sizeof (mm_head_t));
          }
        else
          {
            actual_size = up_to - current - (sizeof (mm_head_t));
          }
        //
        // Si mostra lo stato del blocco di memoria.
        //
        if (head->allocated)
          {
            printf ("[%s] used %08X..%08X size %08zX\n",
                    __func__,
                    current + (sizeof (mm_head_t)), up_to, actual_size);
          }
        else
          {
            printf ("[%s] free %08X..%08X size %08zX\n",
                    __func__,
                    current + (sizeof (mm_head_t)), up_to, actual_size);
          }
        //
        // Si passa alla posizione successiva.
        //
        head = (void *) next;
      }
}
```

«

La funzione ***malloc()*** esegue una scansione della mappa della memoria, alla ricerca del primo blocco di dimensione sufficiente a soddisfare la richiesta ricevuta (*first fit*). Una volta trovato, se il blocco libero è abbastanza grande, lo divide, in modo da utilizzare solo lo spazio richiesto. Gli spazi allocati sono sempre multipli della dimensione di '**`mm_head_t`**', pertanto, se necessario, si alloca uno spazio leggermente più grande del richiesto.

Listato u170.7. '`./05/lib/malloc.c`'

```
#include <stdlib.h>
#include <kernel/mm.h>
void
*malloc (size_t size)
{
    uintptr_t  start = os.mem_ph.available_s;
    mm_head_t *head  = (void *) start;
       size_t  actual_size;
    uintptr_t  current;
    uintptr_t  next;
    uintptr_t  new;
    uintptr_t  up_to;
          int  counter;
    //
    // Arrotonda in eccesso il valore di «size», in modo che sia un
    // multiplo della dimensione di «mm_head_t». Altrimenti, la
    // collocazione dei blocchi successivi può avvenire in modo
    // non allineato.
    //
    size = (size + (sizeof (mm_head_t)) -1);
    size = size / (sizeof (mm_head_t));
    size = size * (sizeof (mm_head_t));
    //
    // Cerca un blocco libero di dimensione sufficiente.
    //
    counter = 2;
    while (counter)
      {
        //
        // Annota la posizione attuale e quella successiva.
        //
        current = (uintptr_t) head;
           next = head->next * (sizeof (mm_head_t));
        //
        if (next == start)
          {
            up_to = os.mem_ph.available_e;
          }
        else
          {
            up_to = next;
          }
        //
        // Se è stato raggiunto il primo elemento, decrementa il
        // contatore di una unità. Se è già a zero, esce.
        //
        if (current == start)
          {
            counter--;
            if (counter == 0) break;
          }
        //
        // Controlla se si tratta di un blocco libero.
        //

        if (! head->allocated)
          {
            //
            // Il blocco è libero: si deve determinarne la dimensione.
            //
            if (current == start && next == start)
              {
                //
                // Si tratta del primo e unico elemento della lista.
                //
                actual_size = os.mem_ph.available_e - (sizeof (mm_head_t));
              }
            else
              {
                actual_size = up_to - current - (sizeof (mm_head_t));
              }
            //
            // Si verifica che sia capiente.
            //
            if (actual_size >= size + ((sizeof (mm_head_t)) * 2))
              {
                //
                // C'è spazio per dividere il blocco.
                //
                new = current + size + (sizeof (mm_head_t));
                //
                // Aggiorna l'intestazione attuale.
                //
                head->allocated = 1;
                head->next = new / (sizeof (mm_head_t));
                //
                // Predispone l'intestazione successiva.
                //
                head = (void *) new;
                head->allocated = 0;
                head->next = next / (sizeof (mm_head_t));
                //
                // Restituisce l'indirizzo iniziale dello spazio libero,
                // successivo all'intestazione.
                //
                return (void *) (current + (sizeof (mm_head_t)));
              }
            else if (actual_size >= size)
              {
                //
                // Il blocco va usato per intero.
                //
                head->allocated = 1;
                //
                // Restituisce l'indirizzo iniziale dello spazio libero,
                // successivo all'intestazione.
                //
                return (void *) (current + (sizeof (mm_head_t)));
              }
          }
        //
        // Il blocco è allocato, oppure è di dimensione insufficiente;
        // pertanto occorre passare alla posizione successiva.
        //
        head = (void *) next;
      }
    //
    // Essendo terminato il ciclo precedente, vuol dire
    // che non ci sono spazi disponibili.
    //
    return NULL;
}
```


La funzione ***free()*** libera il blocco di memoria indicato e poi scandisce tutti i blocchi esistenti alla ricerca di quelli liberi che sono adiacenti, per fonderli assieme. Va osservato che la funzione non verifica se il blocco da liberare esiste effettivamente e per evitare errori occorrerebbe una scansione preventiva dei blocchi, a partire dall'inizio.

Listato u170.8. '`./05/lib/free.c`'

```
#include <stdlib.h>
#include <kernel/mm.h>
#include <stdio.h>
void
free (void *ptr)
{
    mm_head_t *start        = (mm_head_t *) os.mem_ph.available_s;
    mm_head_t *head_current = ((mm_head_t *) ptr) - 1;
    mm_head_t *head_next;
    //
    // Verifica il blocco attuale e, se è possibile, lo libera.
    //
    if (head_current->allocated == 1)
      {
        head_current->allocated = 0;
      }
    else
      {
        printf ("[%s] ERROR: cannot free %08X!\n",
                __func__, (uintptr_t) head_current + (sizeof (mm_head_t)));
      }
    //
    // Scandisce i blocchi liberi, cercando quelli adiacenti per
    // allungarli. Se il blocco successivo è il primo, termina,
    // perché non può avvenire alcuna fusione con quello precedente.
    //
    head_current = start;
    while (true)
      {
        //
        // Individua il blocco successivo.
        //
        head_next = (mm_head_t *) (head_current->next * (sizeof (mm_head_t)));
        //
        // Controlla se è il primo.
        //
        if (head_next == start)
          {
            break;
          }
        //
        //
        //
        if (head_current->allocated == 0)
          {
            //
            // Controlla se si può espandere.
```

```
            //
            if (head_next->allocated == 0)
              {
                head_current->next = head_next->next;
              }
            else
              {
                head_current = head_next;
              }
          }
        else
          {
            head_current = (mm_head_t *)
                           (head_current->next * (sizeof (mm_head_t)));
          }
      }
}
```

La funzione ***realloc()***, nel caso sia richiesto un blocco più grande del precedente, si avvale di ***malloc()***, ***memcpy()*** e ***free()***.

Listato u170.9. '`./05/lib/realloc.c`'

```
#include <stdlib.h>
#include <string.h>
#include <kernel/mm.h>
#include <stdio.h>
void
*realloc (void *ptr, size_t size)
{
    uintptr_t  start     = os.mem_ph.available_s;
       size_t  actual_size;
    mm_head_t *head      = ((mm_head_t *) ptr) - 1;
    mm_head_t *head_new;
         void *ptr_new;
    //
    // Verifica che il puntatore riguardi effettivamente
    // un'area occupata.
    //
    if (! head->allocated)
      {
        printf ("[%s] ERROR: cannot re-allocate %08X that is "
                "not already allocated!", __func__, (uintptr_t) ptr);
      }
    //
    // Arrotonda in eccesso il valore di «size», in modo che sia un
    // multiplo della dimensione di «mm_head_t». Altrimenti, la
    // collocazione dei blocchi successivi può avvenire in modo
    // non allineato.
    //
    size = (size + (sizeof (mm_head_t)) -1);
    size = size / (sizeof (mm_head_t));
    size = size * (sizeof (mm_head_t));
    //
    // Determina la dimensione attuale.
    //
    if ((head->next * (sizeof (mm_head_t))) == start)
      {
        actual_size = os.mem_ph.available_e - ((uintptr_t) ptr);
      }
    else
      {
        actual_size = (head->next * (sizeof (mm_head_t))) - ((uintptr_t) ptr);
      }
    //
    // Se la dimensione richiesta è inferiore, può ridurre
    // l'estensione del blocco.
    //
    if (size == actual_size)
      {
        return ptr;
      }
    else if (size <= (actual_size - (sizeof (mm_head_t)) * 2))
      {
        //
        // Si può ricavare lo spazio libero rimanente.
        //
        head_new = (mm_head_t *) (((char *) ptr) + size);
        //
        head_new->next = head->next;
        head_new->allocated=0;
        //
        head->next = ((uintptr_t) head_new) / (sizeof (mm_head_t));
        //
        return ptr;
      }
    else if (size < actual_size)
      {
        //
        // Anche se è minore, non si può ridurre lo spazio usato
        // effettivamente.
        //
        return ptr;
      }
    else
      {
        //
        // La dimensione richiesta è maggiore.
        //
        ptr_new = malloc (size);
        //
```



```
        if (ptr_new)
          {
            //
            // Ricopia i dati nella nuova collocazione.
            //
            memcpy (ptr_new, ptr, actual_size);
            //
            // Libera la collocazione vecchia.
            //
            free (ptr);
            //
            return ptr_new;
          }
        else
          {
            return NULL;
          }
      }
}
```

Verifica del funzionamento

Per utilizzare le funzioni ***mm_init()*** e ***mm_list()*** occorre aggiornare il file '`kernel_main.c`', aggiungendo in modo particolare delle istruzioni per verificare il funzionamento delle funzioni di allocazione della memoria.

Figura u170.10. Modifiche da apportare al file '`./05/kernel/kernel_main.c`'

```
#include <kernel/kernel.h>
#include <kernel/build.h>
#include <stdio.h>
#include <kernel/gdt.h>
#include <kernel/mm.h>
#include <stdlib.h>
…
        mboot_info (info);
        mboot_show (info);
        //
        // Raccoglie i dati sulla memoria fisica.
        //
        kernel_memory (info);
        //
        // Predispone la tabella GDT.
        //
        gdt ();
        //
        // Predispone la memoria libera per l'utilizzo.
        //
        mm_init ();
        void *p0 = malloc (0x100);
        void *p1 = malloc (0x1000);
        void *p2 = malloc (0x10000);
                  malloc (0x100000);
        p0 = realloc (p0, 0x1000);
        p1 = realloc (p1, 0x100);
        p2 = realloc (p2, 0x100000);
        free (p1);
        mm_list ();
…
```

Dopo avere ricompilato, riavviando la simulazione si deve ottenere una schermata simile a quella seguente, dove si può osservare la mappa della memoria alla fine delle operazioni di allocazione e riallocazione eseguite:

```
 05 20070820133728
 [mboot_show] flags: 00000000000000000000011111100111 mlow: 027F mhigh: 00007BC0
 [mboot_show] bootdev: 00FFFFFF cmdline: "(fd0)/kernel"
 [kernel_memory_show] kernel 00100000..0010C65C  avail. 0010C65C..01EF0000
 [kernel_memory_show] text   00100000..001039D8  rodata 001039D8..00103CD4
 [kernel_memory_show] data   00103CD4..00103CD4  bss    00103CE0..0010C65C
 [kernel_memory_show] limit  00001EF0
 [gdt_print] base: 0x0010BD28 limit: 0x0017
 [gdt_print] 0 00000000000000000000000000000000 00000000100000000010000000000000
 [gdt_print] 1 00000000000000000000111011110000 00000000110000001001101000000000
 [gdt_print] 2 00000000000000000000111011110000 00000000110000001001001000000000
 [mm_init] available memory: 31340960 byte
 [mm_list] free 0010C660..0010D764 size 00001104
 [mm_list] used 0010D768..0011D768 size 00010000
 [mm_list] used 0011D76C..0021D76C size 00100000
 [mm_list] used 0021D770..0021E770 size 00001000
 [mm_list] used 0021E774..0031E774 size 00100000
 [mm_list] free 0021E778..01EF0000 size 01BD1888
 [kernel_main] system halted
```

Nelle figure successive viene mostrato, schematicamente, ciò che accade. La prima figura mostra lo stato della lista della memoria dopo le prime quattro allocazioni; le figure successive mostrano le riallocazioni che vengono fatte dopo, una a una, per finire con la liberazione della zona associata alla variabile ***p1***. Purtroppo, nelle figure non è stato possibile usare delle proporzioni realistiche.

Figura u170.12. La lista della memoria dopo le prime quattro allocazioni.

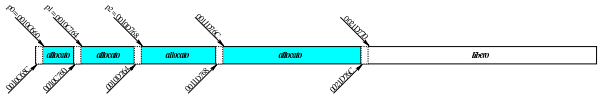


Figura u170.13. La lista della memoria dopo la riallocazione di ***p0***.

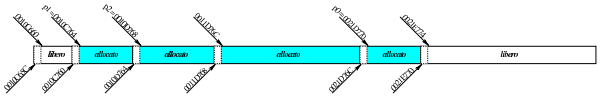


Figura u170.14. La lista della memoria dopo la riallocazione di ***p1***.

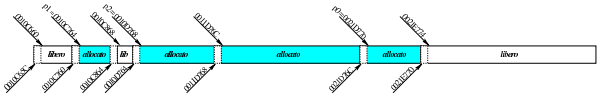


Figura u170.15. La lista della memoria dopo la riallocazione di ***p2***.

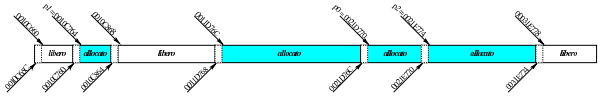


Figura u170.16. La lista della memoria dopo l'eliminazione di ***p1***.