PROGRAMMAZIONE DI SISTEMA

A.A. 2022/2023

S308786 OLIVA MATTIA

S319103 SEFA ENDRI

Politecnico di Torino

IMPLEMENTAZIONE DEGLI ALGORITMI

BUDDY SYSTEM

PER

ALLOCAZIONE E DEALLOCAZIONE

DELLA

MEMORIA

IN

MENTOS

BUDDY SYSTEM IN MENTOS: C'È O NON C'È?

- Il buddy system in MentOS è la strategia d'allocazione della memoria fisica usata dal kernel. Risulta quindi essenziale per il funzionamento del sistema.
- La parte di codice inerente alle due funzioni principali per la realizzazione del buddy system, **bb_alloc_pages** e **bb_free_pages**, è praticamente vuota: delle due funzioni sono presenti solo la dichiarazione (*inc/mem/buddysystem.h*) e uno scheletro della funzione. Eppure, una volta compilato con le opzioni di default, il sistema funziona.
- Ciò è possibile grazie all'uso di una libreria statica, *buddysystem.a*, che viene linkata al momento della compilazione del kernel. Alla riga 191 del file *CMakeLists.txt* usando il comando *target_link_libraries*

```
# ============================================================================
# Add kernel library.
if(USE_BUDDY_SYSTEM)
    # Add the library with `buddysystem.c`.
    add_library(${KERNEL_NAME} ${KERNEL_SOURCES} src/mem/buddysystem.c)
else(USE_BUDDY_SYSTEM)
    # Add the library without `buddysystem.c`.
    add_library(${KERNEL_NAME} ${KERNEL_SOURCES})
    # Link the exercise libraries.
    target_link_libraries(${KERNEL_NAME} ${BUDDY_SYSTEM_FILE})
endif(USE_BUDDY_SYSTEM)
```

- A lato il codice che gestisce la compilazione condizionale del codice di gestione della memoria: l'opzione di default ignora il codice del kernel e linka la libreria statica.

È possibile effettuare la compilazione condizionale del kernel mediante il comando *ccmake*.

```
cd build

cmake ..

ccmake ..
```

Si aprirà un menù con la possibilità di selezionare diverse opzioni per diverse caratteristiche (es enable/disable debug, selezione dell'algoritmo di scheduling, etc).

L'opzione da cambiare per compilare il codice del kernel invece di utilizzare la libreria fornita è `ENABLE_BUDDY_SYSTEM` (da settare su ON)

- Ovviamente, selezionare l'opzione e compilare il codice presente nel sistema di base comporta il pressoché immediato crash del kernel, non appena si tenti di effettuare la prima allocazione di memoria. Questo perché, come abbiamo visto, le due funzioni sono di fatto vuote.
- Per poter implementare queste due routine essenziali, si è dovuto, nell'ordine:
  1. Comprendere gli algoritmi di allocazione e rilascio della memoria, propri della strategia buddy system
  2. Analizzare e comprendere le strutture dati scelte dai progettisti di MentOS per il buddy system così da poter operare su di esse. Sì è scelto di mantenere queste strutture in quanto la loro modifica avrebbe comportato la necessità di cambiare altre parti di codice. Per ridurre al minimo la possibilità di bug inattesi, abbiamo quindi optato per non modificarle
  3. Implementare gli algoritmi visti basandosi sulle strutture dati fornite e le funzioni di gestione delle stesse.
- Tutte le modifiche illustrate di seguito e il codice citato sono situati in *inc/mem/buddysytem.h* o *src/mem/buddysystem.c*, salvo diversamente specificato.

GLI ALGORITMI: ALLOCAZIONE

Il buddy system opera tenendo traccia di una serie di liste di blocchi liberi (in MentOS sono 11 liste, con blocchi da 1 a 1024 frames).

- Quando vengono richieste delle pagine (frames) si cerca nella prima lista in grado di soddisfare la richiesta, ovvero nella prima lista di ordine sufficientemente grande. Se non ci sono blocchi liberi nella lista, si passa a quella di ordine superiore, fino a trovare un blocco libero.
- Una volta trovato, si procede a ridurlo di dimensioni fino ad arrivare alla minima potenza di due in grado di soddisfare la richiesta.
- Ad ogni riduzione (che consiste in una divisione a metà del blocco) si produce un buddy, cioè un blocco di frame liberi, che dovrà essere inserito nella lista di ordine corretto

Una rappresentazione in pseudocodice della ricerca del primo blocco adatto (e rimozione dello stesso dalla sua lista), dati un vettore di liste di blocchi liberi **fr** e l'ordine richiesto **ro**

```
1.  fo = ro
2.  while fo < MAX_ORDER do
3.      if (!empty(fr[fo])) then
4.          goto FOUND
5.      end if
6.      fo = fo + 1
7.  end while
8.  return NULL
9.  FOUND:
10. block = getFirstBlock(fr[fo]) //restituisce il primo blocco della lista di blocchi liberi di ordine fo
11. removeBlock(fr[fo], block) //elimina il blocco trovato dalla lista in cui si trovava
```

1. **while** fo > ro do
2. fo = fo - 1 //diminuisco l'ordine, perché adesso il blocco è dimezzato
3. free_block = splitRight(block) //restituisce la metà destra del blocco
4. addBlock(fr[fo], free_block) //aggiungo la metà dx alla
5. block = splitLeft(block) //restituisce la metà sinistra del blocco
6. end **while**

- Ad ogni iterazione, finché non si raggiunge la dimensione minima richiesta (a.k.a. finché non si raggiunge l'ordine ro), si opera sulla metà sinistra del blocco, salvando quella destra nella lista di blocchi liberi di ordine **fo**

GLI ALGORITMI: DEALLOCAZIONE

- L'algoritmo utilizzato per la liberazione della memoria consiste nel rintracciare ad ogni iterazione il buddy del blocco attuale, dopo aver posto come liberi i frame del blocco.
- Se lo si trova, si effettua un merge dei due blocchi ottenendone uno di ordine superiore (e quindi di dimensione doppia). Per essere un buddy, un blocco deve essere adiacente al blocco in esame, dev'essere libero e della medesima dimensione (ordine).
- L'algoritmo termina quando non ci sono più buddy validi o si è raggiunto l'ordine massimo per il sistema

Dati l'ordine **ord,** un blocco da liberare **bl** ed il vettore di liste di blocchi liberi **fr**:

```
1.  while (ord < MAX_ORDER – 1) do
2.      buddy = getBuddy(bl, ord)  //ottengo il blocco adiacente a bl
3.      if (!free(buddy) | order(buddy) != ord) then
4.          Break //se non ho trovato un buddy valido, esco dal ciclo
5.      end if
6.      removeBlock(fr[ord], buddy) //rimuovo il buddy dalla lista perché ord aumenterà
7.      if (buddy < bl) then
8.          bl = buddy //prendo il blocco più a sx dei due: unisco i due blocchi
9.      end if
10.     ord = ord + 1 //aumento l'ordine
11. end while
12. addBlock(f[o], b) //aggiungo il blocco alla free list dell'ordine più grande che ho trovato
```

LE STRUTTURE DATI

- MentOS è un sistema operativo didattico; di conseguenza, pur non risultando completo, presenta già una struttura ben definita che dovrebbe essere usata come punto di partenza per l'implementazione delle funzionalità mancanti. Per quanto riguarda l'implementazione del buddy system, sono già presenti delle strutture dati utili e la funzione d'inizializzazione di tali strutture.
- Le tre strutture dati, inizializzate dalla funzione *buddy_system_init* sono:
  - *bb_page_t*
  - *bb_free_area_t*
  - *bb_instance_t*

BB_PAGE_T

```
/// The base structure representing a bb page
typedef struct bb_page_t {
    /// The flags of the page.
    volatile unsigned long flags;
    /// The current page order.
    uint32_t order;
    /// Keep track of where the page is located.
    union {
        /// The page siblings when not allocated.
        list_head siblings;
        /// The cache list pointer when allocated but on cache.
        list_head cache;
    } location;
} bb_page_t;
```

- È la minima unità gestita dal buddy system, e corrisponde ad uno dei frame della memoria principale.
- Al suo interno tiene traccia dell'ordine del blocco a cui appartiene la pagina
- *Flags* contiene i bit che identificano se la pagina è **libera** e/o è una **pagina root**

- L'union *location* contiene la successiva root page della lista di blocchi liberi (se non allocata) oppure il puntatore alle altre pagine in cache (se allocata ed utilizzata in cache).

BB_FREE_AREA_T

```
/// @brief Buddy system descriptor: collection of free page blocks.
/// Each block represents 2^k free contiguous page.
typedef struct bb_free_area_t {
    /// free_list collectes the first page descriptors of a blocks of 2^k frames
    list_head free_list;
    /// nr_free specifies the number of blocks of free pages.
    int nr_free;
} bb_free_area_t;
```

- Struttura dati utilizzata per rappresentare una delle liste di blocchi liberi
- Tiene traccia del numero di blocchi di frame liberi disponibili (non del numero totale di frame)

- La *free_list* punta al primo blocco libero della lista. In verità, i nodi sono solo la prima pagina di ogni blocco (*bb_page_t* con il flag *root page* settato), tra loro collegate mediante il campo *location.siblings*.

BB_INSTANCE_T

```
/// that represents a memory area managed by the buddy system
typedef struct bb_instance_t {
    /// Name of this bb instance
    const char *name;
    /// List of buddy system pages grouped by level.
    bb_free_area_t free_area[MAX_BUDDYSYSTEM_GFP_ORDER];
    /// Pointer to start of free pages cache
    list_head free_pages_cache_list;
    /// Size of the current cache
    unsigned long free_pages_cache_size;
    /// Buddysystem instance size in number of pages.
    unsigned long size;
    /// Address of the first managed page
    bb_page_t *base_page;
    /// Size of the (padded) wrapper page structure
    unsigned long pgs_size;
    /// Offset of the bb_page_t struct from the start of the whole structure
    unsigned long bbpg_offset;
} bb_instance_t;
```

- Rappresenta una delle zone gestite dal buddy system in cui è divisa la memoria in MentOS
- Il vettore *free_area* contiene, per ogni entry, il puntatore alla lista di blocchi liberi (*bb_free_list_t*) di ordine pari alla posizione nel vettore.
- Tra le varie informazioni di cui tiene traccia, sono particolarmente importanti per l'implementazione effettuata l'indirizzo della prima pagina gestita e la dimensione della struttura rappresentante le pagine

IMPLEMENTAZIONE: BB_ALLOC_PAGES

```
bb_page_t *bb_alloc_pages(bb_instance_t *instance, unsigned int order)
{
    bb_page_t *page      = NULL;
    bb_free_area_t *area = NULL;

    // Cyclic search through each list for an available block,
    // starting with the list for the requested order and
    // continuing if necessary to larger orders.
    unsigned int current_order;
    for (current_order = order; current_order < MAX_BUDDYSYSTEM_GFP_ORDER; current_order++){
        area = __get_area_of_order(instance, current_order);
        if(!list_head_empty(&area->free_list)){
            goto block_found;
        }
    }
    // No suitable free block has been found.
    return NULL;

block_found:
```

- La funzione richiede come parametri un'istanza del buddy system e l'ordine del blocco di pagine che vogliamo ottenere. Restituisce la prima pagina del blocco
- Nell'immagine possiamo vedere l'implementazione della ricerca di un blocco libero sufficientemente grande. La *get_area_of_order* si limita a restituire la *bb_free_area* in posizione *current_order* nel vettore *free_area* dell'istanza

- Abbiamo utilizzato il costrutto **goto** per mantenere il codice il più simile possibile all'algoritmo illustrato precedentemente. Sarebbe stato possibile ottenere il medesimo funzionamento con l'uso di una variabile di controllo e una modifica nella condizione di ciclo

```
block_found:
    // Get a block of pages from the found free_area_t. Here we have to manage
    // pages. Recall, free_area_t collects the first page_t of each free block
    // of 2^order contiguous page frames.
    page = list_entry(area->free_list.next, bb_page_t, location.siblings);
    //remove the page from the list of area's free pages
    list_head_remove(&page->location.siblings);

    //reduce the number of free blocks of the area
    area->nr_free--;

    //check that the page is actually a root one and free
    assert(__bb_test_flag(page, FREE_PAGE) && __bb_test_flag(page, ROOT_PAGE));

    //set the page as not free
    __bb_clear_flag(page, FREE_PAGE);

    //while we are above the order required, we take the buddy and put it in the lower area as free
    unsigned long size = 1UL << current_order;
```

- A questo punto si procede a recuperare la prima pagina della lista (area) mediante la funzione *list_entry* (che in realtà è una #define usato come alias per una *container_of(…)*) e successivamente si rimuove la pagina trovata dalla lista con la *list_head_remove* e si decrementa il numero di blocchi liberi per l'area
- L'assert serva a verificare che la pagina trovata sia effettivamente valida (libera e root page, cioè la prima di un blocco)

- Dopo aver segnato la pagina come non più libera mediante la funzione *__bb_clear_flag* (che a sua volta chiama una funzione atomica *clear_bit*) andiamo a calcolare la dimensione attuale del blocco

```
    while(current_order > order){
        //new order, we act on the lower order to insert the buddy
        current_order--;
        area = __get_area_of_order(instance, current_order);
        //changed order, the size is halved
        size = size/2;
        //get the buddy, that is current_order pages after the root
        bb_page_t *buddy = __get_page_from_base(instance, page, size);
        //check that the buddy is a valid one
        assert(__bb_test_flag(buddy, FREE_PAGE) && !__bb_test_flag(buddy, ROOT_PAGE));
        //set the buddy as correct order, as a root and add it to the current area's free list
        buddy->order = current_order;
        __bb_set_flag(buddy, ROOT_PAGE);
        list_head_insert_after(&buddy->location.siblings, &area->free_list);
        //increase the current area free blocks
        area->nr_free++;
    }
    //set the page order
    page->order = order;
    return page;
}
```

- Dopo aver ridotto il blocco alla minima dimensione possibile, aggiorniamo l'ordine della sua prima pagina e la restituiamo

- A questo punto riduciamo le dimensioni del blocco trovato fino ad arrivare all'ordine richiesto. Ad ogni iterazione:
  1. Riduciamo l'ordine, otteniamo l'area di quell'ordine e dimezziamo il valore della dimensione del blocco, *size*
  2. Recuperiamo la prima pagina del blocco buddy (quella a *size* pagine dalla prima) tramite la funzione *__get_page_from_base* e verifichiamo sia un buddy valido, cioè libero e non ancora root.
  3. Aggiorniamo l'ordine del buddy trovato, lo marchiamo come root page e lo inseriamo nella free list dell'area, ricordandoci di aumentare il numero di blocchi disponibili per quell'area

IMPLEMENTAZIONE: BB_FREE_PAGES

```
void bb_free_pages(bb_instance_t *instance, bb_page_t *page)
{
    bb_free_area_t *area = NULL;

    // Take the first page descriptor of the zone.
    bb_page_t *base = instance->base_page;
    // Take the page frame index of page compared to the zone.
    unsigned long page_idx = __get_page_range(instance, base, page);
    // Set the page freed, but do not set the private
    // field because we want to try to merge.
    unsigned int order = page->order;

    // Check that the page is used, or that it is not a root page.
    if (__bb_test_flag(page, FREE_PAGE) || !__bb_test_flag(page, ROOT_PAGE)) {
        kernel_panic("Double deallocation in buddy system!");
    }

    //mark as free
    __bb_set_flag(page, FREE_PAGE);
```

- La funzione prende come parametri l'istanza del buddy system su cui si opera e la prima pagina del blocco da liberare
- Si comincia recuperando la prima pagina gestita dall'istanza e l'indice (relativo ad essa) della prima pagina da liberare, usando la ___get_page_range
- Si salva l'ordine del blocco da liberare e si verifica che la pagina non sia già stata deallocata. In caso negativo, si può procedere settando la root page come libera (anche qui andando a richiamare una funzione atomica)

```
while (order < MAX_BUDDYSYSTEM_GFP_ORDER -1){
    //get area in which we operate
    area = __get_area_of_order(instance, order);
    //get new page because we could have a new address in case the buddy is on the lower adddr
    page = __get_page_from_base(instance, base, page_idx);
    //get buddy
    unsigned long buddy_idx = __get_buddy_at_index(page_idx, order);
    bb_page_t *buddy = __get_page_from_base(instance, base, buddy_idx);
    //if the page is not a buddy (not free and/or not of the same order), stop
    if(!__page_is_buddy(buddy, order)){
        break;
    }
    //remove the buddy from the area
    list_head_remove(&buddy->location.siblings);
    area->nr_free--;
    //clear page and buddy root flag
    __bb_clear_flag(buddy, ROOT_PAGE);
    __bb_clear_flag(page, ROOT_PAGE);
    //page_idx becomes the lower address between the two
    page_idx &= buddy_idx;
    order++;
}
```

- Di fatto quello che otteniamo in questo ciclo è un merge del blocco a ed il suo buddy, ripetuto fino ad esaurire i buddies o al raggiungimento dell'ordine massimo

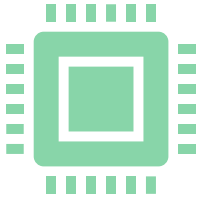
- Eseguiamo un ciclo nel quale:
  1. Recuperiamo l'area dell'ordine attuale e la pagina situata all'indirizzo trovato
  2. Recuperiamo il buddy (quindi la root page del blocco contiguo a quello in esame) ed il suo indice e verifichiamo che tale buddy sia valido, cioè che abbia lo stesso ordine e sia libero. In caso contrario, usciamo dal ciclo
  3. Rimuoviamo il buddy trovato dalla *free list* dell'area, decrementando inoltre il contatore di blocchi liberi della stessa
  4. Azzeriamo il flag root sia del buddy sia della page (in alternativa si può vedere quale dei due diverrà la prima pagina del blocco unito e azzerare solo l'altro)
  5. Il nuovo indice della prima pagina è il più piccolo tra i due indici. Passiamo all'ordine superiore

```
    //get the final block and set the first page as free and root
    page = __get_page_from_base(instance, base, page_idx);
    __bb_set_flag(page, ROOT_PAGE);

    //set page order
    page->order = order;


    //insert in the first position of the free list
    area = __get_area_of_order(instance, order);
    list_head_insert_after(&page->location.siblings, &area->free_list);

}
```

- Una volta terminato il ciclo, indipendentemente dalla condizione di terminazione, abbiamo l'indice della prima pagina di un blocco di 2^{order} frames liberi
- Recuperiamo quindi la prima pagina a partire dall'indice e la settiamo come root e ne aggiorniamo l'ordine

- Come ultima operazione recuperiamo l'area dell'ordine corrente dall'istanza e aggiungiamo il nuovo blocco in cima alla sua *free list*

SOURCES & CREDITS



Sources:

MentOS: https://mentos-team.github.io/doc/doxygen/index.html

Linux Kernel: "Understanding the Linux Kernel, Third Edition 3rd Edition", M. Cesati, D. P. Bovet

Credits and Thanks:

All of the previous

Copyright Licence:

CONTATTI:

ENDRI.SEFA@STUDENTI.POLITO.IT

MATTIA.OLIVA@STUDENTI.POLITO.IT