

PROGRAMMAZIONE DI  
SISTEMA

A.A. 2022/2023

S308786 OLIVA MATTIA

S319103 SEFA ENDRI



Politecnico  
di Torino

# GESTIONE DELLA MEMORIA IN MENTOS

# INDICE

- Registri dell'architettura
- Gestione della memoria fisica
- Gestione della memoria virtuale
- Implementazione in OSI6I



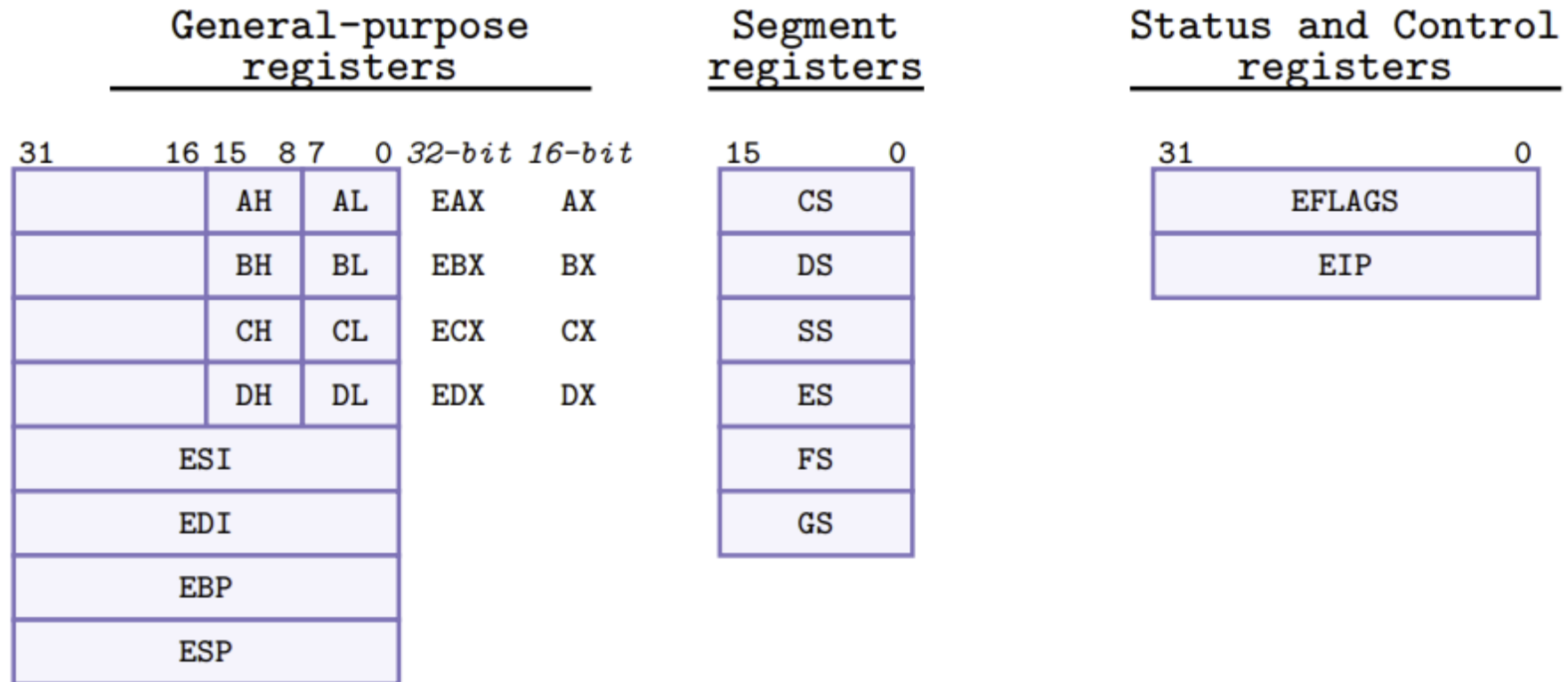
# REGISTRI DELL'ARCHITETTURA

- REGISTRI

# REGISTRI

- MentOS è pensato per dispositivi Intel x86. Le CPU di questa famiglia presentano 16 registri:
  - 8 General-purpose Data Registers a 32 bit;
  - 6 Segment Registers a 16 bit;
  - 2 Status e Control Register a 32 bit (1 che funge da program counter e 1 che mantiene una serie di flag di controllo per il sistema).

- Rappresentazione schematica dei registri dell'architettura x86:



# GESTIONE DELLA MEMORIA FISICA

- DESCRITTORE DI PAGINA
- ZONE
- ZONED PAGE FRAME ALLOCATOR
  - ALLOC\_PAGES
  - FREE\_PAGES
  - WITH CACHE
- BUDDY SYSTEM
- SLAB SYSTEM

# GESTIONE DELLA MEMORIA FISICA

- La dimensione della ram è di 4GB ( $2^{32}$  bits).
- La dimensione dei frame fisici (l'unità di base per la gestione della memoria, per il kernel) è di 4KB, quella standard adottata da Linux, garantendo quindi un totale di  $2^{20} = 2\text{M}$  frames.

## GMF – DESCRITTORE DI PAGINA

- Il sistema operativo alloca la memoria disponibile in multipli di frame.
- Deve quindi tener traccia delle informazioni legate al singolo frame, tra cui:
  - Se esso è libero o occupato;
  - Se contiene codice/strutture dati del kernel;
  - Se appartiene o meno ad un processo utente.

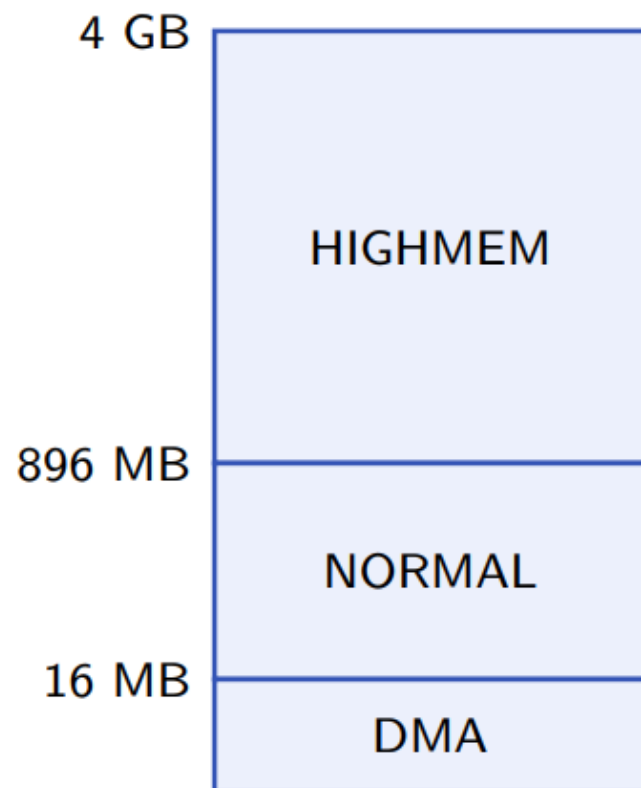


Al contrario di quanto avviene in OS/61, che nella versione base contiene un gestore di memoria che effettua unicamente allocazione contigua di memoria reale, senza mai rilasciarla, e quindi non tiene traccia dei frame assegnati (se non come “ultimo indirizzo assegnato”), in MentOS queste informazioni vengono gestite in una struttura dati, *page\_t* (definita in *inc/mem/zone\_allocator.h*).

```
typedef struct page_t {
    /// @brief Array of flags encoding also the zone number to which the page
    /// frame belongs.
    unsigned long flags;
    /// @brief Page frame's reference counter. 0 free, 1 used, 2+ copy on write
    atomic_t count;
    /// @brief Buddy system page definition
    bb_page_t bbpage;
    /// @brief Contains pointers to the slabs doubly linked list of pages.
    list_head slabs;
    // altro...
} page_t;
```

# GMF - ZONE

- In MentOS la memoria fisica viene suddivisa in 3 intervalli d'indirizzi, detti zone:



- **HIGHMEM** (>896 MB): contiene i frame non mappati permanentemente a memoria kernel. È la memoria utilizzabile per i processi utente. Accessibile tramite Page Tables mapping;
- **NORMAL** (16-896 MB): contiene la memoria assegnata al kernel in maniera permanente. Viene inizializzata all'avvio con l'immagine del kernel (e spazio extra);
- **DMA** (<16 MB): contiene gli indirizzi di memoria utilizzabili per il *Direct Memory Access* (Nota: nella versione più recente di MentOS, questa zona è inclusa in NORMAL).

- La struttura dati usata dal kernel per gestire le informazioni su una data zona (situata in *inc/mem/zone\_allocator.h*):

```
typedef struct zone_t {  
    unsigned long free_pages; // Number of free pages in the zone  
    bb_instance_t buddy_system; // Buddy system managing this zone  
    page_t *zone_mem_map; // Pointer to first page descriptor of the zone.  
    uint32_t zone_start_pfn; // Index of the first page frame of the zone.  
    char *name; // Zone's name.  
    unsigned long size; // Zone's size in number of pages.  
} zone_t;
```

```

// Get the starting address of the physical pages at the end of the modules.
uint32_t kernel_phy_page_start = __align_rup(boot_info.module_end, PAGE_SIZE);
// Get the starting address of the virtual pages.
uint32_t kernel_virt_page_start = __align_rdown(kernel_virt_low, PAGE_SIZE);
// Compute the absolute offset of the first virtual page, by subtracting
// the starting address of the virtual pages and the lowest virtual address
// of the kernel.
uint32_t kernel_page_offset = kernel_virt_page_start - kernel_virt_low;
// If we add the offset we computed earlier to the physical address where
// the modules ends, we obtain the starting address of the physical memory.
boot_info.kernel_phy_start = kernel_phy_page_start + kernel_page_offset;
// The ending address of the physical memory is just the start plus the
// size of the kernel (virt_high - virt_low).
boot_info.kernel_phy_end = boot_info.kernel_phy_start + boot_info.kernel_size;
boot_info.lowmem_phy_start = __align_rup(boot_info.kernel_phy_end, PAGE_SIZE);
boot_info.lowmem_phy_end = 896 * 1024 * 1024; // 896 MB of low memory max
uint32_t lowmem_size = boot_info.lowmem_phy_end - boot_info.lowmem_phy_start;
boot_info.lowmem_start = __align_rup(boot_info.kernel_end, PAGE_SIZE);
boot_info.lowmem_end = boot_info.lowmem_start + lowmem_size;
boot_info.highmem_phy_start = boot_info.lowmem_phy_end;
boot_info.highmem_phy_end = header->mem_upper * 1024;
boot_info.stack_end = boot_info.lowmem_end;

```

- Al momento del boot, in maniera simile alla *ram\_bootstrap* di OS/61, viene definita la suddivisione della memoria.
- In MentOS i vari calcoli e chiamate sono effettuati direttamente all'interno della funzione *boot\_main* (situata in *src/boot.c*), che corrisponde all'entry point del bootloader.

```

/// @brief Entry point of the kernel.
/// @param boot_informations Information concerning the boot.
/// @return The exit status of the kernel.
int kmain(boot_info_t *boot_informations)
{
    pr_notice("Booting...\n");
    // Make a copy for when paging is enabled
    boot_info = *boot_informations;

    //altro...
    //=====
    pr_notice("Initialize physical memory manager...\n");
    printf("Initialize physical memory manager...");
    if (!pmmngr_init(&boot_info)) {
        print_fail();
        return 1;
    }
    print_ok();
    //altro...
    for (;;) {}
    // We should not be here.
    pr_emerg("Dear developer, we have to talk...\n");
    return 1;
}

```

- L'inizializzazione vera e propria del gestore della memoria fisica avviene all'interno del *kmain* grazie alla funzione *pmmngr\_init*.
- (in *src/mem/zone\_allocator.c*) che prende come parametro le informazioni di boot.

# GMF - ZONED PAGE FRAME ALLOCATOR

- Lo *Zone Allocator* è il sottosistema kernel che si occupa dell'allocazione contigua dei frame (e deallocazione, cosa che in OS/61 base non avviene).
- Mette a disposizione due funzioni per gestire la richiesta ed il rilascio dei frames sia per processi kernel sia per quelli utente: *alloc\_pages* e *free\_pages*.

## GMF – ALLOC PAGES

- **page\_t \* alloc\_pages(zone, order):**
  - Usata per richiedere  $2^{\text{order}}$  frame contigui ad una data zona. Restituisce la prima pagina del blocco di  $2^{\text{order}}$  oppure NULL se non ha successo;
  - Può essere visto come il corrispettivo di *getppages* in OS/61. Infatti, come *getppages* chiamava la *ram\_stealmem* per ottenere effettivamente i frames, qui viene invocata la *bb\_alloc\_pages* (definita in *src/mem/buddysystem.c*), e che adotta la strategia *buddy system*.

```

page_t *_alloc_pages(gfp_t gfp_mask, uint32_t order)
{
    uint32_t block_size = 1UL << order;

    zone_t *zone = get_zone_from_flags(gfp_mask);
    page_t *page = NULL;

    // Search for a block of page frames by using the BuddySystem.
    page = PG_FROM_BBSTRUCT(bb_alloc_pages(&zone->buddy_system, order), page_t, bbpage);

    // Set page counters
    for (int i = 0; i < block_size; i++) {
        set_page_count(&page[i], 1);
    }

    assert(page && "Cannot allocate pages.");

    // Decrement the number of pages in the zone.
    if (page) {
        zone->free_pages -= block_size;
    }

    return page;
}

```

- `get_zone_from_flags` restituisce la zona corrispondente al flag passato.
- `PG_FROM_BBSTRUCT(bbstruct, page, element)` restituisce l'indirizzo di un dato elemento con un certo tipo di pagina, a partire dalla *bbstruct* fornita.
- `set_page_count(p,v)` è definita come una chiamata alla funzione atomica `atomic_set(&(p)->count, v)`.



## GMF – FREE PAGES

- **void \_\_free\_pages(page\_t \*page):**
  - Funzione utilizzata per rilasciare 2<sup>order</sup> frame contigui in una data zona.
  - Questa funzione non presenta un corrispettivo in OS/6I in quanto lì di base la memoria allocata non viene rilasciata.
  - Come alloc\_pages, utilizza una funzione del buddy system, bb\_free\_pages (definita in src/mem/buddysystem.c).

```

void __free_pages(page_t *page)
{
    zone_t *zone = get_zone_from_page(page);
    assert(zone && "Page is over memory size.");

    assert(zone->zone_mem_map <= page && "Page is below the selected zone!");

    uint32_t order      = page->bbpage.order;
    uint32_t block_size = 1UL << order;

    for (int i = 0; i < block_size; i++) {
        set_page_count(&page[i], 0);
    }

    bb_free_pages(&zone->buddy_system, &page->bbpage);

    zone->free_pages += block_size;
#ifdef 0
    pr_debug("BS-F: (page: %p order: %d)\n", page, order);
#endif
    //buddy_system_dump(&zone->buddy_system);
}

```


- `get_zone_from_page` restituisce la zona a cui appartiene la pagina.
- `set_page_count(p,v)` è definita come una chiamata alla funzione atomica `atomic_set(&(p)->count, v)`.

## GMF – ZONE ALLOCATOR WITH CACHE

- MentOS mette anche a disposizione un semplice sistema di cache, sfruttando la zona NORMAL della memoria, che può essere indirizzata direttamente.
- Per la gestione di questa particolare memoria (in realtà una zona di memoria particolare), il sistema mette a disposizione delle funzioni equivalenti a quelle precedentemente viste, *alloc\_page\_cached* e *free\_page\_cached*. Anche questi vanno poi ad invocare le medesime funzioni del buddy system.

## GMF – BUDDY SYSTEM

- Il *buddy system* è una strategia robusta ed efficiente per l'allocazione di gruppi contigui di frame (in potenze di due). È la strategia standard utilizzata dai sistemi Linux.
- In MentOS tutti i frame liberi sono raggruppati in 11 liste, ce contengono rispettivamente blocchi di 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, e 1024 frame contigui.
- La richiesta minima è di 1 frame singolo, quindi 4KB.
- In MentOS il buddy system non è implementato direttamente nel codice del kernel, bensì viene utilizzato un file binario (*buddysystem.a*) che viene linkato come se fosse una libreria statica al momento della compilazione del sistema.

- 
- L'algoritmo di base del buddy system può essere suddiviso in tre parti:
    1. Ricerca di un blocco grande a sufficienza da soddisfare la richiesta;
    2. Rimozione di un blocco di frame liberi dalla lista;
    3. Separare il blocco a metà ricorsivamente finché non si ottiene la minima potenza di due in grado di soddisfare la richiesta. Ogni divisione genera due metà, quella su cui si ricorre ed un'altra (il *buddy*) che verrà aggiunta alla lista di blocchi liberi della dimensione corrispondente.

## GMF – SLAB SYSTEM

- I sistemi Linux adottano la strategia *slab system* per la gestione delle strutture dati del kernel.
- MentOS mette quindi a disposizione delle vari metodi e strutture dati (in *src/mem/slab.c*) per la creazione, gestione e distruzione di queste slab.
- Quando lo *slab allocator* necessita di più memoria, la ottiene dal buddy allocator visto precedentemente.

```
typedef struct kmem_cache_t {
    /// Handler for placing it inside a lists of caches.
    list_head cache_list;
    /// Name of the cache.
    const char *name;
    /// Size of the cache.
    unsigned int size;
    /// Size of the objects contained in the cache.
    unsigned int object_size;
    /// Alignment requirement of the type of objects.
    unsigned int align;
    /// The total number of slabs.
    unsigned int total_num;
    /// The number of free slabs.
    unsigned int free_num;
    /// The Get Free Pages (GFP) flags.
    slab_flags_t flags;
    /// The order for getting free pages.
    unsigned int gfp_order;
    /// Constructor for the elements.
    void (*ctor)(void *);
    /// Destructor for the elements.
    void (*dtor)(void *);
    /// Handler for the full slabs list.
    list_head slabs_full;
    /// Handler for the partial slabs list.
    list_head slabs_partial;
    /// Handler for the free slabs list.
    list_head slabs_free;
} kmem_cache_t;
```

- La struttura dati che definisce una cache usata dallo slab allocator.
- Nota: in OSI6I non c'è nulla di simile, in quanto non esiste un allocatore dedicato per le strutture del kernel.



# GESTIONE DELLA MEMORIA VIRTUALE

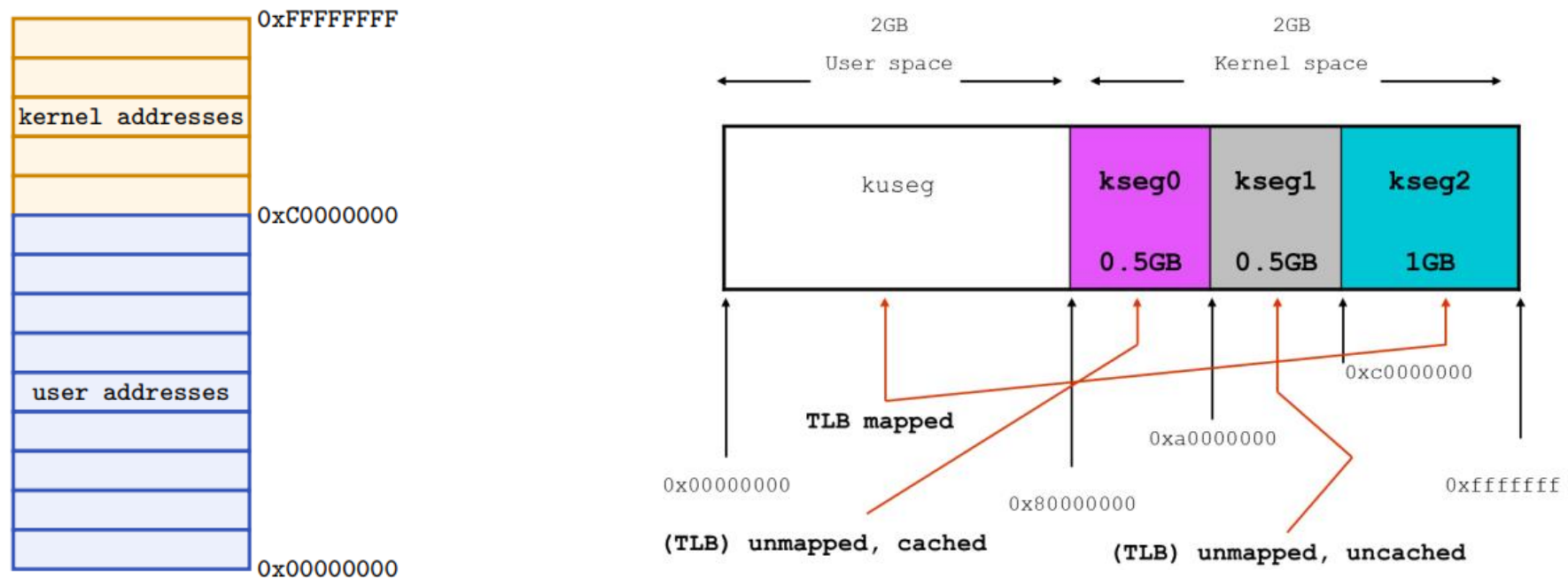
- MMU E TLB
- DEMAND PAGING
- PAGE TABLE
- MEMORY DESCRIPTORS
- SEGMENT DESCRIPTORS



# GESTIONE DELLA MEMORIA VIRTUALE

- Il kernel utilizza la Memoria Virtuale per mappare indirizzi logici a fisici. Ciò fornisce diversi vantaggi, tra cui:
  - Separazione dello spazio kernel e utente;
  - Ogni frame può avere diversi permessi d'accesso;
  - Un processo può accedere solo ad un preciso sottoinsieme della memoria fisica disponibile (protezione);
  - I processi possono venire rilocati.

- Come avviene in OS/61, gli spazi d'indirizzamento virtuale sono separati, con lo spazio kernel sito agli indirizzi più alti. I processi kernel vedono gli indirizzi utente come nel loro spazio d'indirizzamento, mentre non è vero il contrario.
- In maniera identica ad OS/61, gli indirizzi fisici più bassi sono riservati al kernel.



- In MentOS, lo spazio d'indirizzamento kernel (che equivale all'incirca a kseg0 e kseg1 di OS/61) include la Zone\_DMA e Zone\_NORMAL viste in precedenza.

## GMV – MMU E TLB

- MentOS è in grado di utilizzare la *Memory Management Unit* con annesso *Translation Lookaside Buffer* messi a disposizione dall'architettura x86, per la traduzione da indirizzi virtuali a fisici.
- Al contrario di come avviene con OS/61, che adopera architettura *MIPS*, in MentOS i TLB miss sono gestiti in maniera trasparente dall'hardware. L'unica azione intrapresa dal kernel è l'invalidazione delle entry del TLB in caso di (logical) page fault con sostituzione di pagina.

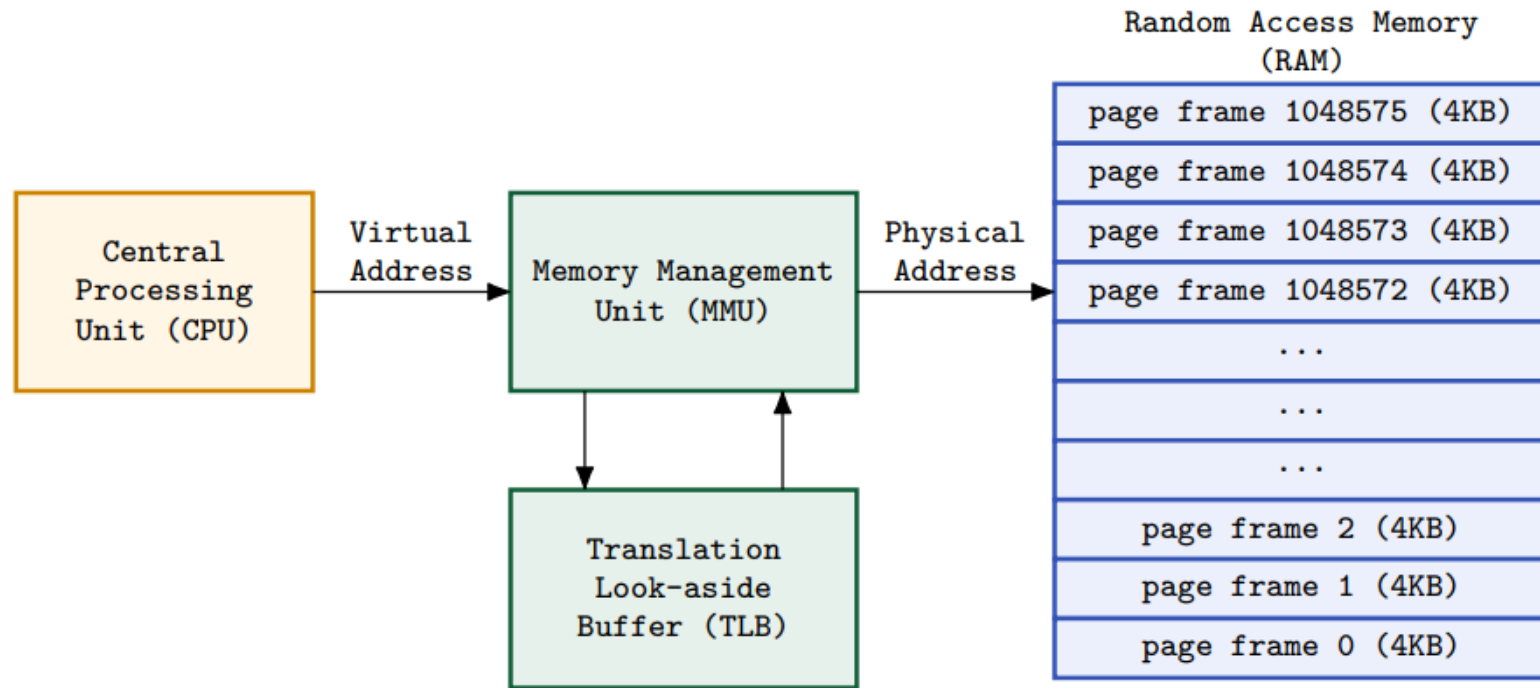


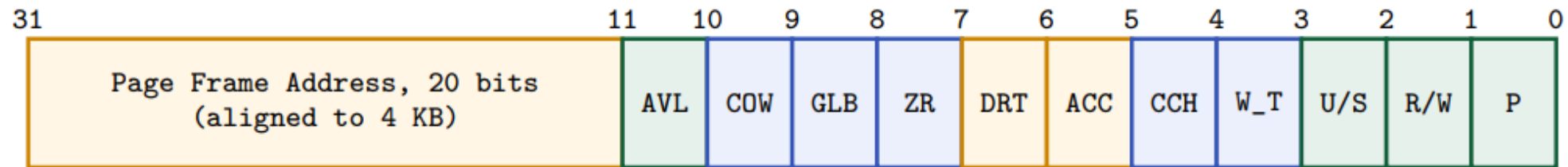
Figure: Memory Management Unit (MMU)

## GMV - DEMAND PAGING

- Come da linee guida Linux, anche MentOS utilizza un sistema di *demand paging* per la gestione delle pagine dei processi.
- Con questa strategia, una pagina viene portata in memoria solo quando si tenta di far accesso ad un indirizzo al suo interno, consentendo di ridurre l'occupazione di memoria effettiva ed il tempo di caricamento del processo in memoria.

# GMV – PAGE TABLE

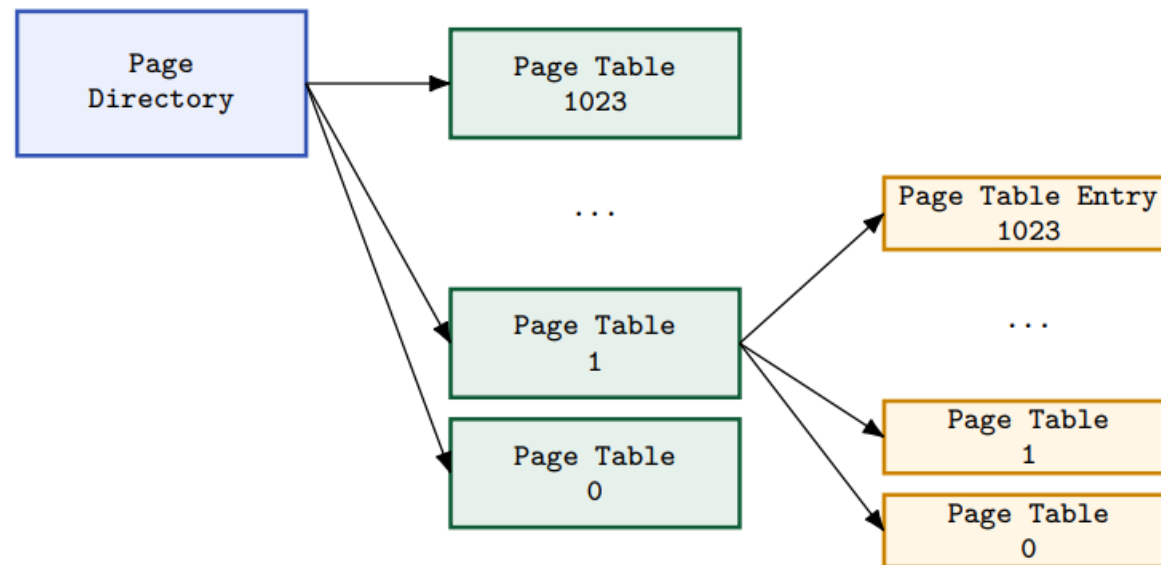
- Per tenere traccia del mapping tra pagine virtuali e frames, il kernel mantiene una Page Table Entry (PTE) per ogni pagina.



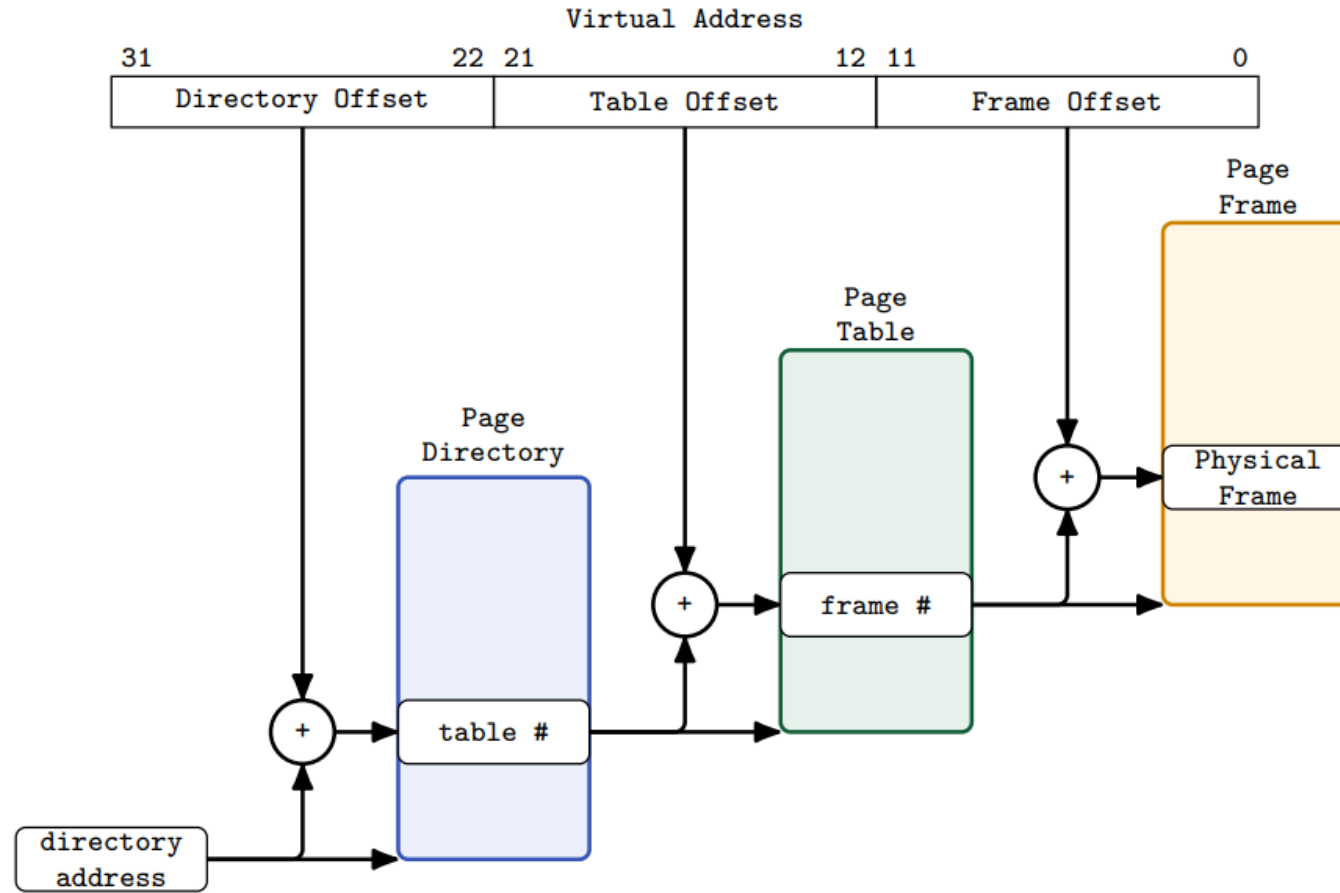
- P posto ad 1 se la pagina si trova in memoria fisica.
- R/W posto a 0 per read, 1 per write.
- U/S se posto a 1 vuol dire che la pagina è accessibile a tutti.
- ACC viene settato quando si fa accesso alla pagina.
- DRT è il dirty bit.

MentOS tiene traccia delle PTE di un processo raggruppandole in una struttura gerarchica a due livelli:

- Page Directory: contiene fino a 1024 indirizzi a Page Table. Viene individuata dai primi 10 bit d'indirizzo;
- Page Table: contiene fino a 1024 PTE. Individuata dai bit 12-22.



- Esempio schematico di traduzione di un indirizzo:





# GMV – MEMORY DESCRIPTORS

- Per mantenere traccia dello spazio d'indirizzamento di un processo, MentOS utilizza delle apposite strutture dati denominate *Memory Descriptors* (*src/mem/paging.h*), che contengono informazioni sulle aree di memoria del processo. La struct *task\_struct*, che descrive un processo, contiene un riferimento alla *mm\_struct* del processo.

```
typedef struct task_struct {  
    /// The pid of the process.  
    pid_t pid;  
    /// The session id of the process  
    pid_t sid;  
    /// The Process Group Id of the process  
    pid_t pgid;  
    /// The Group ID (GID) of the process  
    pid_t gid;  
    /// The User ID (UID) of the user owning the process.  
    pid_t uid;  
    (...)  
    /// Task's segments.  
    mm_struct_t *mm;  
    (...)  
} task_struct;
```

```
/// @brief Memory Descriptor, used to store details about the memory of a user process.  
typedef struct mm_struct_t {  
  
    list_head mmap_list; /// List of memory area (vm_area_struct reference).  
    vm_area_struct_t *mmap_cache;  
  
    page_directory_t *pgd; /// Process page directory.  
  
    int map_count; /// Number of memory area.  
  
    list_head mm_list; /// List of mm_struct.  
    //Indirizzi dei segmenti di testo, data, stack, etc (...)  
    uint32_t env_start; /// ENVIRONMENT start.  
    uint32_t env_end; /// ENVIRONMENT end.  
    unsigned int total_vm; /// Number of mapped pages.  
} mm_struct_t;
```

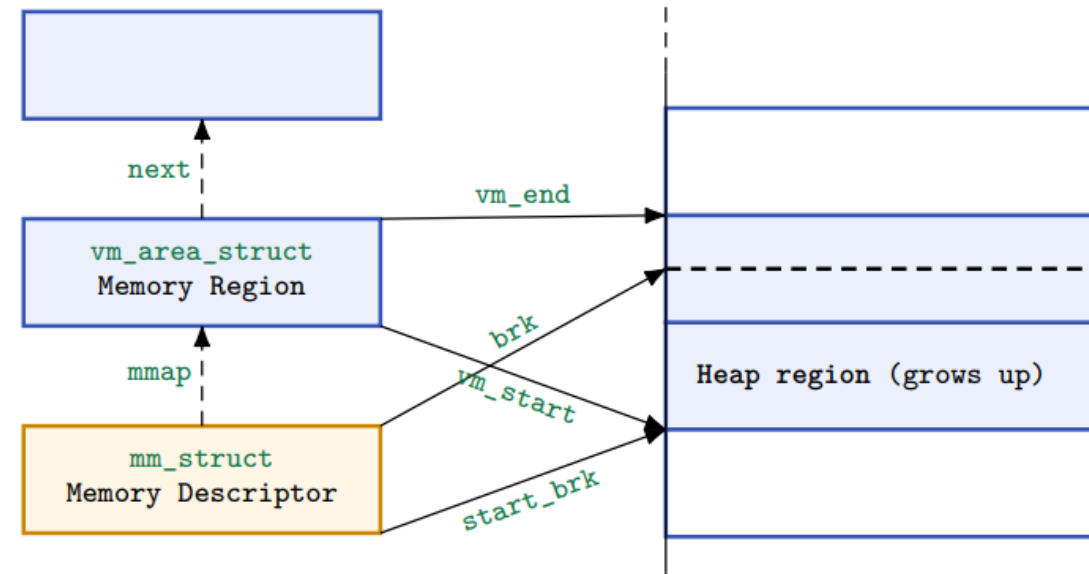
## GMV – SEGMENT DESCRIPTOR

- Il memory descriptor di un processo punta ad una lista di *segment descriptor* (*src/mem/paging.h*).
- Ogni segment descriptor definisce un'area contigua di memoria virtuale di dimensione multipla di 1 pagina logica (4KB).
- Ogni segment descriptor contiene a sua volta l'indirizzo d'inizio e fine del segmento, come la *mm\_struct*. Ciò è dovuto al fatto che, mentre il memory descriptor riporta l'ultimo indirizzo usato per ogni segmento, nel segment descriptor viene riportato l'ultimo indirizzo disponibile per quel segmento.

- La struttura usata per l'implementazione del *segment descriptor*. L'attributo *vm\_list* punta al descrittore di segmento successivo. I flag forniscono informazioni sulle pagine del segmento (es: read only, r/w).

```
typedef struct vm_area_struct_t {  
    /// Memory descriptor associated.  
    struct mm_struct_t *vm_mm;  
    /// Start address of the segment, inclusive.  
    uint32_t vm_start;  
    /// End address of the segment, exclusive.  
    uint32_t vm_end;  
    /// List of memory areas.  
    list_head vm_list;  
    /// Permissions.  
    pgprot_t vm_page_prot;  
    /// Flags.  
    unsigned short vm_flags;  
} vm_area_struct_t;
```

- Rappresentazione grafica della differenza degli indirizzi finali di un segmento, tra memory descriptor e segment descriptor.





## IMPLEMENTAZIONE IN OS/61

- Buddy system
- TLB Handling
- Paging

## OSI61 - BUDDY SYSTEM

- L'allocatore di memoria kernel utilizzato da OSI61 non utilizza nessun algoritmo particolare, ma si limita a cedere il quantitativo di memoria richiesta a partire dal primo indirizzo disponibile, e senza mai rilasciarla.
- Per fare ciò, il kernel utilizza la funzione *getppages* di *dumbvm* come wrapper per *ram\_stealmem*, in maniera simile a come *alloc\_page* dello zone allocator richiama la *bb\_alloc\_pages*.
- Si potrebbe quindi pensare di partire da questo parallelismo, per implementare un gestore della memoria più efficiente.

- Si potrebbe partire implementando l'algoritmo buddy system visto precedentemente nella *ram\_stealmem* (*kern/arch/mips/vm/ram.c*).
- Ciò richiederebbe la realizzazione di strutture dati opportune per tenere traccia delle liste di ordini dei blocchi. Queste potrebbero venire implementate mediante liste di struct (le *bb\_free\_area\_t* di MentOS) aventi tra gli attributi l'ordine del blocco e il puntatore all'elemento successivo della lista.
- Questa struct potrebbe venire definita all'interno di *dumbvm.c/h*, e sarebbe necessario una struttura che contenga i riferimenti al primo elemento di ogni lista, ad esempio un vettore staticamente dichiarato in *dumbvm* (oppure come quello presente nella struct *bb\_instance\_t* di MentOS).

- In OS161 di base non esiste nessun parallelo della `ram_stealmem` che si occupi del rilascio delle pagine occupate. Sarebbe quindi necessario implementare una funzione (es: `ram_freemem`) che si occupi di rilasciare la memoria, seguendo l'algoritmo previsto dal buddy system (a.k.a. individuare il “buddy” della memoria rilasciata, di ordine  $n$  ed effettuare il merge in un unico blocco di ordine  $n+1$ , se disponibile).
- Si tratterebbe quindi di andare a realizzare il corrispettivo della `bb_free_pages` di MentOS.
- Questa funzione potrebbe quindi venire invocata da un wrapper, una `freeppages`, realizzato in `kern/arch/mips/vm/ram.c` come la `getppages`.
- Sarebbe inoltre necessaria una struttura “parallela” alle liste di blocchi liberi, che tenga invece traccia di quelli allocati.

- Mentre, se si vuole limitarsi alla semplice allocazione mediante buddy system, il numero di strutture di supporto e di metodi necessari può essere semplificato e ridotto rispetto alla versione usata su MentOS, ciò non è più vero in caso si voglia implementare la deallocazione (ovviamente a patto di volere un codice comprensibile e mantenibile).
- Sarebbe quindi opportuna l'implementazione dei metodi e delle strutture di supporto del buddy system in file ad hoc, ad esempio in un *buddysystem.c* (e corrispettivo *buddysystem.h*).
- Tra le strutture che si suggerisce di realizzare, vi sono certamente una in grado di gestire i frame allocati, tenendo traccia dell'ordine di allocazione e degli indirizzi d'inizio e fine ( della dimensione), e quelle precedentemente nominata, usata per tenere traccia dei blocchi di frame ancora liberi.



## OSI61 - TLB HANDLING

- In OSI61 non vi è una gestione dei *fault* del TLB con sostituzione, il che significa che una volta riempite tutte le entry del TLB il kernel andrà in crash.
- Questo perché al momento il gestore dei fault del TLB, *vm\_fault* (in *kern/arch/mips/vm/dumbvm.c*) non gestisce effettivamente i casi possibili (fault su entry in sola scrittura o r/w), ma si limita a cercare la prima entry libera del TLB e nel caso in cui non la trovi, restituisce un valore di errore alla funzione chiamante, ovvero *mips\_trap*, l'exception/trap handler di OSI6 (sito in *kern/arch/mips/locore/trap.c*).

- OS161 di base presenta delle funzioni di gestione del TLB definite *kern/arch/mips/include/TLB.h*:
  - *TLB\_random*: scrive la entry passata come parametri (HI e LOW) in uno slot del TLB scelto a (pseudo)random dalla CPU;
  - *TLB\_write*: come la random, ma lo slot in cui scrivere viene specificato;
  - *TLB\_read*: salva il valore della entry TLB specificata nei parametri passati;
  - *TLB\_probe*: cerca una entry corrispondente al valore passato. Se questa non viene trovata, restituisce un valore negativo, altrimenti ritorna la posizione nel TLB della entry.
- Sono funzioni assembly (*kern/arch/mips/vm/TLB-mips161.S*) e quindi architettura-dipendenti. Fortunatamente sono sufficienti per gestire appieno il TLB, bisogna però implementare la loro chiamata.

- Quando viene scatenata un'eccezione dovuta ad un TLB fault, il gestore delle eccezioni di OS/61 invoca la *vm\_fault*. È quindi questa la funzione da modificare per gestire i possibili casi:
  - VM\_FAULT\_READONLY: eccezione per TLB write su una pagina readonly. Non dovrebbe mai accadere, effettuiamo la kill del processo richiedente;
  - VM\_FAULT\_READ o VM\_FAULT\_WRITE: eccezione per TLB miss on load o store rispettivamente. Verifichiamo che la pagina mancante sia in memoria, in caso contrario scateniamo un *page fault* per inserirla. Con la pagina in memoria, si aggiorna l'entry del TLB con il nuovo frame. In caso di TLB pieno, si può sovrascrivere una entry con *TLB\_random*, oppure adottare algoritmi più sofisticati (magari basati sulla località) in combinazione con *TLB\_write*, per selezionare e sovrascrivere la pagina vittima.

- In ogni caso bisogna sempre controllare la validità dell'indirizzo a cui si tenta di fare accesso (modo d'accesso, utente o kernel, spazio d'indirizzamento).
- Bisogna inoltre gestire il flush del TLB ad ogni context switch. In realtà le TLB entry presentano dei bit, detti Address Space Identifier (ASID) che possono essere usati per mantenere entry di processi diversi da quello attuale. L'opzione d'invalidazione delle entry (il flush) resta più semplice da attuare.
- **Importante:** poiché il TLB è una risorsa condivisa, è necessario l'uso di meccanismi di sincronizzazione per la sua modifica. In OS/61 di base ci si limita a disabilitare gli interrupts sulla cpu in cui sta venendo eseguita la *vm\_fault*, ma non è un meccanismo sufficiente in caso di sistemi multiprocessore.

# OSI61 - PAGING

- Utilizzare il paging permette al sistema operativo di non dover necessariamente caricare in memoria tutte le pagine dello spazio d'indirizzamento di un processo.
- Per poter implementare la paginazione in OSI61 bisogna soddisfare dei punti fondamentali:
  1. tenere traccia delle pagine di un processo;
  2. inizializzare le tabelle delle pagine ed il paging stesso;
  3. gestire l'allocazione e deallocazione dei frame a tali pagine.

- Per tenere traccia delle pagine possiamo adottare la medesima struttura vista in MentOS: una tabella delle pagine gerarchica a due livelli.
- Serviranno quindi una struct per la tabella (con un campo array di union che corrispondono al riferimento alla tabella di secondo livello oppure ad una PTE) e una per definire le PTE. Sarebbe anche possibile realizzare una struct diversa per i diversi livelli della tabella, se servissero attributi particolari.
- Un campo che punti alla propria tabella di primo livello sarà inserito nella *struct addrspace* di ogni processo.

```
typedef struct page_table_entry_t {
    unsigned int present : 1;
    unsigned int rw : 1;
    unsigned int user : 1;
    unsigned int w_through : 1;
    unsigned int cache : 1;
    unsigned int accessed : 1;
    unsigned int dirty : 1;
    unsigned int zero : 1;
    unsigned int global : 1;
    unsigned int kernel_cow : 1;
    unsigned int available : 2;
    unsigned int frame : 20;
} page_table_entry_t;
```

- Questa è la rappresentazione di una PTE all'interno di MentOS. I vari campi corrispondono ai flag (diritti di accesso, lettura/scrittura, se è in memoria o no, etc.). L'entry occupa 32bit. Sarebbe possibile realizzare un'implementazione simile anche in OS161 (non tutti i flag sarebbero necessari, come ad esempio il *kernel\_cow*, non essendo implementata la *copy on write*).
- Ovviamente le strutture usate occupano a loro volta memoria. È quindi opportuno realizzare delle funzioni dedite alla loro istanziazione e deallocazione. Tali funzioni verrebbero poi invocate al momento della creazione/distruzione dello spazio d'indirizzamento del processo, ad esempio dentro *as\_create* e *as\_destroy*.

- Allocare i frames alle pagine di un processo è più complicato. In primis, supponendo di voler seguire le linee guida Linux come per MentOS, bisogna ricordare che il paging è limitato allo spazio utente, mentre lo spazio kernel continua ad essere mappato direttamente in memoria.
- Per prima cosa sarà quindi necessario modificare l'allocazione dello spazio richiesto da un processo utente. Ad esempio, immaginiamo di implementare la *as\_prepare\_load* (in *kern/vm/addrspace.c*) in modo tale che vada a chiamare la *getppages* solo per istanziare lo spazio necessario (segmenti di data, stack, etc), ma non richiedendo memoria pari a quella richiesta dallo spazio d'indirizzamento.
- In pratica vogliamo andare ad ottenere un meccanismo di *demand paging* come quello implementato da MentOS, in cui al momento della creazione di un processo si alloca lo stretto indispensabile, e si caricano le pagine in memoria solo quando vengono richieste.

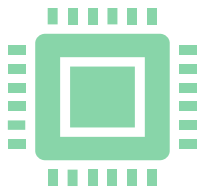


- Bisognerà poi realizzare una funzione in grado di gestire i page fault, ovvero che si occupi di recuperare le pagine dal disco, allocare lo spazio necessario in ram (eventualmente rimuovendo pagine già trovate) e far ripartire il processo.
- In MentOS questi compiti sono svolti dalla *page\_fault\_handler* (in *src/mem/paging.c*). In OS161 si potrebbe pensare di realizzare una funzione omonima in *dumbvm* che venga invocata in caso di page fault. Vale a dirsi quando, a seguito di un TLB miss, la *vm\_fault* rilevi la mancanza della pagina cercata anche in memoria principale. In questo caso la *page\_fault\_handler* dovrebbe solo occuparsi di gestire l'allocazione della nuova pagina.

- Nel caso in cui si dovesse rimuovere una pagina già in memoria per allocare quella richiesta, la `page_fault_handler` dovrebbe anche occuparsi di chiamare le subroutine per salvare la pagina su disco (se il dirty bit è settato) e invalidare l'eventuale entry del TLB della pagina rimossa.
- Nelle funzioni messe a disposizione da OS/61 per la gestione del TLB non ce n'è una per implementare direttamente l'invalidazione di una entry. Di conseguenza ci si presentano due opzioni:
  - Implementare una funzione apposita per invalidare (settando a 0 il valid bit sito nei 32 bit più bassi della TLB entry) l'entry nel TLB della pagina rimossa (se esiste);
  - Restituire al `vm_fault` l'identificativo della pagina rimossa così da inserire direttamente al suo posto nel TLB (se c'era) la nuova pagina.
- In entrambi i casi possiamo sfruttare le funzioni `TLB_write` e `TLB_probe` per la realizzazione.

- L'inserimento di una nuova pagina in memoria comporta la necessità di aggiornamento delle strutture dati viste precedentemente (page table di primo e secondo livello) per tenere traccia della nuova page table entry.
- Allo stesso modo, quando una pagina viene rimossa dalla memoria principale, è necessario aggiornare i campi delle struct in modo tale da segnalare che la pagina non è più presente in memoria.
- Come per la gestione del TLB, bisogna tenere a mente che la versione di OS/61 studiata è pensata per *sistemi multicore*. Ne consegue che la gestione delle *risorse condivise* (qui la memoria) necessita di sistemi di sincronizzazione per ridurre la possibilità di errore. Ad esempio, al momento dell'allocazione della memoria alle pagine, sarebbe opportuno l'uso di uno *spinlock* (o altri costrutti, che non sono tuttavia implementati in OS/61 di base) sulle page table (che risultano associate ad un processo, quindi in comune tra i suoi thread).

# SOURCES & CREDITS



## Sources:

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

OS/161: <http://www.os161.org/>

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



## Credits and Thanks:

All of the previous



## Copyright Licence:

Creative Commons CC2023



CONTATTI:

[ENDRI.SEFA@STUDENTI.POLITO.IT](mailto:ENDRI.SEFA@STUDENTI.POLITO.IT)

[MATTIA.OLIVA@STUDENTI.POLITO.IT](mailto:MATTIA.OLIVA@STUDENTI.POLITO.IT)