

# Virtual Memory with Demand Paging

**Aghilar Lorenzo**  
s334086@studenti.polito.it

**Rosace Giovanna**  
s332938@studenti.polito.it

## Introduzione

Questo progetto riguarda l'implementazione di un sistema di paginazione a richiesta per OS161. In seguito è riportata la divisione del lavoro e una valutazione in termini di tempo e impegno:

- Fase di pianificazione preliminare e progettazione concettuale: Aghilar Lorenzo e Rosace Giovanna ★ ★ ★ ★
- Codice modificato e/o aggiunto nei programmi già esistenti: Aghilar Lorenzo e Rosace Giovanna ★
- File `addrspace.c` e `addrspace.h`: Aghilar Lorenzo ★ ★ ★ ★
- Gestione dell'eccezione `VM_FAULT_READONLY`: Rosace Giovanna ★
- File `coremap.c` e `coremap.h`: Rosace Giovanna ★ ★ ★ ★
- `pt.c`, `pt.h`, `segments.c` e `segments.h`: Aghilar Lorenzo ★ ★ ★
- `swapfile.c` e `swapfile.h`: Rosace Giovanna ★ ★ ★
- `vm_tlb.c` e `vm_tlb.h`: Aghilar Lorenzo ★
- `vmstats.c` e `vmstats.h`: Aghilar Lorenzo ★
- Fase finale di debug e testing: Aghilar Lorenzo e Rosace Giovanna ★ ★ ★ ★ ★

É possibile consultare la repository dal seguente link: <https://github.com/midious/os161-vm-paging/tree/main>

## Fase di pianificazione preliminare, progettazione concettuale e codice modificato

Con il sistema di allocazione contigua precedente (DUMBVM), l'header del file elf veniva letto per definire zone di memoria contigue dei segmenti di text, data e stack. In particolare, dopo l'esecuzione del programma, veniva lanciata la funzione `runprogram` che a sua volta richiamava la funzione `load_elf` che aveva il compito di:

1. leggere l'header,
2. definire gli indirizzi logici di partenza e la dimensione in pagine dei segmenti con la funzione `as_define_region`,
3. definire gli indirizzi fisici di partizioni contigue in memoria con la funzione `as_prepare_load`,
4. caricare il contenuto letto dall'elf in memoria nelle partizioni trovate,
5. salvare l'entrypoint del programma

In seguito a questo caricamento, il programma aveva lo spazio di indirizzamento (`addrspace`) che conteneva tutte le informazioni per leggere e scrivere i dati dei tre segmenti in memoria (contigui).

Nel progetto sono stati eliminati i punti 3 e 4, con l'ausilio di `OPT_PAGING` definita in `opt-paging.h`, in quanto il sistema da progettare prevede la paginazione su richiesta. Inoltre la struttura dati riguardante lo spazio di indirizzamento è stata ripensata adeguatamente per questo obiettivo.

## File `addrspace.h`, `pt.h` e `segments.h`

La struttura dati dell'addrspace è la seguente:

Struttura dati di addrspace

```
struct addrspace {
    struct pt* page_table;
    struct vnode *vfile; //puntatore al ELF file del programma
};
```

dove `vfile` è il puntatore al file ancora aperto dell'header, in quanto verrà chiuso solo alla terminazione del processo con `as_destroy`. La page table invece è lo spazio di indirizzamento vero e proprio, definito dalla struttura dati `pt` in `pt.h`:

Struttura dati della page table

```
struct pt{
    struct segment* code;
    struct segment* data;
    struct segment* stack;
};
```

dove i tre segmenti rappresentano text, data e stack. Ogni segmento è rappresentato nel seguente modo:

Struttura dati del segmento

```
struct entry{
    paddr_t paddr; //indirizzo fisico corrispondente a offset zero della pagina desiderata
    bool valid_bit; //indica se la pagina e' in memoria oppure no
    int swapIndex; //Se e' uguale a -1, allora vuol dire che lo swap della pagina non e' avvenuto
};

struct segment{
    struct entry* entries;
    vaddr_t v_base;
    size_t npages;
    bool readonly; //indica se il segmento e' di sola lettura (segmento code)
};
```

Ogni segmento ha dunque un vettore di entry, allocato dinamicamente nella funzione `as_define_region`, dove inizialmente gli indirizzi fisici saranno settati a 0 e il bit di validità a 0 (entry non valida). Lo swap index invece è settato inizialmente a -1 e conterrà l'eventuale indice dentro lo swap file.

Sono inoltre salvate il numero di pagine, il bit di readonly e l'indirizzo virtuale della prima pagina (utile per calcolare l'indice di una entry).

## `addrspace.c`

Dentro `addrspace.c` sono presenti tutte le funzioni per la corretta gestione della memoria virtuale di ogni processo. Oltre a tutte le funzioni che riguardano la creazione, l'inizializzazione e la distruzione degli spazi d'indirizzamento, la funzione senza dubbio più importante è `vm_fault`.

Essa gestisce i tlb fault, dunque ogni volta che non viene trovata una entry valida nella TLB corrispondente all'indirizzo logico richiesto, viene generata un'eccezione, gestita da questa funzione. Il primo passo che svolge è capire di che segmento dei tre si tratta.

Se si tratta di code e data, per prima cosa verifica se ha bit di validità (dentro la entry corrispondente nella page table) uguale a 0. In tal caso significa che la pagina non è in memoria e dunque il caricamento deve avvenire dal disco. In seguito, si verifica se lo swapindex è uguale a -1, il che significa che tale pagina non è presente nello swap. Ciò comporta una lettura dal file elf, mentre invece se swap index è diverso da -1 allora significa che la lettura avviene dallo swap file. Il frame della memoria fisica viene allocato dalla funzione alloc\_upage (vedi sezione coremap) e in seguito viene riempito dalla funzione load\_page (vedi sezione segment), se la lettura avviene dall'elf, o da swapin (vedi sezione swapfile), se avviene dallo swap file. Come ultimo passaggio abbiamo il caricamento in tlb (vedi sezione TLB).

Se invece il bit di validità è settato a 1 allora significa che la pagina è presente in memoria, dunque richiede solo una validazione nella TLB.

#### Gestione del TLB fault in vm\_fault di una pagina del segmento code

```
...
...

index_page_table = (int) (faultaddress - vbase1)/PAGE_SIZE;
if (as->page_table->code->entries[index_page_table].valid_bit
    == 0)
{
    //Non e' in memoria perche' il valid bit della page table
    e' uguale a 0

    paddr = alloc_upage(faultaddress); //gestisce anche un
        eventuale swap out per liberare un frame
        //se non e' possibile fare
        neanche swap, allora termina
        il processo (system call)
        //alloc upages gestisce anche la
        modifica della coremap e
        l'eventuale swap

    vmstats_increment(PAGE_FAULTS_DISK);

    KASSERT((paddr & PAGE_FRAME) == paddr);

    as->page_table->code->entries[index_page_table].valid_bit
        = 1; // convalido la pagina
    as->page_table->code->entries[index_page_table].paddr =
        paddr;

    if (as->page_table->code->entries[index_page_table].swapIndex
        == -1)
    {
        //niente swap: non e' nello swapfile
        result = load_page(as, index_page_table, paddr, 0);
        if (result)
        {
            return result;
        }

        vmstats_increment(PAGE_FAULTS_ELF);
    }
    else
```

```

    {
        //SWAP IN -code
        swapin(as->page_table->code->
                entries[index_page_table].swapIndex, paddr);
        vmstats_increment(PAGE_FAULTS_SWAP);
    }

    tlb_insert(faultaddress, paddr, 1);
}
else
{

    paddr =
        as->page_table->code->entries[index_page_table].paddr;

    KASSERT((paddr & PAGE_FRAME) == paddr); // deve avere gli
        ultimi bit (dell'offset) uguali a 0

    tlb_insert(faultaddress, paddr, 1); //l'1 indica che e'
        readonly, quindi il dirty bit della tlb sara' settato
        a 0

    vmstats_increment(TLB_RELOADS);
}

...
...

```

Se si tratta del segmento di stack, i passaggi sono del tutto simili agli altri due segmenti. L'unica differenza la troviamo se il valid bit è settato a 0 e lo swap index a -1. in tal caso non legge dal file elf ma alloca semplicemente la memoria per la pagina dedicata allo stack. Anche qui, alla fine avviene il caricamento in tlb.

### Gestione dell'eccezione VM\_FAULT\_READONLY

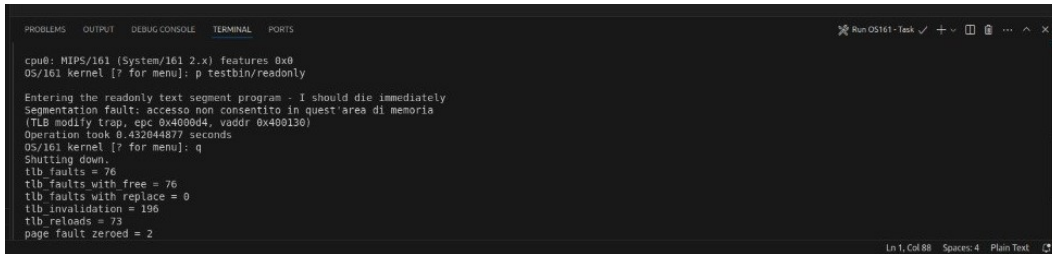
Nel sistema DUMBVM tutti e 3 i segmenti dello spazio di indirizzi (testo, dati, stack) erano sia leggibili che scrivibili dall'applicazione. Con la nuova configurazione in Demand Paging invece, il segmento di testo ("code", vedi sezione segment) di ogni applicazione è di sola lettura. Viene gestito quindi l'inserimento delle voci in TLB con il TLBLO\_DIRTY non impostato, attraverso la funzione tlb\_insert() (vedi sezione vm\_tlb), in modo tale che qualsiasi tentativo di un'applicazione di modificare la sua sezione di testo provochi la generazione di un'eccezione di memoria di sola lettura EX\_MOD -> (VM\_FAULT\_READONLY).

L'eccezione è riconosciuta all'interno della funzione vm\_fault attraverso il parametro faulttype, che con uno switch case ritorna al chiamante un errore EACCES: permission denied. Come da specifica, il processo termina con una system call exit, senza che il kernel vada in crash.

Per testare questa situazione è stato realizzato il test "testbin/readonly" che prova a scrivere all'interno di una area di memoria del segmento text. È possibile vedere il risultato nella figura 1.

### File pt.c e segment.c

In pt.c è presente solamente la funzione get\_pt\_entry che restituisce una entry della page table (indirizzo fisico, valid bit e swap index) dato un indirizzo virtuale e un addrspace.



```
cpu0: MIPS/161 (System/161 2.x) features 0x0
05/161 kernel [? for menu]: p testbin/readonly

Entering the readonly text segment program - I should die immediately
Segmentation fault: accesso non consentito in quest'area di memoria
(TLB modify trap, epc 0x4000d4, vaddr 0x400130)
Operation took 0.432044877 seconds
05/161 kernel [? for menu]: q
Shutting down.
tlb faults = 76
tlb faults with free = 76
tlb faults with replace = 0
tlb invalidation = 196
tlb reloads = 73
page fault zeroed = 2
```

Figure 1: Eccezione VM\_FAULT\_READONLY

In segment.c invece è definita la funzione `load_page` che ha il compito di caricare la pagina dal file elf nello spazio di memoria fisico allocato. Il caricamento avviene tramite l'opportuna definizione dell'offset da cui leggere l'elf, il parametro `memsz` che indica la quantità di byte da caricare in memoria e `filz` che indica quanto deve leggere dal file. Infine il buffer definito sarà un buffer kernel di cui l'indirizzo è `0x80000000 + paddr`, dove `paddr` è l'indirizzo fisico del frame in cui caricare la pagina.

## File `coremap.h` e `coremap.c`

Per tenere traccia dell'uso della memoria fisica è stato creato il file `coremap.c` con il suo corrispondente file header `coremap.h`. Viene implementato quindi un sistema per mappare le pagine di memoria virtuale in quella fisica, sia per i processi user che per lo spazio kernel.

Il file contiene diverse funzioni e strutture che si occupano di allocare e liberare la memoria fisica, mantenendo traccia dello stato di ciascuna pagina e gestendo la memoria tramite una `coremap`. La `coremap` è rappresentata come un array di strutture, in cui ogni struttura (`struct coremap_entry`) tiene traccia dello stato di una singola pagina fisica.

### Struttura dati `coremap_entry` in `coremap.h`

```
struct coremap_entry {
    bool occupied;           // Indica se la pagina e' occupata (1) o
                             // libera (0).
    bool freed;              // Indica se la pagina e' stata liberata
                             // (1) o meno (0). Questo campo e' utile durante la gestione
                             // delle pagine libere.

    int allocSize;           // Indica quante pagine consecutive sono state
                             // allocate a partire dalla pagina attuale.

    // Gestiscono la linked list interna per tracciare l'ordine di
    // allocazione delle pagine, utile per la politica FIFO
    int prevAllocated;
    int nextAllocated;
    // solo per User
    struct addrspace *as;    // Puntatore all'addrspace del processo
                             // che ha allocato la pagina (solo per le pagine utente).
    vaddr_t vaddr;           // Indirizzo virtuale della pagina allocata
                             // (solo per le pagine utente).
};
```

Attraverso la funzione `coremap_init` viene creata la `coremap` con una dimensione pari al numero di frame della RAM. Successivamente, ogni voce della `coremap` è inizializzata con valori che indicano che la pagina è libera.

Funzioni come `alloc_kpages` e `free_kpages` fungono da wrapper per gestire specificamente le pagine riservate al kernel, mentre `alloc_uvm`, un wrapper di `get_uvm`, si occupa dell'assegnazione di singole pagine ai processi utente, garantendo un'organizzazione della memoria efficace e strutturata.

Per le strutture dati del kernel, l'allocazione rimane contigua e dunque possono essere allocate più pagine contigue in memoria.

Per i processi utente, la funzione `get_uvm` si occupa dell'allocazione di singole pagine e, se necessario, attiva il meccanismo di swap (vedi sezione Swap). La deallocazione è affidata a funzioni come `free_uvm` e `free_uvm`, che aggiornano la `coremap` per indicare le pagine come nuovamente disponibili, assicurando la coerenza della struttura dati.

L'allocazione della memoria avviene principalmente tramite la ricerca di pagine precedentemente liberate o, in caso di esaurimento delle risorse, utilizzando una funzione che preleva memoria fisica direttamente dalla RAM. La ricerca di pagine disponibili è gestita da funzioni come `get_free_uvm`, che analizzano la `coremap` per individuare intervalli precedentemente liberati e restituiscono l'indirizzo fisico della prima pagina utilizzabile. Se non ci sono pagine libere, il sistema utilizza `ram_steal` per recuperare memoria fisica.

### **Politica di sostituzione delle pagine**

Il codice implementa una politica di sostituzione FIFO (First In, First Out) per la gestione delle pagine quando la memoria è piena. Le pagine vengono allocate e liberate seguendo una politica FIFO, utilizzando i campi `prevAllocated` e `nextAllocated` per tracciare l'ordine di allocazione delle pagine utente. Quando una pagina viene liberata o aggiunta, i puntatori vengono aggiornati per mantenere l'integrità della lista.

Quando non ci sono più pagine libere, una pagina "vittima" viene selezionata (la prima pagina allocata secondo la FIFO - head), e il suo contenuto viene scritto nel file di swap (vedi sezione Swap).

Successivamente, la pagina vittima viene rimpiazzata con la nuova pagina richiesta, aggiornando di conseguenza la `page table` e `tlb`.

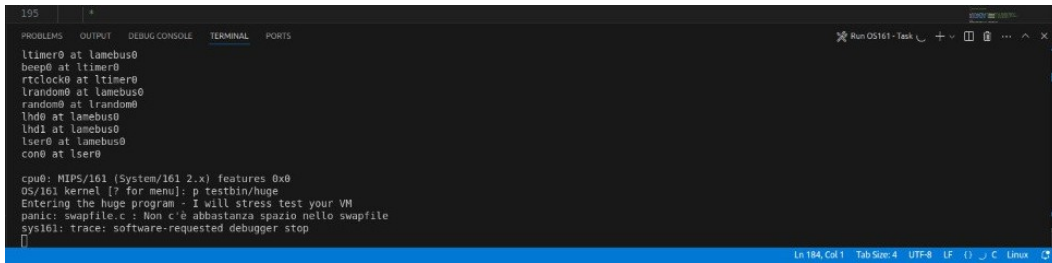
### **File `swapfile.h` e `swapfile.c`**

I file `swapfile.c` e `swapfile.h` implementano un sistema per gestire lo spazio di swap, utilizzando un file dedicato (SWAPFILE) come estensione della memoria fisica per ospitare pagine della RAM quando quest'ultima è satura. Questo meccanismo consente di aumentare la capacità effettiva della memoria disponibile, gestendo dinamicamente il trasferimento delle pagine tra RAM e disco.

Le strutture dati considerate per l'implementazione di questa funzionalità sono una bitmap (`swapfilemap`), utilizzata per monitorare lo stato delle pagine all'interno dello swap file, indicando se sono occupate o libere, e un puntatore `vnode` al file su cui verranno fatte le operazioni di lettura e scrittura (`swapfile`).

Il processo di scrittura nello swap (`swapout`) consente di trasferire una pagina dalla RAM al file di swap. Durante questa operazione, viene allocato un indice libero nella bitmap, calcolato l'offset corrispondente nel file, e la pagina viene salvata utilizzando l'operazione di scrittura `VOP_WRITE`. In caso di errore nella scrittura o di esaurimento dello spazio disponibile nello swap, viene generato un Panic. Al termine, la funzione restituisce l'indice della bitmap dove la pagina è stata salvata nel file di swap. Questo `swapIndex` viene memorizzato nella rispettiva entry della `page table` per indicare che la pagina è stata trasferita nello swapfile (vedi sezione Segment).

La lettura dallo swap (`swopin`) permette di ricaricare una pagina salvata nel file di swap nella RAM. Durante l'operazione, si verifica che l'indice specificato nella bitmap sia valido e che la pagina sia effettivamente presente nel file di swap. Successivamente, la pagina viene trasferita nella memoria



```
195
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [?] for menu]: p testbin/huge
Entering the huge program - I will stress test your VM
panic: swapfile.c: Non c'è abbastanza spazio nello swapfile
sys161: trace: software-requested debugger stop

Ln 184, Col 1 Tab Size: 4 UTF-8 LF C Linux
```

Figure 2: testbin/huge con solo 1 MB allocato per lo swap file

fisica utilizzando un'operazione di lettura VOP\_READ. Una volta completato il caricamento, il bit corrispondente nella bitmap viene liberato per indicare che lo spazio è nuovamente disponibile.

La costante principale definita in swapfile.h è la dimensione massima del file di swap, impostata a 9 MB tramite la variabile SWAPFILE\_SIZE.

Al termine dell'uso, la funzione swap\_shutdown chiude il file di swap e rilascia le risorse associate, mentre swap\_free è responsabile di segnare le pagine come disponibili per future scritture, assicurando che non vi siano duplicati o errori.

### Eccezione "out of swap space"

Se il processo richiede più memoria di quella che è fisicamente disponibile tra RAM e file di swap, allora il sistema si arresta con il messaggio "swapfile.c: Non c'è abbastanza spazio nello swap file". Per testare questa possibilità, abbiamo diminuito la memoria di swap fino a 1 MB (definibile nel file swapfile.h) e abbiamo eseguito "testbin/huge". È possibile vedere il risultato nella figura 2.

### File vm\_tlb.h e vm\_tlb.c

Il file vm\_tlb.h dichiara unicamente tre funzioni.

La prima è tlb\_invalid che invalida tutta la TLB. Questa funzione è chiamata da as\_activate nel file addrspace.c ogniqualvolta avviene un context switch o un nuovo processo viene avviato.

La seconda è tlb\_insert che ha l'obiettivo di inserire dentro la TLB una entry valida (valid bit settato a 1) dove viene associata la pagina a un frame. Inoltre viene anche settato il dirty bit in base al segmento: in quello di code viene settato a 0 in quanto è definito solo come readonly. La politica di sostituzione, come da specifica, è basata sull'algoritmo di round-robin.

La terza è tlb\_invalid\_one che ha l'obiettivo di invalidare l'entry della tlb corrispondente all'indirizzo fisico che ha appena subito uno swap out.

### File vmstats.h e vmstats.c e fase di test

Per quanto riguarda le statistiche, riteniamo interessante approfondire quella riguardante l'azzeramento della pagine. Questa statistica richiede il conteggio delle volte in cui una pagina viene azzerata completamente senza poi caricare nulla. Questo avviene in due occasioni:

- il processo vuole utilizzare una nuova pagina di stack che non sia già stata allocata prima, dunque che non sia già in memoria o nello swap file,
- il TLB fault avviene per una pagina compresa tra maxfilesz e maxmemsz del file elf, dunque nel terzo segmento (BBS).

In seguito sono riportate le statistiche di una serie di programmi già definiti, più un programma (testbin/readonly) realizzato con l'obiettivo di scrivere in un area di memoria readonly (ovvero nel segmento di code).

	palin	huge	sort	matmult	readonly	cctest
TLB Faults	13936	7534	7194	4854	76	248661
TLB Faults with Free	13936	7529	7169	4854	76	248661
TLB Faults with Replace	0	5	25	0	0	0
TLB Invalidations	7835	7798	3728	1997	73	250151
TLB Reloads	13931	3948	5458	4045	73	123697
Page Faults (Zeroed)	1	512	289	380	2	257
Page Faults (Disk)	4	3074	1447	429	1	124707
Page Faults from ELF	4	3	4	3	1	3
Page Faults from SwapFile	0	3071	1443	426	0	124704
Swapfile Writes	0	3517	1661	735	0	124889