- Paging
  - Introduzione
  - TLB Support
    - Entry non Presente
    - Entry Presente
  - 2-Level Page Table
  - On-Demand Page Loading
  - Page Replacement
  - Memoria di Swap
  - Buddy System
  - Statistiche
  - List e HashTable
  - Atomic
  - Conclusione

Paging

Introduzione

Lo scopo di questo progetto e' qello di creare un supporto per la memoria virtuale su OS161, che nella sua forma iniziale presenta una rudimentale forma di memoria virtuale per supportare l'isolazione tra i processi utente, in questa versione viene completamente rimossa e rimpiazzata con una gestione a *Page Table* gerarchica a due livelli per singolo processo, per riempire questa *Page Table* viene utilizzata una tecnica di *Demand Paging* dove ogni volta che avviene un **Page Fault** se la pagina non e' presente in memoria questa viene caricata dal file sorgente del programma (File ELF). Quando avviene un **Page Fault** esistono 2 possibilita' per mettere una pagina in memoria:

- dal file sorgente (ELF)
- dalla memoria di swap

Piu' in generale quando avviene un **TLB Fault** si ha anche un **Page Fault** se la pagina che stiamo cercando non si trova nella *Page Table*, se invece la pagina si trova nella *Page Table* e il tipo di fault e' **ReadOnly** vuol dire che ci troviamo di fronte ad una pagina condivisa tra piu' processi, precedentemente codivisa attraverso una `fork()`, questa' verra' successivamente copiata ed aggiunta alla *Page Table*.

In OS161 e' presente un sopporto minimo anche per la gestione dei frame (pagine fisiche), in questa versione e' stato implementato un algoritmo **Buddy System** per la gestione delle pagine, per fare cio' viene creato un array con tutte le pagine della memoria fisica e queste vengono gestite durante l'`alloc()` e la `free()`, quando il sistema si sovraccarica e iniziano a finire le pagine disponibili queste inziano ad essere spostate nella **Memoria di Swap**, che verrano rimesse in memoria a seguito di **Page Fault**.

TLB Support

Ogni volta che avviene un **TLB Fault** viene chiamata la funzione `vm_fault()` per gestire l'eccezione, questa funzione gestisce la mancanza di una entry all'interno della TLB oppure la scrittura su indirizzo su cui e' permessa la sola lettura. Analizziamo i due casi separatamente riportati nella seguente sezione di codice.

```
static int vm_handle_fault(struct addrspace *as, vaddr_t fault_address, int
fault_type)
{
  // ...

  pte = pt_get_or_alloc_pte(&as->pt, fault_address);
  if (!pte)
    return ENOMEM;

  pte_entry = *pte;

  /* The page is not present in memory */
  if (!pte_present(pte_entry)) {
    return page_not_present_fault(as, area, pte, fault_address,
fault_type);
  }

  if (fault_type & VM_FAULT_READONLY) {
    return readonly_fault(as, area, pte, fault_address, fault_type);
  }

  vm_tlb_set_page(fault_address, pte_paddr(pte_entry),
pte_write(pte_entry));

  // ...
}
```

Entry non Presente

Se la entry non esiste nella TLB si va a cercare l'indirizzo fisico nella *Page Table* del rispettivo processo, quando si arriva alla PTE (*Page Table Entry*) ovvero all'entry di secondo livello della *Page Table* lo si analizza e si vede lo stato in cui si trova, esso puo' essere nei 3 seguenti stati:

- `none`: la entry ha un valore `NULL`, questo vuol dire che la pagina non e' stata ancora mappata, allora si va a caricare la pagina dall'ELF sorgente del programma e la si porta in memoria
- `swap`: la entry ha un valare che punta alla memoria di swap, allora si copia in memoria la pagina e si decrementa il refcount all'interno della memoria di swap

```
static int page_not_present_fault()
{
  struct page *page = alloc_user_page();
  if (!page)
    return ENOMEM;

  /* load page from swap memory */
  if (pte_swap_mapped(*pte)) {
    retval = swap_get_page(page, pte_swap_entry(*pte));
    // ...
  }
```

```
  /* load page from memory if file mapped */
  else if (pte_none(*pte) && asa_file_mapped(area)) {
    clear_page(page);
    retval = load_demand_page(as, area, fault_address,
page_to_paddr(page));
    // ...
  }
  // ...

  pte_set_page(pte, page_to_kvaddr(page), flags);

  // ...
}
```

- `present`: la pagina si trova nella *Page Table*, allora si aggiunge semplicemente il valore dell'indirizzo della pagina fisica nella TLB

Entry Presente

Quando la entry e' presente puo' solo trattarsi di **Read Only Fault**, i casi che possono averlo causato sono 2:

- l'indirizzo del fault non corrisponde a nessuna `addrspace_area` (area di memoria del processo mappata), o l'area a cui corrisponde non e' scrivibile, che porta ad un segmentation fault
- l'indirzzo corrisponde una pagina **condivisa** tra piu' processi, la pagina viene definita **COW** (*Copy On Write*), la pagina viene quindi copiata ed assegnata alla *Page Table* del processo che ha fatto fault, decrementando il refcount della pagina precedente

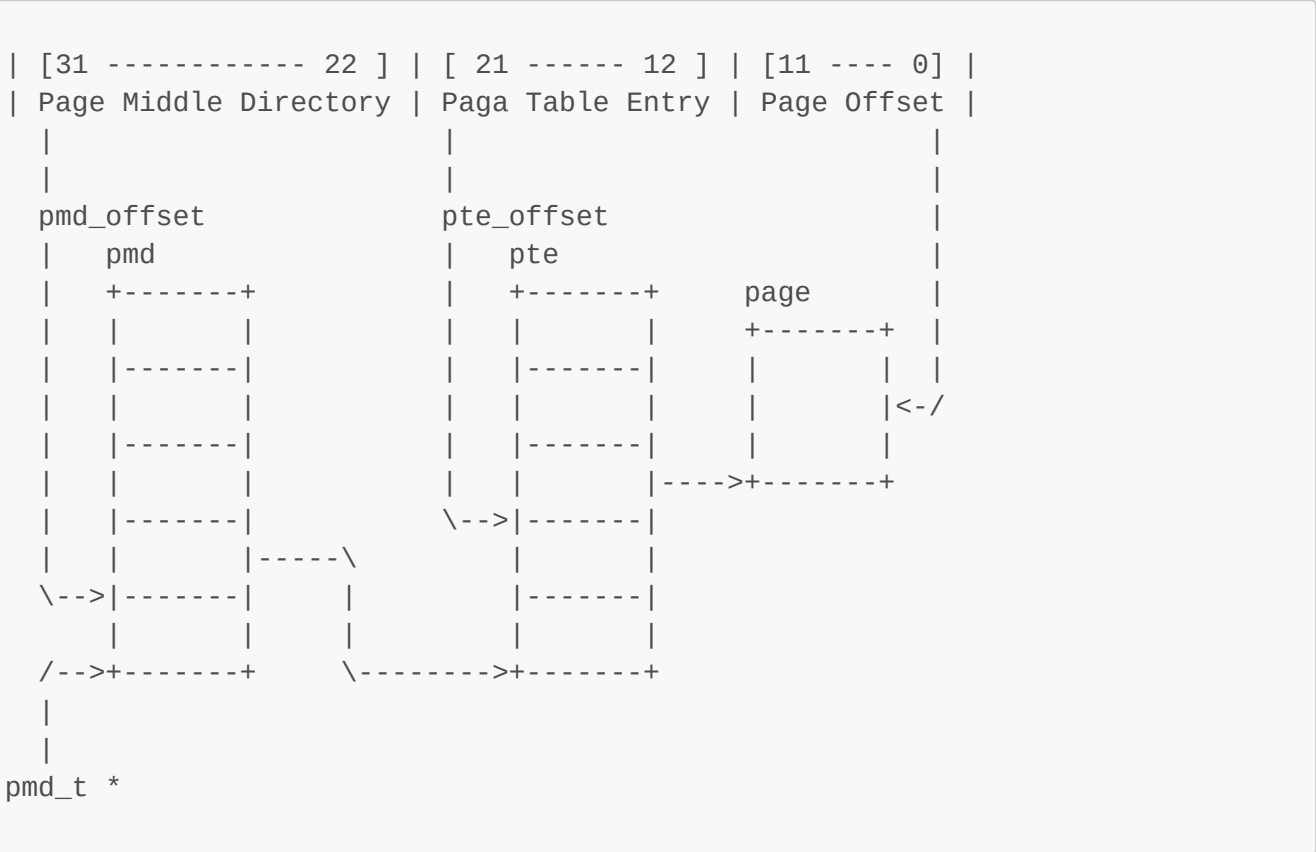
```
static int readonly_fault()
{
  struct page *page = pte_page(*pte);

  // ...

  if (asa_readonly(area))
    return EFAULT;

  if (is_cow_mapping(area->area_flags)) {
    page = user_page_copy(page);
  }

  // ...

  pte_set_page(pte, page_to_kvaddr(page), PAGE_PRESENT | PAGE_RW |
PAGE_ACCESSED | PAGE_DIRTY);
  vm_tlb_set_page(fault_address, page_to_paddr(page), true);

  // ...
}
```

2-Level Page Table

La page table e' messa all'interno della `struct proc` ed ha una struttura a due livelli, dividendo l'indirizzo virtuale in tre parti, rispettivamente in

```
| [31 ------------ 22 ] | [ 21 ------ 12 ] | [11 ---- 0] |
| Page Middle Directory | Paga Table Entry | Page Offset |
```

le parti di indirizzo forniranno l'offset all'interno delle tabelle di livello che dovranno essere contigue



La `struct proc` avra' un campo `pmd_t *pmt` che punta al primo livello della tabella che sara' sempre allocata, mentre le `pte` vengono solamente allocate su richiesta, questo fa guadagnare molta memoria che altrimenti verrebbe sprecata a causa di poco utilizzo. Avere una *Page Table* per processo ha anche il vantaggio che i tempi di attesa non sono lunghi, come accadrebbe se si avesse una *Page Table* globale condivisa da tutti i processi, che necessiterebbe di un pesante sistema di locking.

On-Demand Page Loading

Le pagine di un processo non sono inizialmente caricate in memoria durante `load_elf()`, venongo invece caricati gli header dell'elf che contengono al loro interno la descrizione dell'area di memoria, la loro struttura e' la seguente:

```
struct addrspace_area {
  area_flags_t area_flags;        /* flags of the area */
  area_type_t  area_type;         /* type of the area */
  struct list_head next_area;
```

```
  /*
   * Borders of the area, the end is not included
   * in the interval [area_start, area_end)
   */
  vaddr_t area_start, area_end;

  size_t seg_size;               /* Size of the segment within the source
file */
  off_t seg_offset;              /* Offset of the segment within the
source file */
};
```

Solo quando avviene un **Page Fault** la pagina viene prelevata dall'ELF attraverso `load_elf_page()` e messa in memoria

```
int load_demand_page(struct addrspace *as, struct addrspace_area *area,
vaddr_t fault_address, paddr_t paddr)
{
  int retval;
  off_t page_offset, file_offset;
  size_t memsize, filesz;
  vaddr_t vaddr;

  /*
   * Calculate the offset of the page to be
   * loaded inside the segment
   */
  KASSERT(fault_address >= area->area_start);
  KASSERT(fault_address <  area->area_end);

  /* align the offset with the begenning of a page */
  if ((fault_address & PAGE_FRAME) > area->area_start) {
    page_offset = (fault_address & PAGE_FRAME) - area->area_start;
  } else {
    page_offset = 0;
  }

  file_offset = area->seg_offset + page_offset;
  KASSERT((page_offset == 0) || PAGE_ALIGNED(area->area_start +
page_offset));
  vaddr = PADDR_TO_KVADDR(paddr) + ((area->area_start + page_offset) %
PAGE_SIZE);
  memsize = PAGE_SIZE - ((area->area_start + page_offset) % PAGE_SIZE);
  filesz = (page_offset < area->seg_size) ? area->seg_size - page_offset :
0;

  /*
   * only load the demanded page inside memory,
   * calculate the size of the page to load inside
   */
  retval = load_ksegment(as->source_file,
```

```
        file_offset,
        vaddr,
        memsize,
        MIN(filesz, memsize));
    if (retval)
      return retval;

    return 0;
  }
```

Page Replacement

Quando il sistema inizia a sovraccaricarsi entra in gioco l'algoritmo di *Page Replacement* che permette di mantenere sempre della memoria disponibile in caso di necessita', questo inzia a funzionare superato un certo *threshold* impostato all'80% della memoria occupata. Per rimpiazzare una pagina si scorre la *Page Table* del processo che ha richiesto memoria e se una pagina non e' segnata come `PTE_ACCESSED` (che viene settato quando vi si fa accesso durante un *Page Fault*), questa viene smarcata e si procede con la prossima fino a trovare la prima pagina disponibile, inoltre le pagine condivise tra piu' processi vengono saltate. Questa processo puo' anche portare a nessuna pagina mossa nella memoria di swap, le funzioni utilizzate sono `choose_victim_page()` e `pt_walk_page_table()`.

Memoria di Swap

La **Memoria di Swap** viene rappresentata attraverso `struct swap_memory`, questa struttura contiene il numero di pagine allocate nella memoria di swap ed un array con il numero di `refcount` attuali che ha una pagina in swap, se il `refcount` e' a zero questo vuol dire che quella posizione e' libera.

```
struct swap_entry {
    unsigned refcount;
};

struct swap_memory {
    size_t swap_pages;
    size_t swap_size;

    struct spinlock swap_lock;
    struct lock *swap_file_lock;

    struct vnode *swap_file;

    struct swap_entry swap_page_list[SWAP_ENTRIES];
};
```

Il numero di entry totale e' definito attraverso una macro a *compile-time*, questo fa si che non ci possa essere un ridimensionamento dinamico. Per usare tale memoria viene creato un file `/swap` che permette l'immagazzinamento temporaneo delle pagine, per fare `swap-in` si utilizza `swap_mem.swap_page_list` per decidere il primo spazio libero disponibile, l'algoritmo e' una semplice scansione lineare dove si cerca la prima entry con `refcount` uguale a 0 e successivamente la pagina viene copiata nella swap.

```
static int handle_swap_add_page(struct swap_memory *swap, struct page
*page, swap_entry_t *entry)
{
  // ...

  /*
   * Lock for the file access goes this early
   * beacause there is a race condition after the
   * spinlock ends, another thread might come
   * before this one taking the lock, thus
   * reading garbage from the memory.
   */
  lock_acquire(swap->swap_file_lock);
  spinlock_acquire(&swap->swap_lock);

  first_free = swap_get_first_free(swap);

  swap->swap_page_list[first_free].refcount += 1;
  swap->swap_pages += 1;

  KASSERT(swap->swap_page_list[first_free].refcount == 1);
  spinlock_release(&swap->swap_lock);

  file_offset = first_free * PAGE_SIZE;
  uio_kinit(&iovec, &uio, (void *)page_to_kvaddr(page), PAGE_SIZE,
file_offset, UIO_WRITE);

  retval = VOP_WRITE(swap->swap_file, &uio);
  lock_release(swap->swap_file_lock);
  if (retval)
      goto bad_write_cleanup;

  // ...
}
```

La funzione per cercare la prima entry disponibile e' molto semplice

```
static size_t swap_get_first_free(struct swap_memory *swap)
{
    for (size_t i = 0; i < SWAP_ENTRIES; i += 1) {
        if (swap->swap_page_list[i].refcount != 0)
            continue;

        return i;
    }

    panic("Out of swap space!\n");
}
```

Buddy System

Per allocare e liberare la pagine del sistema si e' utilizzata una strategia di **Buddy System**, il buddy system ha un ordine massimo di 6 cio' significa che il numero massimo di pagine contigue allocabili e' di `2^6 = 64`, durante la fase di bootstrap tutto la memoria rimasta (quella non presa dal codice del kernel) viene suddivisa nel numero massimo di `order-6` pagine. Ogni pagina ha al suo interno una lista che permette ad una pagina di far parte dei vari livelli e di essere cercata con `O(1)`, infatti il check per vedere se un ordine contiene delle pagine si limita a prendere un elemento dalla lista.

```
static struct page *get_page_from_free_area(struct free_area *area)
{
  return list_first_entry_or_null(&area->free_list, struct page,
buddy_list);
}
```

Statistiche

Sono disponibili anche delle statistiche, queste mostrano dettagli riguardanti lo stato del sistema e sono accessibili attraverso dei comandi nel menu

- `mem`: info sullo stato del buddy system e delle pagine del sistema
- `fault`: info sui **TLB Fault**, contiene statistiche riguardanti la TLB e gli spostamenti in memoria delle pagine
- `swap`: statistiche sulla memoria di swap
- `swapdump [start end]`: fa un dump di ogni entry della memoria di swap nel range specificato

List e HashTable

Le liste e le tabelle di hash sono prese dalle libreria di Linux, includendo i rispettivi file `list.h` e `hash.h` (con leggere modifiche per essere adattati ad OS161), l'utilizzo delle liste e' molto esteso in questa versione grazie all'uso generico che ha per sua natura la list di Linux, infatti se si guarda la definizione della lista linkata ci accorgiamo che ha solo due campi

```
struct list_head {
  struct list_head *next, *prev;
}
```

potrebbe sembrare confusionaria ad un primo impatto, infatti se percorriamo la lista abbiamo solo puntatori ma nessun campo per la struttura dati, in realta' e' molto potente, infatti grazie una funzione di gcc `offsetof()` che prende come input il nome della struttura ed il nome di un suo membro ritorna l'offset dal membro all'inzio della struttura, potendo ricavare cosi' un puntatore alla struttura dove la `struct list_head` si trova.

Atomic

Il supporto per gli atomic e' implementato in assembly, il codice e' molto simile a quello del `testandset()` dello `spinlock`, nella `atomic_fetch_add()` vengono prese delle precauzioni:

- il codice assembly viene preceduto da una `membar` (l'istruzione `sync`) e viene anche susseguito da una barriera di memoria (il *clobber* `"memory"` che fa parte della sintassi di gcc)
- la `llsc` puo' sempre fallire, se cio' accade si ripete la sezione di codice, questo viene fatto grazie ad un'istruzione di jump
- l'incremento del contatore e' visibile a tutte le cpu in modo non ordinato rispetto alla cpu che chiama la prima istruzione di `ll`, questo perche' se un'altra cpu cerca di scrivere nella zona di memoria la prima fallira'

```
static inline int
atomic_fetch_add(atomic_t *atomic, int val)
{
    int temp;
    int result;

    asm volatile(
    "    .set push;"          /* save assembler mode */
    "    .set mips32;"        /* allow MIPS32 instructions */
    "    sync;"               /* memory barrier for previous read/write */
    "    .set volatile;"      /* avoid unwanted optimization */
    "1:  ll   %1, 0(%2);"     /*   temp = atomic->val */
    "    add  %0, %1, %3;"    /*   result = temp + val */
    "    sc   %0, 0(%2);"     /*   *sd = result; result = success? */
    "    beqz %0, 1b;"
    "    .set pop;"           /* restore assembler mode */
    "    move %0, %1;"        /*   result = temp */
    : "=&r" (result), "=&r" (temp)
    : "r" (&atomic->counter), "Ir" (val)
    : "memory");              /* memory barrier for the current assembly
block */

    return result;
}
```

Per testare questa funzione e' stato aggiunto il file `test/atomic_unit.c`, che spawna `n` thread che aumentano contemporaneamente la stessa variabile atomica, questo test e' eseguibile grazie al comando `atmu1 <n-thread>`.

Conclusione

In conclusione questo progetto mira ad implementare la gestione della memoria virtuale in un sitema in cui i processi hanno un solo thread, questo e' molto limitante per un SO, motivo per il quale molte strutture dati sono semplificate e non e' presente un grande utilizzo delle primitive di sincronizzazione, ad esempio in Linux per accedere alla *Page Table* di un processo si deve possedere il semaforo `mmap_lock` che blocca l'intero address space di un processo (`mm_struct` = MemoryMap in Linux), in OS161 si puo' accedere all'address space di un processo senza bisogno di lock.

L'implementazione della `fork()` con COW comporta da una parte dei miglioramenti nelle prestazioni ma dall'altra presenta limitazioni non risolte, ad esempio quando si cerca di portare una pagina nella memoria di swap, nel momento in cui un processo ha solo delle pagine condivise, questo non e' possibile. Il motivo e' che spostare una pagina senza notificare il suo movimento agli altri possessori porta a delle *race condition* con l'accesso a quella zona di memoria, in OS161 esiste un sopporto per notificare alle altre CPU di fare un flush della loro TLB attraverso la `struct tlbshootdown` e la funzione `ipi_tlbshootdown()`. Col tempo questa mancanza di spostamenti puo' portare ad una saturazione della memoria. Un eventuale miglioramento sarebbe quello di creare una `page cache` che gestisce gli scambi con la memoria di swap, in Linux la `page cache` gestisce anche gli scambi con altre zone di memoria, come le NUMA, ...

In generale le prestazioni di esecuzione sono leggermente peggiorate a causa dell'overhead causato dalla doppia indirezione della *Page Table*, dalle dimensioni maggiori della `vm_fault()`, dall'introduzione del *Page Replacement* e dalla ricerca nella *Memoria di Swap*, come pro si ha un sistema in grado allocare e dellocare memoria fino al riempimento della memoria di swap e se gestita in modo consono permette di non causare mai un **Out Of Memory**, inoltre quando un processo fa delle operazioni illegali esso viene terminato invece di crashare l'intero sistema.