Progetto 1.1 “Analisi Comparativa tra OS161 e altri Sistemi Operativi Open-Source all’Avanguardia per Sistemi Embedded e Computer General Purpose”

S308538 - Gaetano Insinna

Programmazione di Sistema - a.a. 2023-24

Indice

1 Memoria Virtuale

In questo capitolo tratteremo della memoria virtuale: a una breve introduzione teorica seguirà la comparazione tra OS161 e xv6 ovvero il sistema operativo che stiamo analizzando.

1.1 Introduzione

La memoria virtuale è una tecnica di gestione della memoria che simula un aumento della memoria principale vista da parte di ogni processo. Questa tecnica introduce diversi vantaggi:

1. un processo in esecuzione può essere più grande della memoria fisica, quindi un processo non è limitato dalla quantità di memoria attualmente a disposizione
2. un processo può essere eseguito anche se non è caricato interamente in memoria principale, quindi possono essere eseguiti più programmi contemporaneamente
3. i processi possono condividere librerie e files
4. vengono effettuate un numero minore di operazioni di page swapping in quanto in memoria saranno caricate sole le pagine più utilizzate e non tutte le pagine, anche se attualmente non utilizzate, di un singolo processo

1.1.1 Implementazione della memoria virtuale

La tecnica su cui si basa la memoria virtuale è chiamata paging e consiste nel dividere la memoria in porzioni fisse. Esse prendono il nome di:

- pagine: porzione della memoria virtuale
- frames: porzione della memoria fisica

è importante sottolineare, tuttavia, che la dimensioni di una pagina è uguale a quella di un frame e viceversa.

Gli indirizzi prodotti da una CPU sono degli indirizzi virtuali che non corrispondono immediatamente agli indirizzi della memoria fisica, occorre infatti eseguire delle operazioni per trovare questa corrispondenza. Per eseguire ciò viene introdotto il concetto di page table che altro non è che una tabella nella quale vengono salvati gli indirizzi dei frames. Essi poi possono essere ricavati tramite una ricerca per mezzo del pagina.

Esula da questo lavoro scendere nel dettaglio sul funzionamento della page table e sulla comparazione tra le diverse page table esistenti, ma si esprime una introduzione per poter trattare un confronto tra i sistemi operativi OS161 e xv6.

Per capire il funzionamento di una page table e come avviene la traduzione tra indirizzo logico a fisico, l'indirizzo generato dalla CPU viene diviso in due parti che prendono il nome di:

- page number: viene utilizzato come indice per accedere alla page table, la entry corrispondente è il frame number
- page offset: indica dove si trova il dato richiesto all'interno della pagina e, di conseguenza, all'interno del frame

Unendo il frame number, trovato come entry della page table, e il page/frame offeset si ottiene l'indirizzo della memoria fisica.

1.1.2 Memory Management Unit e Translation Lookaside Buffer

Il lavoro descritto nelle sezioni precedenti è svolto da un particolare dispositivo hardware chiamato Memory Manangement Unit o MMU. Per eseguire la traduzione degli indirizzi esso utilizza, oltre alla modalità già presentata, una seconda ovvero lavora con un hardware dedicato, implementato con memoria cache, chiamato Translation Lookaside Buffer o TLB. Esso possiede una quantità finita di elementi delle tabella delle pagine e viene utilizzato come vera e propria memoria cache degli indirizzi che vengono tradotti

più spesso. Gli elementi del TLB vengono chiamati entries e, grazie ad un accesso diretto, qualora il frame ricercato sia in TLB (TLB Hit) l'indirizzo da tradurre viene generato in maniera molto più veloce, altrimenti accade un TLB Miss e il frame viene calcolato nella maniera descritta sopra.

capire se aggiungere o meno Page Fault e TLB Miss - Demand Paging

1.2 Memoria Virtuale in OS161

Il sistema operativo OS161 è basato su un'architettura `MIPS` a 32-bit. La memoria in OS161 è allocata in modo contiguo e misura `4 GB`. I primi `2 GB` sono riservati alla memoria utente e i secondi sono dedicati alla memoria kernel. Il kernel è mappato in una porzione dello spazio di indirizzamento di ogni processo in modo che esso possa vedere i processi user in maniera semplice. Per poter accadere ciò ci sono dei metodi di protezione attivi per far sì che quando si lavora in modo non privilegiato si possa vedere solamente una parte specifica dell'address space. Nello specifico la memoria è divisa in 4 porzioni:

1. `kuseg`: User and supervisor mode; TLB-mapped, cacheable.

   È un segmento di `2 GB` che va da `0x00000000` a `0x7fffffff` e rappresenta la memoria dove risiedono i programmi a livello utente. È mappato in TLB così da poter effettuare la traduzione da indirizzo logico a indirizzo fisico.

2. `kseg0`: Supervisor mode only; direct-mapped, cached.

   È un segmento che va da `0x80000000` a `0x9fffffff`, rappresenta la memoria kernel quindi non è mappato in TLB per far sì che ci sia una traduzione veloce tra indirizzo logico e indirizzo fisico semplicemente sottraendo all'indirizzo l'indirizzo di base della stessa porzione e ha la cache.

3. `kseg1`: Supervisor mode only; direct-mapped, uncached.

   È un segmento che va da `0xa0000000` a `0xbfffffff` e rappresenta la memoria dei dispositivi di I/O per questo motivo non è mappato in TLB e non prevede la cache

4. `kseg2`: Supervisor mode only; TLB-mapped, cacheable.

   È un segmento di memoria che va da `0xc0000000` a `0xffffffff` ed è inutilizzato

Figure 1: Rappresentazione della memoria di OS161

1.2.1 Inizializzazione e avvio del kernel

Il kernel viene avviato tramite la funzione assembler `kern/arch/sys161/main/start.S`. Questa è una panoramica del kernel all'avvio di OS161 in una memoria RAM di `1 MB`:

Come possiamo vedere quando si fa bootstrap del kernel in RAM si trovano diverse sezioni:

1. gestori delle eccezioni
2. il kernel
3. stringhe per il boot (anche se nella versione base di OS161 non è richiesta alcuna stringa, ma potrebbe essere inserita) e allineamento di pagine
4. lo stack del primo thread di kernel
5. memoria libera

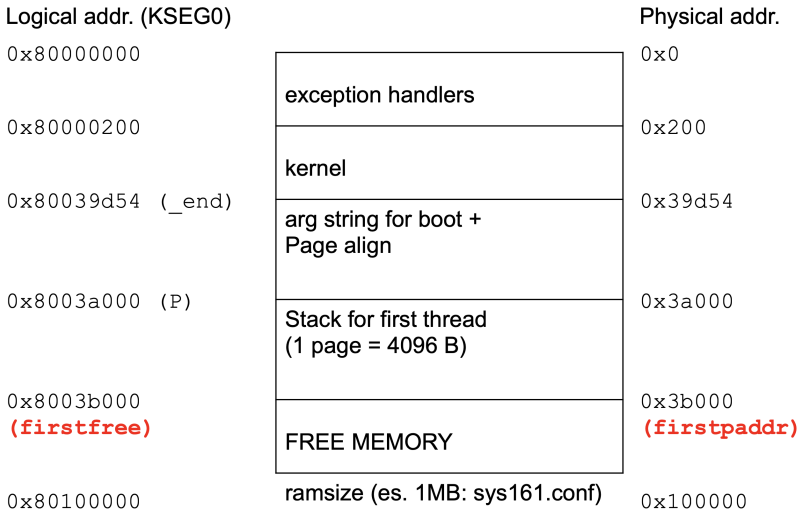

Figure 2: Configurazione iniziale del kernel

Questa configurazione può cambiare, il kernel può aumentare se vengono implementate nuove funzioni.

Si può notare che finito il boostrap, vengono salvati gli indirizzi di memoria fisica (`firstpaddr`) e logico (`firstfree`).

```
 /* Gestione dello stack per il primo thread di kernel */
.frame sp, 24, $0 /* 24-byte sp-relative frame; return addr on stack */
.mask 0x80000000, -4 /* register 31 (ra) saved at (sp+24)-4 */
addiu sp, sp, -24
sw ra, 20(sp)
la s0, _end  /* stash _end in a saved register */

/* Gestione della bootstring e allineamento di pagina*/
move a1, a0  /* move bootstring to the second argument */
move a0, s0  /* make _end the first argument */
jal strcpy  /* call strcpy(_end, bootstring) */
nop       /* delay slot */
/. . ./
sw t0, firstfree /* remember the first free page for later */
/. . ./

/* Gestione delle eccezioni */
li a0, EXADDR_UTLB
la a1, mips_utlb_handler
la a2, mips_utlb_end
sub a2, a2, a1
jal memmove
nop
/. . ./

/* Inizializzazione della TLB */
jal tlb_reset
nop
/. . ./

/* Chiamata del kernel */
jal kmain
move a0, s0   /* in delay slot */
```

```
    /. . ./
```

Code 1: Parte del codice di avviamento del kernel

1.2.2 Allocazione

L'allocazione della memoria, di default, è gestita da un allocatore che prende il nome di `dumbvm` che prevede solamente le operazioni di base. Le allocazioni di memoria sono fatte per pagine (ogni frame occupa `4096` byte) e, inizialmente, nella versione base di OS161, non ci sono strutture che tengono traccia delle pagine salvate, ma solamente un'allocazione contigua in RAM a partire da un indirizzo di base che viene incrementato ad ogni allocazione. Tuttavia, l'allocatore `dumbvm` non prevede la deallocazione della memoria.

L'allocazione è basata su una funzione fondamentale: `ram_stealmem`. Essa si trova all'interno del file `kern/arch/mips/vm/ram.c` ed è chiamata dalla funzione `getppages` (che si trova in `kern/arch/mips/vm/dumbvm.c`) che a sua volta è chiamata da diverse funzioni.

```
static
paddr_t
getppages(unsigned long npages)
{
 paddr_t addr;

 spinlock_acquire(&stealmem_lock);

 addr = ram_stealmem(npages);

 spinlock_release(&stealmem_lock);
 return addr;
}
```

Code 2: `getppages` in `ram.c`

```
paddr_t
ram_stealmem(unsigned long npages)
{
 size_t size;
 paddr_t paddr;

 size = npages * PAGE_SIZE;

 if (firstpaddr + size > lastpaddr) {
  return 0;
 }

 paddr = firstpaddr;
 firstpaddr += size;

 return paddr;
}
```

Code 3: `ram_stealmem` in `ram.c`

Gli elementi da analizzare in questo codice sono i seguenti

- `paddr_t` è un tipo physical address
- `PAGE_SIZE` è una costante definita in `vm.h` e vale 4096 bytes

- `firstpaddr` rappresenta la prima pagina fisica libera. Essa al bootstrap viene inizializzata dopo la chiamata alla funzione assembly `start.S` e, dopo l'allocazione della memoria necessaria all'avvio del sistema, viene salvata come `firstpaddr = firstfree - MIPS_KSEG0` (dove `MIPS_KSEG0` è una costante che rappresenta il primo indirizzo di memoria di `kseg0`)

La funzione è molto basilare: come argomento chiede il numero di pagine da allocare e ritorna l'indirizzo di base della memoria allocata. Se la grandezza della pagine è più grande della memoria disponibile allora la funzione ritorna il valore 0, che indica che quel numero di pagine non può essere allocato, altrimenti aggiorna il `firstpaddr` e ritorna l'indirizzo di inizio della pagine richieste.

L'allocazione delle memoria è comune sia alla memoria utente sia alla memoria di kernel in quanto:

- la memoria utente utilizza la funzione `as_prepare_load` che chiama la funziona `getppages` per inizializzare l'address space
- la memoria kernel viene allocata tramite la funzione `kmalloc` che chiama `alloc_kpages` che a sua volta chiama la funzione `getppages`

1.2.3 Codice allocazione memoria

A livello utente la memoria viene allocata tramite la funzione `as_prepare_load` che svolge il ruolo di inizializzare l'address space e si trova in `kern/include/addrspace.h`.

```
struct addrspace {
    vaddr_t as_vbase1;
    paddr_t as_pbase1;
    size_t as_npages1;
    vaddr_t as_vbase2;
    paddr_t as_pbase2;
    size_t as_npages2;
    paddr_t as_stackpbase;
};
```

Code 4: struttura `addrspace`

Gli spazi di indirizzamento virtuali per un dato processo sono descritti tramite degli oggetti `addrspace` i quali contengono la corrispondenza tra indirizzi virtuali e fisici. Questa struttura dati contiene due segmenti di memoria utente (uno per il codice e uno per i dati) e uno stack. Entrambi i segmenti sono espressi tramite indirizzo di base memoria virtuale e fisica e la grandezza del segmento espressa in numero di pagine, quindi in maniera contigua. Infine c'è il puntatore allo stack il quale non ha bisogno dalla relativa size perché in `dumbvm` è fissa ed è definita da una costante (questo potrebbe non valere per altri allocatori).

Una volta definito l'address space, può essere richiesto alla RAM tramite le funzioni persenti in `dumbvm.c` tra le quali troviamo `as_prepare_load` che come argomento chiede una struttura `addrspace` e, dopo una serie di controlli, richiede il numero di pagine tramite la funzione `getppages`

```
int
as_prepare_load(struct addrspace *as)
{
 KASSERT(as->as_pbase1 == 0);
 KASSERT(as->as_pbase2 == 0);
 KASSERT(as->as_stackpbase == 0);

 dumbvm_can_sleep();

 as->as_pbase1 = getppages(as->as_npages1);
 if (as->as_pbase1 == 0) {
  return ENOMEM;
 }
```

```
 as->as_pbase2 = getppages(as->as_npages2);
 if (as->as_pbase2 == 0) {
  return ENOMEM;
 }

 as->as_stackpbase = getppages(DUMBVM_STACKPAGES);
 if (as->as_stackpbase == 0) {
  return ENOMEM;
 }

 as_zero_region(as->as_pbase1, as->as_npages1);
 as_zero_region(as->as_pbase2, as->as_npages2);
 as_zero_region(as->as_stackpbase, DUMBVM_STACKPAGES);

 return 0;
}
```

Code 5: `as_prepare_load` in `dumbvm.c`

A livello di memoria kernel le pagine possono essere richieste tramite la funzione `alloc_kpages`.

```
vaddr_t
alloc_kpages(unsigned npages)
{
 paddr_t pa;

 dumbvm_can_sleep();
 pa = getppages(npages);
 if (pa==0) {
  return 0;
 }
 return PADDR_TO_KVADDR(pa);
}
```

Code 6: `alloc_kpages` in `dumbvm.c`

Gli elementi da analizzare in questo codice sono:

- `dumbvm_can_sleep()` è una funzione che controlla che la memoria non vada in uno stato inconsistente o instabile
- `getppages` richiede `npages` numero di pagine e ritorna l'indirizzo fisico della prima pagina libera dopo l'allocazione
- `PADDR_TO_KVADDR(pa)` traduce l'indirizzo fisico in logico a livello del kernel e lo ritorno

1.3 MMU in OS161

1.3.1 OS161 e MMU

Come accennato nelle sezioni precedenti, il lavoro della MMU è quello di tradurre gli indirizzi virtuali in quelli fisici. In OS161 la MMU cerca di tradurre ogni indirizzo virtuale utilizzando le entries presenti nella TLB all'interno della quale vengono salvati un numero fisso di indirizzi fisici. In OS161 gli indirizzi salvati all'interno del TLB sono 64 e qualora non vi sia trovato l'indirizzo virtuale che si vuole tradurre, viene scatenato un page fault.

```
int
vm_fault(int faulttype, vaddr_t faultaddress)
{
    /*...*/
 spl = splhigh();

 for (i=0; i<NUM_TLB; i++) {
  tlb_read(&ehi, &elo, i);
  if (elo & TLBLO_VALID) {
   continue;
  }
  ehi = faultaddress;
  elo = paddr | TLBLO_DIRTY | TLBLO_VALID;
  DEBUG(DB_VM, "dumbvm: 0x%x -> 0x%x\n", faultaddress, paddr);
  tlb_write(ehi, elo, i);
  splx(spl);
  return 0;
 }

 kprintf("dumbvm: Ran out of TLB entries - cannot handle page fault\n");
 splx(spl);
 return EFAULT;
}
```

Code 7: `vm_fault` in `dumbvm.c`

Come possiamo vedere la virtual memory di base di OS161 non è sofisticata in quanto se la tabella non ha più entries disponibili il sistema operativo smetterà di funzionare infatti `EFAULT` è una costante definita in `kern/include/kern/errno.h` dichiarata nella seguente maniera

```
    ...
    #define EFAULT          6      /* Bad memory reference */
    ...
```

Code 8: In `err.h` si trovano tutti i principali errori del sistema operativo

1.3.2 MIPS R3000 TLB

Il sistema operativo OS161 è basato su un'architettura MIPS R3000 e, di conseguenza, ha come TLB quella presente in tale architettura. Essa ha un MMU particolare in quanto non è presente un supporto hardware per le page tables, le uniche traduzioni di indirizzo fatte via hardware sono quelle definite dal chip TLB. Questo fa sì che ci siano diverse implicazioni nella gestione e divisione della memoria tra l'hardware e il kernel. Infatti il kernel riesce ad accedere alla memoria utente, mentre ovviamente non è possibile il contrario. La page size è di 4 kilobytes quindi gli indirizzi virtuali sono divisi in 20 bit per il numero di pagina e 12 per l'offset. La TLB contiene 64 entries e ogni entry ha una grandezza di 64 bits. Ogni entry contiene un numero di pagina virtuale, un numero di frame fisico, un identificatore di address space (che non è utilizzato in OS161) e diversi flag nello specifico:

- VPN: virtual page number

- PFN: physical frame number
- PID: (a volte chiamato anche ASID address space id) è un campo che si comporta come tag associando ogni TLB entry ad un processo, tuttavia è diverso da un process ID classico in quanto ad ogni processo che poterebbe avere una entry attiva viene assegnato un tblpid tra 0 e 63. Il kernel setta il campo PID nel entryhi register con il valore del tblpid del processo corrente. L'hardware lo confronta con il campo corrispondente nella TLB entry e rifiuta traduzioni che non corrispondono. Questo meccanismo fa sì che la TLB possa contenere entries per lo stesso numero di pagina virtuale che appartengono a diversi processi senza che ci siano conflitti
- N: no-cache è un bit che se settato dice che la pagina non deve andare nella cache date o istruzioni
- D: dirty bit indica se attivato che la entry è protetta in scrittura
- V: valid bit indica se attivato se la pagina è valida
- G: global è un bit che specifica che il PID deve essere ignorato per questa pagina

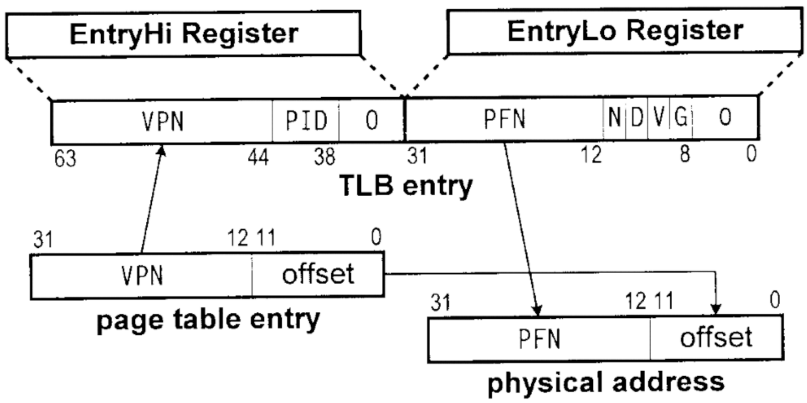

Figure 3: Traduzione degli indirizzi MIPS R3000

da notare che non sono presenti né un referenced bit né un modified bit.

Quando si vuole effettuare una taduzione di indirizzi il numero di pagina virtuale è comparatto ocon tutte le entries della tlb simultaneamente. Se viene trovato un mache e il bit G non è settato, il PID dela entry è confrontato con il tblpid corruente salvato nell'entryhi register. Se sono uguali (o se il bit G è settato) e il bit V è settato il campo PFN contiene il valido numero di frame fisico. Altrimenti viene scatenata una eccezione TLBmiss, mentre per le operazioni di scrittura (store) il bit D deve essere settato altrimenti viene scatenata una eccezione TLBmod. Dal momento che l'hardware non offre ulteriori supporti (come per esempio supporto per una page table), le eccezioni devono essere gestite direttamente dal kernel. Inoltre, visto che non ci sono bit referenced o modified questo comporta un ulteriori sforzo da parte del kernel in quanto deve sapere quali sono le pagine che sono state modificate in quanto devono essere salvate prima di essere riutilizzate. Per rendere questo possibile rendendo tutte le pagine "pulite" protette dalla scrittura (annullando il bit D nelle loro TLB) così da scatenare un'eccezione forzata TLBmod prima di scrivervi.

Questa architettura porta ad un gran numero di page faults visto che ogni TLBmiss deve essere gestita dal software (in questo caso OS161) in quanto il bisogno di tracciare le modifiche delle pagine e dei riferimenti causano un aumento dei page faults. Tuttavia questo è bilanciato da diversi aspetti tra i quali il più importante è che il segmento kseg0 non è mappato in TLB. In quanto viene utilizzato per salvare testo statico e dati del kernel il che aumenta la velocità di esecuzione del codice di kernel in quanto non occorre effettuare address translation. Infine, riduce anche il contenuto della TLB in quanto è utilizzata solamente per gli indirizzi utente e per strutture allocate dinamicamente di dati di kernel.

1.3.3 OS161 e TLB

OS161 offre delle funzioni a basso livello per gestire la TLB tra le quali:

- `tlb_write()`: modifica una specifica entry della TLB
- `tlb_random()`: modifica una entry casuale della TLB
- `tlb_read()`: legge una specifica entry della TLB
- `tlb_probe()`: cerca uno specifico numero di pagina nella TLB
- `tlb_reset()`: inizializza la TLB, è una funzione che viene invocata solamente all'avvio della sistema operativo infatti, a differenza delle altre funzioni che sono dichiarate in `kern/arch/mips/include/tlb.h`, questa viene utilizzata solamente in `start.S` che si trova in `kern/arch/sys161/main`

Queste funzioni sono descritte nel file `kern/arch/mips/vm/tlb-mips161.S` di seguito si mostra un piccolo snippet del codice:

```
    /*. . .*/
    /*
    * tlb_write: use the "tlbwi" instruction to write a TLB entry
    * into a selected slot in the TLB.
    *
    * Pipeline hazard: must wait between setting entryhi/lo and
    * doing the tlbwi. Use two cycles; some processors may vary.
    */
   .text
   .globl tlb_write
   .type tlb_write,@function
   .ent tlb_write
tlb_write:
   mtc0 a0, c0_entryhi /* store the passed entry into the */
   mtc0 a1, c0_entrylo /*   tlb entry registers */
   sll  t0, a2, CIN_INDEXSHIFT  /* shift the passed index into place */
   mtc0 t0, c0_index /* store the shifted index into the index register */
   ssnop  /* wait for pipeline hazard */
   ssnop
   tlbwi  /* do it */
   j ra
   nop
   .end tlb_write
   /*. . .*/
```

Code 9: Parte del file `tlb-mips161.S` nella quale sono descritte le funzioni per gestire la TLB

In questa parte di codice viene descritta una funzione assembly nella quale viene operata una scrittura nella TLB. Nello specifico:

1. vengono scritti nella entry della TLB i valori salvati nei registri `a0` e `a1` rispettivamente nella `c0_entryhi` e `c0_entrylo`
2. viene effettuato uno shift logico a sinistra di una quantità definita per trovare l'indice dello slot della TLB nella quale verrà scritto il contenuto
3. viene salvato nel registro di controllo, questa operazione serve per determinare quale slot sarà sovrascritto
4. le `ssnop` sono delle operazioni di sicurezza (superscalar nope) per assicurarsi non avvengano dei pipeline hazards

5. l'istruzione `tlbwi` scrive effettivamente il contenuto nella TLB

Anche le altre funzioni rimanenti sono dello stesso tipo, tutte di basso livello che lavorano con i registri di controllo e le entries della TLB.

1.4 Memoria Virtuale in xv6

1.4.1 Introduzione

Xv6 è basato sull'architettura RISC-V la quale ha istruzioni (sia kernel sia utente) che utilizzano indirizzi virtuali per essere mappati alla memoria RAM che prevede indirizzi fisici.

Xv6, in particolare, viene eseguito su Sv39 RISC-V che prevede che dei 64-bit a disposizione solamente i 39-bit inferiori vengono utilizzati, i restanti 25 superiori rimangono inutilizzati. Nella configurazione Sv39 una page table è un array di 2^{27} page table entries (PTEs). Ogni entry contiene 44-bit nei quali viene scritto il numero di pagina, mentre nei restanti 10 dei flags.

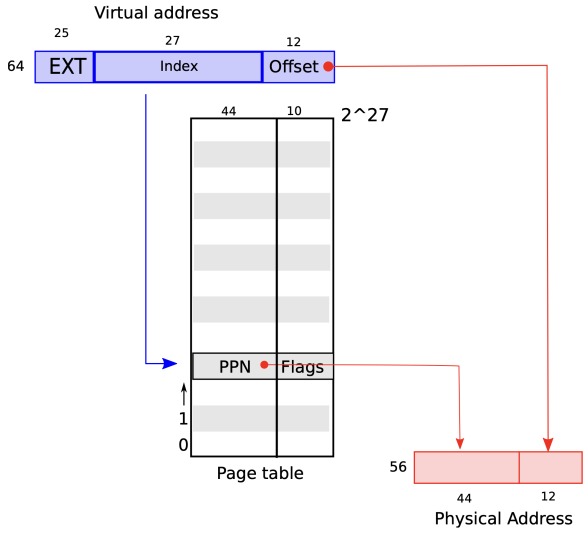

Figure 4: Rappresentazione degli indirizzi virtuali e fisici di Sv39 RISC-V

Come possiamo vedere dalla figura, dei 64 bit logici ne vengono utilizzati solamente 39, suddivisi in 27 in indice (in realtà indici come vedremo nella sezione successiva) della page table e 12 in offset di pagina, quindi ogni pagina misura `4096 byte`. Tramite l'indice si trova il numero di pagina fisico (PPN) all'entry corrispondente nella page table che verrà utilizzato insieme all'offset per la scrittura dell'indirizzo fisico.

Il motivo per il quale è stata scelta la configurazione a 39 bit è giustificato dal fatto che un relativo spazio di indirizzamento virtuale, quindi al quale ogni processo può accedere, è di `512 GB` il che, per gli sviluppatori è stato ritenuto opportuno.

1.4.2 Traduzione degli indirizzi in Sv39 RISC-V

Una CPU RISC-V traduce gli indirizzi virtuali in tre fasi. In memoria fisica vinee salvato un direttorio a tre livelli, la radice è una page table a 4096 byte che contiene 512 entries che contengono l'indirizzo fisico del prossimo nodo il quale, a sua volta è una page table di 512 entries che puntano all'ultima page table.

Per quanto riguarda i flags sono numerosi e rappresentano:

- RSW: riservato per i software superuser e può essere ignorato
- D: dirty flag indica se la pagina è stata modificata dall'ultima volta che il bit è stato pulito
- A: indica se è stato effettuato un qualsiasi tipo di accesso (lettura, scrittura o esecuzione) alla pagina dall'ultima volta che il bit è stato pulito

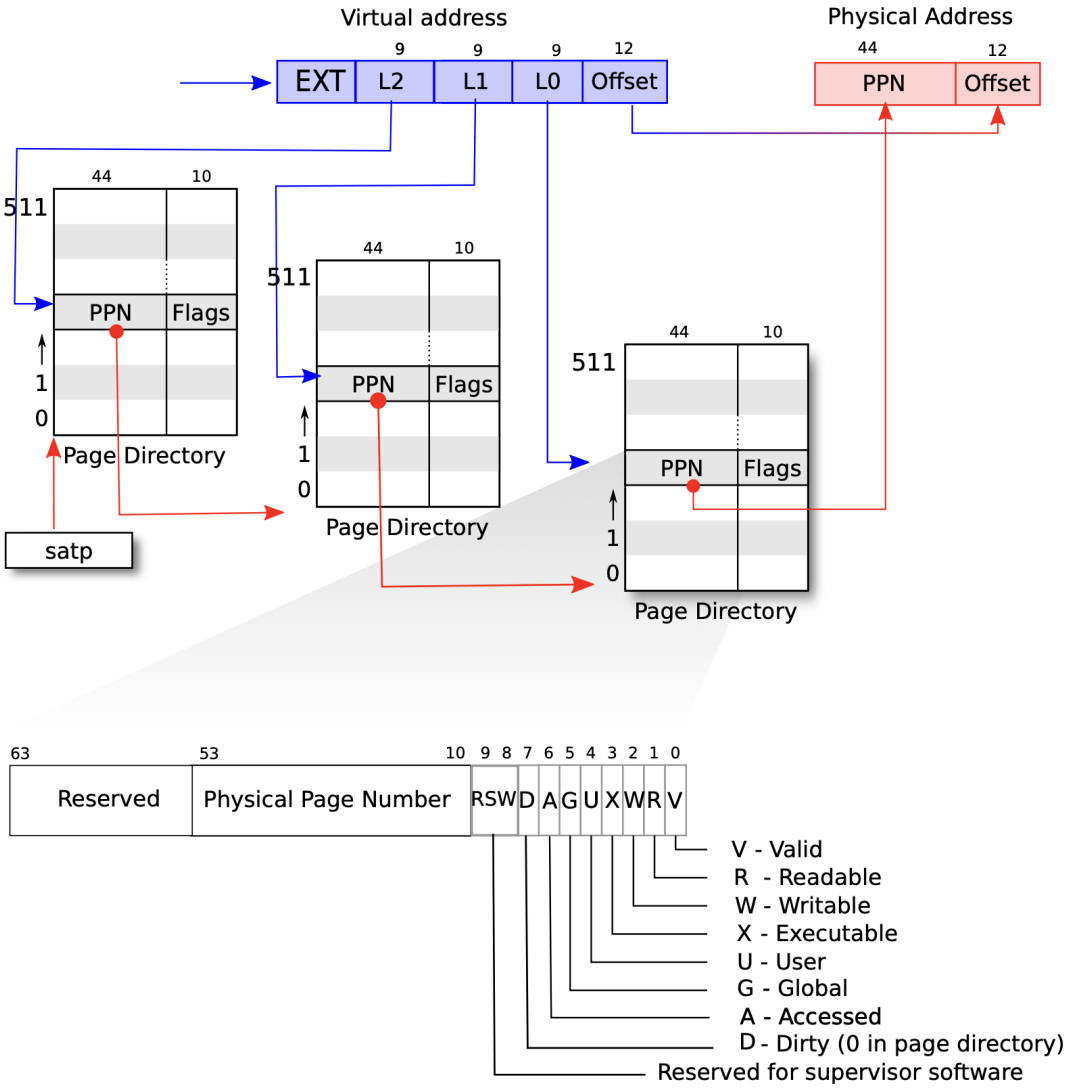

Figure 5: Dettagli della traduzione di Sv39 RISC-V

- G: indica un global mapping ovvero un mapping che esiste in tutti gli spazi di indirizzamento (previene che la TLB possa essere liberata tramite la funzione `LFENCE.VMA`)
- U: indica se si può accedere alla pagina in user-level
- X, W, R: indicano se la pagina può essere eseguita, scritta o letta
- V: indica se la pagina è valida

1.5 MMU in xv6

1.5.1 Il registro `satp`

Tutte le traduzioni eseguite dalla MMU cominciano dal `satp` ovvero il registro Supervisor Address Translation and Protection. Leggendo il valore presente in questo registro si accede all'indirizzo fisico della root page table, in xv6 si può eseguire la seguente operazione grazie alla funzione `r_satp()` che opera a basso livello chiamando una funzione assembler.

```
static inline uint64
r_satp()
{
  uint64 x;
  asm volatile("csrr %0, satp" : "=r" (x) );
  return x;
}
```

Code 10: Funzione per leggere il valore del `satp` presente in `kernel/riscv.h`

Questo registro è composto da tre parti che sono rispettivamente:

1. MODE: indica la modalità di traduzione degli indirizzi adottata (in quanto Sv39 non è l'unica configurazione di RV64, in questo campo il valore 8 rappresenta proprio la configurazione adottata da xv6)
2. ASID: address space identifier
3. PPN: indica il numero di pagina fisico, tuttavia è stato shifato a destra di 12 posizioni per essere salvato, quindi per accedere all'indirizzo originale si deve effettuare `PPN<<12`

Figure 6: La rappresentazione del `satp` per la configurazione Sv39, con la suddivisione dei campi WARL (*write any value, read legal value*)

1.5.2 Traduzione degli indirizzi

La MMU effettua la traduzione degli indirizzi seguendo i seguenti passaggi:

1. legge il registro `satp` e trova la root della page table in PPN calcolando, come detto in precedenza, l'indirizzo originale
2.

2 Algoritmi di Scheduling

2.1 Introduzione

2.1.1 Obiettivi dello scheduling

Un sistema operativo deve gestire molti processi contemporaneamente, ma la CPU può eseguirne solamente uno alla volta. Introduciamo quindi il concetto di scheduling il quale consiste nel decidere quale processo deve essere eseguito in un dato istante. Lo scheduling ha come obiettivo, quindi, quello di ottimizzare l'utilizzo della CPU trovando un bilanciamento tra:

- massimizare il throughput: quanti processi sono eseguiti
- minimizzare la latenza: quanto tempo un processo aspetta per essere eseguito
- starvation: evitare che un processo non venga mai eseguito
- completare una processo entro un tempo predefinito
- massimizzare la percentuale di utilizzo del processore

2.1.2 Preemption

Introduciamo, inoltre, il concetto di preemption il quale indica se un algoritmo di scheduling e/o un processo è interrompibile o meno. Ci sono due possibilità inerenti al preemption:

- la politica preemptive:
  1. un processo può essere interrotto per farne eseguire un altro, di conseguenza un processo non deve aspettare la terminazione di quello corrente prima di essere eseguito
  2. i processi interrotti aspettano che vengano eseguiti in memoria
  3. in ambienti multiprocessore un processo può essere rimosso dalla CPU alla quale è stato assegnato per effettuare un load balancing
  4. ha un costo maggiore in quanto la CPU deve tenere traccia dei processi in attesa, quindi salvare gli stati dei processi al tempo della interruzione, e avere un meccanismo che possa cambiare tra un processo all'altro
- la politica nonpreemptive:
  1. un processo viene eseguito dall'inizio alla fine senza cedere il controllo della CPU a prescindere dalla durata o dalla priorità di altri processi in arrivo
  2. è rischioso in quanto un processo potrebbe bloccarne un altro per un tempo indefinito
  3. ha un costo minore in quanto non bisogna tenere traccia degli stati correnti dei processi in attesa e non c'è un sovraccarico della CPU con il cambio di processo

2.1.3 Politiche degli algoritmi di scheduling

Lo scheduling, quindi, è effettuato da un componente chiamato scheduler che, decide l'ordine temporale per il quale i processi devono accedere alla CPU. Esistono vari modi per scegliere l'ordine temporale e questo dipende da diversi fattori quali:

1. la priorità del processo
2. l'arrivo temporale della richiesta del processo
3. la durata del processo
4. il tempo di attesa in coda

Le politiche di scheduling sono innumerevoli, ma tutti hanno come unico obiettivo quello di ottimizzare il rendimento della CPU. In particolare sono comuni a tutti gli algoritmi di scheduling l'equità, la scalabilità e la predicibilità, ovvero tutti i processi con le stesse caratteristiche sono tratti in maniera uguale che non varia dal numero dei processi da schedulare e, sapendo la politica adottata, si può risalire al risultato finale. Infine, in base a quali sono le caratteristiche che si vogliono rispettare, l'algoritmo di scheduling può essere più o meno complesso.

2.2 Algoritmi di Scheduling in OS161

Lo scheduler della versione base di OS161 offre un algoritmo di Round Robin con politica preemptive. Prima di esaminare in dettaglio il codice del sistema operativo spieghiamo brevemente il funzionamento dell'algoritmo in questione.

2.2.1 Algoritmo di Round Robin

Lo scheduling Round Robin è un algoritmo che si basa sulla divisione della durata dei processi in piccole porzioni chiamate quanta (il plurale di quantum) che vengono eseguite ciclicamente in base all'ordine di arrivo e senza concetto di priorità.

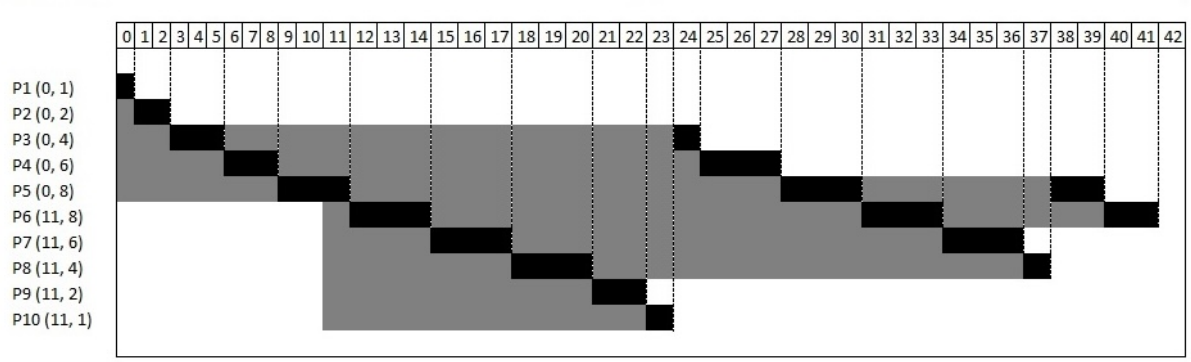

Figure 7: Rappresentazione temporale di uno scheduling round robin. In questo esempio i processi sono descritti da una coppia di valori che rappresentano il tempo di arrivo e la durata. Il time quantum è di 3 unità temporali e l'area grigia rappresenta il tempo di attesa di un processo prima di essere stato eseguito completamente.

Come possiamo notare dalla figura, è uno scheduling di tipo preemptive in quanto ogni quantum di tempo passato, se il processo in corso non è terminato viene sospeso e si passa al successivo. I processi aspettano in una coda ordinata in base al tempo di arrivo, i processi che devono essere eseguiti vengono estratti da questa coda e vengono eliminati da essa solamente una volta terminati completamente. Infine, è anche starvation free in quanto tutti i processi hanno la stessa priorità e possibilità di essere eseguiti a prescindere dalla durata di essi.

2.2.2 Scheduler in OS161

Una volta esaminato il comportamento ad alto livello dello scheduler, passiamo a studiare il funzionamento a basso livello.

I principali dettagli implementativi dell'algoritmo sono descritti in `kern/thread/clock.c` in cui troviamo la definizione temporale

```
#define SCHEDULE_HARDCLOCKS 4
#define MIGRATE_HARDCLOCKS 16
```

Code 11: Vincoli temporali in `clock.c`

che rappresentano rispettivamente ogni quanti cicli di clock hardware avviene lo scheduling e la migrazione (in caso di architettura multiprocessore si può spostare il thread in un'altra CPU se sono libere o meno occupate).

Lo scheduling è gestita dalla funzione `hardclock()` che si trova in `kern/thread/clock.c` che a sua volta chiama la funzione per schedulare o per migrare il thread e viene chiamata un numero fisso `HZ` di volte al secondo.

```
#define HZ  100
```

Code 12: Quanti hardcloks al secondo in `clock.h`

Le funzioni che gestiscono lo scheduling e la migrazione del thread sono rispettivamente `schedule()` e `thread_consider_migration()`, in questa sezione ci concentreremo sulla prima. La funzione che le racchiude viene chiamata (da ogni CPU) 100 volte al secondo e semplicemente incrementa il numero di cicli di clock e se è un multiplo di `MIGRATE_HARDCLOCKS` chiama la funzione dedicata alla migrazione mentre se è multiplo di `SCHEDULE_HARCLOCKS` viene chiamata la funzione per lo scheduling. Essa nella versione base di OS161 è vuota in quanto non è previsto nessuna forma articolata di scheduling, ma qualora si volesse implementare la si dovrebbe modificare presente in `kern/thread/thread.c`

```
void
hardclock(void)
{
 curcpu->c_hardclocks++;
 if ((curcpu->c_hardclocks % MIGRATE_HARDCLOCKS) == 0) {
  thread_consider_migration();
 }
 if ((curcpu->c_hardclocks % SCHEDULE_HARDCLOCKS) == 0) {
  schedule();
 }
 thread_yield();
}
```

Code 13: `hardclock()` in `clock.c`

L'unico scheduling effettuato dal sistema operativo è eseguito dalla funzione `thread_yield()` che a sua volta chiama `thread_switch(S_READY,NULL,NULL)`. Essa ha come parametro fondamentale `S_READY` che rappresenta lo stato del thread, quindi è pronto per essere eseguito che all'interno di uno switch case, indirizzerà il thread corrente all'interno della funzione `thread_make_runnable`.

```
static
void
thread_make_runnable(struct thread *target, bool already_have_lock)
{
 struct cpu *targetcpu;

 /* Lock the run queue of the target thread's cpu. */
 targetcpu = target->t_cpu;

 if (already_have_lock) {
  /* The target thread's cpu should be already locked. */
  KASSERT(spinlock_do_i_hold(&targetcpu->c_runqueue_lock));
 }
 else {
  spinlock_acquire(&targetcpu->c_runqueue_lock);
 }

 /* Target thread is now ready to run; put it on the run queue. */
 target->t_state = S_READY;
 threadlist_addtail(&targetcpu->c_runqueue, target);

 if (targetcpu->c_isidle && targetcpu != curcpu->c_self) {
  /*
```

```
   * Other processor is idle; send interrupt to make
   * sure it unidles.
   */
  ipi_send(targetcpu, IPI_UNIDLE);
 }

 if (!already_have_lock) {
  spinlock_release(&targetcpu->c_runqueue_lock);
 }
}
```

Code 14: La funzione `thread_make_runnable` in `kern/thread/thread.c`

Questa funzione semplicemente inserisce il thread nella coda dei thread in esecuzione. Come possiamo vedere nel file `kern/include/thread_list.h` nella versione base di OS161 la coda è semplicemente una lista doppio puntata. Quindi qualora si volesse implementare un algoritmo più sofisticato all'interno del sistema operativo si dovrebbe modificare la funzione `schedule()`, citata sopra, accedendo alla coda dei thread e modificando il loro ordine per decidere lo scheduling.

```
struct threadlistnode {
 struct threadlistnode *tln_prev;
 struct threadlistnode *tln_next;
 struct thread *tln_self;
};

struct threadlist {
 struct threadlistnode tl_head;
 struct threadlistnode tl_tail;
 unsigned tl_count;
};
```

Code 15: Struttura dati della coda dei thread in esecuzione

2.3 Algoritmi di Scheduling in xv6