

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, S317999 - Luciano Vaccarella, S319331 - Francesca Zafonte

Programmazione di Sistema - a.a. 2023-24

# Indice

<b>1</b>	<b>Introduzione ai sistemi operativi</b>	<b>4</b>
1.1	os161 . . . . .	4
1.2	xv6 vs xv6 RISC-V . . . . .	4
<b>2</b>	<b>System Calls</b>	<b>5</b>
2.1	Introduzione . . . . .	5
2.1.1	Tipi di System Calls . . . . .	6
2.2	System calls in os161 . . . . .	7
2.2.1	os161 Trapframe . . . . .	7
2.2.2	os161 MIPS System Call Handler . . . . .	7
2.2.3	System Calls supportate da os161 . . . . .	8
2.3	System calls in xv6 . . . . .	10
2.3.1	xv6 Trap Handler . . . . .	10
2.3.2	Traps user mode . . . . .	11
2.3.3	uservec . . . . .	12
2.3.4	usertrap . . . . .	13
2.3.5	usertrapret e userret . . . . .	13
2.3.6	Chiamata di una System Call . . . . .	13
2.3.7	Argomenti delle System Calls . . . . .	15
2.3.8	Traps kernel mode . . . . .	16
2.3.9	System Calls presenti in xv6 . . . . .	16
<b>3</b>	<b>Memoria Virtuale</b>	<b>18</b>
3.1	Introduzione . . . . .	18
3.1.1	Implementazione della memoria virtuale . . . . .	18
3.1.2	Memory Management Unit e Translation Lookaside Buffer . . . . .	18
3.2	Memoria Virtuale in OS161 . . . . .	19
3.2.1	Inizializzazione e avvio del kernel . . . . .	19
3.2.2	Allocazione . . . . .	21
3.2.3	Codice allocazione memoria . . . . .	22
3.3	MMU in OS161 . . . . .	24
3.3.1	OS161 e MMU . . . . .	24
3.3.2	MIPS R3000 TLB . . . . .	24
3.3.3	OS161 e TLB . . . . .	26
3.4	Memoria Virtuale in xv6 . . . . .	27
3.4.1	Introduzione . . . . .	27
3.4.2	Traduzione degli indirizzi in Sv39 RISC-V . . . . .	27
3.4.3	Allocazione . . . . .	28
3.5	MMU in xv6 . . . . .	29
3.5.1	Il registro satp . . . . .	29
3.5.2	Traduzione degli indirizzi . . . . .	30
3.5.3	xv6 e TLB . . . . .	30
<b>4</b>	<b>Algoritmi di Scheduling</b>	<b>32</b>
4.1	Introduzione . . . . .	32
4.1.1	Obiettivi dello scheduling . . . . .	32
4.1.2	Preemption . . . . .	32
4.1.3	Politiche degli algoritmi di scheduling . . . . .	32
4.2	Algoritmi di Scheduling in OS161 . . . . .	33
4.2.1	Algoritmo di Round Robin . . . . .	33
4.2.2	Scheduler in OS161 . . . . .	33
4.3	Algoritmi di Scheduling in xv6 . . . . .	35
4.3.1	Scheduler in xv6 . . . . .	35

<b>5</b>	<b>Meccanismi di Sincronizzazione</b>	<b>38</b>
5.1	Introduzione . . . . .	38
5.2	Spinlock in os161 . . . . .	38
5.2.1	Introduzione . . . . .	38
5.2.2	Acquisizione dello spinlock . . . . .	38
5.2.3	Rilascio dello spinlock . . . . .	39
5.2.4	Altre funzioni inerenti allo spinlock . . . . .	40
5.3	Wait Channels in os161 . . . . .	40
5.3.1	Introduzione . . . . .	40
5.3.2	Attesa di un thread . . . . .	41
5.3.3	Risveglio di un thread . . . . .	41
5.3.4	Altre funzioni inerenti ai wait channel . . . . .	42
5.4	Semafori in os161 . . . . .	43
5.4.1	Introduzione . . . . .	43
5.4.2	Le funzioni $P()$ e $V()$ . . . . .	43
5.4.3	Altre funzioni inerenti ai semafori . . . . .	44
5.5	Spinlock in xv6 . . . . .	45
5.5.1	Acquisizione dello spinlock . . . . .	45
5.5.2	Rilascio dello spinlock . . . . .	45
5.5.3	Altre funzioni inerenti allo spinlock . . . . .	46
5.6	Wait channels in xv6 . . . . .	46
5.6.1	Attesa di un processo . . . . .	47
5.6.2	Risveglio di un processo . . . . .	47
5.7	Sleeplocks in xv6 . . . . .	48
5.7.1	Acquisizione dello sleeplock . . . . .	48
5.7.2	Rilascio dello sleeplock . . . . .	48
5.7.3	Altre funzioni inerenti allo sleeplock . . . . .	49
<b>6</b>	<b>Implementazione di una System Call in xv6</b>	<b>50</b>
6.1	Introduzione . . . . .	50
6.2	File modificati all'interno di xv6 . . . . .	50
6.2.1	syscall.h . . . . .	50
6.2.2	syscall.c . . . . .	50
6.2.3	sysproc.c . . . . .	51
6.2.4	kalloc.c . . . . .	51
6.2.5	user.h . . . . .	52
6.2.6	usys.pl . . . . .	52
6.2.7	Makefile . . . . .	52
6.2.8	getfreepages.c . . . . .	52
6.2.9	testkalloc.c . . . . .	52
6.2.10	Immagini del test della syscall . . . . .	53

# 1 Introduzione ai sistemi operativi

## 1.1 os161

os161 è un sistema operativo originariamente sviluppato per essere eseguito sull'architettura MIPS (Microprocessor without Interlocked Pipeline Stages). MIPS è un'architettura di processore RISC (Reduced Instruction Set Computer) nota per la sua semplicità e efficienza. Tuttavia, os161 è stato adattato per funzionare su diverse varianti di architetture MIPS, come MIPS32 e MIPS64. MIPS è noto per la sua semplicità e chiarezza. Le istruzioni sono di dimensioni fisse e seguono un modello regolare. L'architettura MIPS è in grado di esemplificare concetti fondamentali dei sistemi operativi, come la gestione dei processi, la gestione della memoria e la sincronizzazione tra processi.

## 1.2 xv6 vs xv6 RISC-V

xv6 è un sistema operativo educativo basato su Unix V6, creato presso l'Università di Berkeley, con l'obiettivo di illustrare i concetti fondamentali dei sistemi operativi. xv6 è stato inizialmente sviluppato per l'architettura x86 (Intel), ma è stato successivamente portato a diverse altre architetture, inclusa RISC-V. Creato nel 2006 a scopo educativo, nel 2020 ne è stata realizzata una conversione per RISC-V. Parte del codice della non più mantenuta versione per x86 è ricavato da JOS (un precedente sistema operativo Unix-like).

Le principali differenze tra le due versioni sono le seguenti:

1. Architettura di destinazione: x86 e RISC-V sono due architetture differenti, la prima ampiamente diffusa, la seconda sta prendendo piede nei dispositivi odierni in quanto è un progetto open-source
2. Interfaccia hardware: interagiscono con due hardware differenti ad esempio la gestione della memoria è diversa
3. Sistema di istruzioni: RISC-V offre un ISA molto semplice ed efficiente

Tra le due opzioni abbiamo deciso di effettuare il confronto tra os161 e xv6 RISC-V, in primis perchè un progetto più recente e mantenuto ancora al giorno d'oggi (mentre la versione x86 non lo è più), in secondo luogo perchè la tecnologia RISC-V sta prendendo piede nel mondo dell'informatica e, di conseguenza, a nostro avviso più interessante.

## 2 System Calls

### 2.1 Introduzione

Una system calls è un meccanismo usato da un processo a livello utente o applicativo per richiedere un servizio a livello kernel del sistema operativo. Le system calls forniscono dunque un'interfaccia ai servizi forniti da un sistema operativo (esecuzione di programmi, operazioni di I/O, file system ecc..) e sono generalmente scritte in C, C++.

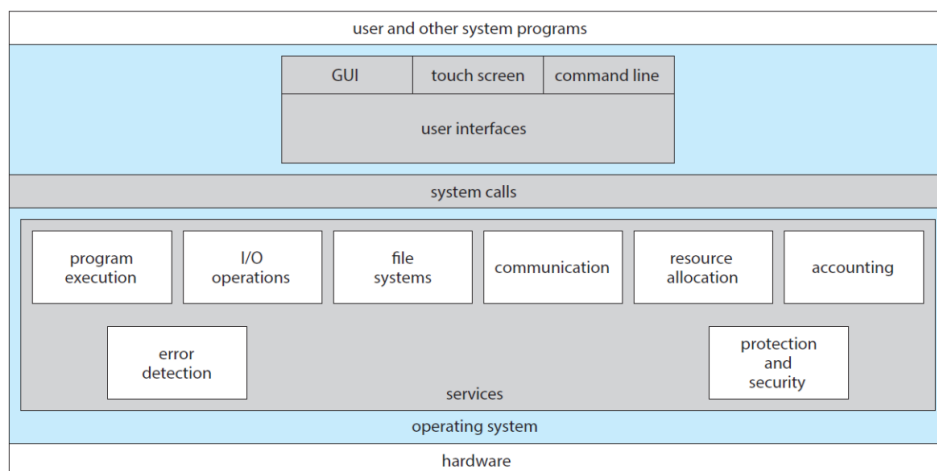


Figure 1: Rappresentazione della interfaccia delle chiamate a sistema

Il sistema operativo associa un numero o un ID ad ogni system call ed una tabella nello spazio del kernel che dice per ogni numero di system call, cosa deve essere fatto. Quando chiamiamo una system call tipicamente avviene uno switching da user level a kernel level, passaggio che avviene attraverso una particolare istruzione: l'istruzione **trap**, passando il numero associato alla system call. Il sistema

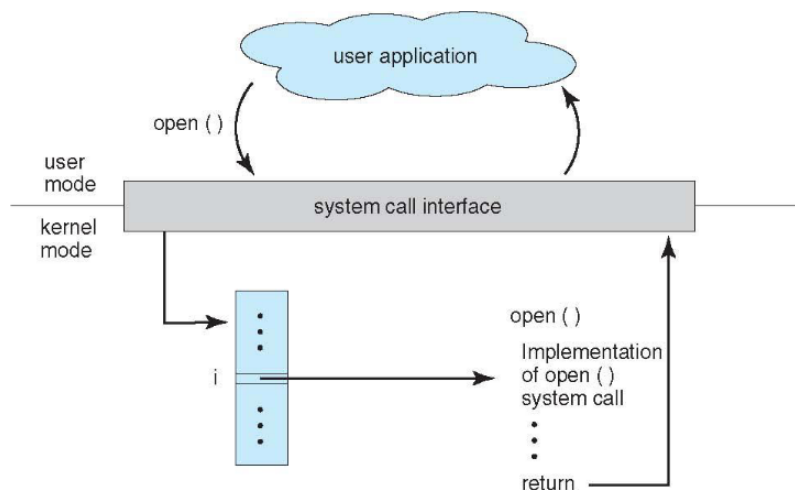


Figure 2: Rappresentazione ad alto livello di una system call, in questo caso la `open ()`

operativo va alla struttura dati del numero di chiamata di sistema ed esegue le operazioni, quando termina l'esecuzione si ritorna all'user mode. Normalmente alle system calls vengono passati dei parametri ed il modo più semplice per passare quest'ultimi dall'user al kernel è quello di metterli nei registri e nello stack. In pratica l'user program, che ha i parametri, va a caricare l'indirizzo di dove si trovano in un registro e quando il sistema operativo eseguirà l'istruzione trap per una determinata sys call oltre ad andare ad

identificare il codice di esecuzione corrispondente userà l'indirizzo di dove si trovano i parametri.

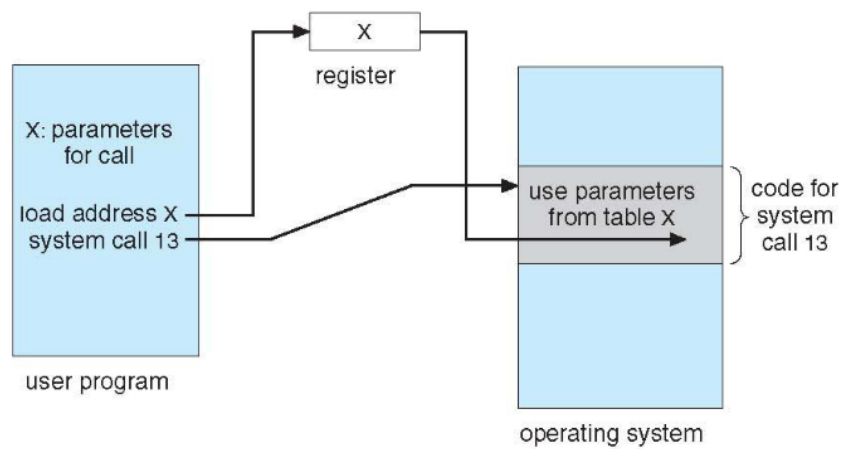


Figure 3: Rappresentazione al alto livello di come il sistema operativo trova la system call da eseguire

### 2.1.1 Tipi di System Calls

1. **Gestione dei processi/thread:** include chiamate di sistema per la creazione, l'esecuzione e la gestione dei processi/thread

- create/terminate processo
- end, abort
- load, execute
- get/info/set attributi dei processi
- wait for time
- wait event, signal event
- allocate e free memoria
- ecc.

2. **Gestione dei file e dei file system:** offre chiamate di sistema per la lettura e scrittura di file, la manipolazione delle directory e l'accesso ai file

- create/delete file
- open/close file
- read/write
- get/set attributi dei file
- ecc.

3. **Gestione dei dispositivi**

- request/release
- read/write
- get/set attributi dei dispositivi
- logically attach/detach
- ecc.

#### 4. Gestione delle informazioni

- get/set time or date
- get/set dati del sistema
- get/set processi, file o attributi dei device
- ecc.

#### 5. Comunicazione

- create/delete della connessione
- send/receive messaggi (dal client al server)
- trasferimento di informazioni di stato
- ecc.

#### 6. Protezione

- controllare l'accesso alle risorse
- get/set permissions
- allow/deny accesso utente

## 2.2 System calls in os161

In OS161 implementato su MIPS c'è un **unico gestore delle eccezioni**, un exception handler, che viene chiamato anche per le syscalls e le interruzioni. La prima cosa che farà l'exception handler è quella di riconoscere da chi è stato chiamato poiché vi è una funzione di gestione diversa per eccezioni, chiamate di sistema e interruzioni. Ogni volta che viene chiamata una system call, l'esecuzione viene passata dalla modalità utente alla modalità kernel. Mentre il kernel gestisce la chiamata di sistema, lo stato della CPU dell'applicazione viene salvato in un **trap frame** sullo stack del kernel del thread e i registri della CPU sono disponibili per mantenere lo stato di esecuzione del kernel.

### 2.2.1 os161 Trapframe

Il trapframe è una struttura che serve per salvare lo stato della CPU cioè i registri dei task in esecuzione quando si attiva una trap.

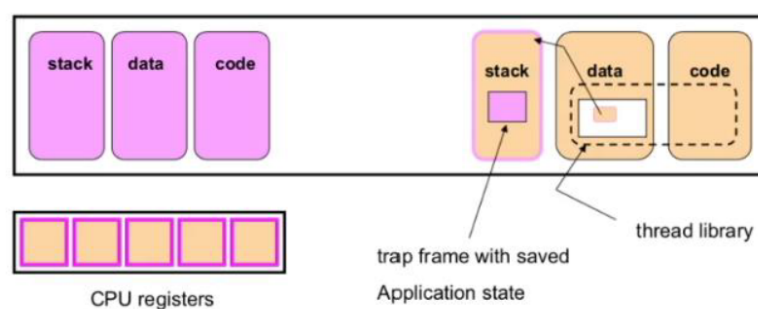


Figure 4: Rappresentazione dei due stack in os161: uno lato utente e l'altro lato kernel

### 2.2.2 os161 MIPS System Call Handler

La funzione che viene attivata quando il gestore di eccezioni si accorge che si tratta di una System call è la seguente:

```

void
syscall(struct trapframe *tf)
{
    / . . . /
    callno = tf->tf_v0;
    retval = 0;

    switch (callno) {
        case SYS_reboot:
            err = sys_reboot(tf->tf_a0);
            break;

        /* Add stuff here */

    default:
        kprintf("Unknown syscall %d\n", callno);
        err = ENOSYS;
        break;
    }
}

```

Code 1: La funzione `syscall` presente in `kern/arch/mips/syscall/syscall.c`

La funzione si chiama `syscall` e riceve come unico parametro il puntatore al `trapframe`. In sostanza è un costrutto `switch case`. La funzione va a guardare uno dei contenuti del `trapframe`, il campo `tf_v0`, cioè il call number, il numero della System Call.

Il numero della eccezione relativo alla `syscall` a livello di costanti è `EX_SYS 8`:

```

#define EX_IRQ      0    /* Interrupt */
#define EX_MOD      1    /* TLB Modify (write to read-only page) */
#define EX_TLBL     2    /* TLB miss on load */
#define EX_TLBS     3    /* TLB miss on store */
#define EX_ADEL     4    /* Address error on load */
#define EX_ADES     5    /* Address error on store */
#define EX_IBE      6    /* Bus error on instruction fetch */
#define EX_DBE      7    /* Bus error on data load *or* store */
#define EX_SYS      8    /* Syscall */
#define EX_BP       9    /* Breakpoint */
#define EX_RI      10    /* Reserved (illegal) instruction */
#define EX_CPU      11    /* Coprocessor unusable */
#define EX_OVF      12    /* Arithmetic overflow */

```

Code 2: Le eccezioni presenti in `kern/arch/mips/include/trapframe.h`

### 2.2.3 System Calls supportate da `os161`

- `exit` - **void** `exit(int status)` - per terminare il processo
- `chdir` - **int** `chdir(const char *pathname)` - per cambiare la directory corrente
- `close` - **int** `close(int fd)` - per chiudere il file
- `dup2` - **int** `dup2(int oldfd, int newfd)` - per clonare file descriptor
- `execv` - **int** `execv(const char *program, char **args)` - per eseguire un programma
- `fork` - **pid\_t** `fork(void)` - copiare il processo corrente



- `fstat` - `int fstat (int fd, struct stat *statbuf)` - ottenere informazioni sullo stato del file
- `fsync` - `int fsync(int fd)` - scaricare i dati del filesystem per un file specifico su disco
- `ftruncate` - `int ftruncate(int fd, off_t filesize)` - impostare la dimensione di un file
- `_getcwd` - ottenere il nome della directory di lavoro corrente (backend)
- `getdirent` - `int getdirent(int fd, char *buf, size_t buflen)` - leggere il nome del file dalla directory
- `getpid` - `pid_t getpid(void)` - ottenere l'ID del processo
- `ioctl` - `int ioctl(int fd, int code, void *data)` - varie operazioni di I/O del dispositivo
- `link` - `int link(const char *oldname, const char *newname)` - creare un collegamento fisico a un file
- `lseek` - `off_t lseek(int fd, off_t pos, int whence)` - cambiare la posizione corrente nel file descriptor
- `lstat` - `int lstat(const char *pathname, struct stat *statbuf)` - ottenere informazioni sullo stato del file
- `mkdir` - `int mkdir(const char *pathname, mode_t mode)` - creare una directory
- `open` - `int open(const char *filename, int flags)` - aprire un file
- `pipe` - `int pipe(int *fds)` - creare un oggetto tubo
- `read` - `ssize_t read(int fd, void *buf, size_t buflen)` - leggere i dati dal file
- `readlink` - `int readlink(const char *path, char *buf, size_t len)` - recuperare il contenuto del collegamento simbolico
- `reboot` - `int reboot(int code)` - riavviare o arrestare il sistema
- `remove` - `int remove(const char *pathname)` - eliminare un file
- `rename` - `int rename(const char *oldname, const char *newname)` - rinominare o spostare un file
- `rmdir` - `int rmdir(const char *pathname)` - rimuovere la directory
- `sbrk` - `void * sbrk(intptr_t amount)` - impostare l'interruzione del processo (alloca memoria)
- `stat` - `int stat(const char *pathname, struct stat *statbuf)` - ottenere informazioni sullo stato del file
- `symlink` - `int symlink(const char *oldname, const char *linkname)` - creare un collegamento simbolico
- `sync` - `void sync(void)` - scaricare i dati del filesystem sul disco
- `_time` - ottenere l'ora del giorno
- `waitpid` - `pid_t waitpid(pid_t pid, int *status, int options)` - attendere l'uscita del processo
- `write` - `ssize_t write(int fd, const void *buf, size_t buflen)` - scrivere i dati su file

## 2.3 System calls in xv6

Il kernel xv6 fornisce un subset di servizi e system calls del tradizionale kernel Unix V6.

System Call: il programma user che è in esecuzione esegue l'istruzione `ecall` per chiedere al kernel di fare qualcosa. Un'applicazione che vuole invocare una funzione del kernel deve switchare da modalità utente a modalità kernel (o supervisore). Una volta che la CPU è passata alla modalità supervisore, il kernel può quindi convalidare gli argomenti della chiamata di sistema, decidere se l'applicazione è autorizzata a eseguire l'operazione richiesta, quindi negarla o eseguirla.

xv6 kernel files:

File	Description
bio.c	Disk block cache for the file system.
console.c	Connect to the user keyboard and screen.
entry.S	Very first boot instructions.
exec.c	<code>exec()</code> system call.
file.c	File descriptor support.
fs.c	File system.
kalloc.c	Physical page allocator.
kernelvec.S	Handle traps from kernel, and timer interrupts.
log.c	File system logging and crash recovery.
main.c	Control initialization of other modules during boot.
pipe.c	Pipes.
plic.c	RISC-V interrupt controller.
printf.c	Formatted output to the console.
proc.c	Processes and scheduling.
sleeplock.c	Locks that yield the CPU.
spinlock.c	Locks that don't yield the CPU.
start.c	Early machine-mode boot code.
string.c	C string and byte-array library.
swtch.S	Thread switching.
syscall.c	Dispatch system calls to handling function.
sysfile.c	File-related system calls.
sysproc.c	Process-related system calls.
trampoline.S	Assembly code to switch between user and kernel.
trap.c	C code to handle and return from traps and interrupts.
uart.c	Serial-port console device driver.
virtio_disk.c	Disk device driver.
vm.c	Manage page tables and address spaces.

Figure 5: xv6 kernel source files

Un processo può effettuare una chiamata di sistema eseguendo l'istruzione `ecall` RISC-V. Questa istruzione aumenta il livello di privilegio hardware e modifica il contatore del programma in un punto di ingresso definito dal kernel. Il codice nel punto di ingresso passa a uno stack del kernel ed esegue le istruzioni del kernel che implementano la chiamata di sistema. Una volta completata la chiamata di sistema, il kernel torna allo stack utente e ritorna nello spazio utente chiamando l'istruzione `sret`, che riduce il livello di privilegi hardware e riprende l'esecuzione delle istruzioni dell'utente subito dopo l'istruzione della chiamata di sistema.

### 2.3.1 xv6 Trap Handler

Ciascuna CPU RISC-V ha una serie di registri di controllo nei quali il kernel scrive per indicare alla CPU come gestire sys call, eccezioni o interrupts e che il kernel può leggere per sapere che tipo di trap

si è verificata. Con trap si indica genericamente una delle seguenti situazioni: system calls, interrupts o exceptions. I registri più importanti sono (in kernel mode):

- `stvec` (trap vector): contiene l'indirizzo del gestore della trap (handler code), al quale saltare per gestirla. Il kernel scrive questo indirizzo qui.
  - `Kernelvec`: gestisce le traps che avvengono in kernel mode
  - `Uservec`: gestisce le traps che avvengono in user mode
- `sepc`: contiene l'indirizzo dal quale è stato effettuato il salto a `stvec`. Quando avviene una trap, la CPU salva il valore del PC (Program counter) qui. L'istruzione `sret` (return from trap) copia `sepc` in `pc`.
- `scause`: contiene il numero che descrive chi ha scatenato la trap. La CPU lo scrive.
- `sscratch`: il kernel inserisce qui un valore che torna utile proprio all'inizio della gestione di una trappola.
- `sstatus`: contiene un bit `SIE` (Interrupts enabled) che controlla se gli interrupt del dispositivo sono abilitati. Se il kernel cancella `SIE`, RISC-V posticiperà gli interrupt del dispositivo finché il kernel non imposta `SIE`. L'`SPP` (Previous Privilege Level, 0=user, 1=supervisor) bit indica se un trap proviene dalla modalità utente o dalla modalità supervisore e controlla cosa la modalità `sret` ritorna.

Quando è necessario forzare una trap, l'hardware RISC-V esegue le seguenti operazioni (delega la gestione della trap al supervisor mode) per tutti i tipi di trap eccetto gli interrupt del timer:

1. se la trap è un device interrupt e il bit di stato `SIE` è azzerato, non viene eseguita nessuna delle operazioni seguenti
2. disabilitazione degli interrupt, cancellando il bit di stato `SIE`
3. copia del valore contenuto in PC in `sepc`.
4. salvataggio della modalità corrente (utente o supervisore) nel bit `SPP` in `sstatus`
5. impostazione di `scause` per riflettere la causa della trappola
6. impostazione della modalità supervisore
7. copia di `stvec` sul PC
8. inizio dell'esecuzione sul nuovo PC

Da notare che la CPU non passa alla tabella delle pagine del kernel, non passa a uno stack nel kernel e non salva nessun registro diverso dal `pc`. Il software del kernel deve eseguire queste attività mentre la CPU effettua un lavoro minimo al fine di avere una maggiore flessibilità con il software.

### 2.3.2 Traps user mode

Una trappola può verificarsi durante l'esecuzione nello spazio utente se il programma utente effettua una chiamata di sistema (istruzione `ecall`), o fa qualcosa di illegale, oppure se un dispositivo interrompe l'esecuzione. Il percorso di alto livello di una trap dello spazio utente è:

1. `uservec` (`kernel/trampoline.S:16`) questo è l'indirizzo salvato in `stvec`
2. `usertrap` (`kernel/trap.c:37`)
3. e al ritorno, `usertrapret` (`kernel/trap.c:90`)
4. `userret` (`kernel/trampoline.S:16`)

### 2.3.3 uservec

Poiché l'hardware RISC-V non cambia le page table durante una trap (il registro `satp` punta alla page table corrente in memoria), la page table utente deve includere una mappatura per `uservec` (il vettore delle istruzioni delle trappole a cui `stvec` punta). `uservec` cambia `satp` per puntare alla page table del kernel. Questo cambio non deve causare problemi, così la page table del kernel deve anche contenere una mappatura a `uservec`. Dunque, `uservec` deve essere mappato allo stesso indirizzo nella tabella delle pagine del kernel e nella tabella delle pagine utente per continuare l'esecuzione delle istruzioni dopo il cambio. `xv6` usa una pagina trampoline (trampolino), che contiene `uservec`, mappata allo stesso indirizzo virtuale (trampoline) nella tabella delle pagine del kernel e in ogni tabella delle pagine utente. Il contenuto della trappola è impostato in `trampoline.S`, e nell'esecuzione del codice utente `stvec` è impostato su `uservec`. Quando `uservec` inizia l'esecuzione, tutti e 32 i registri contengono valori

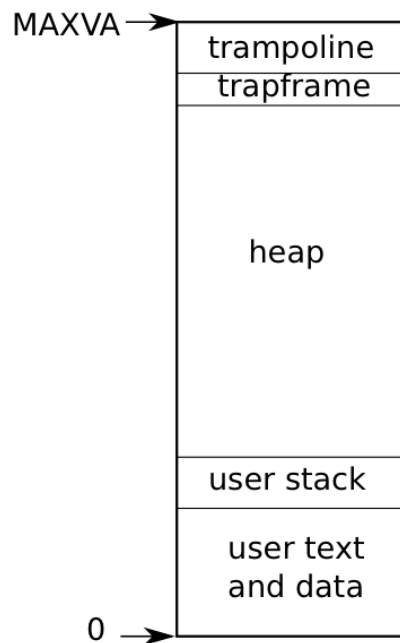


Figure 6: Layout del virtual address space di un processo

posseduti dal codice interrotto. L'istruzione `csrrw` all'inizio di `uservec` scambia i contenuti del primo registro utente `a0` con `sscratch` (registro). Il compito successivo di `uservec` è salvare i registri utente:

- prima di entrare in user mode, il kernel ha precedentemente impostato `sscratch` per puntare a un trapframe specifico per processo che (tra le altre cose) ha spazio per salvare tutti i registri utente (`kernel/proc.h:44`).
- `satp` si riferisce ancora alla tabella delle pagine utente e `uservec` ha bisogno che il trapframe sia mappato nello spazio degli indirizzi utente. Allora quando si crea ogni processo, `xv6` assegna una pagina per il trapframe del processo, e si assicura che sia sempre mappato all'indirizzo virtuale utente `TRAPFRAME`, che è appena sotto `TRAMPOLINE`. Il `p->trapframe` punta anche al trapframe, anche se al suo indirizzo fisico, in modo che il kernel possa usarlo attraverso la tabella delle pagine del kernel.
- dopo lo scambio di `a0` e `sscratch`, `a0` contiene un puntatore al trapframe del processo corrente. `uservec` salva ora tutti i registri utente lì, compreso il `a0` dell'utente, letto da `sscratch`.

Il trapframe contiene puntatori allo stack kernel del processo corrente, all'`hartid` della CPU corrente, all'indirizzo di `usertrap` e all'indirizzo della tabella delle pagine del kernel. `uservec` recupera questi valori, cambia `satp` alla tabella delle pagine del kernel, e chiama `usertrap`.

### 2.3.4 usertrap

Il compito di `usertrap` è determinare la causa della trappola (eccezione, device interrupt, syscall, timer interrupt), elaborarla e ritornare (`kernel/trap.c:37`):

- aggiorna `stvec` in `kernelvec`, (perché gli interrupt utilizzano gestori diversi mentre sono in modalità kernel), in modo che una trappola mentre è nel kernel venga gestita da `kernelvec`
- salva il `sepc` (il contatore del programma utente salvato) ancora perché potrebbe esserci un cambio di processo in `usertrap` che potrebbe causare sovrascrittura di `sepc`

Se la trap è una chiamata di sistema, `syscall()` la gestisce; se è un'interruzione del dispositivo, `devintr()`; altrimenti è un'eccezione, e il kernel termina il processo che ha causato il guasto. Il percorso della chiamata di sistema aggiunge quattro al PC utente salvato perché RISC-V, nel caso di una chiamata di sistema, lascia il puntatore del programma puntato all'istruzione `ecall`. All'uscita, `usertrap` controlla se il processo è stato terminato o se deve cedere la CPU (se questa trappola è un'interruzione del timer).

### 2.3.5 usertrapret e userret

Una volta che la trap è stata processata, il primo passo nel ritorno allo spazio utente è la chiamata a `usertrapret` (`kernel/trap.c:90`). Questa funzione configura i registri di controllo RISC-V per prepararsi a una futura trappola dallo spazio utente. Questo comporta il cambio di `stvec` per fare riferimento a `uservec`, la preparazione dei campi `trapframe` di cui `uservec` ha bisogno, e l'impostazione di `sepc` al contatore del programma utente precedentemente salvato. Alla fine, `usertrapret` chiama `userret` sulla pagina trampolino che è mappata sia nelle tabelle delle pagine utente che in quelle del kernel; il motivo è che il codice assembly in `userret` cambierà le tabelle delle pagine. La chiamata di `usertrapret` a `userret` passa un puntatore alla tabella delle pagine utente del processo in `a0` e `TRAPFRAME` in `a1` (`kernel/trampoline.S:88`). `userret` cambia `satp` alla tabella delle pagine utente del processo. Ricorda che la tabella delle pagine utente mappa sia la pagina trampolino che `TRAPFRAME`, ma nient'altro dal kernel. Ancora una volta, il fatto che la pagina trampolino sia mappata allo stesso indirizzo virtuale nelle tabelle delle pagine utente e del kernel è ciò che consente a `uservec` di continuare l'esecuzione dopo il cambio di `satp`. `userret` copia l'utente del `trapframe` `a0` in `sscratch` in preparazione per un successivo scambio con `TRAPFRAME`. Da questo momento in poi gli unici dati che `userret` potrà utilizzare saranno il contenuto del registro e il contenuto del `trapframe`. Il prossimo `userret` ripristina i registri utente salvati dal `trapframe`, fa uno scambio finale di `a0` e `sscratch` per ripristinare l'utente `a0` e salvare `TRAPFRAME` per la trappola successiva e usa `sret` per tornare allo spazio utente. `usertrapret` e `userret` svolgono rispettivamente le azioni di `usertrap` e `uservec`.

### 2.3.6 Chiamata di una System Call

Vediamo come la chiamata dell'utente alla system call `exec` raggiunge la sua implementazione nel kernel.

- Il codice utente inserisce gli argomenti per la chiamata di sistema nei registri `a0` e `a1`, e inserisce il numero di chiamata di sistema in `a7`. I numeri di chiamata di sistema corrispondono alle voci nell'array `syscalls`, una tabella di puntatori a funzione (`kernel/syscall.c:108`).

```
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork]      sys_fork,
[SYS_exit]      sys_exit,
[SYS_wait]      sys_wait,
[SYS_pipe]      sys_pipe,
[SYS_read]      sys_read,
[SYS_kill]      sys_kill,
[SYS_exec]      sys_exec,
[SYS_fstat]     sys_fstat,
[SYS_chdir]     sys_chdir,
```

```

[SYS_dup]      sys_dup,
[SYS_getpid]   sys_getpid,
[SYS_sbrk]     sys_sbrk,
[SYS_sleep]    sys_sleep,
[SYS_uptime]   sys_uptime,
[SYS_open]     sys_open,
[SYS_write]    sys_write,
[SYS_mknod]    sys_mknod,
[SYS_unlink]   sys_unlink,
[SYS_link]     sys_link,
[SYS_mkdir]    sys_mkdir,
[SYS_close]    sys_close,
};

```

Code 3: Il codice dell'array che mappa il numero della system call alla inerente funzione

- L'istruzione `ecall` fa scattare una trappola nel kernel ed esegue `uservec`, `usertrap` e poi `syscall`, come visto in precedenza
- `syscall` (`kernel/syscall.c:133`) recupera il numero di chiamata di sistema dal registro `a7` salvato nel `trapframe` e lo usa per indicizzare `syscalls`

```

void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,
        // and store its return value in p->trapframe->a0
        p->trapframe->a0 = syscalls[num]();
    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

Code 4: La funzione `syscall.c`

- `a7` contiene `SYS_exec` (`kernel/syscall.h:8`), risultando in una chiamata alla funzione di implementazione della chiamata di sistema `sys_exec`

```

// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup     10
#define SYS_getpid  11
#define SYS_sbrk    12

```

```

#define SYS_sleep 13
#define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21

```

Code 5: I numeri delle system calls in kernel/syscall.h

- Quando la funzione di implementazione della chiamata di sistema restituisce, syscall registra il suo valore di ritorno in `p->trapframe->a0`

```
p->trapframe->a0 = syscalls[num]();
```

Code 6: Dove viene chiamato l'array che mappa le syscalls

La convenzione di chiamata in linguaggio C su RISC-V inserisce i valori di ritorno in `a0`. Le chiamate di sistema convenzionalmente restituiscono numeri negativi per indicare errori e numeri zero o positivi per indicare il successo. Se il numero di chiamata di sistema è invalido, syscall stampa un errore e restituisce -1.

### 2.3.7 Argomenti delle System Calls

Le implementazioni delle chiamate di sistema nel kernel devono trovare gli argomenti passati dal codice utente. Gli argomenti sono inizialmente nei registri. Il codice della trap del kernel salva i registri utente nel trap frame del processo corrente. Le funzioni `argint`, `argaddr` e `argfd` recuperano l'n-esimo argomento della chiamata di sistema dal trap frame come un intero, un puntatore o un descrittore di file. Tutte chiamano `argraw` per recuperare il registro utente salvato appropriato (kernel/syscall.c:35):

```

static uint64
argraw(int n)
{
    struct proc *p = myproc();
    switch (n) {
        case 0:
            return p->trapframe->a0;
        case 1:
            return p->trapframe->a1;
        case 2:
            return p->trapframe->a2;
        case 3:
            return p->trapframe->a3;
        case 4:
            return p->trapframe->a4;
        case 5:
            return p->trapframe->a5;
    }
    panic("argraw");
    return -1;
}

```

Code 7: La funzione `argraw` che ritorna il registro utente salvato

Alcune chiamate di sistema passano puntatori come argomenti, e il kernel deve usare questi puntatori per leggere o scrivere nella memoria utente. Questi puntatori presentano due sfide:

- in primo luogo, il programma utente potrebbe essere difettoso o malevolo e potrebbe passare al kernel un puntatore non valido o un puntatore destinato a ingannare il kernel affinché acceda alla memoria del kernel invece della memoria utente
- in secondo luogo, i mapping della page table del kernel xv6 non sono gli stessi della page table utente, quindi il kernel non può utilizzare istruzioni ordinarie per caricare o memorizzare da indirizzi forniti dall'utente

### 2.3.8 Traps kernel mode

xv6 configura i registri della trap della CPU in modo leggermente diverso a seconda che il codice utente o del kernel stia eseguendo. Quando il kernel è in esecuzione su una CPU, il kernel imposta `stvec` affinché punti al codice assembly in `kernelvec` (`kernel/kernelvec.S:10`). Poiché xv6 è già nel kernel, `kernelvec` può fare affidamento su `satp` impostato sulla page table del kernel e sul puntatore allo stack che fa riferimento ad uno stack del kernel valido. `kernelvec` salva tutti i registri in modo che il codice interrotto possa riprendere eventualmente senza disturbi.

1. `kernelvec` salva i registri nello stack del thread del kernel interrotto. Questo è particolarmente importante se la trappola causa un passaggio a un thread diverso: in questo caso, la trappola tornerà effettivamente nello stack del nuovo thread, lasciando i registri del thread interrotto salvati in modo sicuro sul suo stack
2. `kernelvec` salta a `kerneltrap` (`kernel/trap.c:134`)
3. `kerneltrap` è preparato per due tipi di trappole ovvero interruzioni di dispositivi ed eccezioni:
  - Chiama `devintr()` (`kernel/trap.c:177`) per controllare e gestire le prime. Se la trappola non è un'interruzione di dispositivo, deve essere un'eccezione e questa è sempre un errore fatale se si verifica nel kernel di xv6; il kernel chiama `panic` e smette di eseguire
  - Se `kerneltrap` è stata chiamata a causa di un'interruzione del timer e un thread del kernel di un processo è in esecuzione (piuttosto che un thread dello scheduler), `kerneltrap` chiama `yield()` per dare la possibilità ad altri thread di essere eseguiti. Ad un certo punto, uno di quei thread farà `yield` e permetterà al nostro thread e al suo `kerneltrap` di riprendere di nuovo
4. Quando il lavoro di `kerneltrap` è completo, deve tornare al codice interrotto dalla trap, ripristina i registri di controllo `sepc` e `sstatus` e ritorna a `kernelvec` (`kernel/kernelvec.S:48`)
5. `kernelvec` estrae i registri salvati dallo stack ed esegue `sret`, che copia `sepc` in `pc` e riprende il codice del kernel interrotto.

xv6 imposta `stvec` di una CPU su `kernelvec` quando quella CPU entra nel kernel dallo spazio utente; è possibile vedere ciò in `usertrap` (`kernel/trap.c:29`). C'è una finestra di tempo in cui il kernel sta eseguendo ma `stvec` è impostato su `uservec`, ed è cruciale che le interruzioni del dispositivo siano disabilitate durante quella finestra. Fortunatamente, il RISC-V disabilita sempre le interruzioni quando inizia a prendere una trappola, e xv6 non le abilita nuovamente fino dopo aver impostato `stvec`.

### 2.3.9 System Calls presenti in xv6

- `fork` - `int fork()` - viene utilizzata per creare un nuovo processo figlio identico al processo chiamante (keeping same executable). Essa consente ad un processo user di generare un altro processo user che svolga lo stesso eseguibile. Il valore di ritorno al processo chiamante è il nuovo process ID (PID), mentre al nuovo processo (figlio) ritorna 0.
- `exit` - `int exit(int status)` - viene utilizzata per terminare il processo chiamante. Essa restituisce uno stato di uscita (exitcode) al processo padre. Il sistema operativo rilascia tutte le risorse associate al processo in uscita.
- `wait` - `int wait(int *status)` - sospende il processo chiamante finché uno qualsiasi dei suoi processi figli termina. Restituisce l'ID del processo figlio terminato e imposta lo stato di uscita nel puntatore 'status'.



- `kill` - `int kill(int pid)` - terminare il processo, ritorna 0 o -1 per errori
- `getpid` - `int getpid()` - ritorna l'id del processo corrente (chiamante)
- `sleep` - `int wait(int *status)` - pausa per n clock ticks
- `exec` - `int exec(char *file, char argv[])` - caricare un file e lo esegue con gli argomenti, vi è un ritorno solo in caso di errore
- `sbrk` - `char *sbrk(int n)` - incrementa (o diminuisce) la dimensione del segmento dei dati del processo chiamante. È spesso utilizzata per richiedere o rilasciare memoria dinamicamente.
- `open` - `int open(char *file, int flags)` - apre un file specificato dal percorso 'filename' con le opzioni specificate dai 'flags'. La modalità ('mode') specifica i permessi quando il file viene creato. I flag indicano lettura o scrittura, ritorna un fd(file descriptor)
- `write` - `int write(int fd, char *buf, int n)` - scrive buflen byte nel file specificato da fd, nella posizione nel file specificata dalla posizione di ricerca corrente del file, prendendo i dati dallo spazio puntato da buf
- `read` - `int read(int fd, char *buf, int n)` - legge buflen byte dal file specificato da fd, nella posizione nel file specificata dalla posizione di ricerca corrente del file, e li memorizza nello spazio puntato da buf.
- `close` - `int close(int fd)` - chiude il file identificato da 'filehandle', rilasciando tutte le risorse associate
- `dup` - `int dup(int fd)` - ritornare un nuovo file descriptor
- `pipe` - `int pipe(int p[])` - la chiamata pipe crea un oggetto pipe anonimo nel sistema e lo associa a due handle di file nel processo corrente, uno per la fine di lettura e uno per la fine di scrittura.
- `chdir` - `int chdir(char *dir)` - viene utilizzata per impostare la directory corrente del processo corrente sulla directory denominata dal pathname.
- `mkdir` - `int mkdir(char *dir)` - creare una nuova directory
- `mknod` - `int mknod(char *file, int, int)` - creare un device file
- `fstat` - `int fstat(int fd, struct stat *st)` - recupera le informazioni sullo stato del file a cui fa riferimento l'handle del file fd e le memorizza nella struttura stat puntata da statbuf.
- `stat` - `int stat(int fd, struct stat *st)` - recupera le informazioni sullo stato del file a cui fa riferimento il percorso e le memorizza nella struttura stat a cui fa riferimento statbuf
- `link` - `int link(char *file1, char *file2)` - crea un nuovo nome per il file a cui fa riferimento il vecchio nome
- `unlink` - `int unlink(char *file)` - rimuovere un file

A meno di variazioni tutte le system calls citate ritornano -1 nel caso ci sia un errore altrimenti ritornano 0.

## 3 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.

### 3.1 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

#### 3.1.1 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 dimensione 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 offset si ottiene l'indirizzo della memoria fisica.

#### 3.1.2 Memory Management Unit e Translation Lookaside Buffer

Il lavoro descritto nelle sezioni precedenti è svolto da un particolare dispositivo hardware chiamato Memory Management 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.

## 3.2 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 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

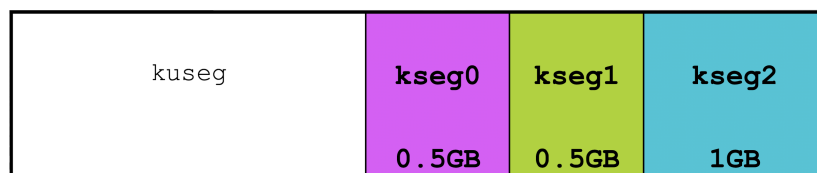


Figure 7: Rappresentazione della memoria di OS161

### 3.2.1 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

Logical addr. (KSEG0)		Physical addr.
0x80000000		0x0
	exception handlers	
0x80000200		0x200
	kernel	
0x80039d54 ( <i>_end</i> )	arg string for boot + Page align	0x39d54
0x8003a000 ( <i>P</i> )	Stack for first thread (1 page = 4096 B)	0x3a000
0x8003b000 <b>(firstfree)</b>	FREE MEMORY	0x3b000 <b>(firstpaddr)</b>
0x80100000	ramsize (es. 1MB: sys161.conf)	0x100000

Figure 8: Configurazione iniziale del kernel

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

Si può notare che finito il bootstrap, 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 8: Parte del codice di avviamento del kernel

### 3.2.2 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 9: `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 10: `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`

### 3.2.3 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 11: 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 della 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 presenti 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 12: 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 13: 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 ritorna

### 3.3 MMU in OS161

#### 3.3.1 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 nel TLB all'interno del 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 14: 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 15: In err.h si trovano tutti i principali errori del sistema operativo

#### 3.3.2 MIPS R3000 TLB

Il sistema operativo OS161 è basato su un'architettura MIPS R3000 e, di conseguenza, ha come TLB quello presente in tale architettura. Essa ha una 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 bits per il numero di pagina e 12 per l'offset. Il 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, ovvero 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 potrebbe 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 il 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

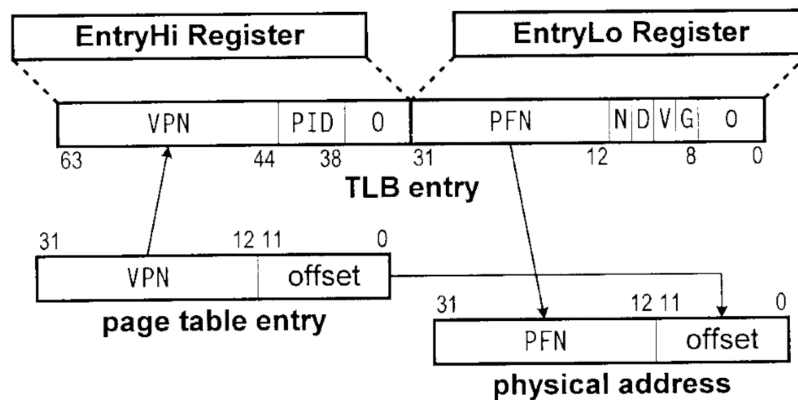


Figure 9: Traduzione degli indirizzi MIPS R3000

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

Quando si vuole effettuare una traduzione di indirizzi il numero di pagina virtuale è comparato a tutte le entries del TLB simultaneamente. Se viene trovato un match e il bit G non è settato, il PID della entry è confrontato con il TBLpid corrente 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 ulteriore 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 nei loro TLB) così da scatenare un'eccezione forzata TLBmod prima di scrivervi.

Il bisogno di tracciare le modifiche della pagine e dei riferimenti portano ad un gran numero di page faults visto che ogni TLBmiss deve essere gestita dal software (in questo caso OS161). Tuttavia, questo è bilanciato da diversi aspetti tra i quali, il più importante, 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 non essendo necessario 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.

### 3.3.3 OS161 e TLB

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

- `tlb_write()`: modifica una specifica entry del TLB
- `tlb_random()`: modifica una entry casuale del TLB
- `tlb_read()`: legge una specifica entry del TLB
- `tlb_probe()`: cerca uno specifico numero di pagina nel TLB
- `tlb_reset()`: inizializza il 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 16: 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 nel TLB. Nello specifico:

1. vengono scritti nella entry del 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 del 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 nel TLB

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

### 3.4 Memoria Virtuale in xv6

#### 3.4.1 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 bits a disposizione solamente i 39 bits 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 bits nei quali viene scritto il numero di pagina, mentre nei restanti 10 dei flags.

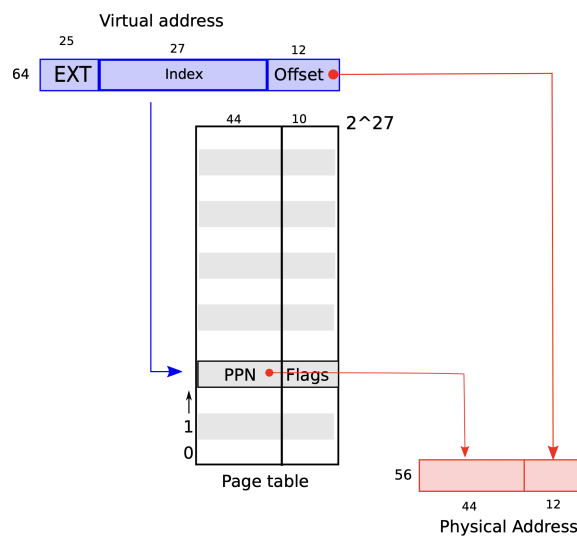


Figure 10: Rappresentazione degli indirizzi virtuali e fisici di Sv39 RISC-V

Come possiamo vedere dalla figura, dei 64 bits 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 bits 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.

#### 3.4.2 Traduzione degli indirizzi in Sv39 RISC-V

Una CPU RISC-V traduce gli indirizzi virtuali in tre fasi. In memoria fisica viene 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

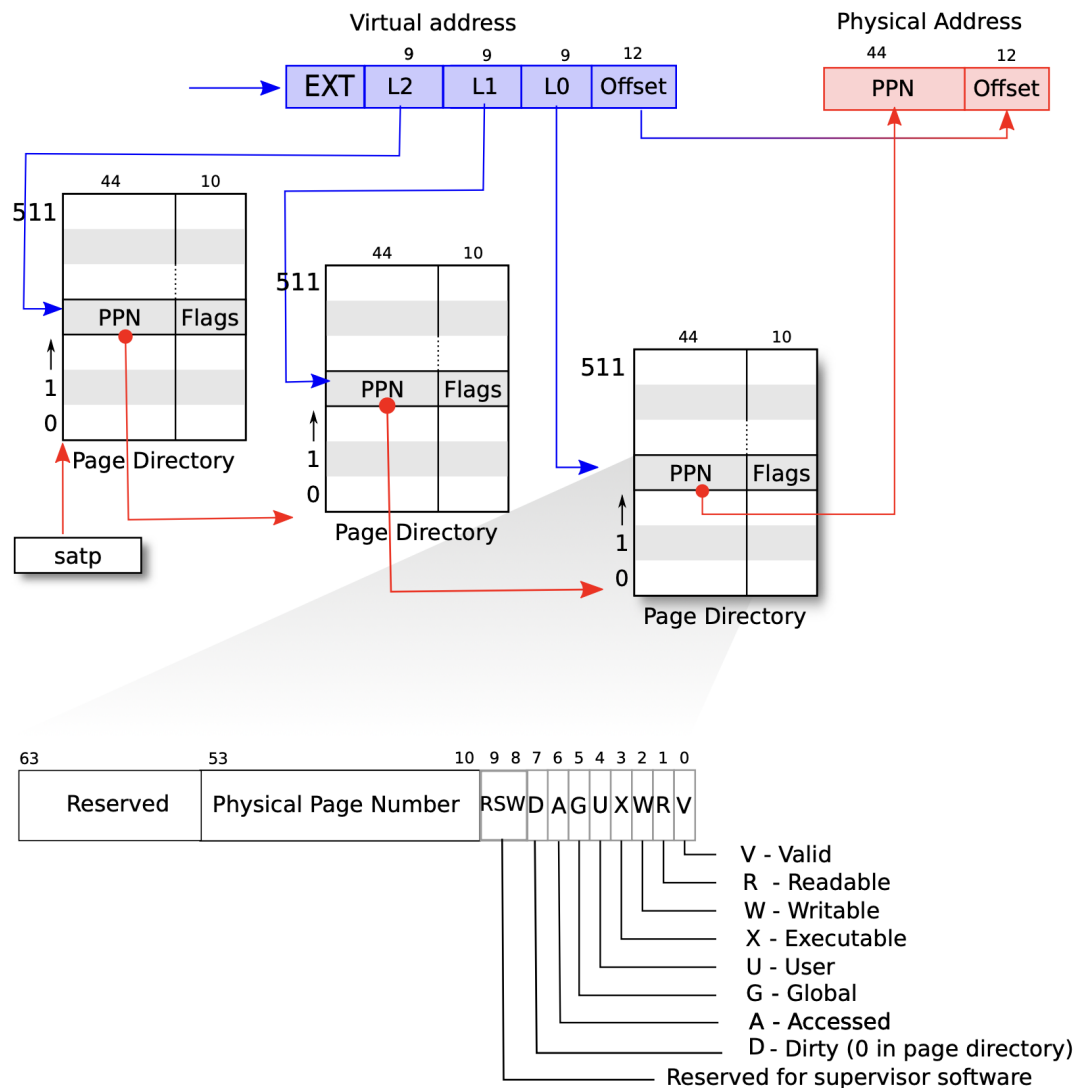


Figure 11: Dettagli della traduzione di Sv39 RISC-V

- G: indica un global mapping ovvero un mapping che esiste in tutti gli spazi di indirizzamento (previene che il 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

### 3.4.3 Allocazione

L'allocatore di memoria risiede in `kernel/kalloc.c` il quale è una lista di pagine di memoria fisica libere che sono disponibili per l'allocazione. Ogni elemento di questa lista è una struttura `run` che contiene un puntatore ad una struttura `run` che rappresenta il prossimo elemento della lista.

```
struct run {
    struct run *next;
};
```

```

struct {
    struct spinlock lock;
    struct run *freelist;
} kmem;

```

Code 17: La struttura delle pagine fisiche libere da allocare

La struttura kmem rappresenta la memoria kernel da allocare (come è possibile notare è protetta da uno spinlock). La funzione main chiama kinit() per inizializzare l'allocatore che deve allocare pagine di 4096 bytes assumendo che xv6 abbia 128 megabytes di memoria RAM.

```

void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE); // fill with junk
    return (void*)r;
}

```

## 3.5 MMU in xv6

### 3.5.1 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 18: 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  $PPN \ll 12$

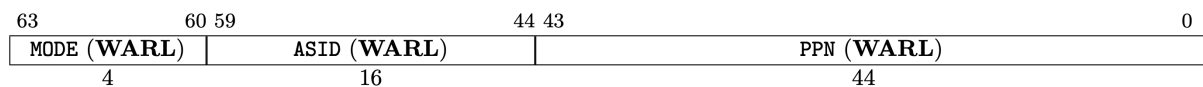


Figure 12: La rappresentazione del `satp` per la configurazione Sv39, con la suddivisione dei campi WARL (*write any value, read legal value*)

### 3.5.2 Traduzione degli indirizzi

Gli indirizzi virtuali di Sv39 sono mappati su 39 bit mentre l'indirizzo fisico è mappato a 56 bit, questo significa che un indirizzo virtuale può essere associato a più indirizzi fisici.

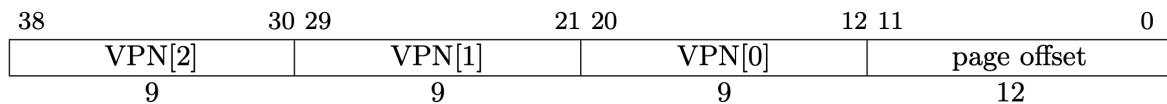


Figure 13: Un virtual address di Sv39, il virtual page number è diviso in 3 parti ognuna è riferita ad un livello della page table

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. calcola la entry prendendo il PPN (preso dal `satp` o dalla entry corrente della page table) e moltiplica per la page size ( $2^{12}$ ). Successivamente aggiunge al `vpn` del livello corrente ( $i$ ) della page table moltiplicato per la size della entry della page table (nel caso di Sv39 è 8 byte)

$$PPN * 4096 + vpn[i] * 8$$

3. legge la entry calcolata, se il bit V (valid) è uguale a 0 allora viene scatenato un page fault
4. se  $V = 1$  e se i bit R, W o X sono tutti 0 la entry è un branch altrimenti è già l'indirizzo ricercato
5. `PPN[2] | PPN[1] | PPN[0]` rappresentano l'indirizzo fisico della memoria dove si trova la prossima page table
6. si ripete il procedimento dal punto (2) finché non si trova l'indirizzo
7. quando si trova `PPN[2]`, `PPN[1]` e `PPN[0]` dicono alla MMU quale sia l'effettivo indirizzo fisico

### 3.5.3 xv6 e TLB

Anche la CPU RISC-V effettua la cache delle page table entries in un TLB (Translation Look-aside Buffer) e quando il sistema operativo cambia una page table deve comunicare alla CPU di invalidare le corrispondenti entries del TLB (altrimenti la entry punterebbe ad una zona della memoria utilizzata da un altro processo). Per fare ciò RISC-V utilizza un'istruzione chiamata `sfence.vma` che permette di pulire il TLB della CPU corrente. Essa viene chiamata all'interno del file `vm.c` dalla funzione `kvmnithart()` la quale prima ricarica il registro `satp` e poi effettua la pulizia del TLB, ma anche nel `trampoline.s` un file assembly (un codice a basso livello che gestisce le traps da livello utente a livello kernel) e ritorna da kernel ad utente).

```
static inline void
sfence_vma()
{
    asm volatile("sfence.vma zero, zero");
}
```

---

Code 19: La funzione `sfence.vma` all'interno del file header `riscv.h`. La dicitura "zero, zero" indica che bisogna eliminare tutte le entries del TLB

Nonostante RISC-V supporti l'eliminazione di TLB entry solamente per uno specifico spazio di indirizzamento (tramite ASIDs) xv6 non sfrutta questa funzione.

## 4 Algoritmi di Scheduling

### 4.1 Introduzione

#### 4.1.1 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 quello di ottimizzare l'utilizzo della CPU trovando un bilanciamento tra:

- massimizzare 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 un processo entro un tempo predefinito
- massimizzare la percentuale di utilizzo del processore

#### 4.1.2 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 bloccare 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

#### 4.1.3 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 trattati 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.

## 4.2 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.

### 4.2.1 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.

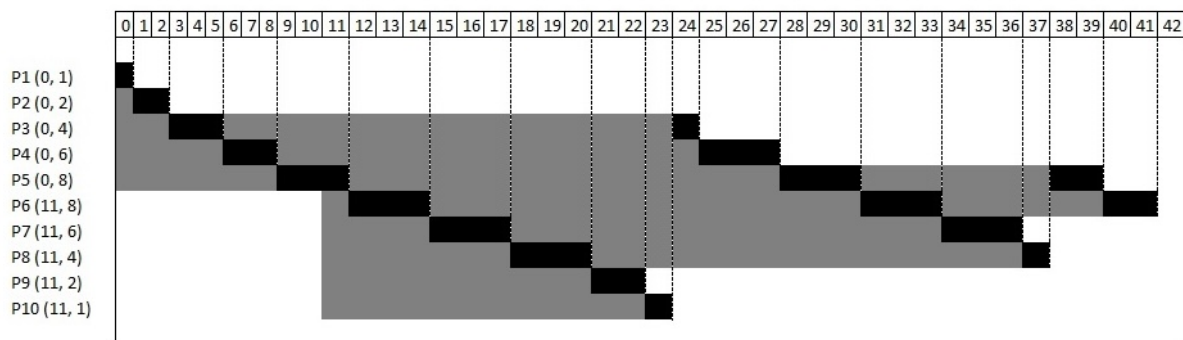


Figure 14: Rappresentazione temporale di uno scheduling round robin. In questo esempio i processi sono descritti da una coppia di valori che rappresentano il tempo di arrivo e la durata. Il time quantum è di 3 unità temporali e l'area grigia rappresenta il tempo di attesa di un processo prima di essere 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.

### 4.2.2 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 20: 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 gestito 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 21: Quanti hardclocks 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_HARDCLOCKS` 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 22: `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 23: 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 24: Struttura dati della coda dei thread in esecuzione

### 4.3 Algoritmi di Scheduling in xv6

Lo scheduler della versione base di xv6 offre anch'esso un algoritmo di tipo Round Robin con politica preemptive, come in OS161 (per dettagli vedere sezione 4.2.1).

#### 4.3.1 Scheduler in xv6

In xv6 lo scheduler è uno speciale thread per ogni CPU, ognuno dei quali esegue la funzione `scheduler`. Questa funzione ha il compito di decidere quale deve essere il prossimo processo ad essere eseguito.

```

void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;
    for(;;) {

        intr_on();

        for(p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if(p->state == RUNNABLE) {
                p->state = RUNNING;

```

```

        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
    }
    release(&p->lock);
}
}
}

```

Code 25: La funzione scheduler in kern/proc.c

Come possiamo notare dal codice, la funzione `scheduler` è un loop infinito che itera tra tutti i processi salvati nella struttura dati `proc`, che non ritorna mai nessun valore ed esegue i seguenti step:

1. sceglie un processo da eseguire tra quelli etichettati `RUNNABLE` e lo etichetta come `RUNNING`
2. chiama la funzione `swtch`
3. riprende il controllo (viene ceduto) dal processo in esecuzione

Inoltre, è importante notare che la caratteristica di `preemption` è garantita dalla funzione `intr_on()` la quale abilita gli interrupt così che lo scheduler possa essere interrotto. La funzione `swtch` è una funzione assembly che semplicemente esegue il context switching salvando i valori dei registri correnti e caricando nei registri i valori del nuovo processo da eseguire.

```

struct cpu {
    struct proc *proc;
    struct context context;
    int noff;
    int intena;
};

```

Code 26: La struttura dati che rappresenta una CPU

Ogni CPU ha la propria struttura dati che contiene, tra gli altri valori, la struttura `context`. Lo scheduler, una volta che esegue un context switching, carica i valori del nuovo processo in questa struttura, così che la CPU, sa che deve eseguire il processo selezionato come prossimo.

Come detto in precedenza, lato processore c'è solamente un thread che continua ad essere seguito all'infinito selezionando i processi da eseguire, ma questi vengono etichettati come `RUNNABLE`, lato processi utente, tramite una catena di chiamate di funzioni.

In breve:

1. un processo esegue una trap
2. viene chiamata la funzione `yield`
3. viene chiamata la funzione `sched` che esegue il context switching caricando come il processo che ha scatenato la trap come corrente etichettandolo come `RUNNABLE`
4. la funzione `scheduler` rileva il nuovo processo e lo esegue etichettandolo come `RUNNING` ed esegue il context switching così che la CPU possa eseguirlo
5. il processo passa in esecuzione ed il controllo passa ad esso

```

void
yield(void)
{
    struct proc *p = myproc();
    acquire(&p->lock);
    p->state = RUNNABLE;
    sched();
}

```

```

    release(&p->lock);
}

void
sched(void)
{
    int intena;
    struct proc *p = myproc();

    /. . ./

    intena = mycpu()->intena;
    swtch(&p->context, &mycpu()->context);
    mycpu()->intena = intena;
}

```

Code 27: La catena di chiamata di funzioni per effettuare lo scheduling

Come possiamo vedere, la funzione `yield` acquisisce il lock, etichetta il processo come `RUNNABLE` e poi chiama la funzione `sched`. Questa, dopo una serie di controlli sul lock e sullo stato del processo (verifica che non sia `RUNNING`) e salva il context del processo corrente all'interno della struttura dati della CPU. La funzione `yield` viene chiamata sia lato utente (`usertrap`) sia lato kernel (`kerneltrap`).

Infine, la funzione `scheduler` viene chiamata ogni decimo di secondo in `xv6` grazie ad una funzione che si collega al clock hardware della CPU RISC-V.

```

void
timerinit()
{
    int id = r_mhartid();

    int interval = 1000000; // cycles; about 1/10th second in qemu.
    *(uint64*)CLINT_MTIMECMP(id) = *(uint64*)CLINT_MTIME + interval;

    uint64 *scratch = &timer_scratch[id][0];
    scratch[3] = CLINT_MTIMECMP(id);
    scratch[4] = interval;
    w_mscratch((uint64)scratch);

    w_mtvec((uint64)timervec);

    w_mstatus(r_mstatus() | MSTATUS_MIE);

    w_mie(r_mie() | MIE_MTIE);
}

```

Code 28: La funzione `timerinit` presente in `start.c`

In questa funzione vediamo come ogni processore ha la propria sorgente di timer interrupt, diversificata dalla funzione `r_mhartid()` che semplicemente ritorna il numero del CORE corrente. Ogni CORE, successivamente, chiede un interrupt locale tramite `CLINT_MTIMECMP`, poi vengono salvate tutte le informazioni inerenti al timer richiesto nella struttura dati chiamata `scratch`, nella quale viene inizializzato `interval = 1000000` come intervallo in numero di cicli tra ogni chiamata dell'interrupt. Questo valore corrisponde a un decimo di secondo nell'ambiente di virtualizzazione QEMU, nella quale viene eseguito il sistema operativo.

## 5 Meccanismi di Sincronizzazione

### 5.1 Introduzione

Con la programmazione concorrente sorge il problema di accedere a risorse condivise o a sezioni critiche in modo consistente e senza perdita di dati. I meccanismi di sincronizzazione servono proprio a ciò, per evitare problemi quali:

- Race conditions: si presenta quando il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti
- Deadlocks: indica una situazione in cui due o più processi si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa
- Resource starvation: si intende l'impossibilità perpetua, da parte di un processo, di ottenere le risorse di cui necessita per essere eseguito

### 5.2 Spinlock in os161

#### 5.2.1 Introduzione

OS161 usa gli spinlock come meccanismo di sincronizzazione. I lock permettono di eseguire la sezione critica impedendo al thread di essere interrotto, in questo modo si evita che altri thread vadano in esecuzione prima che il thread già in esecuzione abbia finito di eseguire la sezione critica, impedendo, così, la lettura o scrittura di dati inconsistenti. Per fare ciò si utilizza un'istruzione hardware atomica chiamata test and set, questa istruzione legge il valore di una variabile booleana, lo imposta a true e ritorna il valore vecchio. Questa istruzione è utile per l'acquisizione del lock: se il lock è già stato acquisito i thread che lo richiedono vengono messi in attesa, la test and set trova il valore della variabile impostato a true, lo reimposta a true e ritorna il vecchio valore che era sempre true, quindi i thread restano sempre in attesa finché il lock non viene rilasciato e la variabile viene impostata a false. Quando il lock viene rilasciato il thread che vuole acquisirlo chiama la test and set sulla variabile che adesso sarà false, questa istruzione modifica il valore della variabile a true e ritorna il vecchio valore, cioè false, permettendo così al thread che l'ha chiamata di uscire dall'attesa, acquisire il lock e accedere alla sezione critica.

Di seguito è riportato il codice dell'implementazione degli spinlock in OS161. La funzione testandset di questo sistema non usa una variabile booleana, ma imposta la variabile a 1 e ritorna il vecchio valore. La struttura spinlock è la seguente:

```
struct spinlock {
    // Contiene 0 se il lock libero, 1 se acquisito
    volatile spinlock_data_t splk_lock;
    // CPU che possiede il lock
    struct cpu *splk_holder;
    // Deadlock detector
    HANGMAN_LOCKABLE(splk_hangman);
};
```

Code 29: La struttura spinlock presente in kern/include/spinlock.h

#### 5.2.2 Acquisizione dello spinlock

```
void spinlock_acquire(struct spinlock *splk)
{
    struct cpu *mycpu;
    // Vengono disabilitati gli interrupt per evitare deadlock
    splraise(IPL_NONE, IPL_HIGH);

    /* this must work before curcpu initialization */
    if (CURCPU_EXISTS()) {
```

```

mycpu = curcpu->c_self;
if (splk->splk_holder == mycpu) {
    panic("Deadlock on spinlock %p\n", splk);
}
mycpu->c_spinlocks++;

HANGMAN_WAIT(&curcpu->c_hangman, &splk->splk_hangman);
}
else {
    mycpu = NULL;
}

while (1) {
    // Ottiene il valore del lock, se diverso da 0 resta in attesa
    if (spinlock_data_get(&splk->splk_lock) != 0) {
        continue;
    }
    // Chiama la test and set, se diverso da 0 il lock gia' acquisito quindi
    // resta in attesa
    // altrimenti lo acquisisce
    if (spinlock_data_testandset(&splk->splk_lock) != 0) {
        continue;
    }
    break;
}

membar_store_any();
// Imposta la cpu corrente come holder del lock
splk->splk_holder = mycpu;

if (CURCPU_EXISTS()) {
    HANGMAN_ACQUIRE(&curcpu->c_hangman, &splk->splk_hangman);
}
}

```

Code 30: La funzione `spinlock_acquire` in `kern/thread/spinlock.c`

Come si evince dal codice OS161 usa una versione più performante degli spinlock, la `test+test and set`. L'istruzione `test and set` essendo atomica è molto onerosa in termini di performance, quindi è preferibile chiamarla il meno volte possibile. Per fare ciò, quando i thread sono in `busy-wait` anziché chiamare la `test and set` per verificare se il lock è stato rilasciato, si usa una `get` per ottenere il valore del lock, se è ancora acquisito si continua l'attesa, altrimenti viene chiamata la `test and set` e si acquisisce il lock. In questo modo si riduce il numero di chiamate di questa istruzione perché viene eseguita soltanto quando il lock è stato rilasciato.

### 5.2.3 Rilascio dello spinlock

```

void spinlock_release(struct spinlock *splk)
{
    /* this must work before curcpu initialization */
    if (CURCPU_EXISTS()) {
        KASSERT(splk->splk_holder == curcpu->c_self);
        KASSERT(curcpu->c_spinlocks > 0);
        curcpu->c_spinlocks--;
        HANGMAN_RELEASE(&curcpu->c_hangman, &splk->splk_hangman);
    }
}

```

```

// Imposta l'holder del lock a NULL
splk->splk_holder = NULL;
membar_any_store();
// Setta il valore del lock a 0, quindi libero
spinlock_data_set(&splk->splk_lock, 0);
spllower(IPL_HIGH, IPL_NONE);
}

```

Code 31: La funzione `spinlock_release` in `kern/thread/spinlock.c`

## 5.2.4 Altre funzioni inerenti allo spinlock

Inoltre sono presenti le seguenti funzioni:

```

void spinlock_init(struct spinlock *splk)
{
    spinlock_data_set(&splk->splk_lock, 0);
    splk->splk_holder = NULL;
    HANGMAN_LOCKABLEINIT(&splk->splk_hangman, "spinlock");
}

```

Code 32: La funzione `spinlock_init` in `kern/thread/spinlock.c` ha il compito di inizializzare lo spinlock

```

void spinlock_cleanup(struct spinlock *splk)
{
    KASSERT(splk->splk_holder == NULL);
    KASSERT(spinlock_data_get(&splk->splk_lock) == 0);
}

```

Code 33: La funzione `spinlock_cleanup` in `kern/thread/spinlock.c` ha il compito di ripulire lo spinlock

```

bool spinlock_do_i_hold(struct spinlock *splk)
{
    if (!CURCPU_EXISTS()) {
        return true;
    }

    /* Assume we can read splk_holder atomically enough for this to work */
    return (splk->splk_holder == curcpu->c_self);
}

```

Code 34: La funzione `spinlock_do_i_hold` in `kern/thread/spinlock.c` ritorna `true` se la cpu corrente ha acquisito lo spinlock, `false` altrimenti

## 5.3 Wait Channels in os161

### 5.3.1 Introduzione

I wait channels permettono di mettere in attesa e risvegliare i thread. Essi contengono una lista di thread in attesa di entrare in una determinata sezione critica e non appena questa è libera, si può scegliere di risvegliarne uno, in modo casuale, o tutti.

La struct di un wait channel in os161 è la seguente:

```

struct wchan {
    // Nome del wait channel
    const char *wc_name;
}

```



```

    // Lista dei thread in attesa
    struct threadlist wc_threads;
};

```

Code 35: La struttura del wait channel

### 5.3.2 Attesa di un thread

Un thread viene messo in attesa tramite la funzione:

```

void wchan_sleep(struct wchan *wc, struct spinlock *lk)
{
    // Verifica che il thread non sia in un interrupt handler
    KASSERT(!curthread->t_in_interrupt);
    // Verifica che posseda lo spinlock
    KASSERT(spinlock_do_i_hold(lk));
    // Verifica che non si posseggano altri spinlock
    KASSERT(curcpu->c_spinlocks == 1);
    // Il thread viene messo in stato SLEEP e inserito nella lista d'attesa
    thread_switch(S_SLEEP, wc, lk);
    // Viene acquisito il lock
    spinlock_acquire(lk);
}

```

Code 36: La funzione wchan\_sleep che mette in attesa un thread

### 5.3.3 Risveglio di un thread

Un thread viene risvegliato tramite le funzioni:

```

void wchan_wakeone(struct wchan *wc, struct spinlock *lk)
{
    struct thread *target;
    // Verifica che si posseda il lock
    KASSERT(spinlock_do_i_hold(lk));

    // Viene preso il primo thread dal wchan
    target = threadlist_remhead(&wc->wc_threads);
    // Se nullo nessun thread e' in attesa, quindi la funzione termina
    if (target == NULL) {
        return;
    }
    // Il thread recuperato dalla lista viene messo in stato RUNNABLE
    thread_make_runnable(target, false);
}

```

Code 37: La funzione wchan\_wakeone che risveglia un thread in attesa sul wait channel

```

void wchan_wakeall(struct wchan *wc, struct spinlock *lk)
{
    struct thread *target;
    struct threadlist list;
    // Verifica che si posseda il lock
    KASSERT(spinlock_do_i_hold(lk));
    // Viene creata una lista
    threadlist_init(&list);
    // Vengono presi tutti i thread in attesa sul wchan e spostati nella lista appena creata
}

```

```

while ((target = threadlist_remhead(&wc->wc_threads)) != NULL) {
    threadlist_addtail(&list, target);
}
// Tutti i thread nella lista vengono messi in stato RUNNABLE
while ((target = threadlist_remhead(&list)) != NULL) {
    thread_make_runnable(target, false);
}
// Viene ripulita la lista
threadlist_cleanup(&list);
}

```

Code 38: La funzione `wchan_wakeall` che risveglia tutti i thread in attesa nel wait channel

### 5.3.4 Altre funzioni inerenti ai wait channel

Sono presenti inoltre le seguenti funzioni:

```

struct wchan * wchan_create(const char *name)
{
    struct wchan *wc;
    // Alloca dinamicamente la memoria necessaria al wchan
    wc = kmalloc(sizeof(*wc));
    if (wc == NULL) {
        return NULL;
    }
    // Crea una lista per contenere i thread in attesa
    threadlist_init(&wc->wc_threads);
    wc->wc_name = name;

    return wc;
}

```

Code 39: La funzione `wchan_create` che crea un wait channel

```

void wchan_destroy(struct wchan *wc)
{
    threadlist_cleanup(&wc->wc_threads);
    kfree(wc);
}

```

Code 40: La funzione `wchan_destroy` che distrugge un wait channel

```

bool wchan_isempty(struct wchan *wc, struct spinlock *lk)
{
    bool ret;
    KASSERT(spinlock_do_i_hold(lk));
    ret = threadlist_isempty(&wc->wc_threads);

    return ret;
}

```

Code 41: La funzione `wchan_isempty` che ritorna `true` se la lista di thread in attesa è vuota, `false` altrimenti

## 5.4 Semafori in os161

### 5.4.1 Introduzione

Un altro meccanismo di sincronizzazione implementato in os161 è il semaforo. Un semaforo contiene una variabile intera usata come contatore e si può accedere ad esso solamente tramite due funzioni atomiche: `wait()` and `signal()`, anche chiamate, rispettivamente, `P()` e `V()`. La funzione `P()` verifica se il contatore del semaforo è minore o uguale a zero, se lo è mette il thread in busy wait. Non appena il contatore sarà maggiore di zero il thread viene risvegliato, avendo, così, la possibilità di accedere alla sezione critica e il contatore viene decrementato, portandolo nuovamente a zero per impedire ad altri thread di poter accedere. La funzione `V()` invece, viene chiamata all'uscita della sezione critica. La funzione semplicemente incrementa il contatore, permettendo a un altro thread di essere risvegliato.

In os161 i semafori sono usati in combinazione con i wait channels e gli spinlock precedentemente descritti. I semafori utilizzano la seguente struttura dati:

```
struct semaphore {
    char *sem_name;      // Nome semaforo
    struct wchan *sem_wchan; // Wait channel
    struct spinlock sem_lock; // Spinlock
    volatile unsigned sem_count; // Counter
};
```

Code 42: La struttura del semaforo

Come si nota, oltre al nome e alla variabile contatore, essa contiene anche il riferimento al wait channel e uno spinlock.

### 5.4.2 Le funzioni `P()` e `V()`

Le funzioni principali del semaforo sono:

```
void P(struct semaphore *sem)
{
    // Verifica che il semaforo non sia nullo
    KASSERT(sem != NULL);
    // Verifica che il thread non sia in un interrupt handler
    KASSERT(curthread->t_in_interrupt == false);
    // Viene acquisito lo spinlock contenuto nel semaforo per proteggere il
    // wchan
    spinlock_acquire(&sem->sem_lock);
    // Finché il contatore è 0, i thread vengono messi in attesa e inseriti
    // nella lista del wchan
    while (sem->sem_count == 0) {
        wchan_sleep(sem->sem_wchan, &sem->sem_lock);
    }
    // Se si esce dal while il contatore è maggiore di 0, ma ciò viene
    // ricontrollato
    // per evitare che un altro thread abbia preceduto quello corrente
    KASSERT(sem->sem_count > 0);
    // Viene decrementato il contatore
    sem->sem_count--;
    // Viene rilasciato lo spinlock
    spinlock_release(&sem->sem_lock);
}
```

Code 43: La funzione `P()`

```
void V(struct semaphore *sem)
{
    // ...
```

```

// Verifica che il semaforo non sia nullo
KASSERT(sem != NULL);
// Acquisisce lo spinlock per proteggere le modifiche al semaforo
spinlock_acquire(&sem->sem_lock);
// Incrementa il contatore e verifica che sia maggiore di 0
sem->sem_count++;
KASSERT(sem->sem_count > 0);
// Risveglia uno dei thread in attesa nel wchan
wchan_wakeone(sem->sem_wchan, &sem->sem_lock);
// Rilascia il lock
spinlock_release(&sem->sem_lock);
}

```

Code 44: La funzione V()

### 5.4.3 Altre funzioni inerenti ai semafori

Inoltre sono present le seguenti funzioni:

```

struct semaphore * sem_create(const char *name, unsigned initial_count)
{
    struct semaphore *sem;

    sem = kmalloc(sizeof(*sem));
    if (sem == NULL) {
        return NULL;
    }

    sem->sem_name = kstrdup(name);
    if (sem->sem_name == NULL) {
        kfree(sem);
        return NULL;
    }

    sem->sem_wchan = wchan_create(sem->sem_name);
    if (sem->sem_wchan == NULL) {
        kfree(sem->sem_name);
        kfree(sem);
        return NULL;
    }

    spinlock_init(&sem->sem_lock);
    sem->sem_count = initial_count;

    return sem;
}

```

Code 45: La funzione sem\_create che crea un semaforo

```

void sem_destroy(struct semaphore *sem)
{
    KASSERT(sem != NULL);

    /* wchan_cleanup will assert if anyone's waiting on it */
    spinlock_cleanup(&sem->sem_lock);
    wchan_destroy(sem->sem_wchan);
    kfree(sem->sem_name);
}

```

```
kfree(sem);
}
```

Code 46: La funzione `sem_destroy` che distrugge un semaforo

## 5.5 Spinlock in xv6

Gli spinlock usano la seguente struttura dati:

```
struct spinlock {
    uint locked;           // Contiene 0 se il lock    libero, 1 se    acquisito
    char *name;           // Nome del lock
    struct cpu *cpu;       // La cpu che possiede il  lock
};
```

Code 47: La struttura `spinlock` in xv6

Come in `os161` anche in `xv6` viene utilizzata una variabile intera binaria per verificare se il lock è libero o acquisito. Non ci sono differenze tra le strutture dei due sistemi.

### 5.5.1 Acquisizione dello spinlock

Lo spinlock viene acquisito tramite la seguente funzione:

```
void acquire(struct spinlock *lk)
{
    // Disabilita le interruzioni per evitare deadlock
    push_off();
    if(holding(lk))
        panic("acquire");

    // Chiama la test and set, se    diverso da 0 il lock    gi    acquisito
    // quindi resta in attesa, altrimenti lo acquisisce
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    __sync_synchronize();

    // Imposta la cpu corrente come holder del lock
    lk->cpu = mycpu();
}
```

Code 48: La funzione `acquire` dello spinlock

Come notiamo dal codice riportato sopra `xv6` non utilizza la versione ottimizzata `test+test and set` degli spinlock, ma continua a richiamare sempre la `test and set`. Questo riduce le performance del sistema, in quanto, essendo un'operazione atomica il thread in esecuzione non può essere interrotto sprecando così cicli di clock senza, sostanzialmente, far nulla.

### 5.5.2 Rilascio dello spinlock

La funzione di rilascio, invece, presenta il seguente codice:

```
void release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");
    // Viene azzerato l holder del lock
}
```

```

lk->cpu = 0;

__sync_synchronize();

// Operazione atomica che imposta il valore di locked a 0, rilasciando,
// cosi' il lock
__sync_lock_release(&lk->locked);

// Riabilita le interruzioni
pop_off();
}

```

Code 49: La funzione release per il rilascio di un spinlock

A differenza della funzione acquire, la funzione release non presenta sostanziali differenze con la versione di os161.

### 5.5.3 Altre funzioni inerenti allo spinlock

Anche qui sono presenti le funzioni:

```

void initlock(struct spinlock *lk, char *name)
{
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}

```

Code 50: La funzione initlock che inizializza lo spinlock

```

void initlock(struct spinlock *lk, char *name)
{
    lk->name = name;
    lk->locked = 0;
    lk->cpu = 0;
}

```

Code 51: La funzione holding restituisce 1 se la cpu corrente possiede il lock e il lock è acquisito, 0 altrimenti

A differenza di os161 in xv6 non è presente una funzione di cleanup.

## 5.6 Wait channels in xv6

A differenza di os161 nei wait channels presenti in xv6 non esiste una lista in cui inserire i processi in attesa. La struttura di un processo contiene un campo wchan. proc

```

struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state;           // Process state
    void *chan;                     // If non-zero, sleeping on chan
    int killed;                     // If non-zero, have been killed
    int xstate;                     // Exit status to be returned to parent's
    wait
    int pid;                         // Process ID
    [ ]
};

```

### 5.6.1 Attesa di un processo

Quando un processo sarà in attesa, in questo campo viene inserito su che channel esso è in attesa, tramite la funzione sleep.

```
void sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = myproc();
    // Viene acquisito il lock del processo
    acquire(&p->lock);
    // E rilasciato quello della funzione che ha chiamato la sleep
    release(lk);

    // Viene inserito il channel su cui e' in attesa il processo
    p->chan = chan;
    // Il processo e' messo in SLEEPING
    p->state = SLEEPING;
    // Viene richiamato lo scheduler per mettere in stato RUNNABLE un altro
    // processo
    sched();

    // Tidy up.
    p->chan = 0;
    // Si rilascia il lock del processo
    release(&p->lock);
    // Si riacquisisce il lock originale
    acquire(lk);
}
```

Code 52: La funzione sleep che serve per mettere in attesa un processo

### 5.6.2 Risveglio di un processo

La funzione di risveglio, di conseguenza, cercherà tra tutti i processi quelli che contengono nel campo chan il channel corretto e risveglierà tutti i processi in attesa su quel channel.

```
void wakeup(void *chan)
{
    struct proc *p;
    // Itera su tutti i processi
    for(p = proc; p < &proc[NPROC]; p++) {
        // Se il processo è uno diverso da quello attuale
        if(p != myproc()){
            // Viene acquisito il lock del processo
            acquire(&p->lock);
            // Verifica che il processo sia in SLEEPING e nel channel corretto
            if(p->state == SLEEPING && p->chan == chan) {
                // Il processo viene messo in stato RUNNABLE
                p->state = RUNNABLE;
            }
            // Viene rilasciato il lock
            release(&p->lock);
        }
    }
}
```

Code 53: La funzione wakeup che serve per risvegliare tutti i processi in attesa su quel thread

Non esiste una funzione per risvegliare un solo processo.

## 5.7 Sleeplocks in xv6

In xv6 è presente una versione differente di lock, gli sleeplock. A volte il sistema ha necessità di possedere il lock per molto tempo, ad esempio se sta scrivendo il contenuto di un file nel disco. Questa è un'operazione che richiede molti cicli di clock, per tanto proteggere il file con uno spinlock sarebbe molto oneroso, in quanto il busy waiting spreca cicli di clock inutilmente impedendo ad altri processi di usare la CPU per del lavoro utile. Gli sleeplock servono per evitare ciò. Utilizzando quest'ultimi, quando il processo è in attesa cede la CPU permettendo così ad altri processi di utilizzarla e di conseguenza di avere performance e efficienza migliori.

Uno sleeplock presenta una struttura di questo tipo:

```
struct sleeplock {
    uint locked;           // Contiene 0 se il lock libero, 1 se acquisito
    struct spinlock lk;    // spinlock che protegge lo sleeplock
    char *name;           // Nome del lock
    int pid;               // Il processo che possiede il lock
};
```

Dalla struct si nota che viene utilizzato uno spinlock per proteggere il campo locked dello sleeplock, questo risulta evidente nella funzione di acquisizione del lock.

### 5.7.1 Acquisizione dello sleeplock

```
void acquiresleep(struct sleeplock *lk)
{
    // Acquisisce lo spinlock
    acquire(&lk->lk);
    // Finche' lo sleeplock e' aquisito si mette il processo in attesa
    while (lk->locked) {
        // La sleep mette in attesa il processo e cede la cpu a un altro
        // processo
        sleep(lk, &lk->lk);
    }
    // Appena lo sleeplock e' libero lo si acquisisce impostando locked a 1
    lk->locked = 1;
    // Viene impostato il processo corrente come possessore del lock
    lk->pid = myproc()->pid;
    // Viene rilasciato lo spinlock
    release(&lk->lk);
}
```

Code 54: La funzione acquiresleep acquisisce lo sleeplock

Il rilascio e l'acquisizione del lock originale, fatta dalla funzione sleep vista in precedenza permette ad altri processi di tentare di acquisire lo sleeplock ed essere inseriti nel wait channel, in questo modo si evita il busy waiting.

### 5.7.2 Rilascio dello sleeplock

La funzione di rilascio, invece, si presenta nel seguente modo:

```
void releasesleep(struct sleeplock *lk)
{
    // Acquisisce lo spinlock
    acquire(&lk->lk);
    // Rilascia lo sleeplock impostando questa variabile a 0
    lk->locked = 0;
    // Viene azzerato il possessore dello sleeplock
}
```



```

lk->pid = 0;
// Viene risvegliato un processo in attesa
wakeup(lk);
// Viene rilasciato lo spinlock
release(&lk->lk);
}

```

Code 55: La funzione releasesleep rilascia lo sleeplock

### 5.7.3 Altre funzioni inerenti allo sleeplock

```

void initsleeplock(struct sleeplock *lk, char *name)
{
    initlock(&lk->lk, "sleep lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
}

```

Code 56: La funzione initsleeplock inizializza il lock

```

int holdingsleep(struct sleeplock *lk)
{
    int r;

    acquire(&lk->lk);
    r = lk->locked && (lk->pid == myproc()->pid);
    release(&lk->lk);
    return r;
}

```

Code 57: La funzione holdingsleep restituisce 1 se il processo corrente possiede il lock e il lock è acquisito, 0 altrimenti

## 6 Implementazione di una System Call in xv6

### 6.1 Introduzione

Le syscalls che abbiamo implementato in xv6 sono due e sono chiamate `getfreepages` e `testkalloc`. La prima ha il compito di visualizzare le pagine libere all'interno di xv6, mentre la seconda serve per testare il corretto funzionamento della prima in quanto alloca una pagina di test all'interno del kernel e stampa quante pagine erano libere prima e dopo il test.

### 6.2 File modificati all'interno di xv6

I files modificati per implementare le syscalls sono stati:

- `syscall.h`: l'header file nel quale sono dichiarati i numeri delle system call
- `syscall.c`: il file nel quale sono gestite le system calls
- `sysproc.c`: il file nel quale sono dichiarate le funzioni da eseguire quando si chiama una syscall
- `kalloc.c`: il file nel quale si gestisce l'allocazione della memoria
- `user.h`: l'header file nella quale sono dichiarati gli header delle funzioni
- `usys.pl`: uno script che genera in automatico il codice in assembly per chiamare una syscall
- `Makefile`: il file che dichiara le dipendenze
- `getfreepages.c` (*\*aggiunto*)
- `testkalloc.c` (*\*aggiunto*)

Di seguito vediamo come abbiamo implementato nel dettaglio queste modifiche.

#### 6.2.1 `syscall.h`

Abbiamo aggiunto due numeri inerenti alle syscall da aggiungere.

```
#define SYS_freepages      22
#define SYS_testkalloc     23
```

Code 58: `kernel/syscall.h`

Essi serviranno per la mappatura tra il numero della syscall e la funzione da chiamare.

#### 6.2.2 `syscall.c`

Per prima cosa abbiamo dichiarato i prototipi della funzioni che saranno chiamate dalla system call:

```
extern uint64 sys_freepages(void);
extern uint64 sys_testkalloc(void);
```

Code 59: La definizione delle funzioni che la syscall chiama `kernel/syscall.c`

E poi successivamente abbiamo aggiunto all'array `*syscall[]` le due entry riferite al numero 22 e 23 (definite precedentemente in `syscall.h`) associandole alle funzioni dichiarate nei prototipi:

```
static uint64 (*syscalls[])(void) = {
/ . . . /
[SYS_freepages]      sys_freepages,
[SYS_testkalloc]     sys_testkalloc,
};
```

Code 60: La mappatura delle fuzioni al numero della syscall in `kernel/syscall.c`

### 6.2.3 sysproc.c

Abbiamo aggiunto le due funzioni `sys_freepages()` e `sys_testkalloc` (che sono inerenti alla mappatura scritta precedentemente).

- La funzione `sys_freepages` ha il compito di ritornare il numero delle pagine libere tramite una funzione che sarà approfondita a breve.

```
uint64
sys_freepages(void)
{
    return kfreepages();
}
```

Code 61: `sys_freepage` aggiunta in `kernel/sysproc.c`

- La funzione `sys_testkalloc` ha il compito di allocare una pagina e si basa sulla funzione `kalloc` che è già presente nel sistema operativo.

```
uint64
sys_testkalloc(void) {
    kalloc();
    return 0;
}
```

Code 62: `sys_testkalloc` aggiunta in `kernel/sysproc.c`

### 6.2.4 kalloc.c

Nel file `kalloc.c` presente nativamente nel sistema operativo abbiamo aggiunto la funzione `kfreepages()` la quale ritorna il numero di pagine libere.

```
int
kfreepages()
{
    int counter = 0;
    struct run *list;
    acquire(&kmem.lock);
    list = kmem.freelist;
    while (list != 0) {
        counter += 1;
        list = list->next;
    }
    release(&kmem.lock);
    return counter;
}
```

Code 63: `kfreepages()` aggiunta in `kernel/kalloc.c`

Questa funzione, basandosi sulle caratteristiche di `xv6`, conta quanti elementi liberi ci sono all'interno della lista `kmem.freelist`. Infatti in `xv6`, esiste una struttura dati composta dal tipo `run` che semplicemente contiene un puntatore al prossimo elemento (anch'esso di tipo `run`) e rappresenta la lista delle pagine libere presenti in RAM. Sfruttando ciò in questa funzione:

1. abbiamo un `counter` per tenere traccia delle pagine che visitiamo
2. acquistiamo un `lock` per garantire che la lista sia visitata in mutua esclusione
3. effettuiamo in loop la visita della lista e incrementiamo `counter` finché non raggiungiamo la terminazione della lista

4. rilasciamo il lock
5. ritorniamo counter

#### 6.2.5 user.h

Dichiariamo i prototipi della funzioni `freepages()` e `test_kalloc` così che possano essere accessibili da altre parti del codice.

```
int freepages(void);  
int testkalloc(void);
```

Code 64: Aggiunta dei prototipi in `user.h`

#### 6.2.6 usys.pl

In questo file script abbiamo aggiunto le entry `freepages` `testkalloc` che serviranno per scrivere del codice assembly in automatico

```
entry("freepages");  
entry("testkalloc");
```

Code 65: Aggiunta delle entry in `usys.pl`

#### 6.2.7 Makefile

Nel file `Makefile` abbiamo aggiunto le due entry dei programmi utenti `getfreepages` e `testkalloc` per far sì che le dipendenze siano rispettate in fase di compilazione

```
$U/_getfreepages\  
$U/_testkalloc\  

```

Code 66: Aggiunta degli user program in `Makefile`

#### 6.2.8 getfreepages.c

Abbiamo aggiunto il file `getfreepages.c` che altro non fa che tornare il numero delle pagine libere chiamando la funzione `freepages()` dichiarata precedentemente

```
#include "kernel/types.h"  
#include "kernel/stat.h"  
#include "user/user.h"  
  
int  
main(int argc, char *argv[])  
{  
    printf("xv6 free pages: %d\n", freepages());  
    exit(0);  
}
```

Code 67: Aggiunta del file `getfreepages.c`

#### 6.2.9 testkalloc.c

Infine abbiamo aggiunto il file `testkalloc.c` che altro non fa che tornare il numero delle pagine libere chiamando la funzione `freepages()` dichiarata precedentemente prima e dopo l'allocazione di una pagina tramite la funzione `testkalloc`

```

#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
    printf("Free pages before allocating:%d\n",freepages());
    testkalloc();
    printf("Allocating a page...\n");
    printf("Free pages after allocating:%d\n",freepages());

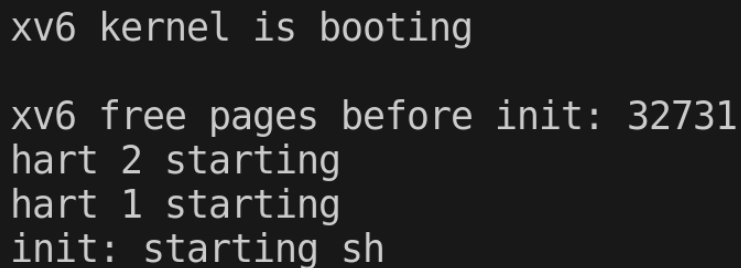
    exit(0);
}

```

Code 68: Aggiunta del file testkalloc.c

### 6.2.10 Immagini del test della syscall

Di seguito degli screenshot che mostrano il funzionamento delle syscalls implementate.



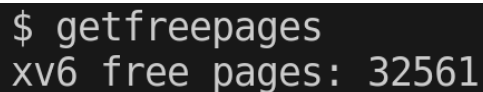
```

xv6 kernel is booting

xv6 free pages before init: 32731
hart 2 starting
hart 1 starting
init: starting sh

```

Figure 15: xv6 all'avvio

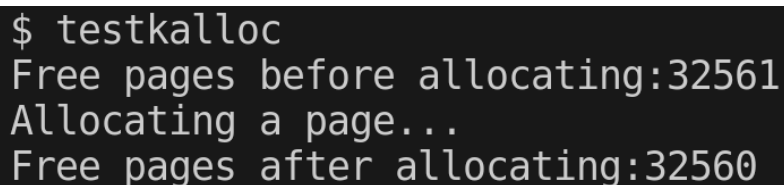


```

$ getfreepages
xv6 free pages: 32561

```

Figure 16: xv6 dopo aver chiesto il numero di pagine libere



```

$ testkalloc
Free pages before allocating:32561
Allocating a page...
Free pages after allocating:32560

```

Figure 17: xv6 dopo aver allocato una pagina di test