UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Sviluppo del supporto per chiamate di sistema e processi utente su architettura RISC-V

Relatore:
Prof. Giuseppe Lettieri

Candidato:
Francesco Barcherini

ANNO ACCADEMICO 2023/2024

Sommario

L'architettura RISC-V, grazie alle elevate prestazioni, al carattere modulare e flessibile dell'ISA e alla sua natura open-source, sta guadagnando sempre più popolarità in una vasta gamma di settori tecnologici. Questo studio si concentra sulla migrazione di parte del nucleo didattico utilizzato nell'esame di Calcolatori Elettronici da x86 a architettura RISC-V. L'obiettivo principale è sviluppare un sistema integrato e modulare in cui poter eseguire un programma fornito dall'utente, con particolare attenzione alla realizzazione dei processi e al supporto per le chiamate di sistema.

Il lavoro si articola in diverse fasi, che comprendono la compilazione e il caricamento del programma utente in memoria, la modifica delle procedure di avvio, la gestione della memoria virtuale e delle interruzioni, lo sviluppo del supporto per i processi e il framework delle primitive di sistema. Particolare attenzione è stata dedicata alla pulizia e all'omogeneizzazione del codice e dei sorgenti.

L'integrazione delle diverse sezioni avviene all'inizializzazione del nucleo e procede all'attivazione dei processi utente. Il risultato finale è un Proof of Concept che permette di testare le funzionalità del sistema attraverso un programma di esempio.

Indice

1 Introduzione **2**
1.1 Caratteristiche di RISC-V 2
1.2 Obiettivi 2

2 Prerequisiti e toolchain di sviluppo **4**

3 Organizzazione delle cartelle del sistema **6**

4 Progettazione e implementazione del sistema **7**
4.1 Compilazione e caricamento 8
4.2 Procedure di bootstrap 10
4.2.1 File entry.s 10
4.2.2 File start.s 11
4.3 Memoria virtuale e dinamica 12
4.3.1 Layout della memoria fisica 12
4.3.2 Memoria virtuale e paginazione 13
Implementazione della memoria virtuale nel nucleo 14
4.3.3 Memora dinamica (heap) 17
4.4 Gestione dei processi 18
4.4.1 Descrittore di processo 18
4.4.2 Salvataggio e caricamento dello stato del processo 19
4.4.3 Creazione dei processi e funzioni di supporto 23
4.5 Inizializzazione del sistema 25
4.5.1 Memoria virtuale e caricamento del modulo utente 25
4.5.2 Creazione dei processi sistema e utente 26
4.6 Supporto per le chiamate di sistema 27
4.6.1 Primitive di sistema e interruzioni 27
4.6.2 Implementazione e meccanismo di chiamata 29
4.7 Implementazione del modulo utente 32
4.7.1 File di supporto 32
4.7.2 Proof of Concept 32

5 Conclusioni e indicazioni per la continuazione del progetto **35**

6 Ringraziamenti **36**

Introduzione

L'architettura RISC-V sta rapidamente guadagnando popolarità in una vasta gamma di settori, tra cui sistemi embedded, Internet of Things (IoT), automotive, telefonia, intelligenza artificiale e calcolo ad alte prestazioni. RISC-V definisce un'ISA (Instruction Set Architecture) di tipo RISC (Reduced Instruction Set Computer) caratterizzata da un numero limitato di istruzioni di lunghezza fissa, più veloci da eseguire rispetto a quelle di un'architettura di tipo CISC come può essere x86. Lo standard, concepito dagli sviluppatori dell'Università di Berkeley in California nel 2010, è completamente open-source: in questo modo non solo aumenta la platea di contributori al progetto, ma se ne rafforza anche la diffusione sul mercato grazie all'abbattimento dei costi di licenza.

1.1 Caratteristiche di RISC-V

RISC-V offre un supporto sia per l'architettura a 32 bit che a 64 bit. A seconda delle esigenze di sviluppo, l'ISA può essere integrata in maniera modulare tramite estensioni, ad esempio per aggiungere istruzioni di moltiplicazione e divisione o operazioni floating point. Le specifiche non definiscono l'implementazione fisica dello standard, permettendo lo sviluppo di hardware custom e/o proprietario.

Le istruzioni di RISC-V sono progettate per essere eseguite in un singolo ciclo di CPU, caratteristica che garantisce performance efficienti e maggiore facilità di analisi delle prestazioni. Inoltre, la lunghezza e la posizione dei campi delle istruzioni sono fisse, semplificandone così la decodifica e l'esecuzione.

L'interfacciamento con la memoria è limitato alle istruzioni di caricamento (load) e salvataggio (store) da memoria a CPU e viceversa. Questo approccio comporta l'esecuzione delle operazioni ALU direttamente all'interno dei registri del processore con significativi vantaggi in termini di performance. L'organizzazione di paginazione, cache e memoria virtuale è simile a x86, con un controllo sui permessi di accesso di tipo rwx (read, write, execute). Le operazioni di input/output avvengono con metodo Memory Mapped I/O, pertanto non tutta la memoria fisica è direttamente accessibile o scrivibile dal kernel.

RISC-V presenta, oltre ai registri operativi, una serie di registri di stato e di controllo (CSR), che consentono di interagire con la configurazione e con lo stato di esecuzione della CPU tramite i comandi `csrr` e `csrw`. Ad esempio, tramite tali registri è possibile determinare il livello di esecuzione (*utente*, *supervisore* o *macchina*), la routine di gestione delle interruzioni, la loro abilitazione e molto altro.

1.2 Obiettivi

L'obiettivo principale di questa tesi è quello di migrare parte del nucleo didattico utilizzato nell'esame di Calcolatori Elettronici da x86 a architettura RISC-V. Tale lavoro, che si basa sul precedente operato dei colleghi Edoardo Geraci, Andrea

Bedini e Chiara Panattoni, prevede inizialmente di realizzare il caricamento in memoria di un modulo utente collegato separatamente dal kernel e eseguito a livello utente. La sezione successiva si focalizza sulla creazione e gestione dei processi e sull'inizializzazione del kernel. L'ultima parte descrive lo sviluppo del supporto per le chiamate di sistema e mostra un semplice esempio di programma utente che sfrutta l'invocazione delle primitive e analizza il funzionamento generale del sistema.

Prerequisiti e toolchain di sviluppo

Per poter compilare ed eseguire il codice del nucleo è necessario installare una toolchain per RISC-V a 64 bit. In questa tesi si fa riferimento alla toolchain GNU [4]. Il pacchetto può essere installato tramite il package manager della propria distro Linux (`apt`, `yum`, `aur`, etc...) sotto uno dei seguenti nomi:

- `riscv64-unknown-elf-gcc`
- `riscv-gnu-toolchain-bin`
- `gcc-riscv64-unknown-elf`

Su Ubuntu, ad esempio, è possibile installare la toolchain tramite il comando

```
sudo apt install gcc-riscv64-unknown-elf
```

Una volta installata la toolchain, occorre preparare un ambiente in cui eseguire codice RISC-V a 64 bit. Per fare ciò, è possibile avvalersi dell'emulatore QEMU [3] installando l'architettura RISC-V 64 bit presente all'interno del pacchetto di architetture `qemu-system-misc`. La versione di QEMU utilizzata per l'emulazione del nucleo è chiamata `qemu-system-riscv64`. Su Ubuntu, basta eseguire il comando

```
sudo apt install qemu-system-misc
```

È possibile effettuare il debug del codice prodotto dalla toolchain tramite l'apposita versione di gdb, chiamata `riscv64-unknown-elf-gdb`.[1] Come per x86, anche questa versione di gdb è compatibile con estensioni come gef. Per debuggare un programma, dunque, è sufficiente avviare QEMU e digitare il comando

```
riscv64-unknown-elf-gdb [executable-file]
```

Per utilizzare la toolchain potrebbe essere necessario rendere visibili i binari alla shell, aggiungendo alla variabile d'ambiente PATH la cartella d'installazione. Il percorso da impostare, di default, risulta in genere essere

```
export PATH="/opt/riscv/bin:$PATH"
```

Nella cartella root del progetto [1] è presente il `Makefile`, in cui sono definite le regole di compilazione del sistema. Una volta installati gli strumenti necessari è possibile eseguire il comando `make` sui seguenti target:

[1] Nelle ultime versioni di `gcc-riscv64-unknown-elf`, il debugger non è incluso nel pacchetto, ma va installato manualmente seguendo le istruzioni nel repository ufficiale [4].

- `make [compile]`
  compila il codice;
- `make run`
  compila ed esegue il codice nell'emulatore QEMU;
- `make debug`
  compila ed esegue il codice con la flag `-S`, esponendo l'emulatore per il debug tramite gdb sulla porta TCP 1234;[2]
- `make libce`
  compila la libreria *libCE* con funzioni di utilità per il codice;
- `make clean`
  rimuove i file oggetto e gli eseguibili.

Inoltre il makefile si occupa di individuare la toolchain per RISC-V e, se non presente, restituisce un errore.

[2]Per debuggare, dopo aver eseguito `make debug`, occorre aprire un nuovo terminale, eseguire gdb sullo stesso file eseguibile passato a QEMU e connettersi all'emulatore tramite il comando `target remote localhost:1234`

Organizzazione delle cartelle del sistema

Cartella	**Descrizione**
/doc	Contiene le relazioni e le slide inerenti al progetto.
/include	Contiene i file header e la libreria `libce.a` dopo la compilazione.
/kernel	Contiene i file relativi al bootloader e al modulo sistema.
/libCE	Contiene la libreria utilizzata dal nucleo, completamente migrata a RISC-V.
/objs	Contiene i file oggetto generati durante la compilazione di libCE, modulo sistema e modulo utente.
/user	Contiene i file relativi all'implementazione del modulo utente.

Progettazione e implementazione del sistema

Per ogni sezione del progetto, dalla compilazione all'esecuzione delle prime istruzioni del nucleo fino alla realizzazione di un programma utente funzionante, approfondiamo prima le scelte progettuali con riferimento alle specifiche dell'architettura per poi analizzare nel dettaglio le scelte implementative e il codice sorgente. Dovendo aumentare la complessità del sistema, l'obiettivo parallelo è anche di porre un'attenzione particolare alla pulizia, all'omogeneità del codice e all'organizzazione dei sorgenti.
Il sistema realizza due moduli:

- *sistema*
- *utente*

I due moduli, i cui sorgenti sono collocati rispettivamente nelle cartelle /kernel e /user, sono programmi a sé stanti collegati separatamente.

Il modulo sistema viene eseguito appena dopo aver ricevuto il controllo della CPU da parte del bootloader di QEMU. I sorgenti al suo interno, oltre a contenere le istruzioni di bootstrap e di configurazione eseguite all'avvio, realizzano i processi, la gestione della memoria e delle interruzioni, l'implementazione di tastiera e vga e il supporto per le chiamate di sistema. Il modulo è eseguito con il processore a livello supervisore ad eccezione delle istruzioni di bootstrap, che girano a livello macchina.

Il modulo utente viene caricato in memoria ram dal modulo sistema e realizza un programma con esecuzione a livello utente che invoca le primitive di sistema per realizzare operazioni non consentite dal proprio livello di privilegio, come ad esempio la manipolazione dei registri CSR, la creazione dei processi oppure la lettura e scrittura in alcune aree di memoria.

Data la crescente complessità del progetto e l'interdipendenza delle sezioni da realizzare, si è scelto di eliminare la testsuite (cartella /test delle versioni precedenti del progetto) e di prediligere l'implementazione di un sistema comunque modulare ma da eseguire in modo organico in tutti i suoi elementi, mantenendo la possibilità di verificarne il funzionamento tramite log sul terminale.

4.1 Compilazione e caricamento

La compilazione e il caricamento del sistema avvengono su una macchina Linux ma gli eseguibili generati devono poter girare nell'ambiente fornito dall'emulatore QEMU; per questo motivo è necessario utilizzare alcune opzioni di compilazione, evitare di utilizzare la libreria standard e specificare indirizzi di collegamento opportuni.

L'implementazione di tipo *multiboot* presuppone che il modulo utente sia allocato dal codice del modulo sistema ad indirizzi fisici non noti in fase di caricamento; per questo motivo compiliamo i sorgenti in modo da generare Position Independent Code tramite l'opzione `-fpic`.[1] Inoltre i due programmi devono eseguire alcune operazioni in maniera indipendente (costruttori C++, gestione dello heap, sovrascrittura di funzioni etc.) perciò colleghiamo ad entrambi i moduli il file di libreria `include/libce.a` tramite l'opzione `-Iinclude -lce`. Nel Makefile, le variabili `OBJS`, `USEROBJS` e `LIBCE_OBJECTS` raccolgono i file oggetto derivanti dalla compilazione dei sorgenti dei moduli sistema e utente e della libCE. Visto l'aumento del numero di file da gestire, l'enumerazione degli object file è stata sostituita dall'utilizzo delle funzioni `pathsubst` e `wildcard`.

Secondo le specifiche dell'emulatore `qemu-system-riscv64`, il sistema deve essere caricato in ram a partire dall'indirizzo fisico `0x80000000`. Per ottenere ciò passiamo al collegatore con l'opzione `-T` il linker script kernel.ld, nel quale riserviamo una sezione di memoria ram di dimensione 128 MiB a partire da quell'indirizzo:

```
1 ENTRY(_entry)
2 MEMORY
3 {
4   ram (rwxai) : ORIGIN = 0x80000000, LENGTH = 0x8000000
5 }
```

Codice 4.1: Indirizzo fisico e dimensione della memoria (file `kernel/kernel.ld`)

La keyword **`ENTRY`** definisce l'indirizzo a cui il bootlader di QEMU salta al termine della sua esecuzione, dunque l'indirizzo della prima istruzione del modulo sistema, individuato dal simbolo `_entry` nel file `kernel/entry.s`.

Il linker script descrive anche il posizionamento in memoria delle sezioni **`.text`**, **`.data`** e di altre zone di memoria adibite ai costruttori C++ oppure alle strutture dati utili per il debug etc. Rispetto all'implementazione predefinita, aggiungiamo una sezione per riservare spazio alla memoria dinamica (heap) del modulo sistema, cosicché non rientri nel computo dei frame liberi in fase di inizializzazione:

```
1     /* SYS_HEAP SECTION */
2     .heap           :
3     {
4         . = ALIGN(0x1000);
5         __heap_start = .;
6         . = . + 0x100000; /* 1MiB */
```

[1] Questo si rende necessario in particolare per evitare che l'indirizzamento delle variabili e delle funzioni globali avvenga relativamente al global pointer salvato nel registro `gp`, metodo che non tiene conto del differente spazio di indirizzamento virtuale dei due moduli.

```
7           __heap_end = .;
8           _end = .; PROVIDE (end = .);
9       }
```

Codice 4.2: Sezione **.heap** (file `kernel/kernel.ld`)

Seppure non strettamente necessario, allineiamo alla pagina lo heap sistema e definiamo una sezione di ampiezza 1 MiB. I simboli come `__heap_start` e `end` possono essere utilizzati dal codice per conoscere gli indirizzi non noti a priori ma stabiliti dinamicamente in fase di collegamento.

Il collegamento del modulo utente avviene specificando l'indirizzo virtuale della sezione **.text**. Nel Makefile viene passata al linker l'opzione `-Ttext` seguita dall'indirizzo `0xffff800000001000`, una pagina più avanti rispetto all'inizio della parte utente/condivisa (vedi Sottosezione 4.3.2); questo è dovuto al fatto che il collegatore GNU per RISC-V calcola poi l'indirizzo di inizio della prima sezione prevedendo una pagina di spazio per gli header dell'ELF.

L'ultimo passaggio per avviare il sistema è eseguire QEMU con le opzioni `-kernel` seguita dall'eseguibile del modulo sistema e `-initrd` seguita dall'eseguibile del modulo utente; in questo modo il bootloader carica in ram sia il modulo sistema che il modulo utente. Anche i dettagli dell'inizializzazione del modulo utente saranno approfonditi più avanti alla Sezione 4.5 e alla Sezione 4.7.

4.2 Procedure di bootstrap

Una volta che il bootloader di QEMU ha terminato le proprie operazioni, viene avviata l'esecuzione del codice del modulo sistema a livello macchina. Come anticipato nella Sezione 4.1, il linker script adotta il simbolo `_entry` definito in `kernel/entry.s` come entry point.

4.2.1 File entry.s

Il sorgente `entry.s` prepara il sistema per poter eseguire codice C++ e inizializza lo stack. Avendo compilato e collegato il modulo con l'opzione `-nostdlib`, è necessario invocare la funzione `ctors` definita nella libCE (file `as/ctors.s`):

```
1  .global ctors
2  ctors:
3      addi sp, sp, -36
4      sd s0, 0(sp)
5      sd s1, 8(sp)
6      sd s2, 16(sp)
7      sd ra, 24(sp)
8      la s0, __init_array_start
9      la s1, __init_array_end
10 1:  beq s0, s1, 2f
11     ld s2, 0(s0)
12     jalr s2
13     addi s0, s0, 8
14     j 1b
15 2:  ld s0, 0(sp)
16     ld s1, 8(sp)
17     ld s2, 16(sp)
18     ld ra, 24(sp)
19     addi sp, sp, 36
20     ret
```

Codice 4.3: Funzione ctors della libCE (file `as/ctors.s`)

Il C++ prevede che si possa eseguire del codice prima dell'entry point (per es. nei costruttori degli oggetti globali). Il compilatore organizza questo codice in una serie di funzioni di cui raccoglie i puntatori nell'array `__init_array_start`. Il C++ run time deve poi chiamare tutte queste funzioni prima di saltare all'entry point. Poiche' abbiamo compilato il modulo con `-nostdlib` dobbiamo provvedere da soli a chiamare queste funzioni, altrimenti gli oggetti globali non saranno inizializzati correttamente. La funzione salva i registri preservati e il return address sullo stack, dopodiché assegna a `s0` e `s1` gli indirizzi di inizio e fine dell'array e cicla al suo interno invocando i costruttori; alla fine ripristina i registri dallo stack e ritorna. Implementando `ctors` è stato possibile migrare tutto il codice precedente da linguaggio C a C++. A questo punto `_entry` chiama `start` nel file `kernel/start.s`.

4.2.2 File start.s

In questa fase il processore gira ancora a livello macchina (M-mode). Nella funzione `start` avvengono i seguenti passaggi:

- Nel registro `mstatus` il campo `MPP[1:0]` è settato a `01` cosicché, dopo l'istruzione `mret`, il livello di privilegio passi a supervisore (S-mode);
- Il registro `mepc` è impostato in modo che, dopo l'istruzione `mret`, il programma salti alla funzione `boot_main` definita nel file `kernel/boot_main.cpp`;
- Viene azzerato il registro `satp` per disabilitare la paginazione;
- Vengono delegate tutte le interruzioni e le eccezioni al livello supervisore scrivendo nei registri `medeleg` e `mideleg` e abilitando le interruzioni esterne, timer e software nel registro `sie`;
- Vengono configurati i registri `pmpaddr0` e `pmpcfg0` per permettere alla modalità supervisore di accedere a tutta la memoria fisica;
- Viene inizializzato il timer e il relativo gestore di interruzione a livello macchina;
- Nel registro `stvec` viene impostata la routine `k_trap` relativa alla gestione delle interruzioni a livello supervisore, definita nel file `kernel/traps_asm.s`;
- Viene settato il bit `SUM` del registro `sstatus` per permettere al codice eseguito a livello supervisore di accedere alla memoria virtuale mappata con il bit U pari a 1.[2]

I dettagli di alcuni dei passaggi verranno approfonditi più avanti.

Viene infine eseguita l'istruzione `mret`, che permette di passare alla modalità supervisore e di iniziare a eseguire il codice contenuto nel file `boot_main.cpp`.

[2]Se `sstatus.SUM` è pari a 0 e il processore, quando gira a livello supervisore, accede a un indirizzo mappato con il bit U settato, viene sollevata un'eccezione di page fault.

4.3 Memoria virtuale e dinamica

Nella gestione della memoria occorre fare riferimento alle specifiche di RISC-V, all'implementazione hardware di QEMU e alle singole scelte progettuali. Prima di tutto definiamo il layout della memoria fisica.

4.3.1 Layout della memoria fisica

Le specifiche dell'ISA RISC-V non definiscono uno standard sul layout della memoria fisica, che quindi può essere considerato platform specific. QEMU organizza la memoria fisica come indicato in Figura 4.1.

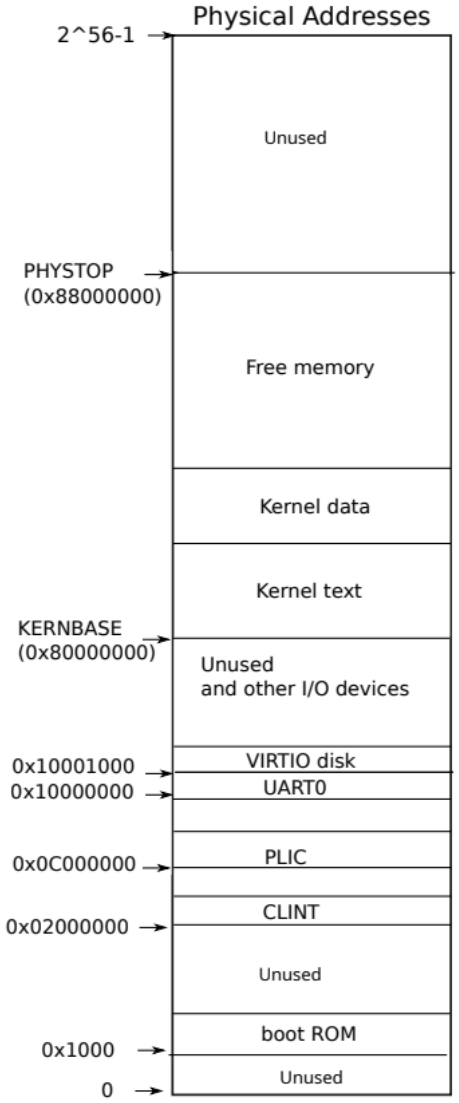

Figura 4.1: Spazio di indirizzamento fisico del sistema

La politica Memory Mapped I/O impone che non tutti gli indirizzi siano accessibili e che alcuni siano riservati a funzionalità specifiche. La ram vera e propria è indirizzata

a partire da `0x80000000` ed ha una dimensione di 128 MiB. Per questo motivo, carichiamo il modulo sistema all'inizio della ram (Sezione 4.1).

4.3.2 Memoria virtuale e paginazione

RISC-V supporta l'astrazione della memoria virtuale e la paginazione tramite un meccanismo molto simile a quello adottato da x86. In particolare sia lo spazio fisico che lo spazio virtuale sono suddivisi in pagine di dimensione 4 KiB; il meccanismo di traduzione associa ai bit più significativi (dal dodicesimo in poi) dell'indirizzo virtuale un numero di pagina fisica (PPN), a cui appende i 12 bit di offset. La versione di base dell'instruction set con cui è implementato il nucleo è RV64I, che prevede operazioni con numeri interi su registri da 8 Byte (XLEN=64). Questo implica anche che gli indirizzi con cui il processore accede in memoria sono lunghi 64 bit, ma non tutti sono effettivamente significativi nel processo di traduzione da indirizzo virtuale a indirizzo fisico. L'ISA definisce diverse modalità di mappatura della memoria virtuale, a seconda del numero di bit che compongono l'indirizzo e del numero di livelli nella traduzione: quella adottata nel nucleo è Sv48, con indirizzi virtuali di 48 bit e 4 livelli di traduzione.

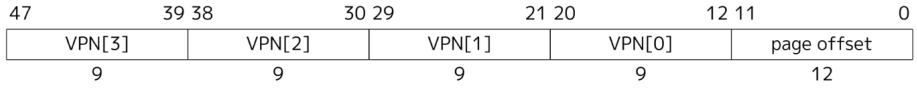

Figura 4.2: Indirizzo virtuale Sv48

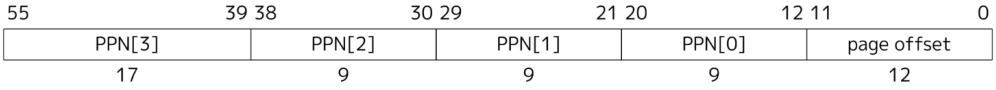

Figura 4.3: Indirizzo fisico Sv48

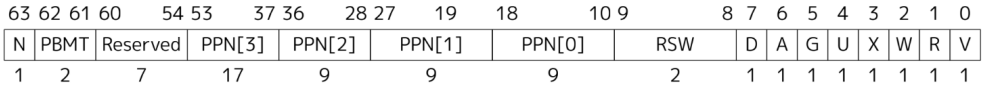

Figura 4.4: Entry di una tabella di traduzione Sv48

Il formato delle entry delle tabelle di traduzione è indicato in Figura 4.4. Ci sono alcune differenze sostanziali rispetto al sistema x86, che sfruttiamo nell'implementazione:

- I permessi di lettura, scrittura ed esecuzione sono codificati dai tre bit R, W e X e non è possibile porre W a 1 e R a 0;[3]
- Se il bit G è settato, la entry della pagina nel TLB sopravvive all'invalidamento del TLB. In questo modo è possibile definire in modo efficiente traduzioni globali valide per tutti i processi;

[3]Se si tenta di scrivere a un indirizzo così mappato viene sollevata un'eccezione di Store page fault.

- Una entry viene considerata non foglia quando possiede i 3 bit RWX a 0. Altrimenti, la traduzione si interrompe in quel punto;
- Il controllo sui permessi (bit RWX e U) è effettuato sulla sola entry foglia;
- Nelle entry, il numero di pagina fisica parte dal bit 10, perciò per estrarre o settare l'indirizzo fisico correttamente è necessario azzerare gli ultimi 10 bit della entry o gli ultimi 12 bit dell'indirizzo e shiftare di due bit a sinistra o a destra.

La modalità di indirizzamento e la posizione in memoria della prima tabella di traduzione devono essere indicate nel registro CSR `satp`, strutturato come in Figura 4.5. Nel campo `MODE` va definita la modalità di indirizzamento, che per Sv48 corrisponde al valore 9, mentre nel campo `PPN` va inserito il numero di frame della tabella di livello massimo, che quindi deve essere allocata ad un indirizzo allineato alla pagina.

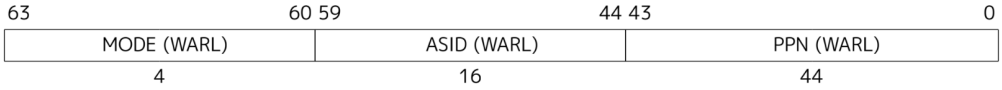

Figura 4.5: Formato del registro `satp`

Implementazione della memoria virtuale nel nucleo

La gestione della memoria virtuale e della paginazione nel nucleo è definita nel sorgente `kernel/vm.cpp` e nell'header `include/vm.h`. Tali file riprendono in gran parte gli script già presenti nella versione in x86 [11] poiché le due architetture adottano un approccio molto simile. Le ultime modifiche hanno permesso di migrare completamente in RISC-V il codice del nucleo afferente alla memoria e allo spazio di indirizzamento dei processi, organizzando il codice in modo modulare.

Le modifiche più rilevanti hanno riguardato l'adeguamento dei sorgenti alle specifiche introdotte nella precedente sezione. Oltre alla scrittura in `sstatus.SUM` trattata nella Sottosezione 4.2.2, è stata introdotta una variazione nelle funzioni di utilità `init_frame`, `alloca_frame` e `rilascia_frame` che serve a tenere conto del fatto che i frame di memoria ram sono allocati a partire dall'indirizzo `0x80000000`, perciò la loro indicizzazione nell'array dei frame `vdf` deve essere opportunamente riscalata.

Nel codice sotto indicato osserviamo, invece, come cambiano `extr_IND_FISICO` e `set_IND_FISICO` per shiftare opportunamente gli indirizzi fisici nelle entry delle tabelle di traduzione.

```
1 // maschera per l'indirizzo (44 bit, offset di 12 bit)
2 const natq ADDR_MASK  = 0xFFFFFFFFFFF000;
3 // maschera per la page table (44 bit, offset di 10 bit)
4 const natq PTE_MASK   = 0x3FFFFFFFFFC00;
5
6 static inline paddr extr_IND_FISICO(tab_entry descrittore)
7 {
8   return ((descrittore & PTE_MASK) << 2);
9 }
```

```
10
11 static inline void  set_IND_FISICO(tab_entry& descrittore, paddr
       ind_fisico)
12 {
13   descrittore &= ~PTE_MASK;
14   descrittore |= ((ind_fisico & ADDR_MASK)>>2);
15 }
```

Codice 4.4: Indirizzo fisico nelle entry di traduzione (file `kernel/vm.h`)

La nuova `extr_IND_FISICO` viene invocata nella funzione `trasforma`, che restituisce l'indirizzo fisico corrispondente all'indirizzo virtuale passato in input nell'albero di traduzione di radice `root_tab`:

```
1 paddr trasforma(paddr root_tab, vaddr v)
2 {
3   tab_iter it(root_tab, v);
4   while (it.down())
5     ;
6
7   tab_entry& e = it.get_e();
8   if (!(e & BIT_V))
9     return 0;
10
11  paddr p = extr_IND_FISICO(e);
12  int l = it.get_l();
13  natq mask = dim_region(l - 1) - 1;
14  return (p & ~mask) | (v & mask);
15 }
```

Codice 4.5: Modifica della funzione `trasforma` (file `kernel/vm.cpp`)

L'implementazione precedente prevedeva l'utilizzo diretto della entry `e`, causando traduzioni shiftate di due bit nella parte alta dell'indirizzo fisico.

È stata migrata anche la funzione `c_access`, primitiva utilizzata a sua volta dalla primitiva `do_log` per controllare che i buffer passati dal livello utente siano accessibili dal livello utente e non possano causare page fault nel modulo sistema. Riportiamo le parti significative per la migrazione:

```
1 extern "C" bool c_access(vaddr begin, natq dim, bool writeable, bool
      shared)
2 {
3   esecuzione->contesto[I_A0] = false;
4
5   // ...
6
7   for (tab_iter it(esecuzione->satp << 12, begin, dim); it; it.next())
      {
8     tab_entry e = it.get_e();
9
10    // interrompiamo il ciclo non appena troviamo qualcosa che non va
11    if (!(e & BIT_V))
12      return false;
13
14    if (it.is_leaf()) {
```

```
15        if (!(e & BIT_U) || (writeable && !(e & BIT_W)))
16          return false;
17      }
18    }
19    esecuzione->contesto[I_A0] = true;
20    return true;
21  }
```

Codice 4.6: Modifica della funzione `c_access` (file `kernel/vm.cpp`)

Essendo una primitiva, salva il valore di ritorno nel contesto del processo in esecuzione nel campo relativo al registro `a0`, per maggiori dettagli si veda più avanti la Sezione 4.4. Alla Linea 7 viene dichiarato un iteratore per scorrere tutte le entry da controllare. Si noti che il costruttore, così come tutte le funzioni di manipolazione della memoria, si aspetta in input non il valore del registro `satp` bensì l'indirizzo della radice dell'albero di traduzione; una soluzione possibile, basandosi sul formato del registro in Figura 4.5, è passare il valore di `satp` shiftato logicamente a sinistra di 12 bit. Nell'iterazione controlliamo che le entry siano tutte valide, cioè con bit V settato, ma il controllo sui permessi lo eseguiamo solo sulla entry foglia (Linea 15); il metodo `is_leaf` restituisce `true` se non c'è il bit V settato (il controllo che la entry sia valida viene eseguito prima nella funzione) oppure se almeno uno dei bit RWX è a 1 oppure se la tabella è di livello minimo 1.

Per impostare `satp` in modo da puntare ad una tabella radice dell'albero di traduzione, nella libCE forniamo la funzione `writeSATP` così definita:

```
1 #define SATP_SV48 (9L << 60)
2 #define MAKE_SATP(pagetable) (SATP_SV48 | (((paddr)pagetable) >> 12))
3 void writeSATP(paddr addr){
4     writeSATP_asm(MAKE_SATP(addr));
5     invalida_TLB();
6 }
```

Codice 4.7: Funzione `writeSATP` (file `libCE/satp_write.cpp`)

dove `writeSATP_asm` semplicemente scrive nel registro il valore passato come argomento, mentre `invalida_TLB` invalida le entrate della cache TLB in seguito al cambio di albero di traduzione tramite l'istruzione `sfence.vma zero, zero`.[4]

Buona parte degli aspetti sopra indicati sono funzionali per realizzare un indirizzamento dello spazio virtuale dei processi quanto più vicino a quello realizzato nel nucleo in x86 [11], in cui i processi hanno sia zone di memoria condivise tra tutti e mappate nel medesimo sottoalbero di traduzione, sia zone di memoria private per ciascuno. La parte che va dall'indirizzo `0000 0000 0000 0000` all'indirizzo `0000 7fff ffff ffff` ($2^{47} - 1$) è dedicata al sistema (bit U pari a 0), mentre la parte che va da `ffff 8000 0000 0000` a `ffff ffff ffff ffff` è dedicata all'utente (bit U pari a 1). I file `constanti.h` e `vm.h` definiscono le costanti con cui suddividiamo la memoria virtuale di tutti i processi in sezioni: la parte sistema/condivisa contiene la finestra sulla memoria fisica, la parte sistema/privata

[4] L'istruzione effettua anche il flush di tutte le scritture e letture in memoria, poiché RISC-V ottimizza tali operazioni potenzialmente alterando l'ordine con cui vengono eseguite.

contiene la pila sistema del processo, la parte utente/condivisa contiene le sezioni **.text** e **.data** e lo heap del modulo utente, la parte utente/privata contiene la pila del processo utente. Dettagli ulteriori saranno affrontati nella Sezione 4.4 e nella Sezione 4.5.

Le traduzioni delle parti condivise vengono create una sola volta all'avvio e poi condivise con tutti i processi. La sezione sistema/condivisa viene generata tramite la funzione `crea_finestra_FM` definita nel file `vm.cpp`. Tale funzione segue lo standard memory mapped I/O e mappa con traduzione identità gli indirizzi relativi a PLIC, UART0, VIRTIO, VGA e memoria ram (quest'ultima con tabelle di livello 2) con i bit RWX e il bit G settati per preservare le traduzioni condivise in caso di invalidazione del TLB.

Un aspetto di cui occorre tenere conto è che il meccanismo della paginazione non viene adottato dal processore in modalità di esecuzione a livello macchina. Sebbene questo non abbia alcun impatto in fase di boot, cioè prima dell'inizializzazione della memoria virtuale, è invece problematico nella gestione delle interruzioni timer. Infatti il gestore del timer salvato in `mtvec` viene eseguito quando nei registri possono esserci anche indirizzi virtuali.

```
1 timer_machine_handler:
2     csrw sscratch, sp
3     li sp, 0x88000000
4     # ...
5     # corpo della funzione
6     # ...
7     csrr sp, sscratch
8
9     mret
```

Codice 4.8: Registro `sp` nell'handler del timer (file `machine_interrupts.s`)

In particolare, si rende necessario salvare il valore dello stack pointer in un registro d'appoggio (RISC-V fornisce appositamente il CSR `sscratch`), assegnare a `sp` un valore inutilizzato della memoria fisica e alla fine ripristinare `sp` da `sscratch`.

4.3.3 Memora dinamica (heap)

Le funzioni di gestione e le dimensioni dello heap sono definite nella libCE. Al modulo sistema il linker script assegna una zona di memoria dinamica successiva e contigua alle sezioni **.text** e **.data** in fase di collegamento, mentre lo heap utente viene allocato dalla funzione `carica_modulo` descritta nella Sezione 4.5. Nel file `vm.cpp` avviene anche l'overloading degli operatori `new` e `delete`, che si limitano a chiamare funzioni della libCE, con cui si alloca e dealloca memoria nello heap. In particolare, il modulo sistema salva in memoria dinamica i descrittori dei processi.

4.4 Gestione dei processi

Il lavoro di migrazione si è concentrato sulla realizzazione dei processi in modo da mantenere l'impianto progettuale del nucleo in x86 [11] riguardo ai processi di tipo utente e sistema e apportando le modifiche necessarie. Pur essendo implementata buona parte delle funzioni di supporto richieste, l'estensione del progetto per il modulo I/O è lasciata a lavori futuri.

In breve, un processo viene inizialmente attivato da un altro processo in esecuzione (ad eccezione del primo) e vengono create le strutture dati necessarie alla sua gestione (pile, albero di traduzione, descrittore, etc.). Dopo l'attivazione e prima della terminazione il processo si può trovare nello stato *esecuzione*, *pronto* o *bloccato*. Consideriamo il sistema monoprocessore, perciò un solo processo può essere in esecuzione, stato che lascia solo in caso di terminazione, attesa di un evento (es. nei semafori) oppure per preemption (es. interruzione). La schedulazione dei processi avviene con un sistema a priorità fissa: ad ogni processo è assegnata una priorità numerica al momento della creazione e il sistema si impegna a garantire che, ad ogni istante, si trovi in esecuzione il processo che ha la massima priorità tra tutti quelli pronti.

Ora analizziamo le scelte implementative e i principali interventi richiesti dalle specifiche di RISC-V.

4.4.1 Descrittore di processo

Il sistema descrive lo stato del processo tramite la struttura `des_proc` definita nell'header `include/proc.h`[5]:

```
1  #define N_REG 31
2  struct des_proc {
3    natw id;
4    natw livello;
5    natl precedenza;
6    vaddr punt_nucleo;
7    natq contesto[N_REG];
8    natq epc;
9    paddr satp;
10   natw spie;
11   des_proc* puntatore;
12 };
```

Codice 4.9: Descrittore di processo (file `include/proc.h`)

Al suo interno troviamo:

`id`: l'identicatore numerico del processo;

`livello`: il livello di privilegio. L'header `costanti.h` definisce i valori costanti `LIV_UTENTE`, `LIV_SISTEMA` e `LIV_MACCHINA` a 0, 1 e 3 secondo le specifiche RISC-V;

[5]Nel codice riportato mancano le informazioni di debug `corpo` e `parametro`.

precedenza: la priorità nelle code dei processi;

punt_nucleo: il puntatore alla base della pila sistema;

contesto: copia dei 31 registri operativi del processore (escluso `x0`[6]), indicizzati con opportune costanti (es. `I_RA`, `I_SP`, etc.);

epc: indirizzo di ritorno dall'interruzione;

satp: valore del registro `satp` relativo al trie del processo;

spie: campo `SPIE` del registro `sstatus`, memorizza l'abilitazione delle interruzioni a livello supervisore;

puntatore: puntatore al prossimo processo in coda;

Si tenga presente che il registro `satp` rappresenta un valore shiftato e sporcato dell'indirizzo della radice del trie del processo, pertanto va passato alle funzioni di utilità della memoria virtuale shiftato a sinistra di 12 bit, come accennato alle Linea 7 della `c_access`. In RISC-V le interruzioni non saltano automaticamente alla pila sistema, perciò `punt_nucleo` va opportunamente gestito via software. Inoltre non avviene il salvataggio delle informazioni in pila come in x86, pertanto dobbiamo memorizzarle nel `des_proc`: a questo scopo salviamo l'indirizzo di ritorno dal registro exception program counter (`epc`), il livello di privilegio e l'abilitazione delle interruzioni in S-mode.[7] I dettagli della gestione di queste informazioni saranno approfonditi nella Sottosezione 4.4.2.

4.4.2 Salvataggio e caricamento dello stato del processo

Quando si verifica un'interruzione da livello utente o supervisore, l'hardware esegue le seguenti operazioni:

- salva il program counter del processo interrotto nel registro `sepc`;
- scrive nel campo Exception Code del registro `scause` un codice che indica la causa dell'interruzione. Nel caso in cui ciò sia dovuto a un page fault, scrive anche nel registro `stval` l'indirizzo che lo ha causato;
- pone il campo `sstatus.SPIE` uguale al campo `sstatus.SIE`, mentre quest'ultimo viene azzerato, in modo da disabilitare ulteriori interruzioni;
- salva la modalità di privilegio corrente (U o S) nel campo `sstatus.SPP` e passa al livello supervisore;
- copia l'indirizzo salvato in `stvec` in `pc` e riprende l'esecuzione da lì.

[6] Il registro `x0` o `zero` è settato in hardware al valore 0.

[7] L'istruzione **`sret`** scrive nel campo Supervisor Interrupt Enable (`SIE`) il valore di Supervisor Previous Interrupt Enable (`SPIE`) riportando l'abilitazione delle interruzioni da livello supervisore come nel processo interrotto.

Si noti che non avviene in automatico il cambio della pila né viene salvata in memoria alcuna informazione, perciò è necessario gestire il tutto via software. La generica routine di interruzione ha il seguente formato:

```
intr:
    addi sp,sp,-8
    sd ra,0(sp)

    call salva_stato
    # ...
    call carica_stato

    ld ra,0(sp)
    addi sp,sp,8

    sret
```

Codice 4.10: Generica routine di interruzione

Innanzitutto viene salvato in pila, all'indirizzo puntato da `sp` e non modificato nella gestione hardware dell'interruzione, l'indirizzo di ritorno della funzione interrotta. Dopodiché si salva lo stato del processo, si esegue il corpo della routine di interruzione e si carica lo stato del processo da riportare in esecuzione, che potrebbe anche essere diverso da quello interrotto. Alla fine si ripristina il return address dallo stack e viene eseguita `sret`: l'hardware reimposta il program counter al valore salvato in `sepc`, il livello di privilegio al valore in `sstatus.SPP` (azzerandolo) e `sstatus.SIE` a `sstatus.SPIE` (azzerandolo).

Le funzioni `salva_stato` e `carica_stato` sono definite nel file assembly `kernel/processi_asm.s`.

```
salva_stato:
  addi sp, sp, -16
  sd a1, 0(sp)
  sd a0, 8(sp)

  la a0, k_trap
  csrw stvec, a0

  la a1, esecuzione_precedente
  ld a0, esecuzione
  sd a0, 0(a1)

  # copiamo per primo il vecchio valore di a1
  ld a1, 0(sp)
  sd a1, A1(a0)
  # usiamo a1 come appoggio per copiare il vecchio a0
  ld a1, 8(sp)
  sd a1, A0(a0)
  # salviamo il valore che sp aveva prima della chiamata a salva stato
  mv a1, sp
  addi a1, a1, 16
  sd a1, SP(a0)
  # Il chiamante deve aver lasciato RA sullo stack
  ld a1, 16(sp)
  sd a1, RA(a0)
```

```
26    # ...
27    # copia dei registri nel contesto
28    # ...
29    # salviamo l'indirizzo di ritorno in EPC
30    csrr a1, sepc
31    sd a1, EPC(a0)
32
33    # salviamo il valore di sstaus.SPIE
34    csrr a1, sstatus
35    andi a1, a1, SSTATUS_SPIE
36    sw a1, SPIE(a0)
37
38    # Ripristiniamo a1 e a0 dalla pila
39    ld a1, 0(sp)
40    ld a0, 8(sp)
41    addi sp, sp, 16
42
43    # se il processo gira a livello sistema non devo fare altro
44    # se il processo gira a livello utente devo spostare sp alla pila
      sistema
45    # sporco t0 e t1
46    ld t0, esecuzione
47    lh t1, LIVELLO(t0)
48    bne t1, zero, 1f
49    ld sp, PUNT_NUCLEO(t0)
50  1:
51    ret
```

Codice 4.11: Funzione `salva_stato` (file `kernel/processi_asm.s`

La funzione `salva_stato` salva nella pila del processo interrotto `a1` e `a0` come registri di appoggio, imposta `stvec` con l'handler di interruzioni da livello supervisore,[8] aggiorna `esecuzione_precedente` con il valore in `esecuzione` e salva nel contesto di `esecuzione` `a1` e `a0` (in pila). Dopodiché salva nel campo `sp` del contesto l'indirizzo puntato prima della chiamata: a prescindere dal livello di esecuzione del processo, a questo indirizzo è salvato il return address della funzione interrotta. A seguire avviene il salvataggio dei registri operativi nel campo `contesto` del `des_proc`.

A questo punto viene salvato l'indirizzo di ritorno nel campo `epc` e il flag di abilitazione delle interruzioni a livello supervisore nel campo `spie`. Ora è possibile ripristinare `a1` e `a0` dalla pila. L'ultimo passaggio, eseguito solo se il processo interrotto girava a livello utente, sposta lo stack pointer alla pila sistema. Questo è necessario perché non possiamo presupporre di poter scrivere nella pila utente senza alterare funzionalità necessarie all'esecuzione del programma utente. Al termine della `salva_stato` la pila si trova nello stato descritto in Figura 4.6.

Alla fine della routine di interruzione va caricato lo stato del processo in esecuzione tramite la funzione `carica_stato` in modo da recuperare `ra` sullo stack e di ritornare correttamente dall'interruzione con `sret`.

```
1  carica_stato:
2    ld s0, esecuzione
```

[8] L'aggiornamento del registro avviene perché `k_trap` gestisce le interruzione da livello privilegiato ed è bene effettuare l'aggiornamento il prima possibile nella routine di interruzione.

```
3   # Paginazione
4   ld a1, SATP(s0)
5   csrr a2, satp
6   beq a1, a2, 1f
7   sfence.vma zero, zero
8   csrw satp, a1
9   sfence.vma zero, zero
10 1:
11  # Spostamento di sp alla pila sistema di esecuzione
12  lh a1, LIVELLO(s0)
13  beqz a1, 1f
14  ld sp, SP(s0)
15  j 2f
16 1:
17  ld sp, PUNT_NUCLEO(s0)
18 2:
19  # ...
20  # distruzione pila del processo precedente
21  # ...
22
23  # sepc
24  ld a1, EPC(s0)
25  csrw sepc, a1
26
27  # ...
28  # Settiamo sstaus.spp a 0 se il processo e' utente, 1 se e' sistema
29  # Settiamo stvec a s_trap se il processo e' utente, a k_trap se e'
      sistema
30  # Ripristiniamo i registri dal contesto
31  # ...
32
33  ld sp, SP(s0)
34  ld s0, S0(s0)
35
36  ret
```

Codice 4.12: Funzione `carica_stato` (file `kernel/processi_asm.s`

Poiché useremo delle funzioni di supporto, salviamo `esecuzione` nel registro preservato `s0`. Se l'albero di traduzione cambia, carichiamo `satp` dal contesto assicurandoci di eseguire il flush delle operazioni di memoria (prima) e l'invalidamento del tlb (dopo) tramite l'istruzione **`sfence.vma`**. Dopodiché spostiamo lo stack pointer, che fino ad ora ha usufruito della pila sistema del processo preceden-

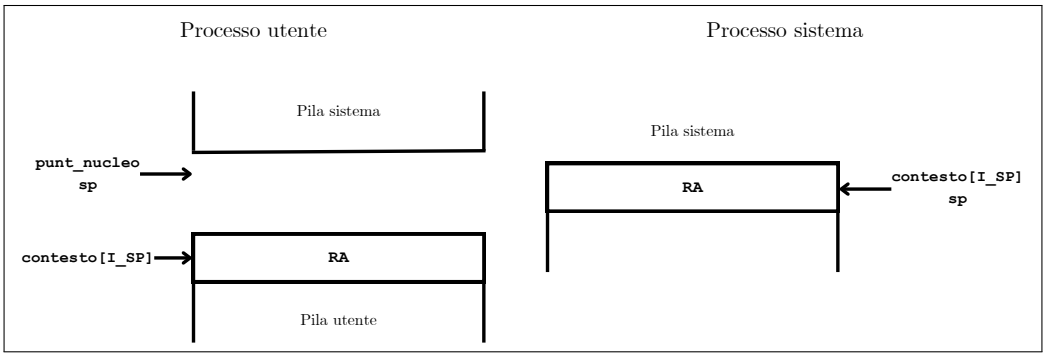

Figura 4.6: Stato della pila al termine di `salva_stato`

te, alla pila sistema di `esecuzione`: se il processo è utente `sp` prende il valore `punt_nucleo` del `des_proc`, cioè la base della pila sistema; se il processo è sistema allora lo stack pointer deve essere posizionato in cima alla pila, nell'indirizzo salvato nel contesto. A questo punto possiamo distruggere la pila del processo precedente, che era stata presa in prestito, se terminato.

Poi carichiamo l'indirizzo di ritorno in `sepc`. Il passaggio successivo è settare `sstatus.SPP` a 0 o 1 a seconda che il processo sia utente o sistema, in modo che `sret` imposti il livello di privilegio corretto. Dopo viene impostata la corretta routine di interruzione (vedi Sezione 4.6) e vengono caricati dal contesto i registri. Solo ora possiamo ripristinare lo stack pointer al valore nel contesto, a prescindere dal livello di esecuzione, perché a quell'indirizzo è salvato il return address della funzione interrotta, e il registro di appoggio `s0`.

La routine di interruzione nel Codice 4.10 può ripristinare `ra` dalla pila ed eseguire `sret`.

4.4.3 Creazione dei processi e funzioni di supporto

La maggior parte delle funzioni di utilità relative alla gestione dei processi non hanno richiesto modifiche per la migrazione a RISC-V e ora sono definite nel file `kernel/processi_c.cpp`. La struttura delle funzioni di salvataggio e caricamento dello stato dei processi impone, tuttavia, alcune variazioni al momento della creazione del processo. La primitiva `activate_p` invoca la funzione `crea_processo`:[9]

```
1  des_proc* crea_processo(void f(natq), natq a, int prio, char liv)
2  {
3    des_proc* p;       // des_proc per il nuovo processo
4    natl      id;      // id del nuovo processo
5
6    p = new des_proc;
7    memset(p, 0, sizeof(des_proc));
8
9    p->precedenza = prio;
10   p->puntatore = nullptr;
11   p->contesto[I_A0] = a;
12
13   id = alloca_proc_id(p);
14   p->id = id;
15
16   p->satp = alloca_tab();
17   init_root_tab(p->satp);
18
19   p->epc = reinterpret_cast<natq>(f);
20   p->spie = 1;
21   crea_pila(p->satp, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
22
23   if (liv == LIV_UTENTE) {
24     crea_pila(p->satp, fin_utn_p, DIM_USR_STACK, LIV_UTENTE)
25     p->contesto[I_SP] = fin_utn_p - sizeof(natq);
26     p->livello = LIV_UTENTE;
```

[9]Per esigenze di spazio, sono state rimosse dal codice le parti relative ai campi di debug del descrittore di processo e alla gestione degli errori tramite **goto**.

```
27      p->punt_nucleo = fin_sis_p;
28    } else {
29      p->contesto[I_SP] = fin_sis_p - sizeof(natq);
30      p->livello = LIV_SISTEMA;
31    }
32
33    p->satp = (p->satp >> 12) | (9L << 60);
34
35    return p;
36
37  // ...
38  // label di gestione degli errori tramite goto
39  // ...
40  }
```

Codice 4.13: Funzione di creazione dei processi (file `kernel/processi_c.cpp`)

La funzione alloca nello heap il descrittore del nuovo processo tramite l'operatore **new** (vedi Sottosezione 4.3.3) e setta preventivamente tutti i suoi campi a 0. In una prima fase i valori già noti, opportunamente controllati dalla primitiva `activate_p`, vengono impostati nel `des_proc`. Salviamo dunque:

- la priorità;
- il puntatore al prossimo descrittore, che sarà aggiornato al momento dell'inserimento in lista `pronti`;
- l'argomento della funzione `f`, eseguita all'avvio del processo, nel campo `I_A0` del contesto. Sarà caricato in `a0` da `carica_stato`;
- il campo `satp`, inizialmente contenente l'indirizzo della radice del trie e poi opportunamente modificato per essere caricato nel registro `satp` alla Linea 33. Le traduzioni comuni a tutti i processi, cioè relative alle sezioni condivise (Sottosezione 4.3.2), vengono opportunamente impostate nella `init_root_tab` copiando le entry della tabella radice del processo in esecuzione nella tabella radice del nuovo processo;
- l'indirizzo a cui saltare dopo l'esecuzione di `sret`, impostato in `sepc` dalla `carica_stato`;
- l'abilitazione delle interruzioni da livello supervisore;

Inoltre creiamo la pila sistema, che viene allocata in nuovi frame non condivisi.

A questo punto, se il nuovo processo gira a livello utente, non serve inizializzare la pila sistema con i campi richiesti da **iretq** come in x86. Creiamo la pila utente e prepariamo il campo `I_SP` del contesto con il primo indirizzo della pila libero, sotto al quale la routine di interruzione si aspetta di recuperare `ra` (Figura 4.6); impostiamo il livello di esecuzione e l'indirizzo della base della pila sistema. Se il processo gira a livello sistema, il codice utilizza solo la pila sistema perciò non serve gestire `punt_nucleo`; anche in questo caso la routine di interruzione si aspetta di trovare in cima alla pila `ra`, perciò alziamo `sp` all'indirizzo precedente.

Terminata l'implementazione del supporto per memoria virtuale e processi, è possibile procedere all'inizializzazione del nostro sistema multiprogrammato.

4.5 Inizializzazione del sistema

Come anticipato nella Sottosezione 4.2.2, il flusso di esecuzione del nostro sistema, una volta terminate le configurazioni, continua alla funzione `boot_main` nel file `kernel/boot_main.cpp` passando da livello macchina a livello supervisore. L'obiettivo è quello di inizializzare il sistema sfruttando le funzioni di utilità descritte nelle sezioni precedenti, in modo da effettuare il setup della memoria virtuale, creare i primi processi sistema e caricare in memoria il modulo utente. Al termine dell'inizializzazione il programma deve essere in grado di abbassare il proprio livello di privilegio ed eseguire il codice utente.

4.5.1 Memoria virtuale e caricamento del modulo utente

Innanzitutto viene inizializzato il VGA e il Platform-Level Interrupt Controller (PLIC) [12]. Poi viene creata la finestra sulla memoria fisica, attivata la paginazione scrivendo nel registro `satp` e inizializzato lo heap sistema sfruttando gli indirizzi nel linker script (vedi Codice 4.2). A questo punto creiamo un primo processo di appoggio per visualizzare correttamente i log del sistema sul terminale[10] e per preparare le entry di traduzione dello spazio condiviso per tutti i processi. L'eseguibile relativo al modulo utente viene passato a QEMU con l'opzione `-initrd` (initial ramdisk, vedi Sezione 4.1). I sorgenti di QEMU [3] definiscono il comportamento di tale opzione nella funzione `riscv_load_initrd` del file `qemu/hw/riscv/boot.c`: per sistemi con memoria inferiore a 1 GiB come il nostro, l'ELF viene caricato a metà della ram, dunque a partire dall'indirizzo `0x84000000`.[11]

Scriviamo una funzione `carica_modulo` che prende in ingresso l'indirizzo attuale di caricamento del modulo in ram, la radice dell'albero di traduzione del processo in cui vogliamo mappare il modulo, un flag che indica se il modulo è accessibile a livello utente e la quantità di memoria che intendiamo destinare allo heap. Con opportune strutture dati, interpretiamo l'header dell'ELF caricato in memoria. Per ogni sezione (**`.text`**, **`.data`**, **`.bss`**, etc.) l'ELF fornisce indirizzo virtuale di collegamento, dimensione e permessi di lettura, scrittura, esecuzione: con queste informazioni possiamo copiare il contenuto delle varie sezioni (o azzerarlo) in frame allocati di seguito al modulo sistema e mappare tali frame agli indirizzi virtuali corrispondenti con il flag G settato.[12] In fondo la funzione alloca dello spazio aggiuntivo per lo heap e ritorna l'entry point del modulo salvato nell'header dell'ELF.

In questo modo è stato allocato e mappato il modulo utente ed è noto l'indirizzo virtuale a cui far saltare il codice nel passaggio da processo sistema a processo utente. L'output sul terminale è indicato di seguito:

[10] Il sistema invia dei log sul terminale tramite la funzione `flog` definita nella libCE.

[11] In alcune versioni di QEMU, come quella distribuita da `apt` nella macchina virtuale del corso, l'header dell'ELF è caricato a partire dalla pagina precedente, all'indirizzo `0x83fff000`. Nel caso in cui il bootloader di QEMU salvi l'header dell'ELF a partire da metà esatta della ram, come accade nella versione fornita da `apt` in Ubuntu, è possibile eseguire il comando `make` con la variabile d'ambiente `UBUNTU` settata.

[12] Come la finestra sulla memoria fisica, anche il modulo utente è condiviso tra i processi perciò non occorre invalidare le traduzione nel tlb al cambio di processo.

```
INF    ?    Running in S-Mode
INF    ?    PCI initialized
INF    ?    VGA inizialized
INF    ?    PLIC Initialized
INF    ?    Numero di frame: 312 (M1) 32456 (M2)
INF    ?    Allocata tabella root
INF    ?    Creata finestra sulla memoria centrale: [0000000000001000, 0000000088000000)
INF    ?    Attivata paginazione
INF    -    Nucleo di Calcolatori Elettronici - RISC-V
INF    -    Heap del modulo sistema: [0000000080038000, 0000000080138000)
INF    -    Suddivisione della memoria virtuale:
INF    -    - sis/cond [0000000000000000, 0000008000000000)
INF    -    - sis/priv [0000008000000000, 0000010000000000)
INF    -    - usr/cond [ffff800000000000, ffffc00000000000)
INF    -    - usr/priv [ffffc00000000000, 0000000000000000)
INF    -    mappo il modulo utente:
INF    -     - segmento utente read-only  mappato a [ffff800000000000, ffff800000003000)
INF    -     - segmento utente read/write mappato a [ffff800000003000, ffff800000004000)
INF    -     - heap:                               [ffff800000004000, ffff800000104000)
INF    -     - entry point: 0xffff800000001000
INF    -    Frame liberi: 32183 (M2)
INF    -    Creato il processo dummy (id = 0)
INF    -    Creato il processo main_sistema (id = 1)
INF    -    Cedo il controllo al processo main sistema...
```

Codice 4.14: Inizializzazione: log sul terminale

4.5.2 Creazione dei processi sistema e utente

La funzione `boot_main` prosegue creando il processo `dummy` a minima priorità, che rimane in attesa di interruzioni tramite l'istruzione RISC-V `wfi` se ci sono ancora processi bloccati ed esegue lo *shutdown* del sistema quando non ci sono più processi attivi.

Di seguito avviene la creazione del primo processo sistema da mandare in esecuzione: il processo viene creato e il suo descrittore è inserito in lista `pronti` e schedulato. L'ultimo passaggio consiste nella chiamata a `carica_stato` che prepara il sistema per l'esecuzione di `sret`.

A questo punto è possibile creare il processo utente impostando come funzione da eseguire al suo avvio l'entry point restituito da `carica_modulo`. Nel processo sistema possiamo anche effettuare il test delle funzionalità della tastiera, già sviluppate dai colleghi Bedini [2] e Panattoni [12]. Infine, terminiamo il processo tramite la primitiva `terminate_p` e cediamo il controllo al processo utente. L'esecuzione da questo punto in poi sarà descritta più avanti nella Sezione 4.7. L'output sul terminale continua così:

```
INF    1    Creo il processo main utente
INF    1    proc=2 entry=ffff800000001000(0) prio=1023 liv=0
Press a character key:
The read character was: A
Write something: Ciao mondo!
You have written: Ciao mondo!
INF    1    Cedo il controllo al processo main utente...
INF    1    Processo 1 terminato
```

Codice 4.15: Processo main sistema: log sul terminale

4.6 Supporto per le chiamate di sistema

In questa sezione progettiamo e implementiamo il framework delle primitive di sistema. L'obiettivo è fornire un supporto affinché il modulo utente possa invocare delle chiamate di sistema atomiche che girino a livello supervisore.

4.6.1 Primitive di sistema e interruzioni

L'istruzione RISC-V con cui il codice può invocare una primitiva sollevando un'interruzione software è **`ecall`** (environmental call), senza argomenti. Tale istruzione provoca gli stessi effetti già elencati nella Sottosezione 4.4.2; in particolare, con riferimento al formato del registro in Figura 4.7, imposta il campo `Interrupt` a 0 e il campo `Exception Code` a 8 per le chiamate dal livello di esecuzione utente o a 9 per le chiamate dal livello supervisore.

Figura 4.7: Formato del registro `scause`, con `SXLEN` pari a 64.

In RISC-V, qualsiasi interruzione gestita a livello supervisore (nel nostro caso tutte ad eccezione del timer) esegue la routine puntata da `stvec`. Implementiamo, dunque, due funzioni di gestione delle interruzioni, definite in `kernel/traps_asm.s`: `k_trap` intercetta le interruzioni generate a livello supervisore nel modulo sistema, mentre `s_trap` quelle sollevate a livello utente nel modulo utente. Per questo, all'avvio impostiamo `stvec` con l'indirizzo di `k_trap` (vedi Sottosezione 4.2.2).

Entrambe seguono il formato della `intr` definita nel Codice 4.10 e tra l'esecuzione di `salva_stato` e di `carica_stato` chiamano rispettivamente le funzioni C++ `kInterruptHandler` e `sInterruptHandler` in `kernel/traps_c.cpp`. Il registro `stvec` viene aggiornato con la giusta routine, a seconda che il processo in esecuzione sia di tipo sistema o utente, nella `carica_stato` (vedi Sottosezione 4.4.2) così da eseguire il cambio più in là possibile nel flusso di esecuzione della routine.

Analizziamo dapprima `sInterruptHandler`, nel Codice 4.16:

```
1  extern "C" void sInterruptHandler(){
2      if ((readSSTATUS() & SSTATUS_SPP) != 0)
3          fpanic("usertrap: not from user mode");
4
5      if (readSCAUSE() == 8) {
6          esecuzione->epc += 4;
7          syscall();
8      }
9      else if (dev_int() != 0) {
10     }
11     else {
12         flog(LOG_WARN, "unexpected scause=%p, id=%p", readSCAUSE(),
       esecuzione->id);
13         flog(LOG_WARN, "sepc=%p, stval=%p", readSEPC(), readSTVAL());
14
```

```
15          c_terminate_p();
16      }
17      disableSInterrupts();
18  }
```

Codice 4.16: Parte C++ della routine utente (file `kernel/traps_c.cpp`)

Per prima cosa si controlla che il campo `SPP` di `sstatus` sia pari a 0, cioè che la modalità di esecuzione prima dell'interruzione fosse utente.

A questo punto occorre individuare via software il tipo di interruzione e gestirla opportunamente. Ai fini della nostra implementazione distinguiamo:

chiamate di sistema: sono caratterizzate da un valore in `scause` pari a 8 in quanto da livello utente. Si noti che al sollevamento di un'interruzione con **`ecall`**, il registro `sepc` viene impostato con l'indirizzo dell'istruzione: di conseguenza, poiché vogliamo che dopo la primitiva il codice continui dall'istruzione successiva a **`ecall`**, aumentiamo il campo `epc` del descrittore della lunghezza di un'istruzione. L'esecuzione della primitiva vera e propria avviene all'interno di `syscall` e verrà trattata nella sottosezione successiva;

interrupt: sono interruzioni caratterizzate da un valore di `scause.Interrupt` pari a 1. Nella funzione `dev_int`, già sviluppata nel precedente lavoro [12], vengono gestite interrupt software e esterne;

eccezioni: tutto ciò che non rientra nelle precedenti categorie è inaspettato e indica verosimilmente un errore o un'eccezione. Stimoliamo il log di alcune informazioni utili e terminiamo il processo.

Poiché potremmo venire da codice che aveva precedentemente eseguito in modalità supervisor con le interruzioni abilitate, ad esempio all'interno di una chiamata di sistema, le disattiviamo tramite una funzione scritta in assembly che annulla `sstatus.SIE`. Infine la funzione termina e si procede con la `carica_stato`.

La parte C++ dell'handler delle interruzioni da livello supervisore non è molto diversa:

```
1  extern "C" void kInterruptHandler(){
2      natq epc = readSEPC();
3      natq status = readSSTATUS();
4      natq cause = readSCAUSE();
5
6      if ((status & SSTATUS_SPP) == 0)
7          fpanic("kerneltrap: not from supervisor mode");
8
9      if (cause == 9) {
10         esecuzione->epc += 4;
11         syscall();
12         return;
13     }
14
15     if (dev_int() == 0) {
16         flog(LOG_WARN, "scause=%p", readSCAUSE());
17         flog(LOG_WARN, "sepc=%p, stval=%p", readSEPC(), readSTVAL());
```

```
18            fpanic("kerneltrap");
19        }
20    }
```

Codice 4.17: Parte C++ della routine supervisore (file `kernel/traps_c.cpp`)

Stavolta controlliamo che `sstatus.SPP` sia settato (S-mode) e gestiamo le chiamate di sistema se `scause` ha valore 9 (environmental call da S-mode). Gestiamo il caso di eccezioni o errori da livello sistema differentemente da prima, perché se l'errore è avvenuto a livello sistema vuol dire che c'è una falla grave e quindi che non possiamo limitarci a terminare il processo ma dobbiamo interrompere l'esecuzione tramite `fpanic`.

4.6.2 Implementazione e meccanismo di chiamata

Occorre implementare un meccanismo di gestione delle chiamate di sistema che permetta al programma che le invoca di "comunicare" all'handler di interruzione quale primitiva eseguire e i valori dei parametri attuali. Infatti, mentre in architettura x86 associamo ad ogni primitiva un codice e un'entrata delle Interrupt Descriptor Table ed è l'hardware a saltare alla funzione corretta, in RISC-V questo controllo è lasciato al software ed in particolare alla routine di interruzione puntata da `stvec`.

Come interfaccia per le chiamate di sistema, il programmatore di sistema crea un header `include/sys.h` che raccoglie i prototipi delle funzioni che il codice può eseguire per invocare una primitiva. Qualunque sorgente che voglia invocare una primitiva lo includerà tramite la direttiva **`#include`**. Supponiamo di voler fornire al modulo utente la seguente primitiva generica:

```
...
extern "C" tipo_r primitiva_i(parametri formali);
...
```

Codice 4.18: Dichiarazione della primitiva in `sys.h`

dove nel prototipo vengono indicati anche il tipo del valore di ritorno e i parametri formali.

Per invocare la primitiva basta, allora, chiamarla come una normale funzione C++. Ad esempio, nel codice seguente:

```
#include <sys.h>
...
void corpo(int a)
{
    ...
    tipo_r res;
    res = primitiva_i(parametri attuali);
    ...
}
...
```

Codice 4.19: Chiamata della primitiva nel codice C++

L'obiettivo di `primitiva_i` è eseguire il corpo vero e proprio della primitiva, definito nel modulo sistema come `c_primitiva_i`. Associamo ad ogni primitiva un codice *TIPO_I*, vale a dire un valore univoco con cui identificarla, definito nell'header `include/costanti.h`. Nel corpo della funzione sopra definita, utilizziamo il registro `a7` per comunicare alla routine di interruzione quale primitiva intendiamo eseguire. Tale funzione è implementata in assembly nel seguente modo:

```
...
.global primitiva_i
primitiva_i:
  li a7, TIPO_I
  ecall
  ret
...
```

Codice 4.20: Codice assembly per l'invocazione della primitiva

Dapprima viene caricato in `a7` il codice relativo alla primitiva, che viene invocata tramite l'istruzione **`ecall`**. Avviene il salto alla routine di interruzione a livello supervisore e il processo perde il controllo della CPU. A questo punto, sia `s_trap` che `k_trap` eseguono diverse istruzioni, poi controllano il valore in `scause` e saltano alla funzione `syscall` definita in `kernel/traps_c.cpp` e riportata schematicamente di seguito:

```
1  void syscall(void) {
2    natq num;
3    des_proc *p = esecuzione;
4
5    num = p->contesto[I_A7];
6
7    switch (num)
8    {
9      ...
10     case TIPO_I:
11         c_primitiva_i(p->contesto[I_A0], p->contesto[I_A1], ...);
12         break;
13     ...
14     default:
15         flog(LOG_WARN, "unknown sys call %d\n", num);
16         p->contesto[I_A0] = -1;
17         break;
18   }
19 }
```

Codice 4.21: Funzione `syscall` per la chiamata alla primitiva

Osserviamo che, dal momento in cui è stata sollevata l'interruzione all'esecuzione di `syscall`, i registri non preservati potrebbero essere stati sporcati. Di conseguenza, per recuperare il valore di `a7` impostato nel Codice 4.20, è necessario leggerlo nella variabile `num` dal contesto del processo, il quale è stato opportunamente salvato dalla `salva_stato`. A questo punto viene chiamata la funzione corrispondente alla primitiva con codice identificativo `num`. Nel caso generale, viene chiamata

`c_primitiva_i`. Si noti che il compilatore C++ traduce il Codice 4.19 muovendo i valori dei parametri attuali all'interno dei registri `a0`, `a1`, `a2` e così via. Di nuovo, la routine di interruzione potrebbe sporcare tali registri perciò passiamo come argomenti di `c_primitiva_i` i valori salvati nel contesto, eventualmente effettuando il casting al tipo corretto (es. puntatore a funzione, attributo **`const`**, etc.). Le primitive di tipo non **`void`** devono scrivere nel campo `I_A0` del contesto nel `des_proc` il valore di ritorno, secondo lo schema indicato di seguito:

```
...
extern "C" void c_primitiva_i(parametri_formali)
{
    ...
    esecuzione->contesto[I_A0] = (natq)ris;
}
...
```

Codice 4.22: Valore di ritorno della primitiva

Allo stato attuale, il sistema fornisce le seguenti primitive:

```
extern "C" natl activate_p(void f(natq), natq a, natl prio, natl liv);
extern "C" void terminate_p();
extern "C" natl sem_ini(int val);
extern "C" void sem_wait(natl sem);
extern "C" void sem_signal(natl sem);
extern "C" void do_log(log_sev sev, const char* buf, natl quanti);
```

Codice 4.23: Primitive implementate (file `include/sys.h`)

Le prime due attivano e terminano i processi; `sem_ini`, `sem_wait` e `sem_signal` sono primitive semaforiche e fanno riferimento al sorgente `kernel/semafori.cpp`, sostanzialmente copiato dal nucleo in x86; `do_log` permette al modulo utente di scrivere log sul terminale chiamando `flog`.

Ovviamente è possibile aggiungere altre primitive di sistema con lo stesso meccanismo spiegato in questa sezione.

4.7 Implementazione del modulo utente

Mentre il modulo sistema cambia raramente ed è sviluppato dal programmatore di sistema, il modulo utente rappresenta il programma, di volta in volta diverso, che l'utente del nostro sistema vuole eseguire. Nella cartella `user` definiamo alcuni sorgenti e header di supporto e proponiamo un esempio di programma utente da eseguire come test.

4.7.1 File di supporto

Forniamo all'utente una serie di funzioni di utilità nel file `lib.cpp` e nel relativo header `lib.h`, oltre a quelle già presenti nella libCE. In particolare definiamo: gli operatori **`new`** e **`delete`** per la gestione dello heap utente; la funzione `panic` per segnalare errori e terminare il processo; il costruttore `lib_init`, che alloca un semaforo di mutua esclusione per le operazioni sulla memoria dinamica e inizializza lo heap.

Il vero e proprio programma utente, scritto in `user_c.cpp`, per prima cosa include l'header `all.h`. In questo modo ha accesso alle costanti del sistema (`costanti.h`), al file di intestazione della libCE (`libce.h`), ai prototipi delle primitive di sistema (`sys.h`) e alla libreria utente (`lib.h`).

All'utente forniamo anche il file assembly `user_asm.s`. Al suo interno sono definite le funzioni di interfaccia per invocare le primitive con **`ecall`** (vedi Codice 4.20). Anche l'entry point del modulo utente è in questo file:

```
.global _start, start
_start:
start:
    call ctors
    call main
    ret
```

Codice 4.24: Entry point del modulo utente (file `user/user_asm.s`)

La funzione, eseguita in modalità di privilegio utente, si limita a chiamare i costruttori del C++ tramite la funzione `ctors` della libCE prima di saltare al main del programma utente. Si noti che tra i costruttori chiamati vi è anche `lib_init` in `lib.cpp`, in quanto dichiarata come
`void` `lib_init() __attribute__((constructor));`
In questo modo è possibile definire procedure di inizializzazione da eseguire prima dell'avvio del programma `main` utente senza dover effettuare modifiche al file `user_c.cpp`.

4.7.2 Proof of Concept

Nel semplice programma utente di test vogliamo contare fino a 1000. L'obiettivo è testare il meccanismo delle primitive e le funzionalità di caricamento del modulo utente da parte del modulo sistema. Il file `user/user_c.cpp` contiene il seguente codice:

```
#include "all.h"

natl mainb_proc, conta_proc;
int count = 0;
natl end_count;

void conta(natq quanti) {
    for (int i = 0; i < quanti; i++) {
        count++;
    }

    flog(LOG_DEBUG, "Fine conta: count = %d", count);
    sem_signal(end_count);
    terminate_p();
}

void main_body(natq tot) {
    natq quanti1 = tot/2;

    end_count = sem_ini(0);

    flog(LOG_DEBUG, "Creo il processo conta1");
    conta_proc = activate_p(conta, quanti1, MIN_PRIORITY, LIV_UTENTE);
    flog(LOG_DEBUG, "Aspetto che il processo conta1 conti fino a %d", quanti1);
    sem_wait(end_count);

    flog(LOG_DEBUG, "Creo il processo conta2");
    conta_proc = activate_p(conta, tot-quanti1, MIN_PRIORITY, LIV_UTENTE);
    flog(LOG_DEBUG, "Aspetto che il processo conta2 conti fino a %d", tot-quanti1);
    sem_wait(end_count);

    if (count == (int)tot) {
        flog(LOG_DEBUG, "Test completato con successo", count);
        flog(LOG_DEBUG, ">>>FINE<<<");
    }

    terminate_p();
}

extern "C" void main() {
    flog(LOG_DEBUG, ">>>INIZIO<<<");

    flog(LOG_DEBUG, "Creo il processo main_body utente");
    natl mainb;
    mainb = activate_p(main_body, 1000, MIN_PRIORITY+1, LIV_UTENTE);

    flog(LOG_DEBUG, "Cedo il controllo al processo main_body utente...");
    terminate_p();
}
```

Codice 4.25: Esempio di programma utente (file `user/user_c.cpp`)

In particolare, il processo `main` utente attiva un ulteriore processo che esegue la funzione `main_body` e termina. La `activate_p` passa come argomento `tot` a `main_body` il valore da contare, in questo caso 1000.

`main_body` inizializza il semaforo di sincronizzazione `end_count` con la primitiva `sem_ini`, dopodiché crea due processi `conta` che, in due step, portano la variabile globale `count` al valore `tot`. Più nel dettaglio, `main_body` attiva `conta` tramite la primitiva `activate_p` e si sospende in attesa della fine del conteggio sulla primitiva `sem_wait`. `conta` per due volte esegue il conteggio, risveglia `main_body` con la primitiva `sem_signal` e termina.

Al termine del conteggio, viene inviato un log sul terminale per segnalare il successo e tutti i processi utente terminano. Rimane attivo solo il processo `dummy`, che esegue lo shutdown del sistema. L'ultima parte del log del nostro sistema, dall'esecuzione del primo processo utente allo shutdown, è dunque:

```
INF    2    Heap del modulo utente: 000000000100000 [ffff8000000034d0, ffff8000001034d0)
DBG    2    >>>INIZIO<<<
DBG    2    Creo il processo main_body utente
INF    2    proc=3 entry=ffff80000000132e(1000) prio=2 liv=0
DBG    2    Cedo il controllo al processo main_body utente...
INF    2    Processo 2 terminato
DBG    3    Creo il processo conta1
INF    3    proc=4 entry=ffff8000000012b4(500) prio=1 liv=0
DBG    3    Aspetto che il processo conta1 conti fino a 500
DBG    4    Fine conta: count = 500
DBG    3    Creo il processo conta2
INF    3    proc=5 entry=ffff8000000012b4(500) prio=1 liv=0
DBG    3    Aspetto che il processo conta2 conti fino a 500
INF    4    Processo 4 terminato
DBG    5    Fine conta: count = 1000
DBG    3    Test completato con successo
DBG    3    >>>FINE<<<
INF    3    Processo 3 terminato
INF    5    Processo 5 terminato
INF    0    Shutdown
```

Codice 4.26: Processi utente: log sul terminale e shutdown

A questo punto, anche il modulo utente è completamente implementato in architettura RISC-V. Qualunque utente del nostro sistema potrebbe caricare il proprio programma ed eseguirlo correttamente.

Questa è l'ultima fase di migrazione del progetto in architettura RISC-V relativamente a questo progetto di tesi. Nel Capitolo 5 vengono ricapitolate le principali fasi dello sviluppo del sistema e vengono proposti alcuni suggerimenti per la prosecuzione del lavoro.

Conclusioni e indicazioni per la continuazione del progetto

Il lavoro svolto ha condotto ad una migrazione quasi completa dei moduli sistema e utente in architettura RISC-V. Il primo approccio seguito è stato quello di realizzare o ampliare le funzionalità essenziali del nucleo separatamente, implementando la compilazione e il caricamento del programma utente in memoria, modificando opportunamente le procedure di avvio, la gestione della memoria virtuale e delle interruzioni, sviluppando il supporto per i processi utente e il framework delle primitive di sistema. Durante questa fase si è dedicata attenzione anche alla pulizia e all'omogeneizzazione del codice e dei sorgenti.

Tali sezioni sono state poi integrate per inizializzare il nucleo in modo consistente e per cedere il controllo ai processi utente. Il risultato finale è un Proof of Concept in grado di testare il setup e le funzionalità del sistema attraverso un programma di esempio che si basa su tutte le sezioni implementate nel nucleo.

Ovviamente il lavoro per sviluppare il nucleo in architettura RISC-V non è terminato. In alcuni passaggi è stata tralasciata la migrazione di operazioni comode per l'interazione con il sistema ma non indispensabili al suo funzionamento. Altre sezioni rimangono, invece, completamente da sviluppare. Nel caso in cui qualche studente voglia cimentarsi nella prosecuzione del lavoro, alcuni suggerimenti per la continuazione del progetto sono:

- integrare i processi e il timer per realizzare la primitiva `delay`;
- ripristinare le funzionalità di dump e di debug per aumentare la verbosità dei log sul terminale;
- studiare il protocollo VIRTIO e l'interazione con le periferiche per realizzare il modulo I/O;
- realizzare le estensioni per il debugger in modo da ottenere informazioni e comandi aggiuntivi rispetto a quelli di base di gef e gdb;
- proporre variazioni nel nucleo basate sui temi d'esame o sul funzionamento di kernel reali, come ad esempio la paginazione su domanda, la schedulazione time-sharing, il file system, etc.

Il compito ha presentato un certo grado di complessità e ha implicato la soluzione di una serie di imprevisti non banali. Tuttavia, l'impegno profuso è stato ampiamente ricompensato dalla soddisfazione finale e dal valore formativo dell'esperienza. A chiunque desideri proseguire l'opera va il mio sostegno e l'augurio di un buon lavoro.

Ringraziamenti

Questa tesi segna la conclusione di un percorso troppo intenso, ricco e complesso per poter essere analizzato con lucidità. Guardo a questo momento come a un punto di svolta, vissuto con un pizzico di nostalgia ma anche con profonda speranza verso il futuro.

Vorrei esprimere la mia profonda gratitudine al Prof. Giuseppe Lettieri per la sua straordinaria disponibilità, per la sua chiarezza e per avermi dato l'opportunità di impegnarmi in questo lavoro, che ho trovato estremamente stimolante. Desidero ringraziare anche i miei predecessori, Edoardo Geraci e Andrea Bedini per l'imponente lavoro svolto e Chiara Panattoni per i preziosi consigli.

Un ringraziamento speciale va alla mia famiglia, perché non è scontato poter contare in ogni istante su un punto di riferimento così solido e rassicurante. Non posso che essere grato per la loro presenza costante e per essere allo stesso momento comprensivi ed esigenti. Ciò che sono lo devo a loro, a chi c'è oggi e a chi ci sarebbe voluto essere.

Ringrazio gli amici che ho dovuto lasciare la mattina di quel 4 ottobre 2021, perché non è scontato mantenere vivo il legame con le persone con cui sei cresciuto insieme nonostante i chilometri di distanza. È un privilegio poter tornare e continuare a chiamare quel luogo "casa".

Ringrazio tutte le persone che ho incontrato lungo questo percorso a partire dal pomeriggio di quel 4 ottobre 2021, perché non è scontato piombare in un posto nuovo e trovare un luogo da chiamare "casa", delle persone da chiamare "famiglia". Ringrazio chi sta sveglio di notte, chi fa cose stupide, chi mi sopporta, chi c'è sempre, chi mi aspetta, chi mi segue e chi mi trascina, chi ci mette l'anima, chi mi ispira, chi mi fa stare bene.

Ringrazio chi mi ha permesso di vivere sino ad ora con serenità. Spero di meritare tutto ciò che ricevo.

Elenco delle figure

4.1 Spazio di indirizzamento fisico del sistema 12
4.2 Indirizzo virtuale Sv48 13
4.3 Indirizzo fisico Sv48 13
4.4 Entry di una tabella di traduzione Sv48 13
4.5 Formato del registro `satp` 14
4.6 Stato della pila al termine di `salva_stato` 22
4.7 Formato del registro `scause`, con `SXLEN` pari a 64. 27

Elenco dei codici

2.1 Installazione della toolchain 4
2.2 Installazione di QEMU 4
2.3 Debug del sistema 4
2.4 Aggiunta al PATH della cartella di installazione dei binari 4
4.1 Indirizzo fisico e dimensione della memoria (file `kernel/kernel.ld`) 8
4.2 Sezione **`.heap`** (file `kernel/kernel.ld`) 8
4.3 Funzione ctors della libCE (file `as/ctors.s`) 10
4.4 Indirizzo fisico nelle entry di traduzione (file `kernel/vm.h`) 14
4.5 Modifica della funzione `trasforma` (file `kernel/vm.cpp`) 15
4.6 Modifica della funzione `c_access` (file `kernel/vm.cpp`) 15
4.7 Funzione `writeSATP` (file `libCE/satp_write.cpp`) 16
4.8 Registro `sp` nell'handler del timer (file `machine_interrupts.s`) 17
4.9 Descrittore di processo (file `include/proc.h`) 18
4.10 Generica routine di interruzione 20
4.11 Funzione `salva_stato` (file `kernel/processi_asm.s` 20
4.12 Funzione `carica_stato` (file `kernel/processi_asm.s` 21
4.13 Funzione di creazione dei processi (file `kernel/processi_c.cpp`) 23
4.14 Inizializzazione: log sul terminale 26
4.15 Processo main sistema: log sul terminale 26
4.16 Parte C++ della routine utente (file `kernel/traps_c.cpp`) 27
4.17 Parte C++ della routine supervisore (file `kernel/traps_c.cpp`) . 28
4.18 Dichiarazione della primitiva in `sys.h` 29
4.19 Chiamata della primitiva nel codice C++ 29
4.20 Codice assembly per l'invocazione della primitiva 30
4.21 Funzione `syscall` per la chiamata alla primitiva 30
4.22 Valore di ritorno della primitiva 31
4.23 Primitive implementate (file `include/sys.h`) 31
4.24 Entry point del modulo utente (file `user/user_asm.s`) 32
4.25 Esempio di programma utente (file `user/user_c.cpp`) 33
4.26 Processi utente: log sul terminale e shutdown 33

Bibliografia

[1] Francesco Barcherini. *Repository del progetto*. https://github.com/Francesco-Barcherini/RISC-V. 2024.

[2] Andrea Bedini. *Migrazione del nucleo di un calcolatore da architettura x86 ad architettura RISC-V*. Università di Pisa - Ingegneria Informatica. 2023.

[3] Fabrice Bellard. *Official QEMU mirror*. https://github.com/qemu/qemu/tree/master/hw/riscv. 2024.

[4] RISC-V Collaboration. *GNU toolchain for RISC-V, including GCC*. https://github.com/riscv-collab/riscv-gnu-toolchain. 2024.

[5] Russ Cox, Frans Kaashoek e Robert Morris. *xv6: a simple, Unix-like teaching operating system*. MIT. 2022.

[6] Five EmbedDev. *An Embedded RISC-V Blog*. https://www.five-embed dev.com/. 2024.

[7] Edoardo Geraci. *Porting su architettura RISC-V di parte del nucleo didattico - Con particolare attenzione all'utilizzo di VGA*. Università di Pisa - Ingegneria Informatica. 2022.

[8] RISC-V International. *RISC-V ABIs Specification*. https://drive.google.com/file/d/1Ja_Tpp_5Me583CGVD-BIZMlgGBnlKU4R/view. 2022.

[9] RISC-V International. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view. 2024.

[10] RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. https://drive.google.com/file/d/17GeetSnT5wW3xNuAHI95-SI1gPGd5sJ_/view. 2024.

[11] Giuseppe Lettieri. *Materiale per l'esame di Calcolatori Elettronici*. https://calcolatori.iet.unipi.it. 2024.

[12] Chiara Panattoni. *Gestione delle interruzioni in architettura RISC-V*. Università di Pisa - Ingegneria Informatica. 2023.