Appunti sul nucleo multiprogrammato didattico (v6.5)

Gabriele Frassi

A.A 2020-2021 - Secondo semestre

Indice

La dispensa vuole porsi come strumento per conoscere in maniera approfondita il nucleo multiprogrammato didattico presentato durante il corso di Calcolatori Elettronici. Essa descrive i punti nevralgici del nucleo, cercando di approfondire in alcuni punti. Le fonti utilizzate sono:

- le lezioni dell'ultimo mese di Calcolatori elettronici;
- le dispense del professore su `https://calcolatori.iet.unipi.it/`;
- il codice del nucleo.

Ne approfitto per ringraziare due persone: Alessandro Corsi (per la pazienza che ha avuto nei miei confronti in questi mesi estivi) e Giacomo Sansone (per i suoi appunti, che mi hanno permesso di sistemare alcune parti della dispensa).

Capitolo 1

Recap orientativo del programma

- **Protezione**.
  - Origine del meccanismo. Esempio introduttivo del *batch processing*. Legame tra interruzioni e protezioni.
  - Necessità di introdurre ulteriori modifiche hardware: distinzione tra modalità sistema (priva di limiti) e modalità utente (limitata nelle istruzioni eseguibili e negli indirizzi a cui si può accedere).
  - **Implementazione del meccanismo nei processori Intel**.
    * Architettura segmentata, organizzazione della memoria proposta da Intel e mai diventata popolare. Necessario tenerla a mente per gestire il meccanismo della protezione.
    * Registro *Code Selector* e bit *Current Privilege Level* per verificare l'attuale livello di privilegio del processore.
    * Regole per i cambi di modalità: passaggio obbligato dal gate per innalzamento del privilegio, passaggio obbligato dall'istruzione IRETQ per passare a un livello di privilegio minore. Possibilità di mantenere inalterato il livello di privilegio.
    * Contenuto del gate: indirizzo di routine, flag P (implementazione della routine), *Descriptor Privilege Level* (livello minimo di privilegio per attraversare il gate), selettore di codice relativo alla *Global Descriptor Table*. Uso della *Global Descriptor Table* per ottenere il livello di privilegio dopo l'attraversamento del gate.
    * Passaggio da pila sistema a pila utente e viceversa. Necessità di memorizzare alcuni dati nella pila sistema per poter tornare alla pila utente.
    * Uso dei *Task State Segments* per ottenere l'indirizzo della pila sistema, passando dal *Task Register*.
    * Flag *IO Privilege Level* (IOPL) relativo a istruzioni IN/OUT/CLD/STD.
    * Esempio di programma: passaggio a livello utente. Programma che gestisce il primo passaggio da pila sistema a pila utente dall'avvio dell'emulatore. Necessità di preparare la pila per tale passaggio.

- **Nucleo multiprogrammato didattico: introduzione**.
  - Concetti base:
    * Il processo come sequenza degli stati attraverso cui passa il sistema eseguendo un programma. Differenza tra un processo e un programma, il contenuto di un fotogramma ideale.
    * Primitiva come funzionalità base (in un certo senso funzione di libreria) atomica (indivisibile, interruzioni esterne ed eccezioni vietate).
    * Contesto come stato del processo, mezzo per la realizzazione delle astrazioni di processo e primitiva. Cambio di contesto nel passaggio da un processo a un altro. Cambio di privilegio con l'invocazione di una primitiva.
  - Divisione del nucleo multiprogrammato didattico in moduli: modulo sistema, modulo I/O e modulo utente. Cartelle principali del nucleo.
  - Programma *Hello world!*: esempio di primitiva *writeconsole*.
  - Compilazione con comando *make*.
  - *backtrace*, ulteriore strumento per il debugging.
  - Libreria *libce*: possibilità di usare le cose viste fino ad oggi nel nucleo multiprogrammato didattico.
  - Lo stato dei processi: *esecuzione* (unico processo in esecuzione nel processo), *pronto* (processo pronto per andare avanti, ma non in esecuzione) e *bloccato* (processo che non può andare avanti, in attesa di un particolare evento).
  - Passaggio dei processi da uno stato a un altro:
    * schedulazione, passaggio da pronto a esecuzione;
    * blocco, passaggio da esecuzione a bloccato;
    * risveglio, passaggio da bloccato a esecuzione
    * preemption (o prelazione), passaggio da esecuzione a pronto.

    L'ultimo non è sempre presente, conseguenze della sua assenza in vecchi sistemi come Windows 98 (non si vedono processi di priorità maggiore finchè il processo in esecuzione non si ferma per qualche motivo).
  - Processo dal punto di vista dell'utente: parti condivise (text, data, bss, heap) e parte non condivisa (pila utente e pila sistema).
  - Processo dal punto di vista del programmatore: idea di una struttura per memorizzare le informazioni relative a un processo, differenza tra routine di sistema e routine di interruzione (indivisibilità), passaggio a contesto sistema e passaggio da un processo a un altro attraverso routine di sistema (modifica della variabile esecuzione). Perchè è necessario avere una routine atomica.
  - Esempio di processo dal punto di vista funzionale: *activate_p*. Inizializzazione della pila sistema (cosa che avviene sempre). Uso dei Task State Segments e del Task Register (unico TSS, spostamento del punt_nucleo nel TSS, si tenga conto che la cosa risulterà semplificata con la paginazione)
  - **Implementazione di processi e primitive**:
    * File *costanti.h*

* Inizializzazione della *Interrupt Descriptor Table* per gestire i gate. Inizializzazione in parte immediata e in parte rimandata ad altri momenti (si pensi alla parte relativa al modulo I/O). Uso dell'istruzione LIDT per porre l'indirizzo della tabella nel registro IDTR. Priorità massima (basata sul tipo, ricordare) data al timer. Un gate per primitiva limitante: alternativa moderna, ma non problema nel nostro nucleo (poche primitive).
* Inizializzazione della *Global Descriptor Table*. Uso diverso rispetto a quanto concepito dalla Intel, entrata per gestire il passaggio da modalità utente a modalità sistema, entrata per attraversare il gate e rimanere in modalità utente. Inizializzazione del *Task State Segment* (unico elemento, che uso per porre l'indirizzo della pila sistema).
* Processo dummy e utilità (gestire situazione dove tutti i processi sono bloccati e non c'è altro da fare). Funzioni di utilità per creare il processo dummy e per terminare il tutto nel caso in cui i processi risultino tutti terminati (si provoca la tripla eccezione).
* Descrittore di processo, array di processi, variabile per conteggiare il numero di processi attivi.
* Funzioni per la manipolazione delle liste: inserimento di un descrittore di processo in una lista sulla base delle priorità (con logica FIFO a parità di precedenza), rimozione della testa di una lista, funzione *inspronti* per l'inserimento di un processo in esecuzione nella lista pronti, funzione schedulatore per porre nella variabile esecuzione la testa della lista pronti. Lo schedulatore non comporta un cambio di processo immediato (quello si avrà con la carica_stato, tenerlo a mente).
* Funzione *salva_stato* per la memorizzazione dello stato del processo nel relativo descrittore di processo: aggiornamento di tutti i registri nell'apposito array del descrittore.
* Funzione *carica_stato* per il caricamento dello stato di un processo (posto in un descrittore di processo) nel processore. Eventuale cambio di processo qualora il valore di esecuzione sia stato modificato nella routine di sistema. Aggiornamento di CR3 con JMP condizionato (per evitare invalidamenti inutili della TLB), aggiornamento di tutti i registri a partire dall'array nel descrittore di processo. Distruzione della pila sistema del processo precedente nel caso in cui il processo sia stato terminato (se terminiamo un processo la pila sistema relativa al processo non ci serve più).
* Funzione di utilità *liv_chiamante* per la verifica del livello di privilegio del processo chiamante. Si osservi che la cosa funziona solo se si è chiamato precedentemente la *salva_stato*. Si recupera il valore dalla pila sistema, precisamente dal vecchio Code Selector.
* Esempio di primitiva dal punto di vista implementativo: *activate_p*. Verifica di alcune condizioni relative ai parametri interni, uso della funzione di utilità *crea_processo*.
* La nostra prima primitiva: *getprec*, che restituisce il livello di privilegio del processo in esecuzione. Parte Assembler e parte C++, uso dell'array contesto nel descrittore di processo per valori restituiti dalle primitive. Funzione di appog-

gio, in Assembler, che usa l'istruzione INT e permette il passaggio dal gate per eseguire la primitiva.

- **Nucleo multiprogrammato didattico: semafori**.
  - Problema della mutua esclusione: evitare che azioni vengano svolte in contemporanea da più parti.
  - Problema di sincronizzazione: fare in modo che un'azione si manifesti sempre prima di un'altra.
  - Primitive semaforiche come soluzione ai due problemi: idea delle scatole con gettoni, cosa succede quando levo un gettone e lo rilascio, perchè si parla di *soluzione cooperativa*.
  - Esempio del buffer con produttore che inserisce un contenuto e consumatore che lo utilizza.
  - **Implementazione**:
    * Descrittore di semaforo *des_sem* e array dei semafori *array_dess*. Divisione dell'array in due metà: la prima per i processi di livello utente e la seconda per i processi di livello sistema.
    * Funzione di utilità *sem_valido* per la verifica dell'esistenza di un semaforo (secondo i propri livelli di privilegio). Variabili contatore per il numero di semafori allocati nella prima metà e per il numero di semafori allocati nella seconda metà.
    * Primitiva *sem_ini* per l'allocazione del semaforo (con funzione di utilità *alloca_sem*).
    * Primitiva *sem_wait* per la presa del gettone. Uso del counter nel descrittore semaforico per capire se il processo deve essere posto in attesa (quindi inserito nella coda del descrittore).
    * Primitiva *sem_signal* per il rilascio del gettone. Uso del counter nel descrittore semaforico per capire se ci sono processi da risvegliare (presenti nella coda del descrittore).

- **Nucleo multiprogrammato didattico: paginazione**.
  - **Premesse**.
    * Indirizzi relativi a periferiche I/O posti nello spazio di memoria. Necessità di gestire la cosa relativamente alla cache (disattivarla o gestire gli indirizzi con politica *write-through*).
    * Meccanismo di protezione: garantire l'isolamento tra $M1$ ed $M2$ (cioè i processi utenti non possono accedere ad $M1$ e a tutto ciò che non li riguarda in $M2$). Divieto di utilizzo dell'indirizzo 0, usato come *nullptr*.
    * Gestione della multiprogrammazione: necessaria la memorizzazione del contesto. Pignoleria richiederebbe memorizzazione della RAM. Noi non facciamo questo: poniamo nella RAM informazioni relative a più processi, prese dall'hard disk.
    * Problemi conseguenti: determinare la dimensione massima del processo (si ha una parte di cui è possibile determinare la dimensione a priori e una parte che varia runtime, lo stack - si risolve ponendo una dimensione massima); garantire che un processo non acceda a informazioni di altri processi (idea temporanea dei registri LINF ed LSUP); questione degli indirizzi variabili (cioè non possiamo

sapere gli indirizzi assoluti a priori se la posizione del processo nella RAM è incerta).
    * Introduzione degli indirizzi virtuali per risolvere l'ultimo problema: l'utente indica solo gli offset, e si ottiene l'indirizzo virtuale sommando LINF ed offset (soluzione temporanea).
    * Scopo del Kernel: garantire un uso controllato dell'hardware ai processi in esecuzione.
    * Problemi finali che ci portano a parlare di paginazione: necessità di condividere informazioni tra processi, necessità di gestire in modo oculato lo spazio di memoria della RAM (e quindi necessità in alcuni casi di spezzare le info di un processo in più parti sparse per la RAM).
- Introduzione alla paginazione. Distinzione di uno spazio di indirizzamento virtuale da uno spazio di indirizzamento fisico. Divisione dello spazio di indirizzamento virtuale in regioni da 4 KiB dette pagine. Divisione dello spazio di indirizzamento fisico in regioni da 4 KiB dette frame. Perchè si divide in regioni piccole.
- Memory Management Unit per la traduzione degli indirizzi virtuali in indirizzi fisici. Dove viene collocata nell'architettura, quali componenti ragionano in termini di indirizzi fisici e in quali in termini di indirizzi virtuali. Tabella di corrispondenza (una per processo) dove ponendo il numero di pagina si ottiene il corrispondente numero di frame. Contenuto delle tabelle di corrispondenza: numero di frame, bit P per la validità dell'entrata, bit U/S per la validità della traduzione a livello utente, bit R/W per le operazioni di scrittura, bit PWT per la politica Page Write Through nella cache, bit PCD per la disattivazione della cache relativamente all'indirizzo. Bit A e bit D che useremo a sistemi operativi per la paginazione su domanda.
- Necessità di mappare l'intero modulo $M1$ nella MMU: questo per gestire i cambi di processo nelle routine di sistema. Necessità per lo sviluppatore di sistema di distinguere indirizzi virtuali da indirizzi fisici. L'utente utilizza solo indirizzi virtuali non rendendosi conto dell'esistenza di indirizzi fisici.
- Esempio di programma ed MMU all'opera: diapositive del professore.
- Introduzione alla struttura dati della tabella di corrispondenza: introduzione al *bitwise trie*. Tabella di corrispondenza troppo grande, necessità di introdurre una struttura dati più flessibile. Archi dell'albero rappresentano terne di bit relativamente all'indirizzo virtuale (dai più significativi ai meno significativi), la foglia contiene l'indirizzo fisico, si hanno al più quattro livelli. Ogni nodo è una tabella di 512 entrate, avente dimensione di 4096 byte. Registro CR3 contiene l'indirizzo all'albero utilizzato dal processo (si ricordi la variabile cr3 posta nel descrittore di processo). Dimensioni a confronto: apparentemente più peso, in realtà più leggero grazie alla traduzione non completa degli indirizzi e alla condivisione di sottoalberi tra *trie* (si pensi al modulo $M1$, la traduzione è uguale in tutti i *trie* possibili). Bit presenti in ogni entrata (più o meno gli stessi già detti), convenzione sull'uso dei bit R/W e U/S (bit uguali per tutto il percorso se si vuole scrittura e/o accesso da parte dell'utente).
- Scoperta: fino ad ora abbiamo sempre lavorato con la paginazione attiva. Processore parte con modalità di compatibilità: inizialmente a 16 bit, passa a 32 bit con l'attivazione della protezione, passa a 64 bit con l'attivazione della paginazione. Indirizzi posti nel trie sono indirizzi virtuali: perchè deve essere così.

- Introduzione ai registri CR0 (attivazione della paginazione), CR2 (che nel caso di page fault, errore di traduzione nella MMU, contiene l'indirizzo virtuale che si è cercato di tradurre), CR3 che contiene l'indirizzo al *trie* considerato (precisamente il numero di frame, i nodi del trie richiedono un allineamento a 4096).
- Eccezione *page fault*.
- Introduzione al bit *Page Size Flag*, bit che permette di definire nel *trie* pagine di dimensioni superiori. Posto a 1 alla fine del percorso del *trie* (pertanto non devo percorrere, per forza, quattro livelli).
- Introduzione alla Transmit Lookaside Buffer, memoria cache dedicata esclusivamente alla MMU. Struttura identica a quella già vista quando abbiamo parlato di memorie cache, bit presenti o meno all'interno delle tabelle delle etichette. Invalidazione delle entrate della MMU: completa aggiornando il registro CR3, parziale utilizzando l'istruzione Assembler invlpg (che chiede un operando). Uso in C++ della funzione *invalida_entrata_TLB*, che chiama la invlpg.
- **Implementazione**.
  - *typedef* per distinguere indirizzi fisici da indirizzi virtuali (*paddr* e *vaddr*), *typedef* per identificare entrate di una tabella del trie (*tab_entry*).
  - Maschere per gestire il contenuto di una *tab_entry*: bit presenti nell'entrata, maschera per prendere solo la parte relativa all'indirizzo (ricordare l'allineamento di 4096), maschera per prendere solo la parte relativa ai bit di configurazione.
  - Funzione *norm* per normalizzare un indirizzo (ricordare la regola che abbiamo introdotto per il buco).
  - Funzione *dim_region* che restituisce la dimensione di una regione dato un livello: 0 è 4096 byte, 1 è 2 MiB, 3 è 1 GiB.
  - Funzione *base* per trovare, dato un indirizzo e un livello, l'indirizzo della base della relativa regione (restituisce un vaddr).
  - Funzione *limit* per trovare, dato un indirizzo e un livello, la base della prima regione di livello indicato a destra dell'intervallo in cui ci si trova (rispetto all'indirizzo indicato).
  - Funzione *extr_IND_FISICO* per estrarre da una entrata di tabella l'indirizzo fisico (ricordare l'allineamento delle tabelle rispetto a 4096).
  - Funzione *set_IND_FISICO* per impostare un nuovo indirizzo fisico nell'entrata di tabella indicata.
  - Funzione *get_entry* per ottenere l'indirizzo fisico di una entrata di tabella a partire dall'indirizzo della base della tabella stessa e un indice.
  - Classe *tab_iter*: iteratore per scorrere il *trie*.
  - Suddivisione dello spazio di indirizzamento: modulo sistema, modulo I/O e pila sistema. Modulo $M2$: text, data e heap (parte condivisa) e stack (parte privata).
  - Costanti dedicate alla memoria virtuale, con cui indichiamo il numero di pagine riservate a ciascuna sezione dello spazio di indirizzamento virtuale.
  - Costanti dedicate alla memoria fisica, con cui indichiamo la dimensione totale di un ogni sezione della memoria fisica.
  - Indirizzi virtuali e fisici nel file ELF.

    - Gestione dei frame: descrittore di frame *des_frame* (uso della union per minimizzare lo spazio, numero di entrate valide e lista di frame liberi), array di descrittori, variabile con numero di frame, variabile *primo_frame_libero*, variabile con numero di frame in $M1$ e numero di frame in $M2$. Funzioni di utilità per gestire il contatore *nvalide* nel descrittore di frame. Necessità di gestire un meccanismo che renda disponibile al volo un qualunque frame all'interno della memoria RAM: funzioni di utilità *alloca_frame* e *rilascia_frame*.
    - Gestione delle tabelle del trie nella MMU: funzione *alloca_tab* (restituisce il relativo indirizzo fisico) e funzione *rilascia_tab*. Funzioni per gestire settaggio e copia: *set_entry* per impostare una entrata di tabella (dato l'indirizzo di base di una tabella, un indice e il contenuto di un'entrata), *copy_des* per copiare n entrate da una tabella di indirizzo *src* a una tabella di indirizzo *dst* (a partire dall'elemento di indice i), *set_des* per impostare n descrittori allo stesso modo in una tabella (a partire dall'elemento di indice i).
    - Funzioni *map* e *unmap* per la gestione della MMU.

- **Nucleo multiprogrammato didattico: periferiche**.
  - Recupero dell'esempio del *batch processing*: job equivalente a processo, necessità di fare operazioni di I/O e porsi in attesa finchè la periferica non conclude l'operazione.
  - Necessità di limitare l'utente: niente accesso diretto alle periferiche, ma accesso controllato attraverso primitive da noi definite.
  - Schema delle possibili primitive di I/O: IN read e OUT write.
  - Necessità di più interruzioni in alcuni casi, soprattutto quando richiediamo più di un byte e la periferica non è in grado di inviarci più di un byte alla volta.
  - Problema di mutua sincronizzazione di sincronizzazione, applicati al caso delle periferiche. Uso dei semafori per gestire la situazione e rottura dell'atomicità delle primitive.
  - Primitive per avviare l'operazione di I/O e bloccare il processo
  - Driver per gestire il trasferimento dei byte e sbloccare il processo a operazione conclusa.
  - Prototipo di descrittore di periferica e array di descrittori. Implementazione in Assembler e in C++ della primitiva.
  - Implementazione in Assembler e in C++ del driver.
  - Il problema del cavallo di Troia: risoluzione con la primitiva *access*.

- **Nucleo multiprogrammato didattico: inizializzazione**.
  - Funzione *start* implementata in Assembler: recupero del puntatore alla struttura dati *multiboot_info*, esecuzione di costruttori relativi ad oggetti globali attraverso un ciclo. Chiamata della funzione *main* implementata in C++ nel modulo sistema.
  - **Nella funzione main**.
    - Inizializzazione di Global Descriptor Table con la chiamata della funzione *init_gdt* implementata in Assembler (e vista precedentemente).

    * Inizializzazione dell'APIC con le funzioni già viste nella libreria *libce*. Associazione del piedino 2 al timer. Più avanti inizializzazione effettiva del timer.
    * Modulo $M1$ già caricato dal bootloader, inizializzazione delle strutture dati relative alla gestione dei frame.
    * Definizione di costanti (indirizzi virtuali *vaddr*) con cui rappresentiamo i limiti delle varie aree da noi raggiungibili attraverso lo spazio di indirizzamento virtuale.
    * Creazione della cosiddetta *finestra* con la funzione *map*: attenzione ai primi step, si ignora il primo frame per non usare l'indirizzo 0 e inizialmente si mappa una piccola regione per gestire memoria video (politica page write through) e apic (politica page cache disable).
    * Creazione dello spazio condiviso utilizzando la struttura dati *multiboot_info*, si ricorre a informazioni dettagliate del file ELF (non affrontato in modo profondo). Inizializzazione della MMU e caricamento del relativo indirizzo nel registro CR3.
    * Inizializzazione dello heap.
    * Creazione di un processo sistema che useremo per concludere alcune inizializzazioni. Approfondimento sulla funzione *crea_processo*.
    * Creazione del processo dummy e chiamata della funzione schedulatore per passare al processo sistema creato poco prima. Passaggio al processo con la funzione *salta_a_main*, che è implementata in Assembler e consiste semplicemente in una chiamata di *carica_stato* e IRETQ.
- **Nella funzione *main_sistema*.**
    * Inizializzazione effettiva del timer (smascheriamo il piedino) e chiamiamo una funzione *avvia_timer*. Ricordarsi che il timer, contrariamente ad altre periferiche, è implementato direttamente nel modulo sistema.
    * Inizializzazione delle periferiche: creazione di un apposito processo con livello di privilegio sistema (con cui eseguiremo la funzione *main* implementata nel modulo I/O), chiamata della *sem_wait* su un semaforo inizializzato senza gettoni, che ci obbliga a passare al processo precedentemente creato. La *sem_signal* sarà chiamata in questo processo e permetterà di riprendere l'esecuzione della *main_sistema*.
    * Creazione del primo processo con livello di privilegio utente (si esegue la funzione *main* implementata dall'utente nel modulo utente).
- **Nel modulo I/O: funzione main e cose collegate**.
    * File header con le intestazioni delle varie primitive. Attraverso questi file si distinguono quali primitive possono essere usate in certi moduli e quali no.
    * Inizializzazione della parte di Interrupt Descriptor Table relativa al modulo I/O.
    * Funzione *main* che utilizziamo per inizializzare tutto ciò che va inizializzato: inizializzazione dello heap, ridefinizione di operatori uguali a prima, ma con chiamate di primitive di sistema (non si ha l'atomicità nel modulo I/O); inizializzazione della console con la chiamata di *console_init*, inizializzazione dell'hard disk con la chiamata di *hd_init*.
    * Chiamata della *sem_signal* per mettere un gettone nel semaforo che abbiamo toccato alla fine della funzione *main_sistema* del modulo sistema. Terminazione del processo e ritorno nella funzione *main_sistema* del modulo sistema.

Capitolo 2

Protezione

2.1 Introduzione

Premessa

Il meccanismo di protezione non nasce per difendersi dagli hacker.

Come nasce l'idea La necessità della protezione nasce dall'utilizzo dei grandi calcolatori degli anni 60 per eseguire più programmi contemporaneamente.

1. L'utente richiede l'utilizzo del calcolatore lasciando il suo codice.
2. Gli operatori inseriscono il codice nel calcolatore (pensare alla *Calcolatrice Elettronica Pisana*) e lo eseguono.
3. L'utente ritira dopo un po' di tempo l'output del programma lasciato.

Abbiamo un *batch processing*. Devo caricare programmi complessi, con molti dati che potrebbero provocare tempi di attesi lunghi prima dell'esecuzione: l'idea è che mentre attendo passo a occuparmi di un altro programma da eseguire. Chiaramente posso fare una cosa di questo tipo solo col meccanismo delle interruzioni: al termine del caricamento dei dati lancio un'interruzione.

Quanto fatto fino ad ora non basta Le istruzioni che abbiamo visto fino ad ora (relative alle interruzioni) possono essere eseguite senza problemi, dunque un programma potrebbe interferire con il meccanismo delle interruzioni. Il programma può dare fastidio in vari modi (per esempio con istruzioni mov che modificano il codice del programma).

Soluzione La soluzione al problema è di tipo hardware, non possiamo fare diversamente. Dobbiamo distinguere due modalità per il processore:

- **modalità sistema**, destinata al codice dell'operatore (nessun limite di esecuzione);
- **modalità utente**, destinata al codice degli utenti (non è ammessa l'esecuzione di tutte le istruzioni, o l'esecuzione di operazioni di scrittura/lettura in certe aree di memoria).

Dobbiamo prevedere meccanismi che permettono il passaggio da una modalità all'altra. In alcuni casi vengono implementate più modalità: avere più livelli di privilegio è una questione di efficienza, cose che non ci interessano.

Memoria Nella memoria di sistema abbiamo il codice stabilito dagli operatori, ma anche i codici dei vari *job* da eseguire. Ogni tanto si salta al codice degli operatori:

1. interruzione esterna,
2. eccezione,
3. l'utente ha esplicitamente invocato una routine del sistema.

In tutti e tre i casi vogliamo che il processore si porti in modalità sistema. La cosa più semplice è associare anche il terzo elemento alle interruzioni. Possiamo invocare una routine del sistema con la seguente istruzione

```
INT $tipo
```

L'innalzamento del livello di privilegio non può che passare dall'IDT.

Codice del kernel Ci dovrebbe preoccupare che il codice dell'operatore sia, adesso lo possiamo dire, il codice del kernel?

- L'area di memoria con questo codice NON deve essere accessibile a programmi eseguiti in modalità utente. Gli utenti, come già anticipato, **NON** devono avere libero accesso a tutta la memoria!

Modalità utente In una modalità privilegiata esistono limiti su:

- istruzioni eseguibili (si distinguono quelle eseguibili, *privilegiate*, da quelle eseguibili, *non privilegiate*);
- indirizzi a cui si può accedere (operazioni di lettura e scrittura).

Il processore in modalità utente non esegue istruzioni privilegiate e non accede a indirizzi dove non può accedere: se gli si chiede di fare queste cose genera un'*eccezione di protezione*. Queste cose limitano l'utente su tutti e tre gli elementi fondamentali dell'architettura di un elaboratore: CPU (limitano il controllo della CPU impedendo l'esecuzione di particolari istruzioni), la RAM e l'I/O (non tutti gli indirizzi sono accessibili).

Motivazioni Abbiamo già detto che la questione non è proteggersi dagli hacker, ma proteggere un utente da un altro, impedire a un utente di proseguire il suo lavoro a causa di un altro utente (*la polizia, che protegge i cittadini da altri cittadini*, cit.).

Regola base per ogni cosa che gestiamo Quando siamo in modalità sistema non possiamo fidarci di ciò che potrebbe essere stato gestito in modalità utente. Dobbiamo pensare all'utente come una persona che farà tutto in suo potere per sovvertire il sistema.

Modalità del processore all'avvio All'avvio del sistema siamo in modalità sistema. Ecco perché in QEMU siamo liberi di fare ciò che ci pare!

2.2 Protezione nel processore Intel

L'Intel ha introdotto la protezione nel 286 (a 16 bit), un sistema derivato da *multics* (*Multiplexer information and communication system*), un sistema operativo sviluppato da MIT e *Bell Labs*. Questo sistema operativo possedeva diversi livelli di privilegio: l'Intel introdusse un architettura segmentata, meccanismo di cui dobbiamo tenere conto.

Cosa si intende con architettura segmentata? Divido la memoria in più segmenti (codice, dati). Ogni volta che si deve riferire un certo byte bisogna specificare sia il segmento che ci interessa che l'offset all'interno del segmento. La cosa, su cui Intel ha insistito, non è risultata popolare: l'architettura a cui noi siamo abituati è un architettura *flat*.

Approfondiamo i meccanismi Il meccanismo dell'Intel è abbastanza elaborato: fa tante cose, tutte importanti (dobbiamo imparare un po' a memoria, cit.). Vediamo più nel dettaglio...

2.2.1 Livello di privilegio attuale del processore (*Current Privilege Level*)

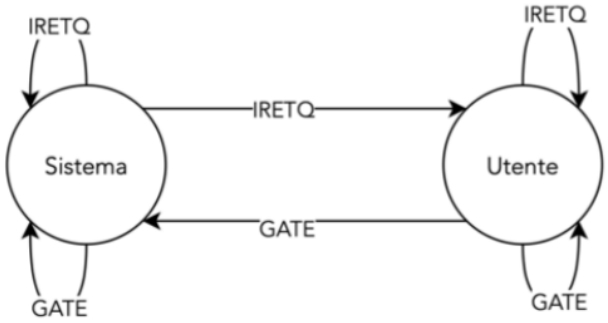
All'interno del processore Intel è presente un registro CS (*Code Selector*) che contiene negli ultimi due bit il livello di privilegio e l'identificatore del segmento codice corrente. La segmentazione non ci interessa, dunque guarderemo solo i primi due bit (che definiamo CPL, *Current Privilege Level*). I valori possibili sono

- 00 (sistema), e
- 01 (utente).

2.2.2 Protezione e interruzioni, cambi di modalità (gate e IRETQ)

L'altra cosa su cui si basa l'architettura Intel è l'associare tutti i possibili cambi di livelli di privilegio al meccanismo delle interruzioni.

- Il passaggio dal gate può avvenire solo attraverso un'interruzione esterna, un'interruzione interna o l'esecuzione dell'istruzione INT. Il livello di privilegio successivo al passaggio del gate deve essere maggiore o uguale: non è possibile che un calcolatore in modalità sistema si ritrovi in modalità utente dopo aver attraversato il gate, al più si ritrova nella stessa modalità.
- L'istruzione IRETQ confronta il vecchio CPL col nuovo e permette solo passaggi con livello di privilegio minore o uguale rispetto a quello corrente: sono possibili passaggi da modalità sistema a modalità utente, ma non il contrario.



2.2.2.1 Contenuto del gate e *Global Descriptor Table*

- All'interruzione abbiamo sempre associato un tipo: in alcuni casi è stabilito dall'APIC (interruzioni esterne), in altri dall'operando di una istruzione (istruzione INT), in altri ancora è implicito (eccezioni).
- Il tipo permette di identificare un'entrata dell'IDT. Ciascuna entrata contiene delle informazioni. Quelle che ci interessano maggiormente sono:
  - l'indirizzo della routine da mandare in esecuzione (nulla di nuovo);
  - un bit I/T che indica il tipo del gate (*interrupt* o *trap*, nulla di nuovo);
  - un flag P, che specifica se la routine è stata implementata o no;
    - Logicamente non saranno implementate tutte le 256 routine possibili.
    - Se $P = 0$ il processore non può attraversare il *gate* e genera una nuova eccezione di tipo *gate non implementato*. I programmatori di sistema solitamente implementano questa eccezione: se ciò non avviene il processore subisce una doppia eccezione (*abort*) e si spegne.
  - il *Descriptor Privilege Level*, che indica il livello di privilegio minimo che il processore deve avere prima di attraversare un certo gate;
    - Notare che **con un'interruzione esterna e un'eccezione il gate viene attraversato comunque**, la limitazione sta solo nell'uso dell'istruzione INT. La cosa è utile per prevenire l'esecuzione attraverso INT di routine normalmente associate a interruzioni esterne e/o eccezioni.
  - Un selettore di codice relativo a un'entrata della *Global Descriptor Table*, quella del segmento dove si trova la routine.

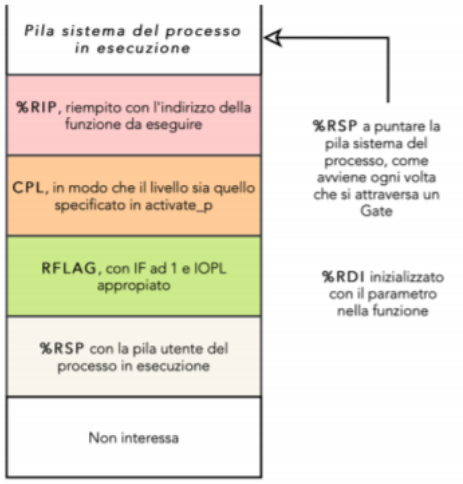
A cosa ci serve il selettore di codice? Per trovare il livello di privilegio dopo aver attraversato il gate. Detto in altro modo: cosa dobbiamo scrivere nel *Current Privilege Level* dopo aver attraversato il gate?

- L'informazione non si trova nella tabella delle interruzioni ma nella *Global Descriptor Table* (GDT), tabella legata al meccanismo della segmentazione. La tabella, dato il segmento codice dove si trova la routine, indica una serie di informazioni:
  - da che indirizzo parte;
  - quanto è grande il segmento;
  - **il livello di privilegio**!
- Il meccansimo della Intel è flessibile: si da la possibilità di decidere se innalzare o no il livello di privilegio: ciò per noi è una semplice constatazione, innalzeremo sempre il livello di privilegio.

2.2.2.2 Necessità di una nuova pila

Alcune informazioni vengono memorizzate in pila. Non possiamo utilizzare in modalità sistema il contenuto di una pila manipolato dall'utente (non possiamo fidarci dell'utente, MAI). Segue la necessità di fare un cambio di pila qualora vi sia un cambio di privilegio.

- Distinguiamo due pile: la *pila utente* e la *pila sistema*.
- **Se il livello di privilegio si innalza il processore passa da pila utente a pila sistema** (si confronta il *Current Privilege Level* col nuovo livello di privilegio trovato nella *Global Descriptor Table*). Cambiare pila significa modificare il valore del registro RSP. Nella pila sistema andiamo a porre delle informazioni:



  1. il vecchio RIP.
  2. il vecchio registro CS (in particolare il vecchio CPL);
  3. i vecchi flag (RFLAGS);
  4. il vecchio RSP (cioè dov'era RSP prima di fare il passaggio);
  5. il selettore pila vecchio (che non ci interessa poichè legato alla segmentazione);

  queste informazioni rendono possibile il ritorno nella pila utente.

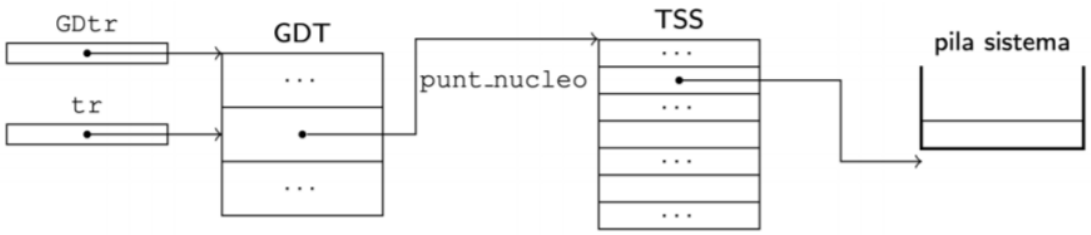
- Salvate tutte queste informazioni il processore preleva dal gate il nuovo RIP e il nuovo CS (in particolare il nuovo CPL).
- La IRETQ confronta il *Current Privilege Level* e il CPL preso dal vecchio valore di CS in pila. Per prima cosa si verifica che il vecchio valore sia al più uguale (e non maggiore) del CPL attuale (controllo necessario per impedire violazioni del meccanismo della protezione[1]). Successivamente si verifica se è necessario riportare in RSP la pila utente (confronto tre *Current Privilege Level* e il CPL preso dal vecchio valore di CS in pila). La IRETQ ripristina ciò che va ripristinato recuperando le informazioni dalla pila corrente (quella puntata con RSP).

[1] L'utente potrebbe fare casini modificando RSP prima di lanciare la IRETQ.

Osservazione *pushf* e *popf* non permettono l'alterazione di tutti i flag. Abbiamo visto che per modificare il trap flag utilizziamo una funzione con istruzioni *pushf* e *popf*. Quindi è possibile modificare qualunque flag in questo modo? No, *iretq* e *popf* non permettono la sovrascrizione di tutti i flag: contrariamente ad altre situazioni non si ha un'eccezione, semplicemente si ignorano le richieste di modifica senza effetti visivi.

2.2.2.3 Da dove viene preso il puntatore alla nuova pila?

L'Intel nel 286 prevedeva la possibilità di gestire la microprogrammazione quasi completamente in hardware (anche questa cosa sostanzialmente non usata). Dai GDT è possibile raggiungere i *Task State Segment* (TSS). Questi segmenti, previsti di suo dal processore Intel e identificati da una certa entrata della GDT, contengono informazioni relative a un particolare programma in esecuzione. Inoltre si ha un registro *Task Register* (TR) contenente l'indice del *task/job* corrente. Nel TSS si hanno tante informazioni, tra queste **il puntatore alla base della pila sistema**.



Quando c'è un'interruzione il processore usa TR per accedere al segmento TSS a partire dalla GDT, trovando così l'indirizzo della pila sistema.

2.2.3 Livello di privilegio e istruzioni IN/OUT/CLD/STD (flag IOPL)

Le istruzioni IN/OUT/CLD/STD non sono automaticamente vietate a livello utente. Esiste un altro campo, nel registro dei flag, detto *IO Privilege Level* (IOPL), che per noi assume come valore

- 00 (sistema) o
- 11 (utente).

Con questo definiamo il livello privilegio che dobbiamo avere per eseguire le istruzioni IN/OUT/CLD/STD. L'informazione è statica e rimane lì anche dopo passaggi di gate. Le modifiche al campo sono ignorate nell'esecuzione di POPF e IRETQ.

2.2.4 Morale della favola

Per far funzionare le interruzioni abbiamo bisogno di:

- almeno un segmento TSS (ci serve un solo TSS per gestire i passaggi tra due livelli di privilegio),
- almeno un descrittore nella GDT,
- e il registro TR inizializzato in modo da contenere l'identificativo di questo descrittore.

Fortunatamente possiamo fare questa cosa all'avvio.

2.3 Esempio di programma: passaggio a livello utente (*liv_utente*)

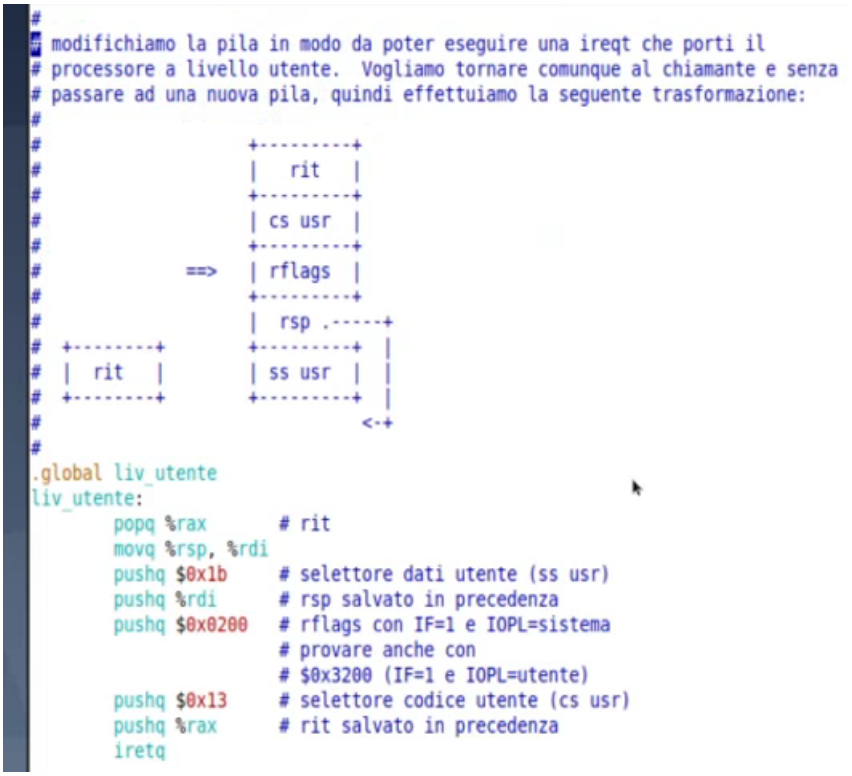
Perchè ci interessa questa cosa? Il nostro programma parte in modalità sistema, a un certo punto vogliamo passare in modalità utente. La IRETQ da sola non basta, dobbiamo impostare la pila nei modi detti prima per compiere il cambio di privilegio.

Esempio Supponiamo di voler eseguire la seguente funzione due volte: la prima volta in modalità sistema e la seconda in modalità utente.

```
void foo() {
    natb c;
    printf("Provo a leggere RBR\n");
    c = inputb(0x60);
    printf("Ho letto RBR: %2x\n", c);
}
```

Gestiamo il passaggio a livello utente attraverso la funzione *liv_utente*, che poniamo nel nostro codice tra le due chiamate di *foo*.

```
foo();
liv_utente();
foo();
```



La funzione si bassa sulla necessità di passare dalla IRETQ per effettuare un passaggio da modalità sistema a modalità utente. Sappiamo che la IRETQ recupera del contenuto dalla pila e lo utilizza: segue che per utilizzare la IRETQ dovremo mettere in pila ciò che la funzione si aspetta. Le cose sono quelle già elencate prima

1. Tolgo l'indirizzo di ritorno con la pop
2. Pongo il selettore dati utente (il vecchio contenuto del registro SS, non ci interessa)
3. Pongo l'RSP
4. Pongo il registro dei flag aggiornato
5. Pongo il selettore codice utente (il contenuto del registro CS, ci interessa il CPL)
6. Pongo l'indirizzo di ritorno rimosso all'inizio
7. Chiamo la IRETQ

```
INF     -       entry point 00200120
WRN     -       Eccezione 13, err=0000000000000000, EIP=0000000000200cb8
studenti@debian-ce-2:~/tmp/prot2$
```

A questo punto ogni istruzione che richiede un livello di privilegio maggiore rispetto a quello corrente provocherà il lancio di un'eccezione di tipo 13 (*eccezione di protezione*). L'eccezione è di tipo *fault*, dunque l'indirizzo stampato del warming è quello dell'istruzione che ha causato il problema (la IN).

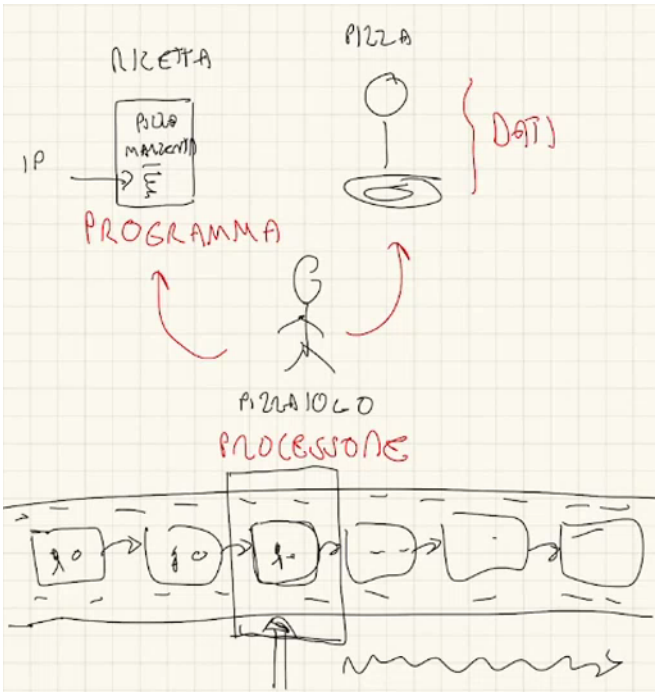
Parte I

Nucleo multiprogrammato didattico

Capitolo 3

Introduzione

Con nucleo multiprogrammato intendiamo un sistema in grado di svolgere più *processi*.



Il sistema che creiamo fornisce ai suoi utenti due astrazioni: *processi* e *primitiva*. Il meccanismo che realizza queste due astrazioni è detto *contesto*.

3.1 Concetti base

3.1.1 Processo

Il processo è un programma in esecuzione.

$$\text{Processo} \neq \text{Programma}$$

Pensiamo al pizzaiolo inesperto che sta facendo la pizza: ha bisogno di una ricetta, perché non si ricorda come si fa.

- La ricetta è il programma, che posso utilizzare per creare le pizze (anche in contemporanea).
- Il pizzaiolo è il processore, la pizza i dati (l'input è la richiesta del cliente).

Nessuna di queste cose è il processo. Il processo possiamo immaginarlo come il filmato di tutto ciò che accade in questo sistema (*pizza + pizzaiolo*), dall'inizio alla fine (il filmato di tutti gli stati attraverso cui il sistema passa).

Usciamo dalla metafora Il processo è la sequenza degli stati attraverso cui passa il sistema eseguendo il programma. L'esecuzione di ogni istruzione del programma fa passare il processo da uno stato al successivo.

Programma vs Processo Può essere facile confondere il processo col programma.

- Un programma può essere eseguito da più processi "contemporaneamente". I clienti chiedono la pizza margherita, il pizzaiolo ne fa tante, ognuna di queste segue il suo processo indipendentemente dalle altre.
- Un processo può eseguire programmi diversi. Si pensi alla *pizza al metro*, con vari tipi di pizza uno dietro l'altro.

Fotogramma

- Concentriamoci sul fotogramma: pur avendo una sequenza di stati possiamo concentrarci su uno stato in un particolare istante. **Per la natura dei nostri processi dato un qualunque fotogramma avrò le informazioni necessarie per proseguire il processo** (di ogni pizza il pizzaiolo si ricorda dove è arrivato).
- Passare da un processo a un altro significa ricordarsi il fotogramma dove è arrivato il processo da cui stiamo saltando via, caricarlo e spostarci sul fotogramma del nuovo processo che vogliamo portare avanti.

Il processo è una cosa sequenziale. **Non possiamo, tuttavia, definire il processo come l'istanza di un programma**: un processo, ricordiamolo, può cambiare programma.

- Il programma non è la sceneggiatura del processo: la ricetta può contenere delle varianti (quattro stagioni con o senza carciofi...). Il processo di produzione di una singola pizza riguarda una sola delle varianti possibili.
- Il programma potrebbe contenere dei cicli. Nel programma vediamo le azioni ripetute una volta sola, mentre nel processo vediamo le azioni effettivamente ripetute tante volte.

3.1.2 Primitiva

Cosa intendiamo con *primitivo* Primitivo è un qualcosa di *non derivato*, un mattone fornito dal sistema per costruire il resto delle cose. L'esempio sono le funzionalità di base fornite da un programma di grafica. Ricordarsi la caratteristica fondamentale delle primitive:

La primitiva non è scomponibile.

Funzioni di libreria? Si potrebbe dire che le primitive sono funzioni di libreria non scomponibili (di solito lo sono).

Tipi primitivi I tipi primitivi di un linguaggio (int, long, char...) sono offerti dal linguaggio, possono essere usati per definire nuovi tipi derivati, ma non possono essere modificati nel comportamento.

Cosa faremo a breve Creeremo un sistema che fornisce una stazione dei processi, esegue programmi all'interno dei processi, fornisce delle primitive che sono le funzionalità di base presenti all'interno di questi programmi.

- Gli utenti possono lanciare le primitive e usufruire dei servizi da esse offerte.
- Normalmente le primitive di un sistema sono pensate come chiamate di sistema, dunque con privilegio maggiore. Le primitive molto spesso reggono il loro funzionamento su delle strutture dati: queste strutture dati non devono essere manipolabili dall'utente, o al più manipolabili in modo controllato (non raggiungibili dal livello utente se non attraverso una primitiva).
- Gli utenti non possono manipolare le primitive esistenti, ma soprattutto non possono crearne di nuove.

3.1.3 Contesto

Con la parola contesto si intende, nel linguaggio comune, tutto ciò che è necessario sapere per interpretare correttamente un dato testo, ma che non è scritto esplicitamente nel testo stesso. Prendiamo ad esempio la seguente istruzione C++:

```
x = x + 1
```

abbiamo una somma, ma non sappiamo in cosa consista x.

E nel nostro caso? Il contesto permette la realizzazione delle astrazioni *processo* e *primitiva*. Se io ho più processi che vanno avanti, anche processi che eseguono lo stesso programma, avrò per ciascuno uno stato. (i registri relativi a un processo potrebbero non essere i registri relativi a un altro processo). **In un certo senso il *contesto* consiste nello stato del processo**. Associo un contesto diverso ad ogni processo (nel nucleo faremo ciò con un *descrittore di processo*).

Cambio di contesto Quando si passa da un processo a un altro si dice che *si cambia contesto*. Il processore, chiaramente, può dedicarsi a un solo processo alla volta: l'unico modo per avere un'esecuzione apparentemente concorrente dei processi è eseguire in modo frammentato i processi (eseguo un pezzo di un processo, cambio contesto, eseguo un pezzo di un altro processo, cambio nuovamente contesto, e così via...)

Contesto e primitive Questa idea di contesto è valida anche per le primitive: supponiamo di avere un'istruzione HLT, è lecito o non è lecito eseguirla? Di per sè non è nè lecita, nè illecita. Dipende solo da chi la vuole eseguire: un utente non ne ha diritto, il sistema sì. Anche l'invocazione di primitiva da parte di un processo può essere immaginato come un *cambio di contesto* (meno drammatico rispetto al cambio di contesto vero e proprio), che definiamo *cambio di privilegio* per distinguerlo dai cambi di processo.

3.2 Moduli

Il sistema che andiamo a studiare è realizzato in moduli: ogni modulo è un programma a se stante eseguito in un contesto (di privilegio) diverso.

- Il **modulo sistema** deve essere eseguito a livello sistema;
- Il **modulo I/O** è eseguito a livello sistema (la soluzione ideale sarebbe eseguirlo a un livello di privilegio intermedio, ma abbiamo solo due modalità - sistema e utente);
- Il **modulo utente** è eseguito a livello utente.

In sostanza...

- Il programmatore di sistema scrive delle primitive, che svolgono operazioni per conto dell'utente (impedisco all'utente di occuparsi delle periferiche, ad esempio, ma gli dico che posso farlo io). Il codice di sistema, così come le primitive e le strutture dati su cui si reggono, sono posti nel modulo sistema e nel modulo I/O (vedremo verso la fine l'utilità del modulo I/O).
- Le primitive possono essere invocate esclusivamente passando da un gate della IDT, innalzando il livello del processore.
- L'utente si scrive i programmi di cui ha bisogno e li esegue in modalità utente, facendo eventuale ricorso alle primitive offerte dal nucleo.

Il modulo sistema e il modulo I/O costituiscono, di fatto, un sistema operativo non modificato dall'utente, che riceve il programma e lo esegue.

3.3 Cartelle

Modulo sistema I file che compongono il modulo sistema stanno nella directory

`sistema`

tutto è posto in due file: *sistema.cpp* (parte C++) e *sistema.s* (parte Assembler).

Modulo I/O I file che compongono il modulo I/O stanno nella directory

`io`

anche in questo caso abbiamo tutto in due file: *io.cpp* (parte C++) ed *io.s* (parte Assembler).

Modulo utente I file che compongono il modulo utente stanno nella directory

`utente`

abbiamo più files:

- *all.h*
- *lib.cpp*

- *lib.h*
- *utente.s*
- *utente.cpp*

La cosa è necessaria per permettere all'utente di scrivere solo la parte che gli interessa (quella C++, nell'ultimo file).

Approccio L'approccio è il solito: utilizzare Linux come sistema di appoggio per *cross-compiling*. Compiliamo su Linux programmi pensati per girare come sistema.

3.4 Esempio di scrittura del file *utente.cpp*: Hello, world!

```
#include <all.h>

void main() {
    writeconsole("Hello, world!\n", 14);
    pause(); /* Per evitare che il processo venga terminato subito */
    terminate_p();
}
```

- Includiamo *all.h* per avere a disposizione tutte le funzioni.

  ```
  #include <costanti.h>
  #include <libce.h>
  #include <sys.h>
  #include <io.h>
  #include "lib.h"
  ```

- Scriviamo il *main*, che non è standard (attenzione al *void*). Il sistema lo esegue all'interno di un processo: quando il processo termine e non ci sono altre cose da fare il sistema spegne la macchina.
- Un esempio di primitiva è *writeconsole*

  ```
  writeconsole("Hello, world!\n", 14);
  ```

  passiamo la stringa e indichiamo la sua lunghezza.
- *terminate_p* è un altro esempio di primitiva, invocata per terminare esplicitamente il processo.
- Si osservi che entrambe le primitive comportano un cambio di privilegio: la prima sarà eseguita dal modulo I/O (poichè posta nel modulo I/O), la seconda dal modulo sistema (poichè posta nel modulo sistema).

3.5 Compilazione e avvio del sistema

Eseguiamo il comando *make* nella root

```
make
```

la procedura ha creato, nella cartella *build* tre file ELF: modulo utente, modulo sistema e modulo I/O. Ribadiamo che ciò che abbiamo ottenuto non è pensato per essere eseguito su Linux ma su un'altra macchina avente architettura comune.

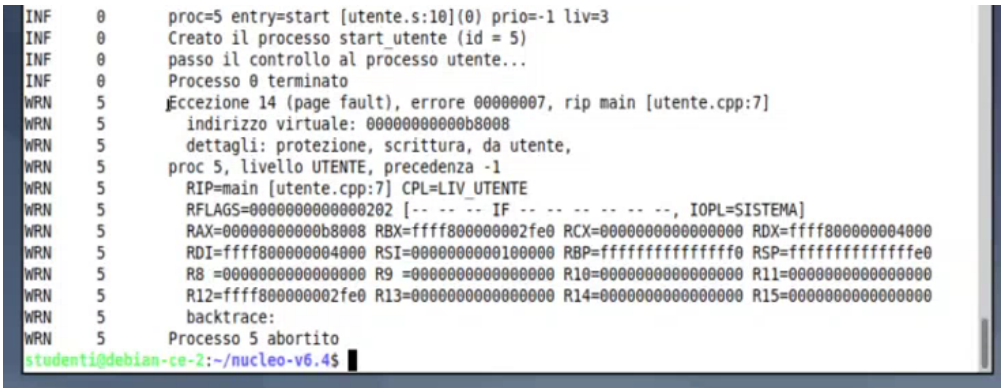
Avvio del sistema

- Il modulo sistema è il primo modulo caricato dal bootloader. Il modulo sistema questa volta è un po' più sofisticato: inizializza le sue strutture dati, prepara i moduli I/O e utente. Il bootloader si limita solo a caricare questi moduli in memoria, il modulo sistema li interpreta. Quando è tutto pronto viene creato un nuovo processo e si avvia l'esecuzione del programma *main* in quel processo, a livello utente.
- Osserviamo che all'avvio viene inizializzata la GDT (accompagnata da alcuni elementi della segmentazione necessari per definire i livelli di privilegio), l'APIC, il timer (che restituisce periodicamente il controllo al sistema). il modulo IO con tutte le periferiche che conosciamo (tastiera, video, busmaster). Dopo tutte queste inizializzazioni passo il controllo all'utente.
- Ogni processo è identificato da un numero, visibile nel terminale (dei vari messaggi e warning si vede il relativo processo) .
- Il processore esegue il sistema o il modulo utente. Risponde a tutte le chiamate di sistema che l'utente potrebbe fare, e a tutte le eccezioni/interruzioni, che provocano la restituzione del controllo al modulo sistema (o al modulo I/O in alcuni casi).

3.6 Esempio di programma bloccato dalla protezione

```
#include <all.h>

void main() {
    volatile natw* video = (natw*)0xb8000;
    video[4] = 0x3F00 | 'a';

    pause(); /* Per evitare che il processo venga terminato subito */
    terminate_p();
}
```

- Proviamo a scrivere in memoria video senza utilizzare la primitiva *writeconsole*.
- Se compiliamo non succede nulla, ma quando avvieremo il sistema il nostro programma verrà interrotto. Nel momento in cui ha provato a scrivere in quella zona di memoria il processore si è rifiutato e ha lanciato un'eccezione 14 di *page fault*: il controllo è stato restituito al sistema, la routine andata in esecuzione ha interrotto il processo e stampato una miriade di informazioni (che dovrebbero aiutarci a capire in che stato si trovava il processo al momento dell'errore).



Vediamo che l'eccezione è stata provocata da una certa riga di *utente.cpp*, vediamo anche l'indirizzo di memoria relativo. Si indica il livello di privilegi corrente al momento di esecuzione dell'istruzione, i valori dei registri (RFLAGS, IF, IOPL, RAX, RBX...). Per quanto riguarda il registro RIP il log è automaticamente decodificato, se l'indirizzo è associabile a una riga del sorgente si pone direttamente il numero di riga. Possiamo evitare questa cosa nel seguente modo

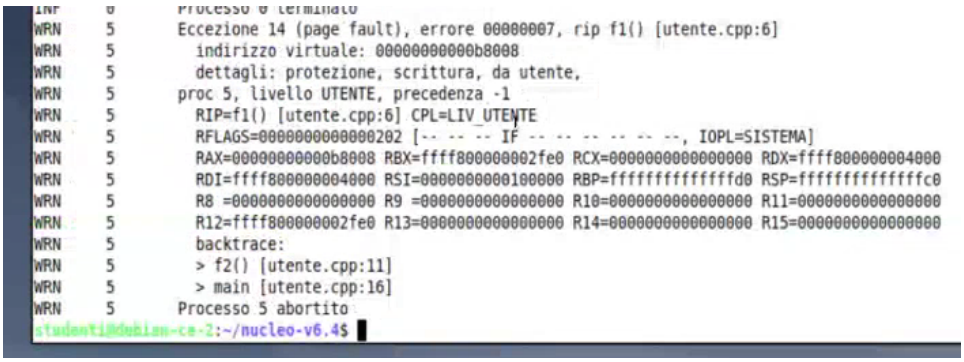
```
CERAW=1 ./run
```

3.6.1 *backtrace*

Il backtrace mostra lo stack delle chiamate. Proviamo a porre il programmino precedente nel seguente modo:

```
#include <all.h>
void f1() {
    volatile natw* video = (natw*)0xb8000;
    video[4] = 0x3F00 | 'a';
}

void f2() { f1(); }

void main() {
    f2();
    pause(); /* Per evitare che il processo venga terminato subito */
    terminate_p();
}
```

Otteniamo un lancio di eccezione anche in questo caso



- Si specifica che al momento dell'errore l'instruction pointer era dentro *f1*.
- Si indica che *f2* è stata chiamata da *f1*, a sua volta chiamata da *main*. In entrambi i casi si indica la riga del sorgente con la chiamata.

Il backtrace è utile perché la funzione potrebbe essere chiamabile da più punti del nostro sorgente.

3.7 Libreria *libce*

```
#include <all.h>

void f1() {
    char_write('a');
}

void f2() {
    f1();
}

void main() {
    f2();
    pause(); /* Per evitare che il processo venga terminato subito */
    terminate_p();
}
```

Il nostro programma è collegato anche con la libreria *libce*, stessa cosa il modulo I/O. Per scrivere su video i codici da scrivere sono i soliti già visti. Dobbiamo ricordarci che non è il codice a dirci il livello di privilegio, ma il contesto.

Capitolo 4

Processo

4.1 Gestione dei processi

```
#include <all.h>

void f1() {
    char_write('a');
}

void f2() {
    f1();
}

void main() {
    natw id = activate_p(f1, 10, 40, LIV_UTENTE);
    terminate_p();
}
```

Una delle cose che possiamo fare con questo sistema è gestire i processi. Il sistema mi fornisce la primitiva *activate_p*, che mi permette di attivare un nuovo processo. Devo indicare

- la funzione che dovrà eseguire questo processo,
- gli eventuali parametri in ingresso,
- la precedenza (si esprime come valore numerico),
- il livello di privilegio (in realtà la primitiva non ci concederebbe la creazione di un processo sistema, solo che questa cosa può essere utilizzata dal modulo I/O, che può chiedere di creare un processo a livello sistema).

La decisione di quali processi vanno avanti e quando è una delle questioni più bollenti riguardo i sistemi operativi.

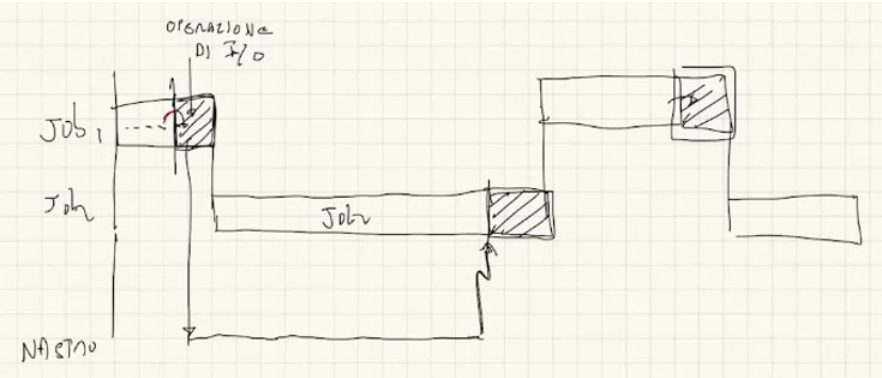
Identificativo del processo La funzione restituisce un valore: l'identificatore del processo appena creato. Restituisce una sequenza di 1 nel caso in cui ci siano stati errori.

Che differenza c'è tra *printf* e *writeconsole*? La *printf* è una libreria di funzione che per stampare caratteri userà delle primitive. La *writeconsole* è una primitiva vera e propria.

Come si realizza l'astrazione dei processi? Durante la sua vita un processo può trovarsi in uno tra diversi stati di esecuzione. Il processore sta eseguendo le istruzioni del programma, sta portando avanti lo stato del processo. Se abbiamo un solo processore solo un processo può trovarsi nello stato **esecuzione**. Tutti gli altri processi presenti nel sistema possono trovarsi in uno tra i seguenti stati:

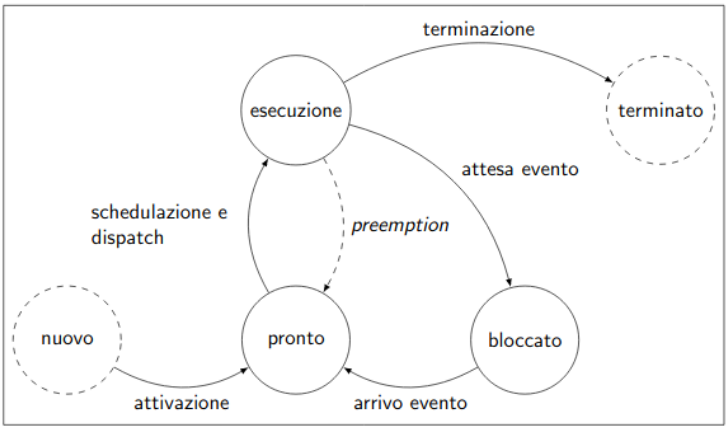
- **pronti** (il processo potrebbe andare avanti se dipendesse solo da lui, non va avanti perché il processore è già occupato da qualcun'altro con precedenza maggiore),
- **bloccati** (il processo sta attendendo il verificarsi di un certo evento, per esempio il termine di un'operazione di I/O, non può andare avanti neanche col processore libero).

Supponiamo di avere due processi: *job1* e *job2*, il primo ha precedenza maggiore.



- Si inizia con l'esecuzione di *job1*, che ha priorità maggiore. Rimane in stato *esecuzione* finchè non avvia l'operazione di I/O output. A quel punto il *job1* si pone nello stato *bloccato* e passa la palla a *job2*.
- Il processo *job2* passa in *esecuzione*. Si eseguono istruzioni finchè l'operazione di I/O non risulta completata. A quel punto si ha una richiesta di interruzione intercettata dal sistema: la routine rimette *job1* in *esecuzione* e pone *job2* nello stato *pronto*.
- Il processore conclude l'esecuzione di *job1* invocando una primitiva di sistema: a quel punto si rimette in *esecuzione* il processo *job2*.

I passaggi sono intramezzati per forza di cose, abbiamo numerose variazioni di stato dei vari processi. Si parla di:



- **schedulazione**, quando un processo passa da *pronto* a *esecuzione*;
- **blocco**, quando un processo passa da *esecuzione* a *bloccato*;
- **risveglio**, quando un processo passa da *bloccato* a *esecuzione*;
- **preemption**, quando un processo passa da *esecuzione* a *pronto*,.

L'ultimo passaggio, che possiamo italianizzare in *prelazione*, avviene quando è possibile eseguire processi con priorità maggiore. La cosa non è presente in tutti i sistemi multiprocesso.

Conseguenza di questa assenza Un processo va avanti finchè non si blocca, ma per bloccarsi è lui che chiama una primitiva di sistema. Se fa un ciclo infinito il processore rimane lì. Se c'è un errore in un programma tutto il sistema si blocca (ripensare a sistemi operativi come *Windows 95 / 98 / ME*). In un sistema del genere il processore si accorge di processi con priorità maggiore solo dopo che il processo in esecuzione è stato bloccato.

Creazione di processi Un processo può creare altri processi assegnando una priorità minore o maggiore. Se il processo ha priorità maggiore potrebbe fare *preemption* subito. Nel nostro caso non sarà così perché imporremo la creazione di processi aventi priorità minore.

Terminazione dei processi Il processo può essere terminato solo mediante esecuzione di primitiva, dunque il processo dovrà essere in *esecuzione*. Per permettere a processi di uccidere altri processi dobbiamo implementare ulteriori meccanismi che non ci interessano. Distinguiamo la cosa da una terminazione dovuta a eccezioni.

Sistema come intermediario I cambi di processo avvengono grazie all'intermediazione del sistema: avvengono solo quando i processi, che normalmente girano a livello utente, sono passati a livello sistema (con uno dei metodi possibili, per eccezione, interruzione esterna o istruzione INT). Tutto ciò è vero per tutti i sistemi multiprogrammati.

Semplificazioni Il nostro sistema adotta alcune semplificazioni:

- una schedulazione che si basa solo sulle precedenze dei processi;
- il sistema è multiprocesso ma non multiutente.

La multiprogrammazione si è resa necessaria andando avanti: è vero che l'utente è unico, ma è vero che l'utente singolo vuole fare più cose in simultanea. L'utente, inoltre, esegue programmi non realizzati da lui, ed esegue in simultanea programmi realizzati da sviluppatori diversi.

4.2 Processo dal punto di vista dell'utente

Un'altra cosa su cui i sistemi possono distinguersi è quanta memoria i processi possono condividere tra di loro (condividere nel senso avere processi con dati in comune). Abbiamo sistemi *a scambio di messaggi* (dove non si condivide niente) e sistemi *a memoria comune* (dove condividono tutto). Quello che faremo noi è un ibrido.

- **Parte condivisa**.

  I processi condividono tutto ciò che è globale nel modulo utente

  ```
  .text
  .data
  .bss
  .heap // anche lo heap, creato a tempo di esecuzione
  ```

- **Parte non condivisa**.

  Ogni processo presenta una *pila privata*.

Le pile sono separate in modo netto: un processo non ha la possibilità di accedere alla pila di un altro processo.

4.3 Processo dal punto di vista del programmatore

Dal punto del sistema abbiamo un programmatore, non un utente, che prepara le strutture dati e le routine necessarie per realizzare quanto visto dall'utente. Ogni processo è descritto da una *struct*: il **descrittore di processo**

```
des_proc *pronti;
des_proc *esecuzione; // <--- uno solo, visto che abbiamo solo un processore
const int N_REG = 16;

struct des_proc {
    natw id;
    natw livello;
    natl precedenza;
    vaddr punt_nucleo; /* Puntatore a pila sistema */
    natq contesto[N_REG];
    paddr cr3;

    des_proc *puntatore;
};
```

- **esecuzione**. Una variabile globale (il puntatore *esecuzione*) si ricorda del processo in esecuzione (ricordiamo, uno solo se abbiamo un solo processore).

- **punt_nucleo**. Il sistema oltre ad avere un descrittore di processo per ogni processo ha anche una pila sistema diversa per ogni processo. perché? Abbiamo detto che ogni processo deve avere la sua pila: la cosa è valida non solo in modalità utente, ma anche in modalità

sistema. Tutte le informazioni che il processore salva quando attraversa un gate devono rimanere sulla pila del processo, poichè costituiscono lo stato del processo quando si verifica l'interruzione/eccezione/int (tutte cose che dobbiamo tenere da parte).

- **contesto**. La cosa più importante che il sistema deve ricordarsi, di ogni processo, è "l'ultima foto" del suo sistema, che consiste in particolare nel contenuto dei registri. Abbiamo un array *contesto* nella struttura, che contiene il valore corrente di tutti i registri del processore (ci semplifichiamo la vita non memorizzando lo stato della memoria).

- **puntatore**. All'interno della struttura abbiamo *puntatore*, un puntatore necessario per costruire delle liste che rappresentano tutti i processi che si trovano nei vari stati.

  - **pronti**. Tutti i processi pronti si trovano in una lista: il nostro sistema avrà una variabile globale (il puntatore *pronti*) che punta al primo elemento della lista. I vari elementi che costituiscono la lista sono le strutture *des_proc*. La lista è ordinata in base al campo *precedenza* del processo, segue che per l'operazione di *schedulazione* basterà leggere il primo elemento della lista (cosa con complessità nulla).
  - Esistono più liste dedicate ai processi bloccati, ciascuna riguardante uno specifico evento: lista per chi aspetta di utilizzare l'hard disk, chi la tastiera, chi il timer...

  Queste liste sono gestite dal sistema, dopo aver ottenuto nuovamente il controllo a seguito del passaggio del gate (ricordiamoci, serve l'atomicità per manipolare delle liste).

- **cr3**. Affrontato più avanti, nel capitolo sulla paginazione.

Differenze tra routine di sistema e routine di interruzione

- Possiamo immaginarci una routine di sistema come una routine di interruzione.

- La differenza sostanziale sta nella proprietà di *atomicità* della routine di sistema, ossia la sua indivisibilità. **La routine di interruzione può essere interrotta da un'altra interruzione, la routine di sistema no**!

- **La routine di sistema non può essere considerata parte del processo** poichè non contribuisce all'avanzare dello stato del processo stesso.

4.3.1 Passaggio a contesto sistema

Quando il processo passa il gate viene fotografato lo stato del processo corrente. Il passaggio al *contesto sistema* avviene con il seguente codice Assembler

```
CALL salva_stato
// ...
// codice della routine di sistema
// ...
CALL carica_stato
IRETQ
```

- Con la *salva_stato* è una routine scritta in assembler che salva, nel campo *contesto* del processo in esecuzione, lo stato dei registri.

- Questo codice in realtà è il solito incapsulamento che abbiamo già usato altre volte. Non ci piace programmare in Assembler, dunque tra le due call chiameremo una routine implementata in C++. Ricordarsi che l'incapsulamento via Assembler è obbligatorio a causa della IRETQ (come la chiamiamo in C++? Non si può).
- Con la *carica_stato* poniamo il contenuto del campo *contesto* (del processo in esecuzione) nei vari registri.

Ribadiamo

Quanto posto tra le due call non deve essere considerato parte del processo.

4.3.2 Passaggio da un processo a un altro attraverso routine di sistema

Il passaggio da un processo a un altro è cosa semplice a questo punto. Abbiamo un codice simile al precedente (il passaggio avviene mediante routine di sistema, visto che il tutto funzione con l'intermediazione del sistema)

```
CALL salva_stato
// ...
esecuzione = ...
// ...
CALL carica_stato
IRETQ
```

ma modifichiamo tra le due call il valore del puntatore *esecuzione* (il valore precedente sarà posto in un'altra coda). Quando la nostra routine di sistema finisce la *carica_stato* andrà a porre nei registri il *contesto* del nuovo processo puntato dalla variabile globale *esecuzione* (**solo a questo punto avviene il passaggio**).

Memoria protetta Dal punto di vista della protezione tutte queste strutture dati (descrittori di processo, pile di sistema, variabili globali *pronti* ed *esecuzione*) devono stare nella parte di memoria non accessibile agli utenti. Distinguiamo due aree di memoria:

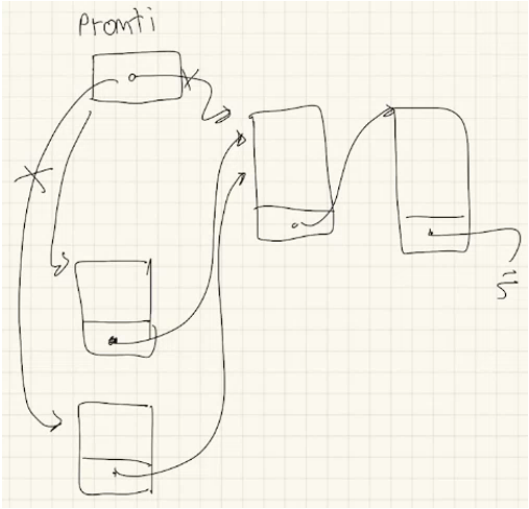
- le strutture dati, i descrittori di processo, le pile sistema stanno nell'area di memoria non accessibile all'utente;
- modulo utente e pila del processo corrente stanno nell'area accessibile all'utente (supponiamo ci sia una pila alla volta in memoria).

4.3.3 Atomicità: perché la routine di sistema deve essere indivisibile?

Dal punto di vista della correttezza limitarsi a proteggere una parte di memoria non è sufficiente. Dobbiamo garantire l'*atomicità* per evitare problemi.

Singole istruzioni Un'istruzione è per definizione indivisibile: abbiamo detto che le interruzioni si manifestano tra un'istruzione e un'altra, non durante l'esecuzione di una singola istruzione.

Consistenza delle liste Prendiamo ad esempio la gestione delle liste con i processi, che è in mano al sistema (l'intermediario, prendiamo ad esempio la coda globale *pronti*).



Abbiamo visto che la manipolazione della lista non richiede una semplice istruzione macchina, ma una serie di istruzioni durante le quali la lista non è consistente. La lista è consistente all'inizio (prima delle modifiche) e alla fine (dopo le modifiche). Lanciare un'interruzione nel bel mezzo della sua manipolazione significa considerarla in uno stato inconsistente (si hanno problemi se mentre stiamo manipolando una lista si ha un'interruzione e la relativa routine interviene anch'essa sulla lista che era in corso di modifica). Ricordarsi che

La lista è consistente solo se tutti gli elementi sono raggiungibili partendo dalla testa

Cosa facciamo? Seguono due strade possibili:

1. scrivere le routine pensando al fatto che lo stato considerato possa essere inconsistente (il programmatore normalmente scrive pensando che nessuno si metta di mezzo tra le istruzioni del suo programma);
2. impedire le interruzioni in modo tale che gli stati inconsistenti non siano visibili.

La prima cosa è dispendiosa, la seconda semplice:

- per le interruzioni esterne si ricorre a un apposito flag (il flag IF);
- le eccezioni devono essere evitate dal sistema;
- stessa cosa le interruzioni software (ad ogni chiamata di routine corrisponde una *salva_stato*, che sovrascrive il contesto precedente[1]).

In questo modo evitiamo tutti i meccanismi che permettono di attraversare il gate e di ritornare ricorsivamente nel sistema. Possiamo dire che **la routine di sistema è diventata, in un certo senso, simile a un'unica istruzione di linguaggio macchina** (è atomica e indivisibile, viene eseguita fino alla sua fine).

[1] Ricordiamo che ciò che viene eseguito in una routine di sistema non è parte di un processo. L'esecuzione di una *salva_stato* comporta la perdita delle informazioni relative alla routine eseguita.

Wait, altro problema Tenere disattivate le interruzioni è una limitazione troppo grande per molti progettisti di sistema, dunque si è elaborata una via di mezzo. Nel modulo I/O, che vedremo più avanti, abbiamo un rilassamento delle restrizioni:

- le eccezioni sono normalmente vietate;
- le interruzioni sono ammesse;
- la chiamate ricorsive sono permesse (forse si, forse no).

Tutto rimane vietato nel modulo sistema (atomicità, per forza).

4.3.4 Prima istantanea dello stato di un processo

Per i processi appena partiti non è mai stata scattata una foto prima. La foto viene scattata quando si entra in modalità sistema: meccanismo delle eccezioni, processore (salva le informazioni nella pila sistema privata, per la IRETQ), in parte con la *salva_stato* (si aggiorna la struttura *des_proc*). Tutto questo funziona quando il processo esiste già e si attraversa un gate: ma per i processi appena creati? Il modo più semplice è fare in modo che l'*activate_p* generi la foto:

```
activate_p(miaproc, 10, 20, LIV_UTENTE);
```

questo significa creare l'istanza relativa della struttura *des_proc* e allocare la pila di sistema.

- Intanto posso impostare in *des_proc* la priorità e il livello di privilegio.
- Successivamente devo inizializzare il contesto e la pila in modo tale che le seguenti istruzioni, eseguite da una qualche routine, diano inizio al processo in modo regolare.

  ```
  CALL carica_stato
  IRETQ
  ```

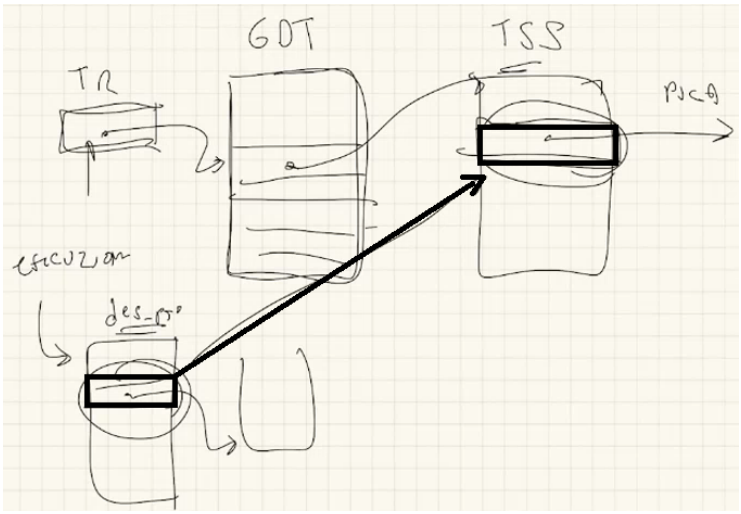
  – In pila mettiamo 5 *quadword* (la IRETQ vuole questo, ricordiamolo). Abbiamo:

  * (RIP) l'indirizzo della funzione da eseguire;
  * (CPL) si imposta in modo che il livello di privilegio sia quello indicato nell'apposito parametro di ingresso;
  * (RFLAG), si impostano i flag con IF ad 1 e IOPL sempre *sistema* (si indica il *Privilege level* necessario per eseguire certe istruzioni);
  * (RSP), si pone l'indirizzo con la pila utente del processo in esecuzione;
  * (TSS) cose relative ai segmenti che non ci interessano.

Si consideri che in un processo a livello sistema si crea solo la pila sistema, mentre in un processo a livello utente si crea sia la pila sistema che quella utente.

- Il contesto ha almeno due registri da inizializzare in modo preciso, gli altri possono essere uguali a zero.
  - Il parametro in ingresso deve andare in RDI (la funzione *miaproc* ha un parametro con valore 10, lo mettiamo subito nel registro - secondo le regole già viste - per permettere l'esecuzione della funzione).
  - Dobbiamo mettere in RSP l'indirizzo della pila sistema, in modo tale che IRETQ possa leggere i dati che abbiamo messo nella pila sistema.

- Si consideri che l'indirizzo della pila sistema non viene preso da *des_proc* ma da quella roba brutta vista qualche lezione fa: passaggio dal TR (*Task Register*) per arrivare alla GDT (*Global Descriptor Table*) e infine al TSS (*Task State Segment*). Per indicare al processore di usare una certa pila abbiamo due strade...
  - **Un TSS per processo**.
    Uso il meccanismo della segmentazione progettato da Intel (quindi *punt_nucleo* punta al TSS) come descrittore di processo (però a quel punto ci serve un TSS per ogni processo e siamo limitati nel numero a causa degli ingressi limitati della GDT) e cambiare ogni volta il valore del registro TR;
  - **Un unico TSS per tutti i processi** (soluzione adottata).
    Ci limitiamo a creare un solo TSS (quindi il valore del registro TR è costante) per fare contento il processore, e quando cambiamo processo la *carica_stato* prende il puntatore alla pila sistema dal *des_proc* e lo pone nel TSS.



Capitolo 5

Implementazione dei processi e delle primitive nel modulo sistema

5.1 Premessa: file *include/costanti.h*

Nel file *sistema.s*, come in altri file di estensione cpp, è incluso con apposita direttiva il file *costanti.h*

```
#include "costanti.h"
```

esso contiene una serie di costanti, tutte introdotte con *define* per permetterne l'uso sia in C++ sia in Assembler. L'uso di queste costanti sarà evidente da qua fino alla fine della dispensa.

5.2 Inizializzazione della *Interrupt Descriptor Table*

La gestione della *Interrupt Descriptor Table* non è così diversa da quella vista con la libreria *libce*.

```
.global init_idt
init_idt:
    //      indice      routine         dpl
    // gestori eccezioni:
    carica_gate 0       divide_error    LIV_SISTEMA
    carica_gate 1       debug           LIV_SISTEMA
    carica_gate 2       nmi             LIV_SISTEMA
    carica_gate 3       breakpoint      LIV_SISTEMA
    carica_gate 4       overflow        LIV_SISTEMA
    carica_gate 5       bound_re        LIV_SISTEMA
    carica_gate 6       invalid_opcode  LIV_SISTEMA
    carica_gate 7       dev_na          LIV_SISTEMA
    carica_gate 8       double_fault    LIV_SISTEMA
    carica_gate 9       coproc_so       LIV_SISTEMA
    carica_gate 10      invalid_tss     LIV_SISTEMA
    carica_gate 11      segm_fault      LIV_SISTEMA
    carica_gate 12      stack_fault     LIV_SISTEMA
    carica_gate 13      prot_fault      LIV_SISTEMA
    carica_gate 14      page_fault      LIV_SISTEMA
    // ... il tipo 15 è riservato
```

```
carica_gate 16      fp_exc          LIV_SISTEMA
carica_gate 17      ac_exc          LIV_SISTEMA
carica_gate 18      mc_exc          LIV_SISTEMA
carica_gate 19      simd_exc        LIV_SISTEMA
carica_gate 20      virt_exc        LIV_SISTEMA
// ... tipi 21-29 riservati
carica_gate 30      sec_exc         LIV_SISTEMA
// ... tipo 31 riservato

// primitive comuni (tipi 0x2-)
carica_gate TIPO_A      a_activate_p    LIV_UTENTE
carica_gate TIPO_T      a_terminate_p   LIV_UTENTE
carica_gate TIPO_SI     a_sem_ini       LIV_UTENTE
carica_gate TIPO_W      a_sem_wait      LIV_UTENTE
carica_gate TIPO_S      a_sem_signal    LIV_UTENTE
carica_gate TIPO_D      a_delay         LIV_UTENTE
carica_gate TIPO_L      a_log           LIV_UTENTE
carica_gate TIPO_GMI    a_getmeminfo    LIV_UTENTE

// primitive per il livello I/O (tipi 0x3-)
carica_gate TIPO_APE    a_activate_pe   LIV_SISTEMA
carica_gate TIPO_WFI    a_wfi           LIV_SISTEMA
carica_gate TIPO_FG     a_fill_gate     LIV_SISTEMA
carica_gate TIPO_AB     a_abort_p       LIV_SISTEMA
carica_gate TIPO_IOP    a_io_panic      LIV_SISTEMA
carica_gate TIPO_TRA    a_trasforma     LIV_SISTEMA
carica_gate TIPO_ACC    a_access        LIV_SISTEMA

// i tipi 0x4- verranno usati per le primitive fornite dal modulo I/O
// (si veda fill_io_gates() in io.s)
// i tipi da 0x50 a 0xFE verrano usati per gli handler
// (si veda load_handler() più avanti)

// la priorità massima è riservata al driver del timer di sistema
carica_gate TIPO_TIMER  driver_td       LIV_SISTEMA

lidt idt_pointer
ret
```

- Abbiamo la *carica_gate*, ma con la possibilità di indicare anche il *Descriptor Privilege Level* del gate (livello considerato solo nel lancio di interruzioni con istruzione INT).

```
// Carica un gate della IDT
// num: indice (a partire da 0) in IDT del gate da caricare
// routine: indirizzo della routine da associare al gate
// dpl: dpl del gate (LIV_SISTEMA o LIV_UTENTE)
.macro carica_gate num routine dpl
    movq $\num, %rdi
    movq $\routine, %rsi
    movq $\dpl, %rdx
    xorq %rcx, %rcx
    call init_gate
.endm
```

- **Attenzione**.

  Le righe di codice dove usiamo ripetutamente la macro *carica_gate* non sono il luogo dove effettivamente stiamo ponendo la nostra *Interrupt Descriptor Table*. In fondo al file *sistema.s* abbiamo uno spazio destinato a tale scopo

```
.bss
.balign 16
idt:
    // spazio per 256 gate
    // verra' riempita a tempo di esecuzione
    .space 16*256, 0
```

  La routine *init_gate*, chiamata nella macro, organizza il contenuto di quell'area di memoria ogni volta che viene inserito un gate.

- **Istruzione LIDT**.

  Con l'istruzione *lidt* andiamo ad aggiornare il registro IDTR

```
lidt idt_pointer
```

  dove *idt_pointer* è una variabile avente una certa forma:

```
.data
// ...
idt_pointer:
    .word 0xFFF // limite della IDT (256 entrate)
    .quad idt // base della IDT
```

  - il *word* ci dicono quanto è grande la *Interrupt Descriptor Table*,
  - il *quad* contiene l'indirizzo della base della IDT (non a caso, *idt*).

- Ricordiamo quanto detto con l'APIC

> Il livello di priorità è codificato nel tipo. L'identificativo può essere scomposto in due parti:
>
> - la classe di priorità (parte più significativa), e
> - la sottopriorità all'interno della classe (parte meno significativa).
>
> La regola che la APIC segue è di inviare una nuova richiesta di interruzione soltanto se tra quelle pendenti ce n'è una con classe di priorità maggiore della massima classe di priorità che è già stata accettata dal processore. In caso di *ex-aequo* si guarda la sottopriorità.

Il primo parametro della *carica_gate* permette di indicare l'identificativo del tipo, dunque la priorità associata: maggiore è il valore numerico, maggiore sarà la priorità. Si osservi a chi viene attribuità la priorità massima: il timer (la cosa non dovrebbe stupirci)!

```
carica_gate TIPO_TIMER  driver_td       LIV_SISTEMA
```

In *include/costanti.h* sono definite una serie di costanti relative ai tipi dei gate

```
// ( tipi delle primitive
//   ( comuni
#define TIPO_A 0x20 // activate_p
#define TIPO_T 0x21 // terminate_p
#define TIPO_SI 0x22 // sem_ini
#define TIPO_W 0x23 // sem_wait
#define TIPO_S 0x24 // sem_signal
#define TIPO_D 0x25 // delay
#define TIPO_L 0x26 // log
#define TIPO_GMI 0x27 // getmeminfo (debug)
//   )

//   ( riservate per il modulo I/O
#define TIPO_APE 0x30 // activate_pe
#define TIPO_WFI 0x31 // wfi
#define TIPO_FG 0x32 // fill_gate
#define TIPO_AB 0x33 // abort_p
#define TIPO_IOP 0x34 // io_panic
#define TIPO_TRA 0x35 // trasforma
#define TIPO_ACC 0x36 // access
//   )

// ...
// tipo del driver del timer (priorità  massima)
#define TIPO_TIMER 0xFF
```

- Possiamo classificare i gate già caricati in:
  - eccezioni (dal gate 0 al gate 31, possibili eccezioni definite dalla Intel);
  - primitive comuni (eseguibili a livello utente);
  - primitive per il livello I/O (primitive che saranno utilizzate nel modulo I/O).

  Altri gate saranno caricati più avanti, nel modulo I/O:

```
// i tipi 0x4- verranno usati per le primitive fornite dal modulo I/O
// (si veda fill_io_gates() in io.s)
// i tipi da 0x50 a 0xFE verrano usati per gli handler
// (si veda load_handler() più avanti)
```

- **Ma non è limitante usare un gate per ogni primitiva?**

  Solitamente per tutte le primitive si usa un unico gate: la primitiva chiamata viene decisa sulla base di un valore posto in un registro (RAX, usa quel numero per accedere a un array di puntatori). In questo modo non siamo più limitati dall'hardware nel numero di primitive! Noi non faremo così: per quello che dobbiamo fare non ci serve un numero spropositato di primitive.

5.3 Inizializzazione della *Global Descriptor Table*

Facciamo cose molto simili anche per la *Global Descriptor Table*, che come già detto dobbiamo utilizzare (non come concepito all'inizio) per le modalità sistema e utente. All'interno della GDT troviamo anche il segmento TSS, che ci serve per gestire il cambio della pila.

```
.balign 8
.global gdt
gdt:
    .quad 0     //segmento nullo
code_sys_seg:
    .word 0b0           //limit[15:0]   not used
    .word 0b0           //base[15:0]    not used
    .byte 0b0           //base[23:16]   not used
    .byte 0b10011010    //P|DPL|1|1|C|R|A|  DPL=00=sistema
```

```
    .byte 0b00100000    //G|D|L|-|-------|  L=1 long mode
    .byte 0b0           //base[31:24]   not used
code_usr_seg:
    .word 0b0           //limit[15:0]   not used
    .word 0b0           //base[15:0]    not used
    .byte 0b0           //base[23:16]   not used
    .byte 0b11111010    //P|DPL|1|1|C|R|A|  DPL=11=utente
    .byte 0b00100000    //G|D|L|-|-------|  L=1 long mode
    .byte 0b0           //base[31:24]   not used
data_usr_seg:
    .word 0b0           //limit[15:0]   not used
    .word 0b0           //base[15:0]    not used
    .byte 0b0           //base[23:16]   not used
    .byte 0b11110010    //P|DPL|1|0|E|W|A|  DPL=11=utente
    .byte 0b00000000    //G|D|-|-|-------|
    .byte 0b0           //base[31:24]   not used
tss_seg:
    .space 16, 0     // riempito da init_gdt
end_gdt:

tss:
    .long 0
    .global tss_punt_nucleo
tss_punt_nucleo:
    .quad 0
    .space 11 * 8, 0
    .word 0
    .word tss_end - tss - 1
tss_end:
```

Ci servono, come già anticipato:

- un'entrata della GDT per gestire il passaggio da modalità utente a modalità sistema o per rimanere in modalità sistema;
- un'entrata della GDT per attraversare il gate e rimanere in modalità utente;
- il *Task State Segment* per gestire il cambio della pila (nei modi già spiegati quando abbiamo affrontato l'implementazione della protezione nel processore Intel), di esso ci interessa solo la riga *tss_punt_nucleo* (un *quad* dove metteremo l'indirizzo della pila sistema ogni volta che cambiamo processo).

A un certo punto, da *sistema.cpp*, verrà chiamata la funzione *gdt_init* (il cui codice si trova in *sistema.s*). Non ci interessa il codice (Lettieri non l'ha neanche guardato), dobbiamo solo ricordarci che:

- inizializza il contenuto del *Task State Segment* (non è possibile farlo staticamente, nelle righe precedenti abbiamo solo preparato l'area di memoria);
- esegue la seguente istruzione

  ```
  lgdt gdt_pointer
  ```

dove *gdt_pointer* è un'area di ottanta byte con sintassi simile alla *idt_pointer*

```
.data
gdt_pointer:
    .word end_gdt - gdt // limite della GDT
    .quad gdt
```

5.4 Processo *dummy* e shutdown con *end_program*

Nella funzione *main* di *sistema.cpp* viene creato il cosiddetto *processo dummy*.

```
// creazione del processo dummy
dummy_id = crea_dummy();
if(dummy_id == 0xFFFFFFFF)
    goto error;
flog(LOG_INFO, "Creato il processo dummy (id = %d)", dummy_id);
```

Esso permette di evitare il caso particolare in cui i processi sono tutti bloccati e non c'è altro da fare: risolvo la cosa con un processo a bassa priorità, dunque lo schedulatore trova sempre qualcosa da fare.

5.4.1 Funzioni *dummy* e *crea_dummy*

```
extern "C" void end_program();
void dummy(natq i) {
    while(processi)
        ;
    end_program();
}

natl crea_dummy() {
    des_proc* di = crea_processo(dummy, 0, DUMMY_PRIORITY, LIV_SISTEMA, true);
    if(di == 0) {
        flog(LOG_ERR, "Impossibile creare il processo dummy");
        return 0xFFFFFFFF;
    }
    inserimento_lista(pronti, di);
    return di->id;
}
```

La funzione *dummy* si limita a guardare in continuazione la variabile *processi*, che consiste nel numero di processi. Gira li finché il numero di processi attivi non sarà nullo[1].

5.4.2 Funzione *end_program*

La funzione *end_program* verrà eseguita solo se non si hanno processi da eseguire, quindi se usciamo dal ciclo presente nella funzione *dummy*. Il trucco consiste nel provocare un'eccezione di tipo *abort*. Mette nella IDT un puntatore nullo (usando la LIDT) e lancia un'interruzione

[1] Ovviamente si tenga conto del passaggio a processi con priorità maggiore, cosa che avviene quando almeno un processo non è occupato.

(con INT): nella IDT viene caricato un puntatore nullo, dunque trova l'indirizzo 0 (non valido) e lancia un'eccezione di *gate non valido*, anche in quel caso non trova l'indirizzo e lancia una terza eccezione che provoca *abort* e quindi lo spegnimento.

```
    .global end_program
end_program:
    lidt triple_fault_idt
    int $1

// ...

.data
triple_fault_idt:
    .word 0
```

5.5 Strutture dati per la gestione dei processi

5.5.1 Descrittore di processo

In *sistema.cpp* abbiamo il descrittore di processo, già ampiamente descritto.

```
const int N_REG = 16;
struct des_proc {
    natw id;
    natw livello;
    natl precedenza;
    vaddr punt_nucleo; /* Puntatore a pila sistema */
    natq contesto[N_REG];
    paddr cr3;

    des_proc *puntatore;
};
```

5.5.2 Array dei processi

Tutti i descrittori di processo sono raccolti in un'array

```
des_proc* proc_table[MAX_PROC];
```

dove *MAX_PROC* è una costante posta in *include/costanti.h*.

```
#define MAX_PROC        1024UL
```

5.5.3 Indici dei registri nell'array *contesto*

Per quanto riguarda l'array *contesto* si consideri l'*enum* contenente i vari indici dei vari registri all'interno dell'array *contesto*.

```
enum { I_RAX, I_RCX, I_RDX, I_RBX, I_RSP, I_RBP, I_RSI, I_RDI, I_R8, I_R9, I_R10,
I_R11, I_R12, I_R13, I_R14, I_R15};
```

quando andremo a manipolare i registri del contesto utilizzeremo queste costanti (non serve quindi conoscere l'indice preciso).

Esempio

```
esecuzione->contesto[I_RAX] = true;
```

5.5.4 Puntatori

Nel codice abbiamo questi due puntatori

```
des_proc *esecuzione;
des_proc *pronti;
```

- **esecuzione** consiste nel puntatore all'unico processo in esecuzione nel processore.
- **pronti** è il puntatore alla testa di una lista di processi in attesa. La lista è costruita utilizzando, nella struttura *des_proc*, il puntatore *puntatore*.

5.5.5 Conteggio del numero di processi

Poco dopo è definita la variabile *processi*, che come già anticipato conteggia il numero di processi attivi.

```
volatile natl processi;
```

Viene posta *volatile* poichè utilizzata nella funzione *dummy*.

5.6 Funzioni per la manipolazione delle liste

In *sistema.cpp* sono presenti delle funzioni che permettono di manipolare le liste. Ricordiamo che le liste relative ai processi sono ordinate esclusivamente in base alla precedenza.

5.6.1 Funzione *inserimento_lista*

```
void inserimento_lista(des_proc* &p_lista, des_proc* p_elem) {
    // inserimento in una lista semplice ordinata
    // (tecnica dei due puntatori)
    des_proc *pp, *prevp;

    pp = p_lista;
    prevp = nullptr;
    while(pp && pp->precedenza >= p_elem->precedenza) {
        prevp = pp;
        pp = pp->puntatore;
    }

    p_elem->puntatore = pp;

    if(prevp)
        prevp->puntatore = p_elem;
    else
        p_lista = p_elem;
}
```

- La funzione *inserimento_lista* permette l'inserimento in una qualunque lista di un elemento *des_proc*. L'inserimento dell'elemento *p_elem* si basa sulle precedenze (chi ha precedenza più alta sta in cima alla lista), con logica FIFO (a parità di precedenza vince l'elemento inserito per primo nella lista).
- Il parametro di ingresso *p_lista* è il puntatore alla lista da manipolare (con riferimento, potrei porre un elemento in cima alla lista).
- Il parametro di ingresso *p_elem* è il puntatore all'elemento da inserire nella lista.
- Il codice è simile, se non identico, alla funzione studiata a *Fondamenti di programmazione* per l'inserimento di un elemento in una lista:
  - scorro la lista finché non trovo il punto dove il processo deve essere inserito;
  - modifico *puntatore* dell'elemento precedente e imposto *puntatore* del processo appena inserito.

5.6.2 Funzione *rimozione_lista*

```
des_proc* rimozione_lista(des_proc* &p_lista) {
    // estrazione della testa
    des_proc *p_elem = p_lista // nullptr se la lista e' vuota

    if(p_lista)
        p_lista = p_lista->puntatore;

    if(p_elem)
        p_elem->puntatore = nullptr;

    return p_elem;
}
```

- La funzione *rimozione_lista* permette la rimozione della testa di una qualunque lista di elementi *des_proc*.
- Il parametro di ingresso *p_lista* è il puntatore alla lista (con riferimento, visto che rimuoviamo la testa).
- Pongo nel puntatore *p_elem* l'indirizzo della testa.
- Se nella lista puntata da *p_lista* sono presenti elementi (prima di procedere all'estrazione) modifico il puntatore *p_lista* rendendo il secondo elemento la nuova testa.
- Se l'elemento estratto non è nullo modifico il suo *puntatore* ponendolo a null (lo abbiamo scollegato dalla lista, non ha senso mantenere un collegamento).
- Restituisco l'indirizzo della testa estratta (*p_elem*).

5.6.3 Funzione *inspronti* per la lista *pronti*

```
extern "C" void inspronti() {
    esecuzione->puntatore = pronti;
    pronti = esecuzione;
}
```

La funzione *inspronti* inserisce il processo attualmente in esecuzione in testa alla lista *pronti*. Si consideri che:

- se abbiamo lavorato bene inseriremo in testa il processo con priorità maggiore;
- abbiamo aggiornato la lista *pronti*, **ma non abbiamo modificato *esecuzione*** (il processo posto in *esecuzione* è sempre quello attualmente in esecuzione).

5.6.4 Funzione *schedulatore* per la lista *pronti*

Promemoria. La schedulazione è il passaggio di un processo da *pronto* a *esecuzione*

La funzione *schedulatore* comporterà il passaggio di un processo (quello con priorità maggiore) dallo stato *pronto* a *esecuzione*. Le cose da fare sono due:

1. estrarre la testa dalla coda *pronti* (in testa abbiamo, secondo logica FIFO, il processo con priorità maggiore);
2. fare puntare il processo appena estratto dal puntatore *esecuzione*.

Grazie alla funzione *rimozione_lista* il codice della funzione *schedulatore* è molto breve.

```
extern "C" void schedulatore(void) {
    esecuzione = rimozione_lista(pronti);
}
```

Attenzione La funzione sceglie soltanto il processo che dovrà andare in esecuzione quando è finita la primitiva! **Non si ha il cambio di processo nel momento in cui chiamiamo schedulatore**. *Se pensate che scrivere dentro una variabile basti per cambiare il processo allora avete le idee un po' confuse (cit.).*

5.7 Funzioni *salva_stato* e *carica_stato*

Nel file *sistema.s* abbiamo due funzioni familiari: *salva_stato* e *carica_stato*. Nulla di strano, solo un qualcosa di intricato.

5.7.1 Premessa: offset

Sono definite in Assembler una serie di costanti per gestire gli offset:

```
// offset dei vari registri all'interno di des_proc
.set PUNT_NUCLEO, 8
.set CTX, 16
.set RAX, CTX+0
.set RCX, CTX+8
```

```
.set RDX, CTX+16
.set RBX, CTX+24
.set RSP, CTX+32
.set RBP, CTX+40
.set RSI, CTX+48
.set RDI, CTX+56
.set R8,  CTX+64
.set R9,  CTX+72
.set R10, CTX+80
.set R11, CTX+88
.set R12, CTX+96
.set R13, CTX+104
.set R14, CTX+112
.set R15, CTX+120
.set CR3, CTX+128
```

In entrambe le funzioni gli offset vengono utilizzati rispetto al registro RBX, dove abbiamo posto *esecuzione* (quindi gli offset sono costruiti tenendo conto della struttura di *des_proc*)

```
movq esecuzione, %rbx
```

5.7.2 salva_stato

```
// copia lo stato dei registri generali nel des_proc del processo puntato da
// esecuzione.  Nessun registro viene sporcato.

salva_stato:
    // salviamo lo stato di un paio di registri in modo da poterli
    // temporaneamente riutilizzare. In particolare, useremo %rax come
    // registro di lavoro e %rbx come puntatore al des_proc.
    pushq %rbx
    pushq %rax

    movq esecuzione, %rbx

    // copiamo per primo il vecchio valore di %rax
    movq (%rsp), %rax
    movq %rax, RAX(%rbx)
    // usiamo %rax come appoggio per copiare il vecchio %rbx
    movq 8(%rsp), %rax
    movq %rax, RBX(%rbx)
    // copiamo gli altri registri
    movq %rcx, RCX(%rbx)
    movq %rdx, RDX(%rbx)
    // salviamo il valore che %rsp aveva prima della chiamata a salva stato
    // (valore corrente meno gli 8 byte che contengono l'indirizzo di
    // ritorno e i 16 byte dovuti alle due push che abbiamo fatto all'inizio)
    movq %rsp, %rax
    addq $24, %rax
    movq %rax, RSP(%rbx)
    movq %rbp, RBP(%rbx)
    movq %rsi, RSI(%rbx)
    movq %rdi, RDI(%rbx)
```

```
movq %r8,  R8 (%rbx)
movq %r9,  R9 (%rbx)
movq %r10, R10(%rbx)
movq %r11, R11(%rbx)
movq %r12, R12(%rbx)
movq %r13, R13(%rbx)
movq %r14, R14(%rbx)
movq %r15, R15(%rbx)

popq %rax
popq %rbx

ret
```

- La funzione deve essere chiamata subito dopo essere entrati nel modulo sistema, per qualunque motivo. Deve memorizzare lo stato di tutti i registri nel momento in cui è stata chiamata.
- I registri che utilizziamo per svolgere le nostre operazioni sono RAX ed RBX. Il primo è utilizzato come registro di lavoro (cioè per fare passi intermedi), mentre RBX lo usiamo per puntare al descrittore di processo corrente (posto nella variabile globale *esecuzione*). Per fare queste cose poniamo in pila il contenuto vecchio di RAX ed RBX

  ```
  pushq %rbx
  pushq %rax
  ```

- Pongo in rbx il contenuto della variabile globale *esecuzione*

  ```
  movq esecuzione, %rbx
  ```

- Pongo nel *contesto* il contenuto di RAX
- Pongo nel *contesto* il contenuto di RBX, utilizzando RAX come registro di lavoro (ho bisogno di un registro dove spostare il mio risultato intermedio, per forza).
- Segue una sfilata di costanti attraverso cui indichiamo i relativi offset e spostiamo in *contesto* il contenuto dei registri rimanenti.
- Per porre il valore vecchio di RSP dobbiamo ricordarci che abbiamo eseguito una chiamata di funzione e due push, dunque RSP è già stato spostato rispetto all'inizio. Segue

  ```
  movq %rsp, %rax
  addq $24, %rax
  ```

  Useremo il valore vecchio calcolato nella IRETQ per la lettura della pila sistema.

- Esegiamo le due pop consuete dopo aver finito di utilizzare i registri RAX ed RBX

  ```
  pop %rax
  pop %rbx
  ```

5.7.3 carica_stato

```
// carica nei registri del processore lo stato contenuto nel des_proc del
// processo puntato da esecuzione.  Questa funzione sporca tutti i registri.
carica_stato:
    movq esecuzione, %rbx

    popq %rcx   //ind di ritorno, va messo nella nuova pila

    // nuovo valore per cr3
    movq CR3(%rbx), %r10
    movq %cr3, %rax
    cmpq %rax, %r10
    je 1f // evitiamo di invalidare il TLB
    // se cr3 non cambia
    movq %r10, %rax
    movq %rax, %cr3 // il TLB viene invalidato
1:
    // anche se abbiamo cambiato cr3 siamo sicuri che l'esecuzione prosegue
    // da qui, perché ci troviamo dentro la finestra FM che è comune a
    // tutti i processi
    movq RSP(%rbx), %rsp    //cambiamo pila
    pushq %rcx              //rimettiamo l'indirizzo di ritorno

    // se il processo precedente era terminato o abortito la sua pila
    // sistema non era stata distrutta, in modo da permettere a noi di
    // continuare ad usarla. Ora che abbiamo cambiato pila possiamo
    // disfarci della precedente.
    cmpq $0, ultimo_terminato
    je 1f
    call distruggi_pila_precedente
1:
    // aggiorniamo il puntatore alla pila sistema usata dal meccanismo
    // delle interruzioni
    movq PUNT_NUCLEO(%rbx), %rcx
    movq %rcx, tss_punt_nucleo

    movq RCX(%rbx), %rcx
    movq RDI(%rbx), %rdi
    movq RSI(%rbx), %rsi
    movq RBP(%rbx), %rbp
    movq RDX(%rbx), %rdx
    movq RAX(%rbx), %rax
    movq R8(%rbx), %r8
    movq R9(%rbx), %r9
    movq R10(%rbx), %r10
    movq R11(%rbx), %r11
    movq R12(%rbx), %r12
    movq R13(%rbx), %r13
    movq R14(%rbx), %r14
    movq R15(%rbx), %r15
    movq RBX(%rbx), %rbx
```

```
retq
```

- La funzione ha un problema: è stata chiamata, il suo indirizzo di ritorno è stato messo nella pila corrente, se il valore di *esecuzione* è stato modificato a un certo punto tra tutti i registri caricherà anche rsp. La pila punterà da un'altra parte, ma l'indirizzo di ritorno si trova nella vecchia pila. Dobbiamo complicarci un attimo la vita per spostare l'indirizzo di ritorno dalla pila vecchia a quella nuova.
- Verifico se il processo precedente è stato terminato o abortito. In quel caso la pila sistema può essere distrutta (se terminiamo un processo allora la pila sistema relativa al processo non sarà più utilizzata).
- Pongo in RBX, come prima, l'indirizzo al descrittore di processo

  ```
  movq esecuzione, %rbx
  ```

- Metto in RCX l'indirizzo di ritorno, lo levo dalla pila precedente.
- Il registro CR3 viene affrontato nel capitolo sulla paginazione.
- Modifico RSP mettendo il valore relativo alla nuova pila.
- Eseguo la POP per inserire nella nuova pila l'indirizzo di ritorno

  ```
  pushq %rcx
  ```

- A questo punto carico tutti i registri.
- In più, aggiorno *tss_punt_nucleo* nel TSS (ricordare quanto deciso sull'uso del TSS - univoco per tutti i processi, con necessità di modificare ogni volta il puntatore alla pila sistema nel TSS stesso).

5.8 Primitive

Il codice di una primitiva è sempre strutturato in due parti:

- una parte in Assembler dove eseguiamo all'inizio la *salva_stato* e alla fine la *carica_stato*;
- una routine scritta in C++, chiamata tra le due funzioni appena citate.

Il codice vero e proprio della primitiva è posto nella routine.

5.8.1 Funzione di utilità *liv_chiamante*

Funziona solo se è stata chiamata la *salva_stato*

La funzione *liv_chiamante* restituisce il livello di privilegio a cui si trovava il processore nel momento in cui è stata invocata la primitiva.

```
int liv_chiamante() {
    // salva_stato ha salvato il puntatore alla pila sistema
    // subito dopo l'invocazione della INT
    natq *pila = reinterpret_cast<natq*>(esecuzione->contesto[I_RSP]);

    // la seconda parola dalla cima della pila contiene il livello
    // di privilegio che aveva il processore prima della INT
    return pila[1] == SEL_CODICE_SISTEMA ? LIV_SISTEMA : LIV_UTENTE;
}
```

- La funzione recupera dal registro RSP nel contesto la pila.
- Dalla seconda *quadword* nella pila (indice 1) viene recuperato il *Current Privilege Level*.
- Abbiamo utilizzato alcune costanti poste in *include/costanti.h*

```
#define SEL_CODICE_SISTEMA        0x8
#define LIV_UTENTE        3
#define LIV_SISTEMA        0
```

Perchè non usare *punt_nucleo*? L'indirizzo in *punt_nucleo* è quello della base della pila. Noi non sappiamo quale sia la distanza tra la base e la *quadword* che ci interessa.

5.8.2 Analisi della struttura della primitiva *activate_p*

5.8.2.1 Parte Assembler

```
    .extern c_activate_p
a_activate_p:
    call salva_stato
    call c_activate_p
    call carica_stato
    iretq
```

Abbiamo il solito incapsulamento con *salva_stato* (eseguita prima della parte C++ della primitiva) e *carica_stato* (eseguita dopo la parte C++ della primitiva). Nulla di particolare rispetto a quanto già detto.

5.8.2.2 Parte C++

```
extern "C" void c_activate_p(void f(natq), natq a, natl prio, natl liv) {
    des_proc *p; // des_proc per il nuovo processo
    natl id = 0xFFFFFFFF; // id da restituire in caso di fallimento

    // non possiamo accettare una priorita'  minore di quella di dummy
    // o maggiore di quella del processo chiamante
    if (prio < MIN_PRIORITY || prio > esecuzione->precedenza) {
        flog(LOG_WARN, "priorita' non valida: %d", prio);
        c_abort_p();
        return;
    }
```

```
    // controlliamo che 'liv' contenga un valore ammesso
    // [segnalazione di E. D'Urso]
    if (liv != LIV_UTENTE && liv != LIV_SISTEMA) {
        flog(LOG_WARN, "livello non valido: %d", liv);
        c_abort_p();
        return;
    }

    // non possiamo creare un processo di livello sistema mentre
    // siamo a livello utente
    if (liv == LIV_SISTEMA && liv_chiamante() == LIV_UTENTE) {
        flog(LOG_WARN, "errore di protezione");
        c_abort_p();
        return;
    }

    // accorpiamo le parti comuni tra c_activate_p e c_activate_pe
    // nella funzione ausiliare crea_processo
    p = crea_processo(f, a, prio, liv, (liv == LIV_UTENTE));

    if (p != nullptr) {
        inserimento_lista(pronti, p);
        processi++;
        id = p->id; // id del processo creato
        // (allocato da crea_processo)
        flog(LOG_INFO, "proc=%d entry=%p(%d) prio=%d liv=%d", id, f, a, prio, liv);
    }

    esecuzione->contesto[I_RAX] = id;
}
```

- La *c_activate_p* è l'implementazione vera e propria della primitiva *activate_p*, scritta in C++.
  - Si osservi il primo parametro di ingresso

    `void f(natq)`

    esso è un puntatore a funzione, una funzione avente come unico parametro in ingresso *natq*. Porremo questo valore attraverso il secondo parametro di ingresso.
  - Vengono posti una serie di controlli per verificare la validità dei parametri in ingresso
    - Verifico se la priorità posta dall'utente col parametro *prio* sia minore della priorità minima e/o maggiore della priorità del processo attualmente in esecuzione. La prima cosa deve essere evitata, visto che non posso avere processi con priorità più bassa di *dummy*, la seconda è una forma di semplificazione.
    - Verifico se il livello posto con *liv* sia un valore valido.
    - Verifico che non venga richiesta l'attivazione di un processo a livello sistema da parte di un chiamante a livello utente (abbiamo già detto che possiamo creare processi di questo livello solo a livello sistema). Il livello del chiamante viene ottenuto con la funzione di utilità *liv_chiamante*.
- Alla fine creo effettivamente il processo con un'altra funzione, che possiamo ignorare (cit.)

```
p = crea_processo(f, a, prio, liv, (liv == LIV_UTENTE));
```

Se tutto va bene la funzione mi restituisce l'ID univoco del processo creato.

- Incremento *processo*, variabile globale che conteggia il numero di processi esistenti. Stampo un avviso con i dati di base relativi al processo.

- La primitiva deve restituire al chiamante l'id del processo chiamato. Questo è stato allocato dalla *crea_processo*, che lo ha posto nel descrittore di processo. Per restituirlo stiamo facendo in modo diverso da una *return*

  ```
  esecuzione->contesto[I_RAX] = id;
  ```

  Come mai?

  Perchè la *carica_stato* sovrascrive quanto posto se poniamo direttamente il valore in RAX. Il valore di *esecuzione*, non è stato modificato, dunque indica il processo che ha invocato l'*activate_p*. Ricordarsi la funzione del registro RAX nella traduzione C++-Assembler.

5.8.2.3 Invocazione della primitiva da parte di un utente

Poniamoci un'ulteriore domanda...

Come fa l'utente a chiamare la primitiva? Noi abbiamo parlato solo di funzioni che hanno prefisso *a* (Assembler) o *c* (C++), ma l'utente chiama cose senza prefissi. Sappiamo che dobbiamo eseguire un'istruzione INT. Noi nel codice poniamo una cosa del genere

```
activate_p(f, 10, 10, LIV_UTENTE);
```

dal punto di vista del C++ stiamo chiamando una normale funzione C++. Vediamo il codice che si cela dietro

```
    .global activate_p
activate_p:
    int $TIPO_A
    ret
```

Abbiamo una **funzione di appoggio** scritta in Assembler (per forza, non posso eseguire la INT in C++). L'identificativo del tipo dell'interruzione relativa alla primitiva deve essere noto: poniamo per comodità le costanti *TIPO_x* nel file *include/costanti.h*.

Effetto dell'esecuzione di INT? Lancio di *a_activate_p*. I parametri posti in ingresso non sono stati modificati e possono essere usati.

5.8.3 Esempio di primitiva creata da noi: *getprec*

Proviamo a creare una nostra primitiva, cosa che tipicamente si fa negli esercizi di esame.

Cosa dobbiamo fare? Facciamo una primitiva con cui un processo può farsi dire dal sistema qual è la sua precedenza.

- **Assegnazione di un identificativo al tipo di interruzione**.

  La prima cosa da fare è assegnare un numero. Vado in *costanti.h* e definisco una nuova costante con la direttiva define

  ```
  #define TIPO_GETPREC     0x28
  ```

  Ovviamente poniamo un numero che non è ancora utilizzato.

- **Aggiungo un gate nell'IDT**.

  Modifico il codice relativo al caricamento dell'IDT in *sistema.s*. Carico un gate

  ```
  carica_gate     TIPO_GETPREC     a_getprec     LIV_UTENTE
  ```

  chiaramente devo mettere *LIV_UTENTE*, altrimenti non potremo chiamare la primitiva.

- **Funzione di "incapsulamento" con *salva* e *carica* stato**.

  In *utenti.s* scriviamo *a_getprec*. La struttura è sempre la stessa

  ```
  a_getprec:
      call salva_stato
      call c_getprec
      call carica_stato
      iretq
  ```

- **Implementazione concreta della primitiva**.

  Scriviamo il succo della primitiva in C++ (cioè scriviamo la funzione *c_getprec* chiamata tra la *salva_stato* e la *carica_stato*).

  ```
  extern "C" void c_getprec() {
      esecuzione->contesto[I_RAX] = esecuzione->precedenza;
  }
  ```

  Chiaramente una cosa del genere può essere fatta solo in modalità sistema, unico modo per accedere ad *esecuzione* e a ciò che punta *esecuzione* stesso.

- **Funzione di appoggio**.

  No, l'utente non può ancora chiamare la primitiva in C++: manca la funzione di appoggio che esegue la INT. Andiamo in *include/sys.h*, dove dichiariamo la nostra primitiva

  ```
  extern "C" natl getprec();
  ```

  Poi, in *utente/utente.s*, scriviamo quanto segue

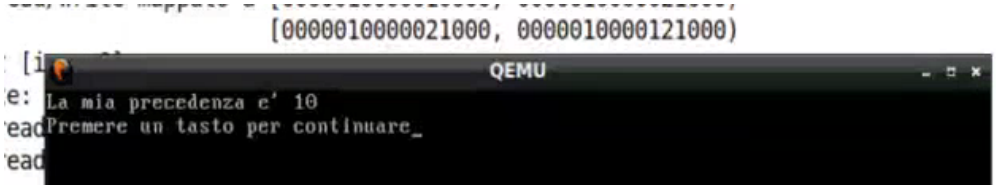
  ```
  getprec:
      int $TIPO_GETPREC
      ret
  ```

  Adesso abbiamo davvero finito.

Esempio di chiamata della primitiva

```
#include <all.h>
void f(natq a) {
    prinft("La mia precedenza e' %d\n", getprec());
    terminate_p();
}

void main() {
    activate_p(f, 10, 10, LIV_UTENTE);
    terminate_p();
}
```

L'output restituisce la precedenza indicata nell'apposito parametro di ingresso di *activate_p*.



Capitolo 6

Semafori

6.1 Problemi di mutua esclusione e di sincronizzazione

Gli utenti del nostro sistema possono creare processi e far eseguire funzioni scelte da loro. Questo utente può definire tanti processi; inoltre, questi processi hanno accesso a una memoria comune (le sezioni *text*, *data* e *bss*).

Altri problemi Questi processi possono mescolarsi fra loro e possono agire su strutture dati condivise. Come può l'utente garantire che tutto funzioni correttamente? Il problema è simile a quello del modulo sistema (ricordare l'esempio delle liste inconsistenti): li abbiamo risolto con l'atomicità (interruzioni esterne disabilitate, evitiamo eccezioni e chiamate ricorsive di primitive del sistema). Il fatto è che non possiamo permettere all'utente di disattivare le interruzioni, potrebbe non attivarle più e impedirci di riprendere il controllo. Classifichiamo le questioni.

- **Problemi di mutua esclusione**.

  L'utente deve fare una o più azioni su una stessa struttura dati, ma non vuole che queste azioni vengano fatte contemporaneamente (cioè non vuole che si mescolino con altre azioni). Si definiscono più processi che possono lavorare sulla stessa lista: queste devono essere mutuamente esclusive, se una cosa è in esecuzione allora l'altra non deve essere in esecuzione.

- **Problemi di sincronizzazione**.

  L'utente vuole che una certa azioni si verifichi sempre prima di un'altra. In presenza di un sistema multiprocesso può essere molto complicato (se non impossibile) capire a priori l'ordine in cui i processi saranno eseguiti. Esempio: definizione della struttura dati e lettura della struttura dati, la seconda operazione non ha senso finchè la prima operazione non viene conclusa (*produttore* vs *consumatore*).

Non ci addentriamo molto in queste questioni, vedremo solo ciò che ci serve. Attenzione a non confondere le due cose:

- nel problema di mutua esclusione non ci interessa l'ordine delle operazioni, ci basta solo non avere il mescolamento delle stesse;

- nel problema di sincronizzazione la questione è proprio l'ordinamento.

6.2 Soluzione ai problemi introdotti: le primitive semaforiche

Il sistemista non può risolvere a priori le questioni introdotte, visto che solo l'utente sa cosa vuole fare. Quello che faremo è fornire all'utente le cosiddette **primitive semaforiche**.

Metafora Diamo all'utente la possibilità di definire delle scatole di gettoni, cioè scatole che possono contenere dei gettoni. Posso fare solo due operazioni: inserire un gettone, prendere un gettone. La particolarità sta nella seconda operazione: la scatole è opaca, non sappiamo quanti gettoni ci sono dentro, se nel provare a prendere il getto non trovo nulla allora l'utente si congela e non fa altro, in attesa che qualcun altro aggiunga un gettone.

6.2.1 Risoluzione del problema della mutua esclusione

Prendiamo una situazione reale per capire meglio.

- Durante l'esame in presenza una sola persona alla volta va in bagno. L'ordine degli studenti non mi interessa, voglio solo evitare che due studenti vadano in bagno insieme.
- Risolviamo la cosa con una scatola avente un solo gettone: chi deve andare in bagno prende il gettone, e lo rimette dopo essere tornato.
- **Problemi di persone che prendono in contemporanea il gettone?**

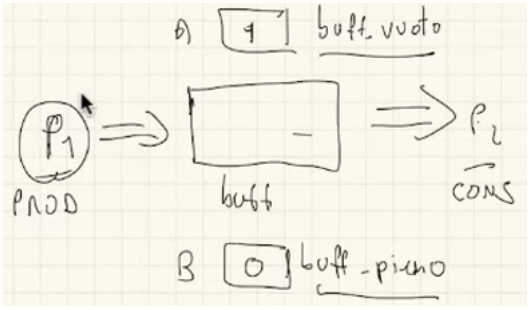
  No, risolto a priori grazie all'atomicità delle primitive di sistema.

Soluzione cooperativa La soluzione appena descritta, coi gettoni, è detta *soluzione cooperativa*. Questo perchè ci si aspetta che chi ha preso il gettone lo restituisca. Se ciò non avviene tutti gli altri non possono muoversi.

6.2.2 Risoluzione del problema della sincronizzazione

Cosa vogliamo fare Supponiamo di avere un'azione A e un'azione B. L'ordine nel complesso non ci interessa, ci basta che A venga eseguita prima di B, SEMPRE!

Esempio con un buffer Prendiamo come esempio un buffer: abbiamo un produttore $P1$, che inserisce un contenuto, e $P2$, consumatore, che lo deve utilizzare. Se $P2$ arriva prima si troverà ad elaborare dati casuali, e noi non vogliamo che ciò avvenga.



- Stavolta non ho una sola scatola, ma due!

- **Cosa c'è inizialmente in queste scatole?**

  Una ha un gettone e una è vuota.

- **Come si usano queste scatole?**

  Possiamo immaginare le due scatole come delle variabili logiche, che hanno come valore 0 o 1. Chiamiamo la scatola A *buff_vuoto* e la B *buff_pieno*.

  - La scatola A indica se il buffer è vuoto, cioè se $P2$ ha già letto o meno il contenuto presente.
  - La scatola B indica se il buffer è pieno, cioè se $P2$ ha del contenuto nuovo da leggere.

  Il produttore $P1$ può procedere solo se il buffer è vuoto, quindi con $A = 1$ e $B = 0$: segue che all'inizio A avrà un gettone e B non avrà gettoni. Il consumatore attende che il buffer si riempa, e agisce solo con $A = 0$ e $B = 1$. L'idea base è che $P1$ e $P2$, quando svolgono A e B (rispettivamente) levano il gettone dove è presente e lo spostano dove non c'è niente. Quindi

  - Il consumatore fa le seguenti cose

    ```
    PRENDERE(buff_pieno) ---> WAIT(buff_pieno)
    CONSUMARE
    AGGIUNGERE(buff_vuoto) ---> SIGNAL(buff_vuoto)
    ```

  - Il produttore fa le seguenti cose

    ```
    PRENDERE(buff_vuoto) ---> WAIT(buff_vuoto)
    PRODURRE
    AGGIUNGERE(buff_pieno) ---> SIGNAL(buff_pieno)
    ```

- Si osservi che risolvendo questo problema abbiamo affrontato pure la mutua esclusione. In generale i due problemi vanno affrontati singolarmente.

Capitolo 7

Implementazione dei semafori nel modulo sistema

7.1 Descrittore di semaforo *des_sem*

Il **descrittore di semaforo** è una struttura contenente le informazioni di un semaforo: un contatore (per ricordarci i gettoni presenti) e un puntatore alla lista dei processi in attesa del gettone.

```
struct des_sem {
    int counter;
    des_proc *pointer;
};
```

Le strutture dati sono poste nel modulo sistema, dunque sono inaccessibili all'utente se non invocando le primitive:

- l'accesso è controllato (l'utente non può fare cose che non ci aspettiamo);
- si ha atomicità (non si pone la questione della mutua esclusione).

7.2 Array dei semafori *array_dess*

I semafori non vengono mai deallocati, quindi possiamo allocarli sequenzialmente in un'array.

```
des_sem array_dess[MAX_SEM*2];
```

Numero massimo di semafori Creare un'array significa stabilire un numero massimo di semafori. La costante *MAX_SEM*, posta in *include/costanti.h*

```
#define MAX_SEM         1024UL
```

indica il numero massimo di semafori per la modalità utente e per la modalità sistema. L'array *array_dess* è composto da MAX_SEM $*$ 2 elementi:

- la prima metà dell'array è riservata ai semafori in modalità utente;

- la seconda metà ai semafori in modalità sistema.

Chiaramente non possiamo permettere l'accesso alla seconda parte dell'array quando ci troviamo in modalità utente.

Numero di semafori allocati Con due variabili teniamo a mente il numero di semafori allocati nella prima e nella seconda parte dell'array:

```
natl sem_allocati_utente = 0;
natl sem_allocati_sistema = 0;
```

Considerata l'allocazione sequenziale quanto memorizzato è sufficiente per trovare il primo semaforo non utilizzato.

7.3 Primitiva per l'allocazione del semaforo

7.3.1 Funzione di utilità *alloca_sem*

```
natl alloca_sem() {
    int liv = liv_chiamante();
    natl i;
    if(liv == LIV_UTENTE) {
        if(sem_allocati_utente >= MAX_SEM)
            return 0xFFFFFFFF;
        i = sem_allocati_utente;
        sem_allocati_utente++;
    }
    else {
        if(sem_allocati_sistema >= MAX_SEM)
            return 0xFFFFFFFF;
        i = sem_allocati_sistema + MAX_SEM;
        sem_allocati_sistema++;
    }
    return i;
}
```

- La funzione di utilità *alloca_sem* viene chiamata dalla primitiva *sem_ini*: essa restituisce l'indice del primo semaforo non utilizzato.
- Con la funzione di utilità *liv_chiamante* recuperiamo il livello di privilegio prima del lancio della primitiva.
- Tengo conto del valore di *MAX_SEM*, e verifico se ho raggiunto il numero massimo di semafori possibili relativamente a una modalità. Se ciò avviene restituisco

  `return 0xFFFFFFFF;`

  altrimenti restituisco l'indice del primo elemento array disponibile, e incremento la relativa variabile contatore.
- Ricordarsi che la prima parte dell'array è dedicata alla modalità utente, mentre la seconda alla modalità sistema.

7.3.2 Primitiva *sem_ini*

```
// parte "C++" della primitiva sem_ini
extern "C" void c_sem_ini(int val) {
    natl i = alloca_sem();

    if(i != 0xFFFFFFFF)
        array_dess[i].counter = val;

    esecuzione->contesto[I_RAX] = i;
}
```

- Per la creazione del semaforo utilizziamo la primitiva *sem_ini*. Il parametro di ingresso *val* contiene il numero di gettoni iniziali del semaforo.
- Con la funzione di utilità *alloca_sem* otteniamo l'indice del primo semaforo non utilizzato dell'array.
- Se la funzione non ha restituito il valore di errore manipolo il semaforo avente indice i: imposto il *counter* col valore del parametro di ingresso *val* e restituisco i aggiornando il contesto di *esecuzione*.

  ```
  esecuzione->contesto[I_RAX] = i;
  ```

7.4 Primitive per la gestione dei semafori

7.4.1 Funzione di utilità *sem_valido*

```
bool sem_valido(natl sem) {
    int liv = liv_chiamante();
    return sem < sem_allocati_utente || (liv == LIV_SISTEMA && sem - MAX_SEM < sem_allocati_sistema);
}
```

- Recupero il livello di privilegio antecedente la chiamata della primitiva con la funzione di utilità *liv_chiamante*.
- Uso il valore restituito (posto in *liv*) e il parametro di ingresso *sem* (identificativo di un semaforo) per verificare se questo è valido.
  - **Modalità utente**.
    Il semaforo è valido se *sem* è minore del numero di semafori allocati in modalità utente (*sem_allocati_utente*) (ci troviamo nella prima metà dell'array).
  - **Modalità sistema**.
    Il semaforo è valido se la differenza tra *sem* e *MAX_SEM* è minore del numero di semafori allocati in modalità sistema (*sem_allocati_sistema*, ci troviamo nella seconda parte dell'array). Dobbiamo anche verificare che il livello del chiamante sia di sistema.

7.4.2 Primitiva *sem_wait* (presa del gettone)

```
extern "C" void c_sem_wait(natl sem) {
    // una primitiva non deve mai fidarsi dei parametri
    if(!sem_valido(sem)) {
        flog(LOG_WARN, "semaforo errato: %d", sem);
        c_abort_p();
        return;
    }

    des_sem *s = &array_dess[sem];
    s->counter--;
    if(s->counter < 0) {
        inserimento_lista(s->pointer, esecuzione);
        schedulatore();
    }
}
```

- Eseguire questa primitiva è un po' come dire *prova a prendere un gettone dal semaforo*.
- Abbiamo come unico parametro in ingresso l'identificativo del semaforo *sem*.
- La prima cosa che facciamo è verificare la validità del semaforo utilizzando la funzione di utilità *sem_valido*. Nel caso in cui l'esito sia negativo si abortisce.
- Se l'esito della verifica è positivo si va avanti: si recupera l'indirizzo del relativo semaforo nell'array e si decrementa il contatore *counter* (cioè si leva un "gettone", lo faccio SEMPRE, il valore negativo ha un significato nella primitiva *sem_signal*).
- Nel caso in cui il contatore *counter* sia negativo il processo attualmente in esecuzione viene posto nella lista *pointer*, successivamente viene chiamata la funzione *schedulatore*.
- La funzione *schedulatore* rimuove la testa di *pronti* e ci porta ad eseguire altri processi. Il processo attuale non potrà essere eseguito finché non ci saranno "gettoni".

7.4.3 Primitiva *sem_signal* (rilascio del gettone)

```
extern "C" void c_sem_signal(natl sem) {
    // una primitiva non deve mai fidarsi dei parametri
    if(!sem_valido(sem)) {
        flog(LOG_WARN, "semaforo errato: %d", sem);
        c_abort_p();
        return;
    }

    des_sem *s = &array_dess[sem];
    s->counter++;
    if(s->counter <= 0) {
        des_proc *lavoro = rimozione_lista(s->pointer);
        inspronti(); // preemption
        inserimento_lista(pronti, lavoro);
        schedulatore(); // preemption
    }
}
```

- Abbiamo come unico parametro di ingresso l'identificativo del semaforo *sem*.
- La prima cosa che facciamo è verificare la validità del semaforo utilizzando la funzione di utilità *sem_valido*. Nel caso in cui l'esito sia negativo abortisco.
- Se l'esito è positivo si va avanti: si recupera l'indirizzo del relativo semaforo nell'array e si incrementa il contatore (cioè metto un "gettone", lo faccio SEMPRE).
- Se il contatore del semaforo è al più uguale a zero allora sono presenti processi che hanno cercato di prendere invano un gettone (e quindi sono in attesa). Quello che facciamo è estrarre uno di questi processi.
- Si ha un meccanismo di *prelazione*.
  - Rimuoviamo il processo dalla lista del semaforo, ci ricordiamo dove sta con una variabile temporanea.
  - Inserisco in testa alla lista *pronti* il processo attualmente in esecuzione con la funzione *inspronti* (se abbiamo lavorato bene il processo attualmente in esecuzione è quello con la priorità maggiore).
  - Inserisco nella lista *pronti* anche il processo appena estratto dalla lista del semaforo.
  - Chiamo *schedulatore*, l'esito sarà la scelta di uno dei due processi appena posti in *pronti*.

Perchè abbiamo messo il processo in esecuzione in pronti?

Chiediamoci: cosa succede se ho processi con la stessa priorità? Ricordiamo il codice di *inserimento_lista*: a parità di priorità diamo un'ordinamento FIFO (chi prima arriva esce per primo). Il fatto è che inserire il processo attualmente in esecuzione in lista *pronti* con *inserimento_lista* (e non con *inspronti*) significherebbe porlo in lista dopo altri processi aventi la stessa priorità, se presenti. Con il codice scritto ci assicuriamo che il processo attualmente in esecuzione non venga sospeso se sono presenti altri processi aventi la stessa priorità: è una sottigliezza, non è sbagliato fare diversamente.

Stati bloccati.
I semafori permettono l'implementazione dei cosiddetti *stati bloccati*: ogni semaforo rappresenta un possibile stato bloccato, dove l'attesa dipende dal diverso significato che attribuiamo a un certo semaforo.

Promemoria sull'atomicità delle primitive.
Le primitive semaforiche non possono essere usate all'interno di un'altra primitiva, ne romperebbero l'atomicità. La questione sarà affrontata nel capitolo sulle periferiche, dove introdurremo delle novità.

Capitolo 8

Premesse sulla paginazione

La paginazione è l'argomento che storicamente confonde di più gli studenti. Si tenga conto che l'argomento sarà affrontato anche nel corso di *Sistemi Operativi*.

8.1 Recap con prime questioni

- **Memory mapped I/O**.

  Lo spazio di indirizzamento include la RAM (che occupa una minima parte dello spazio), ma anche periferiche come APIC, memoria video in modalità testo, memoria video in modalità grafica. La cosa ci ha creato problemi relativamente alle periferiche *memory mapped*: la cache deve essere disattivata, non può memorizzare cacheline relativamente ad istruzioni di I/O. L'unica cosa che possiamo fare è distinguere le operazioni a partire dagli indirizzi.

- **Protezione**.

  Abbiamo diviso la memoria RAM in due parti: la parte $M1$ per il sistema e la parte $M2$ per l'utente. I meccanismi di protezione fanno sì che l'utente non possa accedere in alcun modo alla prima area di memoria (nè operazioni di lettura, nè di scrittura). Abbiamo dei problemi relativi all'**isolamento tra processi**. La protezione agisce anche sulle istruzioni di un programma: posso leggerle ma non modificarle. Inoltre, viene vietato l'utilizzo dell'indirizzo 0, poiché *nullptr* (anche in modalità sistema).

- **Multiprogrammazione**.

  Abbiamo anche detto che ogni volta che attraversiamo il gate ed effettuiamo un cambio di processo dobbiamo memorizzare il *contesto*. Il contesto non si limita ai soli registri, ma anche al contenuto della memoria. Se vogliamo fare i precisi diventa necessario spostare l'intera area di memoria nell'hard disk (cioè in qualcosa di più grande). Questa cosa funziona molto bene con memorie RAM piccole, ma adesso è problematica (oggi fare uno spostamento del genere significa lavorare con diversi GB).

8.2 Idea di base: tenere insieme informazioni di più processi

Abbiamo bisogno di un meccanismo che minimizzi gli spostamenti tra RAM e hard disk, limitandoci al minimo indispensabile. L'idea è tenere in memoria RAM le informazioni relative a più processi, cioè trattare la RAM come una sorta di cache dell'hard disk. Si considerino le seguenti problematiche.

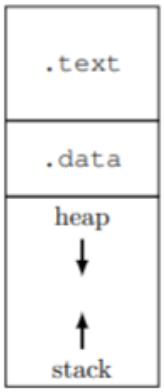
- **Dimensione massima di ogni processo**

  La prima cosa che dobbiamo chiederci è di quanto spazio ha bisogno ogni processo. Dobbiamo tenere conto delle seguenti aree:

  – *text* (codice programma) e *data* (variabili globali), dimensione nota al collegatore e costante per tutta l'esecuzione del programma;

  – pila e *heap*, aree di memoria (inizialmente vuote) che possono espandersi durante l'esecuzione del programma.

  L'area di memoria di un processo è organizzata così



  heap e pila stanno in un'area unica, con direzioni di espansione opposte.

  Soluzione. Quello che faremo è porre dimensione massima per l'area dedicata a *heap* e pila. Solitamente si pone un valore di default.

- **Isolamento tra i processi**

  Abbiamo detto che l'idea base è mettere insieme, nella RAM, più processi. Senza il ricorso al meccanismo della protezione il processo in esecuzione potrebbe intervenire sullo stato dei processi non in esecuzione.

  Soluzione. L'idea risolutiva potrebbe essere l'aggiunta di due nuovi registri nella CPU: LINF (limite inferiore) ed LSUP (limite superiore). Segue l'inserimento di questi due registri nel *contesto* in *des_proc*.

  – Chiaramente i due registri sono scrivibili solo a livello sistema.

  – Se ci troviamo a livello utente la CPU controlla che ogni accesso memoria sia compreso tra LINF ed LSUP, altrimenti lancia un'eccezione.

- Ogni volta che un processo viene caricato dall'hard disk il sistema deve inizializzare gli appositi campi *contesto* con l'indirizzo finale e iniziale della parte di memoria occupata dal processo.
- Ogni volta che si cambia processo si aggiorna il contenuto di LINF ed LSUP nel processore, prendendo come valori quelli relativi al processo entrante.

Attenzione: questa soluzione non è quella definitiva (quando introdurremo la paginazione vedremo perchè non servono questi registri).

- **Caricamento a indirizzi variabili**

  L'indirizzo dove andiamo a caricare il processo non è più chiaro come prima: il processo può essere caricato ovunque e la sua posizione dipende anche dagli altri processi posti nella RAM. L'indirizzo dove carichiamo il processo non è quindi noto durante compilazione e collegamento.

  - **Grosso problema**. Se io rimuovo un processo e lo metto in un punto diverso della memoria gli indirizzi relativi al processo non sono più validi, dovrei correggerli rispetto alla nuova posizione: questa cosa non è possibile, poichè gli indirizzi sono indistinguibili.
  - **Soluzione**. La soluzione è utilizzare il registro LINF come indirizzo di base. L'indirizzo x in un processo è diverso dall'indirizzo x in un altro processo: nel programma ci limitiamo a indicare l'offset, che si somma a LINF per ottenere l'indirizzo vero e proprio a cui puntare.

  Attenzione: questa soluzione non è quella definitiva (quando introdurremo la paginazione vedremo perchè non serve il registro LINF e come facciamo a ricondurci al cosiddetto *indirizzo fisico*).

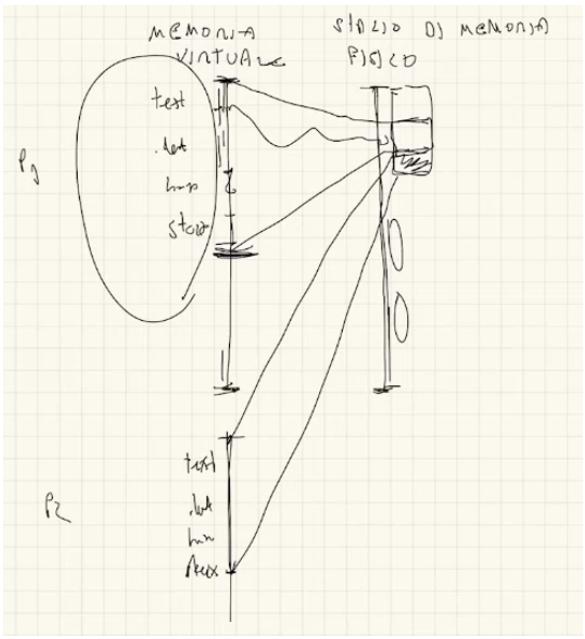
8.3 Step successivo: memoria virtuale e indirizzi virtuali

Riflettiamo un po' di più sull'ultima questione affrontata: quella degli indirizzi variabili. Fino ad oggi abbiamo utilizzato solo ed esclusivamente i cosiddetti **indirizzi fisici**. Adesso introduciamo gli **indirizzi virtuali** e la *memoria virtuale*.

- Il codice del programma relativo a un processo utilizza esclusivamente *indirizzi virtuali*.
- Questi *indirizzi virtuali* sono relativi al processo, cioè l'indirizzo x di un presunto processo $P1$ non sarà uguale all'indirizzo x di un processo $P2$.
- Se consideriamo la soluzione provvisoria per gli indirizzi otteniamo quello fisico così

$$\text{Indirizzo fisico} = \text{LINF} + \text{Indirizzo virtuale}$$

Cioè? Dobbiamo distinguere lo *spazio di indirizzamento fisico* (dove troviamo RAM e qualunque periferica) dallo *spazio di indirizzamento virtuale*. I processi non hanno alcun accesso allo spazio di indirizzamento fisico: vedono solo lo spazio di indirizzamento virtuale, che è una sorta di mondo immaginario dove troviamo la *memoria virtuale*. In questa memoria sono presenti le solite parti: *text*, *data*, *heap* e *stack*.



L'hardware, attraverso una unità che vedremo più avanti, trasforma gli indirizzi di *text*, *data*, *heap* e *stack* in indirizzi dello spazio di memoria fisico (*un po' come i film Western dove di una casa c'è solo la facciata*, cit.).

8.3.1 Scopo del Kernel

Quello che avremo è una situazione dove il programma si interfaccia con una CPU e una RAM virtuali.

- La CPU è il contesto che abbiamo salvato nel descrittore di processo, che verrà caricato nella CPU vera (si aggiorna lo stato della CPU fisica in divisione di tempo, ma si rappresenta lo stato della CPU virtuale).
- La memoria RAM non viene gestita in divisione di tempo (come ipotizzato all'inizio), ma in divisione di spazio: la RAM contiene, in contemporanea, più immagini di processi diversi.

Tutte queste cose sono affrontate dal *kernel*.

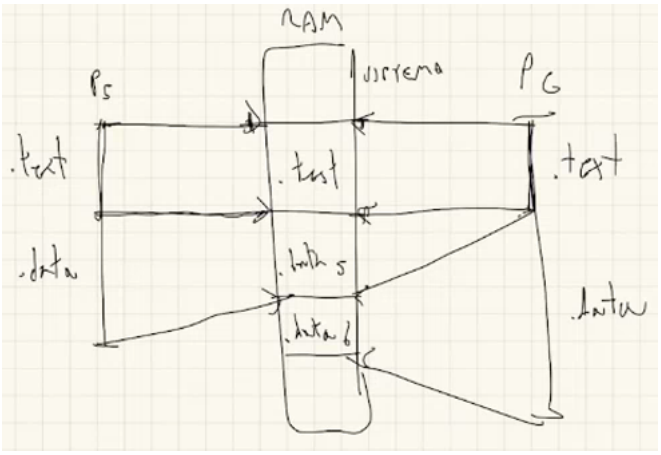
Definizione di *kernel* su Wikipedia *Un kernel, in informatica costituisce il nucleo o core di un sistema operativo, ovvero il software che fornisce un accesso sicuro e controllato dell'hardware ai processi in esecuzione sul computer. Dato che possono eventualmente esserne eseguiti simultaneamente più di uno, il kernel può avere anche la responsabilità di assegnare una porzione di tempo-macchina (scheduling) e di accesso all'hardware a ciascun programma (multitasking).*

8.4 Tutto finito?

Il meccanismo di cui abbiamo parlato fino ad ora presenta due svantaggi:

1. Come possiamo condividere parte della memoria tra due o più processi?
2. Cosa succede se dobbiamo caricare un processo, ma lo spazio in memoria è frammentato in porzioni troppo piccole? Questo problema potrebbe essere risolto
   (a) ricompattando lo spazio (ma questo significherebbe copiare un'intera porzione di memoria da una zona a un'altra), oppure
   (b) spezzare la memoria di un processo in più porzioni e gestire ciascuna di queste separatamente.

La soluzione (b) è la più interessante, visto che permette di risolvere anche il primo problema.



Tutto questo ci porta a parlare della **paginazione**, dove parti diverse dello spazio di indirizzamento di un processo si traducono in modo diverso (il passaggio da indirizzi virtuali a indirizzi fisici).

Capitolo 9

Paginazione

Tradurre gli indirizzi virtuali in indirizzi fisici singolarmente è una cosa veramente improponibile.

Cosa facciamo?

- Dividiamo lo spazio di indirizzamento virtuale in regioni naturali dette *pagine*.
- Dividiamo lo spazio di indirizzamento fisico in regioni naturali dette *frame*.

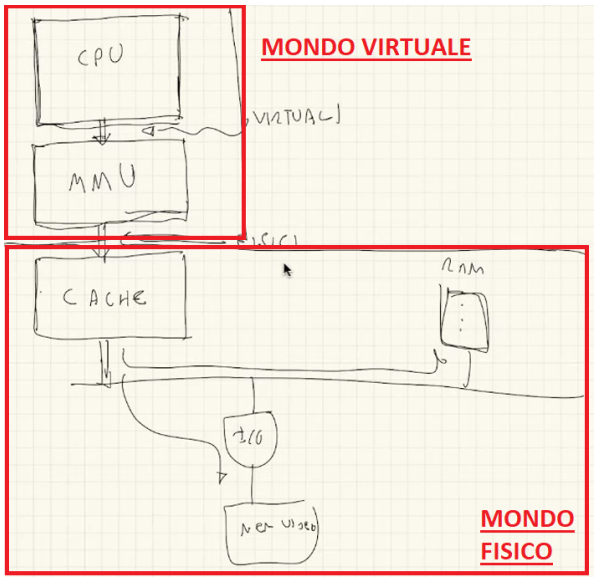
Pagine e frame hanno la stessa dimensione: 4 KiB (0x1000 in esadecimale). La pagina viene inserita in un frame e si identifica perfettamente con esso: non esiste una pagina posta tra frame diversi.

Traduzione degli indirizzi in una pagina Tutti gli indirizzi posti in una pagina sono tradotti in maniera contigua (la traduzione effettiva avviene nei bit più significativi, non nell'offset).

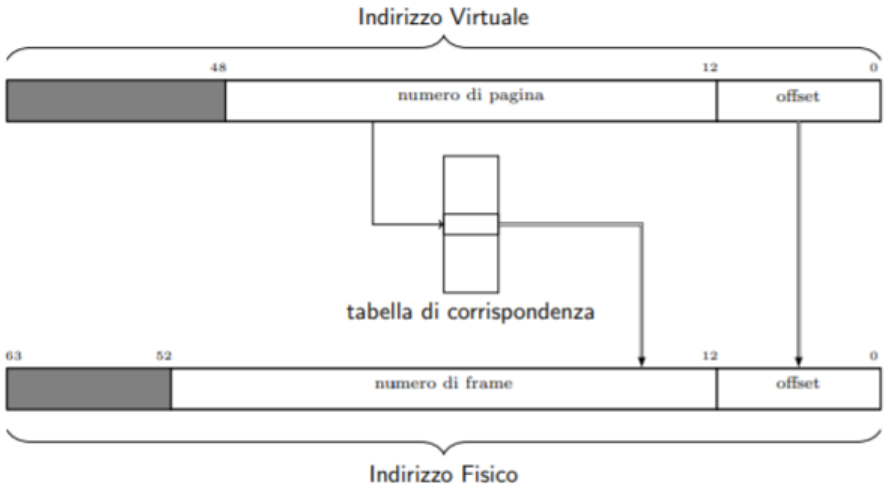
9.1 *Memory Management Unit* (MMU)

Abbiamo introdotto l'organizzazione in pagine, ma non abbiamo ancora detto come gestiremo il passaggio da indirizzo virtuale a indirizzo fisico, tenendo conto che la traduzione dovrà essere diversa per ogni pagina. Introduciamo l'**unità di gestione della memoria** tra CPU e Cache.

- Si osservi che la MMU non ha un buco come lo spazio di indirizzamento: negli indirizzi presenti si ha un salto dall'ultimo indirizzo prima del buco al primo indirizzo dopo il buco.
- La cache fa parte del mondo fisico, può ignorare l'introduzione della *Memory Management Unit* (MMU). La MMU traduce gli indirizzi virtuali in base alla traduzione attiva in un certo istante: la struttura dati utilizzata dalla MMU può essere immaginata come un insieme di *tabelle di corrispondenza* (una per processo). Queste tabelle sono un esempio di struttura dati condivisa tra hardware e software: quest'ultimo (il kernel) stabilisce quali tabelle ci sono e quali sono attive.



9.1.1 Passaggio da indirizzo virtuale a indirizzo fisico



- Nell'indirizzo indicato dalla CPU si distingue l'*offset* (bit meno significativi) dal *numero di pagina* (bit più significativi).
- Il numero di pagina va in ingresso nella tabella di corrispondenza relativa al processo: l'indice posto in ingresso mi restituisce il corrispondente *numero di frame*.
- Ogni tabella di corrispondenza ha una riga per ogni possibile pagina (non poche, visto il numero di bit riservati al numero di pagina).

Attenzione al numero di bit Gli indirizzi fisici fanno riferimento allo spazio di indirizzamento fisico, quelli virtuali allo spazio di indirizzamento virtuale. Il numero di bit massimo è 64, varie generazioni di CPU implementano indirizzi virtuali su un numero inferiore di bit. La questione dipende dal modello della CPU, tutto lì (cit.).

9.1.2 Contenuto delle tabelle di corrispondenza

Ciascuna riga delle tabelle di corrispondenze è grande 8 byte: il numero di frame si trova nella posizione in cui si troverebbe all'interno dell'indirizzo (per comodità del programmatore, in questo caso si va dal bit 12 al bit 52). Le righe non contengono solo il corrispondente *numero di frame*, ma anche una serie di flag (di cui non ci interessa la posizione):

- un flag U/S che mi indica se la pagina può essere acceduta solo a livello utente o anche a livello sistema (quindi distinguo attraverso questo flag i frame relativi all'area riservata al livello sistema dai rimanenti[1]);
- un flag R/W che mi indica se la scrittura nella pagina è permessa o no;
- un flag P (presenza) che mi dice se la traduzione è valida (quindi se l'indirizzo corrisponde, al di la del livello di privilegio). Si utilizza per marcare le pagine che il processo non usa (che avranno $P = 0$);
  - Il flag P può essere utilizzato per vietare l'utilizzo dell'indirizzo 0 (mi basta porre 0 nella pagina relativa). Nel caso di accesso a una pagina con $P = 0$ il processore solleva un'eccezione *page fault* (fault perchè il sistema potrebbe aggiustare lo stato della memoria e rieseguire l'istruzione attraverso la *paginazione su domanda*, che vedremo a Sistemi Operativi[2]).
- i flag PWT (*Page Write Through*) e PCD (*Page Cache Disable*), con cui la MMU trasmette dei comandi alla cache (il secondo ordina alla cache di non intercettare l'operazione, il primo chiede di utilizzare alla cache di usare la politica *write-through*).
  - Il flag PCD è utile per tutte le pagine che contengono indirizzi di registri di I/O mappati in memoria, invece che locazioni di memoria (per esempio l'APIC).
  - Il flag PWT è utile per quanto riguarda la parte di indirizzi relativa alla memoria video: vogliamo che la scrittura vera arrivi nella memoria video, e che non rimanga in cache.

Flag per la paginazione su domanda Abbiamo i bit A e D che sono legati all'implementazione della paginazione su domanda. La MMU setta il bit A di una entrata quando la MMU accede alla relativa entrata per fare una traduzione, mentre setta il bit D se l'accesso all'indirizzo relativo alla pagina era in scrittura.

Bit NX Ulteriore bit è il *Not executable*. Se durante l'operazione di fetch di un'istruzione abbiamo $NX = 1$ per la relativa pagina allora il processore genera un'eccezione.

9.2 MMU e modulo sistema

Facciamoci una domanda: siamo obbligati a mappare gli indirizzi relativi al modulo sistema nello spazio di indirizzamento virtuale dei vari processi? Sì: se siamo a livello utente e sollevo un'eccezione la CPU inizia a fare un po' di cose, come consultare l'IDT e il TSS, o scrivere

[1] Viene meno la necessità di un registro che indica l'indirizzo limite tra area riservata al sistema e il resto.
[2] Uno dei pochi esempi di *fault* effettivamente risolti

dentro la pila sistema. Tutti questi indirizzi saranno tradotti dalla MMU, dunque in assenza di mapping la gestione delle interruzioni non è possibile

Cosa faremo La cosa più semplice è mappare tutta la memoria di sistema: quando siamo a livello utente e avviene un passaggio a livello sistema la CPU troverà tutto ciò che le serve (visto che tutto è mappato). In questa parte di indirizzamento mapperemo tutta la zona di RAM che contiene il codice e le strutture dati presenti nel modulo sistema. Cosa importante

Tutto è mappato agli stessi indirizzi in ogni processo

Tabelle di corrispondenza e cambio di processo Se il sistema vuole fare un cambio di processo allora deve cambiare la tabella di corrispondenza. Se non facciamo ciò l'*Instruction pointer* va avanti interpretando gli indirizzi in modo inaspettate (traduco col contenuto della tabella di corrispondenza del vecchio processo, non con quella del processo in esecuzione).

Aspetto complicato Scrivere codice sistema essendo costantemente consapevoli della differenza tra indirizzi fisici e indirizzi virtuali. Lavorando a livello utente questo ignora che i suoi indirizzi sono virtuali: gli utenti usano solo questi e non hanno interesse sugli indirizzi fisici.

9.3 Esempio: indirizzi virtuali e fisici in un programma in esecuzione

Scriviamo un programma dove dichiariamo una variabile e ne stampiamo l'indirizzo

```
#include <iostream>

int var;
int main() {
     std::cout << "&var: "<< &var << "\n";
}
```

Compiliamo

```
g++ -o myprog.cc -no-pie myprog.cc
./myprog
```

Otteniamo

```
&var: 0x404174
```

L'indirizzo stampato è l'indirizzo scelto dal collegatore: compilatore, collegatore e programmatore sono consapevoli solo dell'esistenza di questo indirizzo virtuale (per noi è l'unico indirizzo vero).

Chiediamo l'indirizzo fisico Se abbiamo livelli di privilegio sufficienti possiamo chiedere l'indirizzo fisico invece di quello virtuale. Modifichiamo il programma per mantenerlo in vita

```
#include <iostream>

int var;
int main() {
     std::cout << "&var: "<< &var << "\n";

     for(;;) {
          sleep(1);
     }
}
```

dove *sleep* è l'equivalente unix della *delay*, col numerino che esprime i secondi. Eseguiamo il programma e mentre questo cicla chiediamo col terminale il processor identifier

```
pgrep myprog
```

successivamente eseguiamo la seguente istruzione

```
sudo ./virt_to_phys 1922 0x404174 // verra' richiesta la password
```

otteniamo l'indirizzo fisico **0x49512174**, completamente diverso dall'indirizzo virtuale (tranne che nell'offset, le tre cifre esadecimali meno significative costituiscono l'offset all'interno della pagina).

- La variabile var si trova nella pagina 404 all'offset 174.
- La MMU trasforma il numero di pagina nel numero di frame 495121, non l'offset.

Proviamo ad eseguire un'altra istanza del programma, senza chiudere la precedente. Con le stesse cose fatte prima otteniamo l'indirizzo fisico **0x4ed18174**: non è cambiato l'offset (174), ma è cambiato il numero di frame (4ed18). Ognuno vive nel suo mondo senza conoscere l'esistenza dell'altro.

9.4 MMU all'opera

Nelle diapositive seguenti viene spiegato nel dettaglio il comportamento della MMU con le sue *tabelle di corrispondenza*.

- Viene introdotto un programma in C++ che agisce su uno spazio di indirizzamento virtuale. Si suppone, per ragioni di semplicità, che lo spazio di indirizzamento virtuale sia di 32 KiB (gli indirizzi vanno da 0000 a 7fff).
- Lo spazio di indirizzamento viene diviso in 8 pagine, ciascuna da 4 KiB (nulla di nuovo).
- Le pagine 0 e 1 sono riservate al sistema.
- La pagina 2 contiene il codice del programma (da 2000 a 2fff).
- Le pagine 3 e 4 contengono la variabile *buf*.
- La pagina 7 viene usata come pila.

Successivamente viene introdotto un secondo processo, più piccolo. Le diapositive mostrano come la MMU operi con due processi.

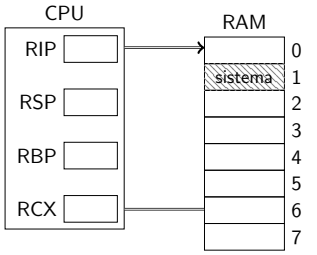
Supponiamo di dover eseguire il seguente programma:

```
char buf[0x2000] = { 2, 6, -1, 200, ...,
  15, 3, -32, 1};
int main()
{
  int sum = 0;
  for (int i = 0; i < 0x2000; i++)
    sum += buf[i];
  return sum;
}
```

L'array buf contiene una serie di numeri di cui vogliamo conoscere la somma.

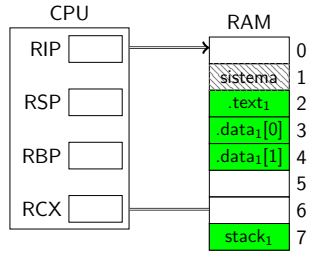
Una possible traduzione in assembly e linguaggio macchina è:

```
                .text
                .global main
2000  main:     pushq %rbp                  55
2001              movq %rsp, %rbp           48 89 e5
2004              subq $8, %rsp             48 83 ec 08
2008              movl $0, -8(%rbp)         c7 45 f8 00 00 00 00
200f              movl $0, -4(%rbp)         c7 45 fc 00 00 00 00
2016  for:        cmpq $0x2000, -4(%rbp)    48 81 7d fc 00 20 00 00
2013              jge fine                  7d 14
2020              movslq -4(%rbp), %rcx     48 63 4d fc
2024              movsbl buf(%rcx), %eax    0f be 81 00 30 00 00
202b              addl %eax, -8(%rbp)       01 45 f8
202e              addl $1, -4(%rbp)         83 45 fc 01
2032              jmp for                   eb e2
2034  fine:       movl -8(%rbp), %eax       8b 45 f8
2037              popq %rbp                 5d
2038              ret                       c3
                .data                       ...
3000  buf:        .byte 2, 6, -1, 200       02 06 ff c8
                  ...                       ...
4ffc              .byte 15, 3, -32, 1       0f 03 e0 01
```



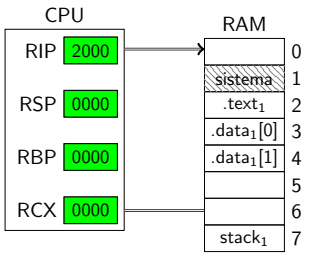

```
2000  55
2001  48 89 e5
2004  48 83 ec 08
2008  c7 45 f8 00 00 00 00
200f  c7 45 fc 00 00 00 00
2016  48 81 7d fc 00 20 00 00
2013  7d 14
2020  48 63 4d fc
2024  0f be 81 00 30 00 00
202b  01 45 f8
202e  83 45 fc 01
2032  eb e2
2034  8b 45 f8
2037  5d
2038  c3
      ...
3000  02 06 ff c8
      ...
4ffc  0f 03 e0 01
```

Vediamo prima l'esecuzione su un sistema non multiprogrammato. Mostriamo solo alcuni dei registri della CPU. Tutti i numeri saranno mostrati in esadecimale. Supponiamo che tutto lo spazio di memoria sia occupato dalla RAM.



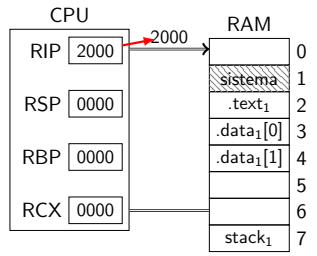

```
2000  55
2001  48 89 e5
2004  48 83 ec 08
2008  c7 45 f8 00 00 00 00
200f  c7 45 fc 00 00 00 00
2016  48 81 7d fc 00 20 00 00
2013  7d 14
2020  48 63 4d fc
2024  0f be 81 00 30 00 00
202b  01 45 f8
202e  83 45 fc 01
2032  eb e2
2034  8b 45 f8
2037  5d
2038  c3
      ...
3000  02 06 ff c8
      ...
4ffc  0f 03 e0 01
```

Carichiamo il programma in memoria.



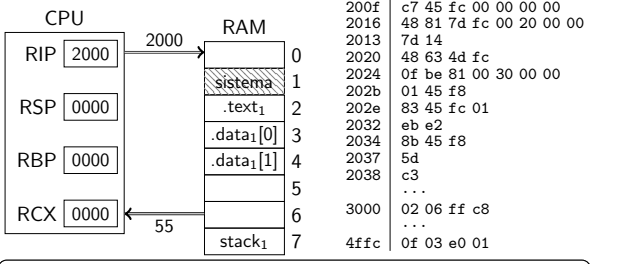

```
2000  55
2001  48 89 e5
2004  48 83 ec 08
2008  c7 45 f8 00 00 00 00
200f  c7 45 fc 00 00 00 00
2016  48 81 7d fc 00 20 00 00
2013  7d 14
2020  48 63 4d fc
2024  0f be 81 00 30 00 00
202b  01 45 f8
202e  83 45 fc 01
2032  eb e2
2034  8b 45 f8
2037  5d
2038  c3
      ...
3000  02 06 ff c8
      ...
4ffc  0f 03 e0 01
```

Inizializziamo tutti i registri



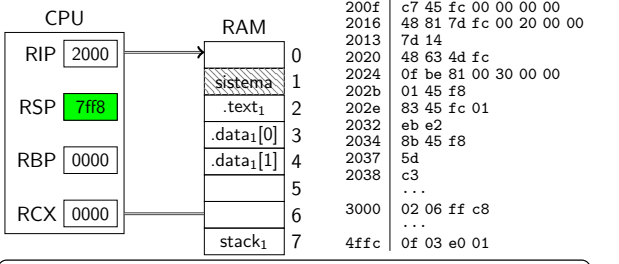

```
2000  55
2001  48 89 e5
2004  48 83 ec 08
2008  c7 45 f8 00 00 00 00
200f  c7 45 fc 00 00 00 00
2016  48 81 7d fc 00 20 00 00
2013  7d 14
2020  48 63 4d fc
2024  0f be 81 00 30 00 00
202b  01 45 f8
202e  83 45 fc 01
2032  eb e2
2034  8b 45 f8
2037  5d
2038  c3
      ...
3000  02 06 ff c8
      ...
4ffc  0f 03 e0 01
```

Avviamo il sistema. La CPU tenta di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia quindi una operazione di lettura in memoria all'indirizzo 2000.



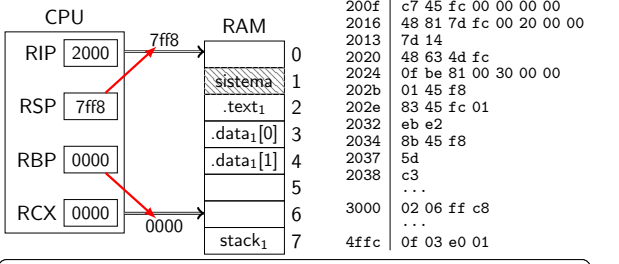
2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

L'operazione di lettura restituisce l'istruzione `pushq %rbp`.



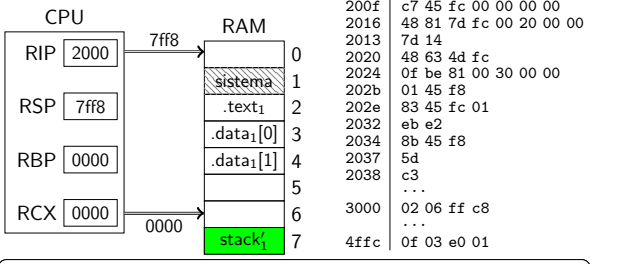
2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

La CPU inizia ad eseguire l'istruzione. Il primo passo è decrementare RSP di 8.



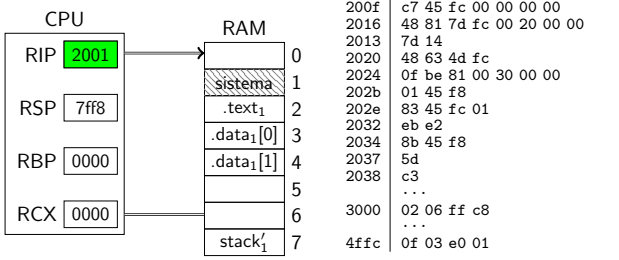
2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

Il secondo passo è scrivere il contenuto di RBP all'indirizzo contenuto in RSP. La CPU inizia quindi una operazione di scrittura all'indirizzo 7ff8.



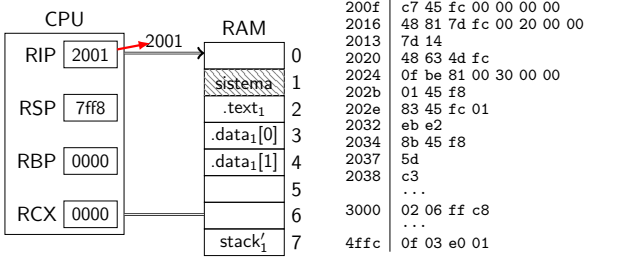
2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

Dopo la scrittura il contenuto dello stack sarà cambiato.



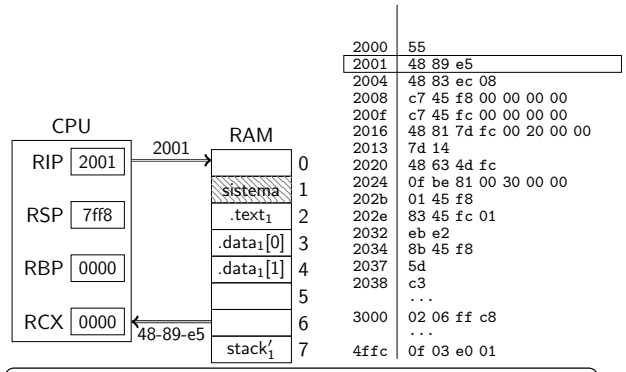
2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

La CPU ha completato l'esecuzione dell'istruzione `pushq %rbp` e può passare alla successiva. Incrementa RIP di 1 (che è la dimensione dell'istruzione appena terminata).

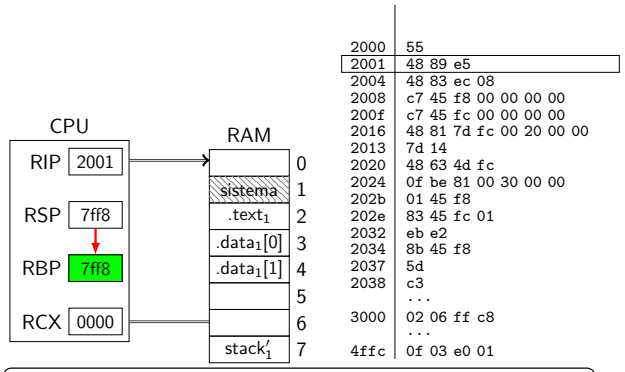


2000	55
2001	48 89 e5
2004	48 83 ec 08
2008	c7 45 f8 00 00 00 00
200f	c7 45 fc 00 00 00 00
2016	48 81 7d fc 00 20 00 00
2013	7d 14
2020	48 63 4d fc
2024	0f be 81 00 30 00 00
202b	01 45 f8
202e	83 45 fc 01
2032	eb e2
2034	8b 45 f8
2037	5d
2038	c3
	...
3000	02 06 ff c8
	...
4ffc	0f 03 e0 01

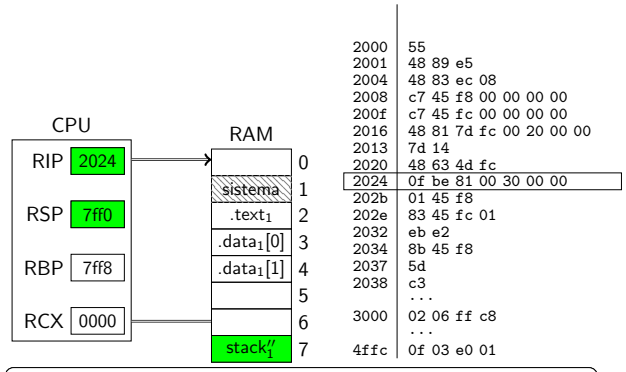
Nuovo ciclo: la CPU tenta di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia una operazione di lettura in memoria all'indirizzo 2001.



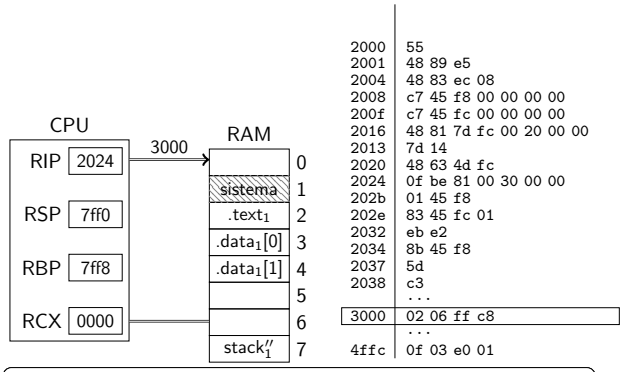
La CPU riceve l'istruzione `movq %rsp, %rbp`.



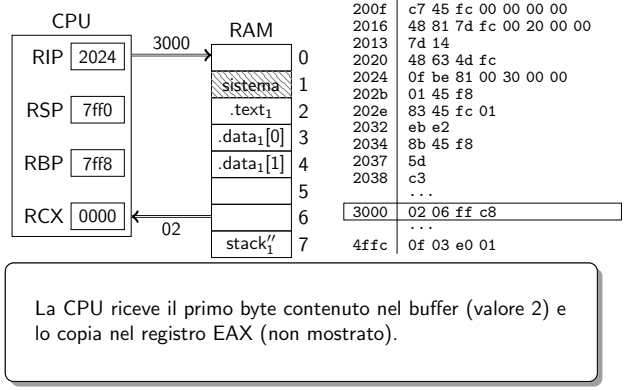
Per eseguirla copia il contenuto di RSP in RBP. L'istruzione non prevede accessi in memoria.



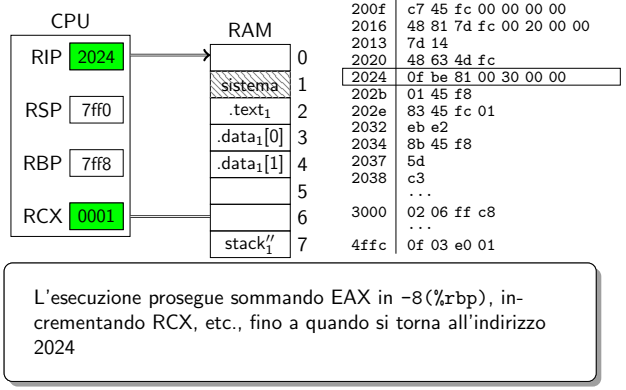
Dopo alcune istruzioni, l'esecuzione del programma arriva all'istruzione "`movsbl buf(%rcx), %rax`" che si trova all'indirizzo 2024. Questa è la prima istruzione che accede alla sezione `.data` del programma.



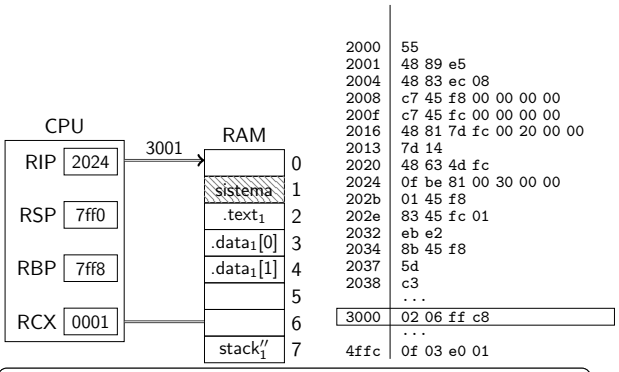
La CPU somma il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3000. Inizia dunque una operazione di lettura a questo indirizzo.



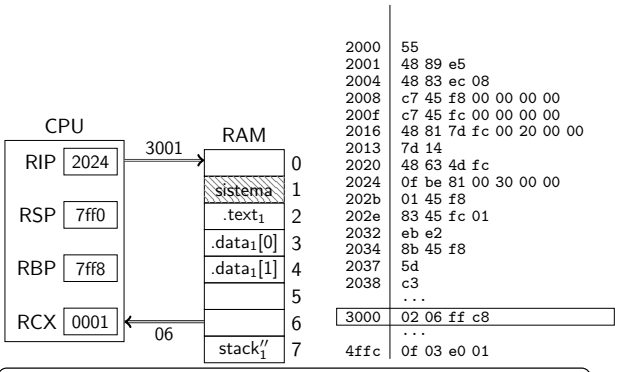
La CPU riceve il primo byte contenuto nel buffer (valore 2) e lo copia nel registro EAX (non mostrato).



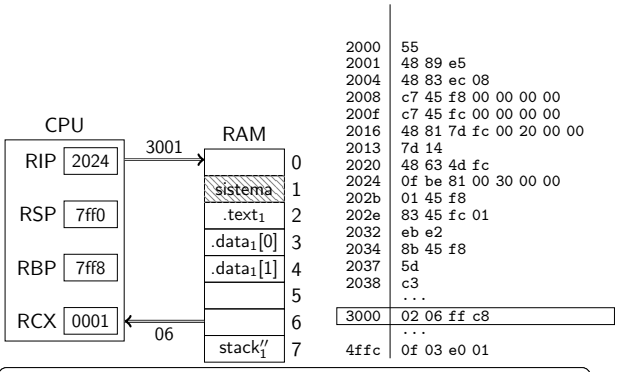
L'esecuzione prosegue sommando EAX in `-8(%rbp)`, incrementando RCX, etc., fino a quando si torna all'indirizzo 2024



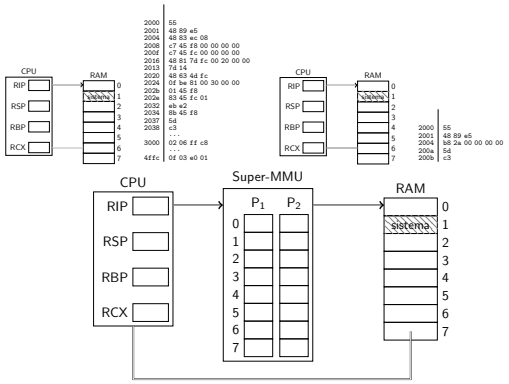
La CPU somma il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3001. Inizia dunque una operazione di lettura a questo indirizzo.



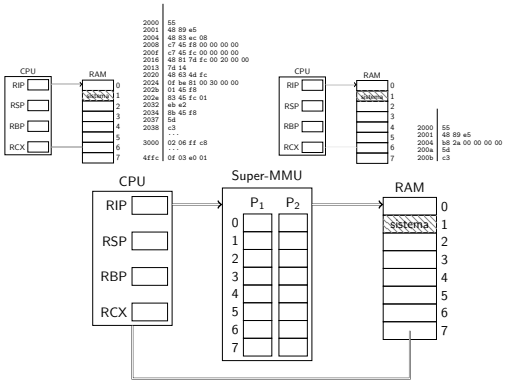
La cpu legge il valore 6 e lo somma al registro EAX (non mostrato).



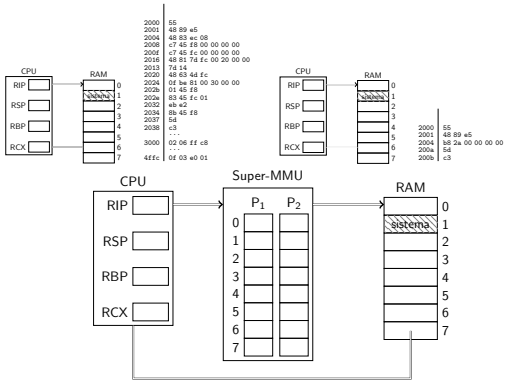
L'esecuzione prosegue in questo modo, sommando tutti i valori di `buf`.



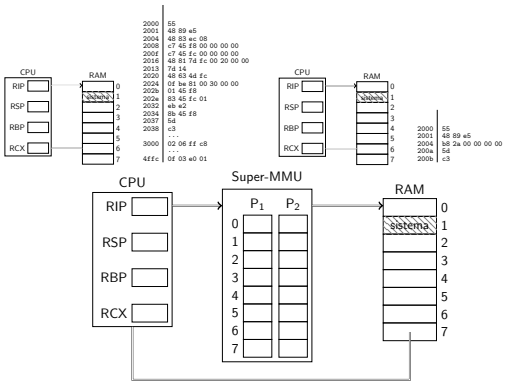
Vediamo ora un esempio di esecuzione con più processi e paginazione.



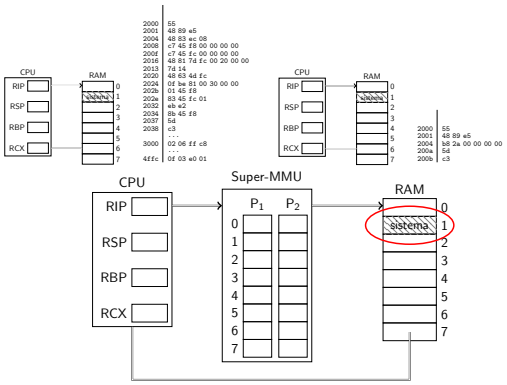
Possiamo immaginare che ciascun processo sia eseguito da un distinto sistema virtuale, con una sua CPU e una sua RAM.



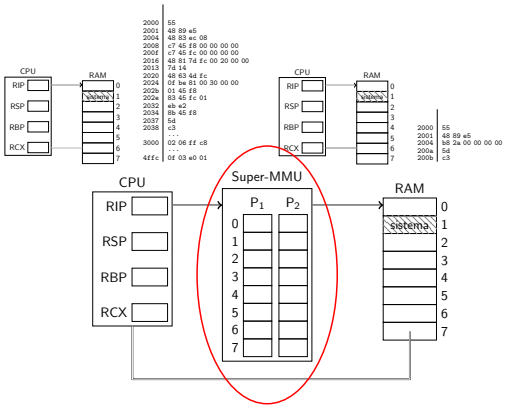
Nella figura, in alto, mostriamo a sinistra lo stato del sistema virtuale che esegue P_1, e a destra quello di un diverso processo P_2.



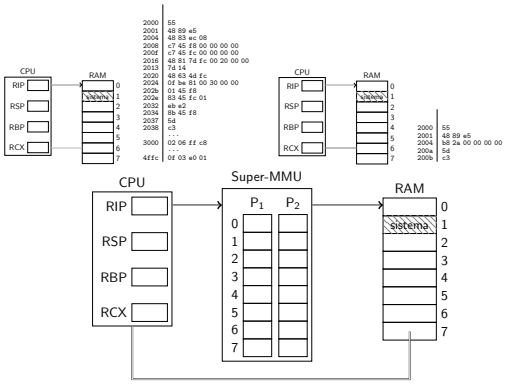
In basso mostriamo invece il sistema fisico, che deve emulare l'evoluzione dei due sistemi virtuali, a divisione di tempo.



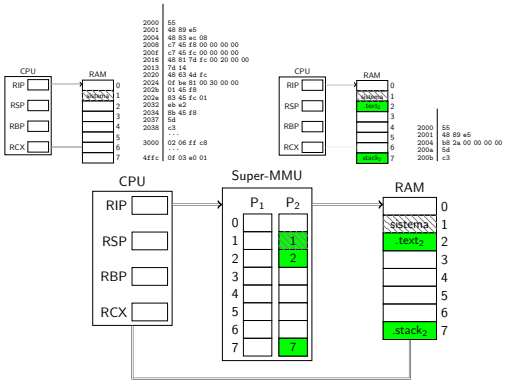
Lo stato delle due CPU virtuali è memorizzato nei descrittori di processo, nella memoria di sistema.



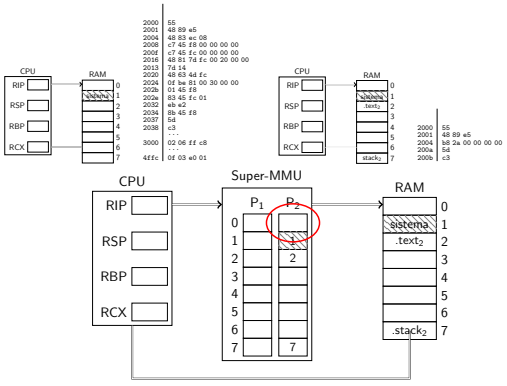
La MMU verrà invece usata per emulare le due memorie virtuali.



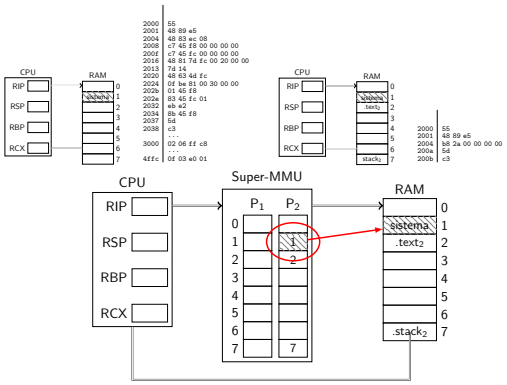
Vogliamo che l'evoluzione del sistema virtuale di P_1 sia identica a quella che abbiamo appena visto.



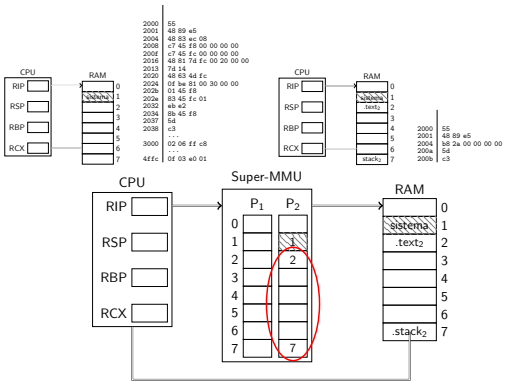
Carichiamo la memoria del processo P_2 e prepariamo la sua tabella di corrispondenza.



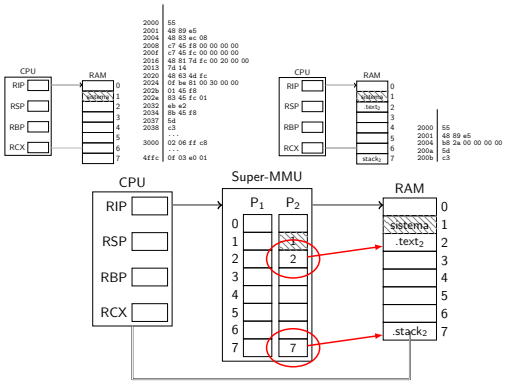
La pagina 0 è lasciata con P=0, per intercettare le dereferenziazioni di puntatori nulli.



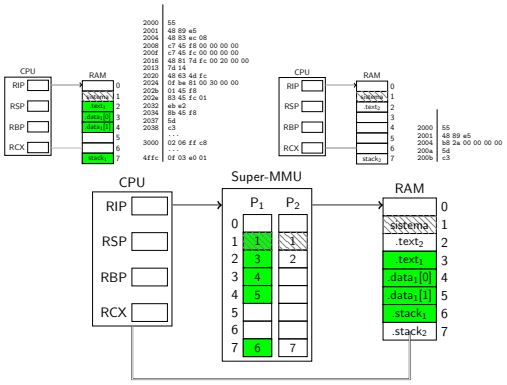
La pagina 1 corrisponde al frame 1, che contiene il sistema.
È marcata come inaccessibile da livello utente.



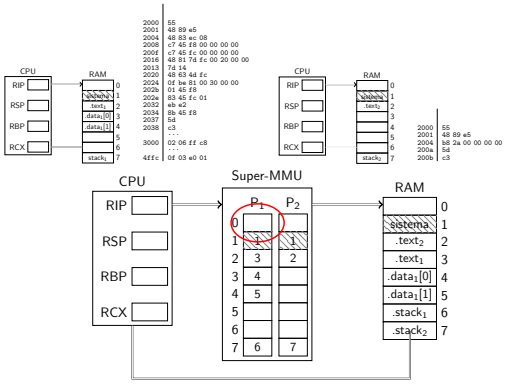
Le pagine da 3 a 6 non sono usate da P_2 (P=0)



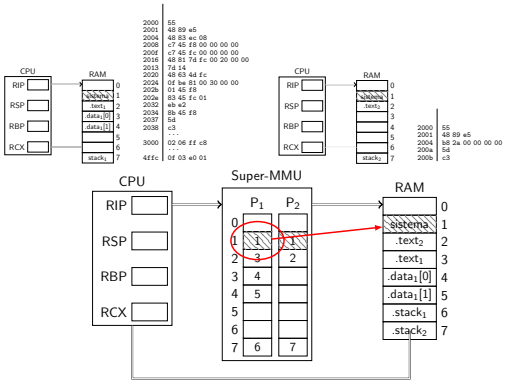
Le altre pagine corrispondono ai frame che le contengono.



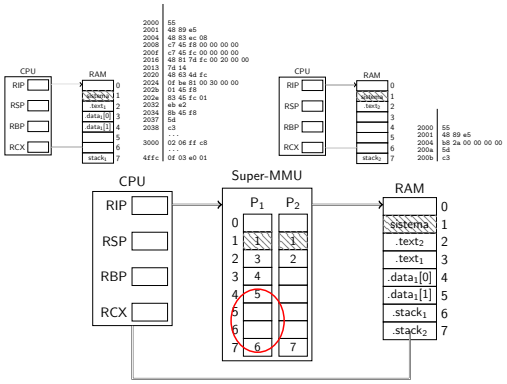
Carichiamo la memoria del processo P_1 nello spazio che rimane e prepariamo anche la sua tabella.



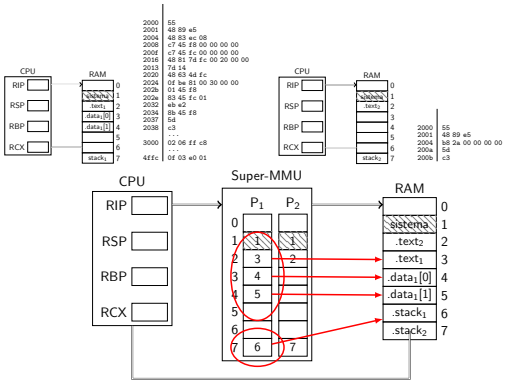
Anche per P_1 La pagina 0 è lasciata con P=0 . . .



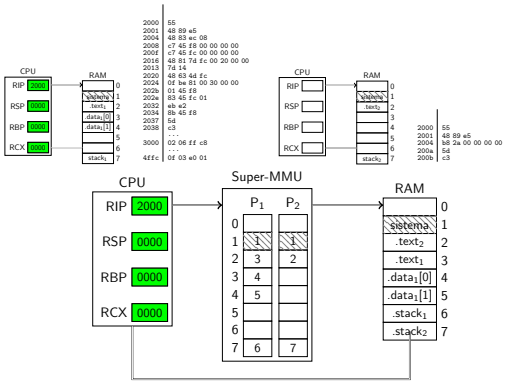
. . . e la pagina 1 corrisponde al frame 1, ma è inaccessibile da livello utente.



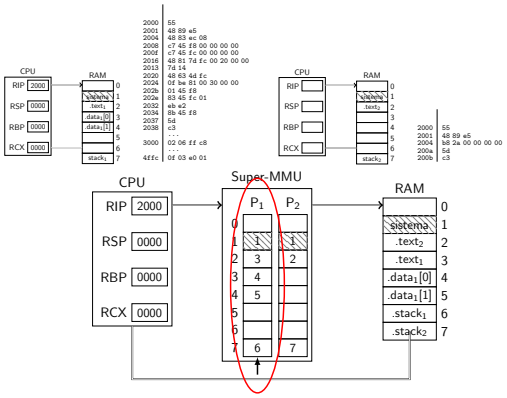
Le pagine 5 e 6 non sono usate (P=0) ...



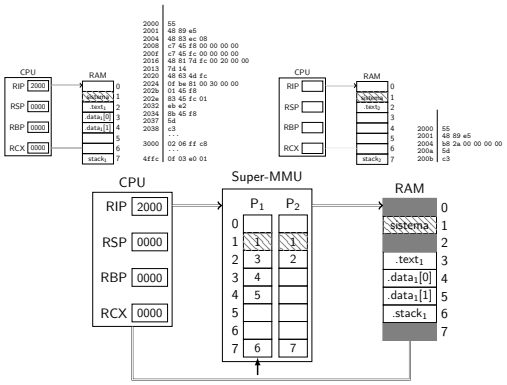
... e le altre corrispondono ai frame che le contengono.



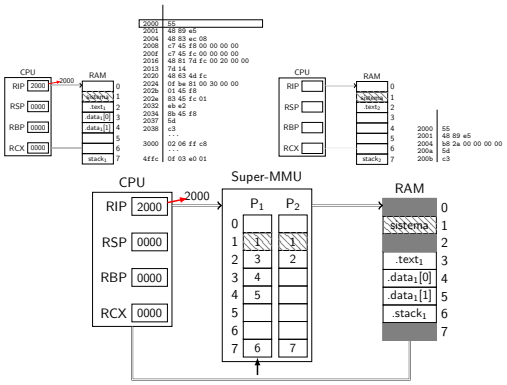
Supponiamo che il sistema metta in esecuzione P_1, caricando lo stato dei registri...



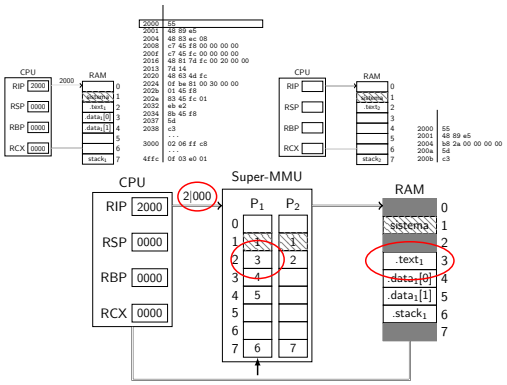
... e rendendo attiva la tabella P_1.



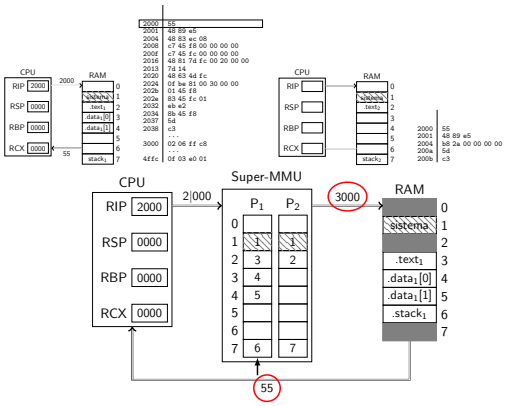
Tutte le pagine che non sono nel codominio di P_1 diventano inaccessibili.



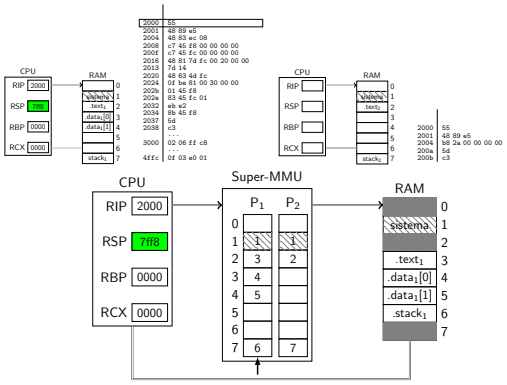
P_1 comincia la sua esecuzione. La CPU fisica esegue una lettura all'indirizzo 2000, esattamente come quella virtuale.



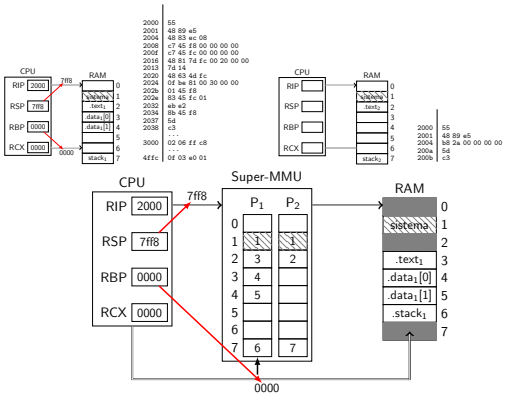
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (2) e offset (000). Consulta quindi l'entrata numero 2 della tabella di corrispondenza e trova il corrispondente numero di frame (3)



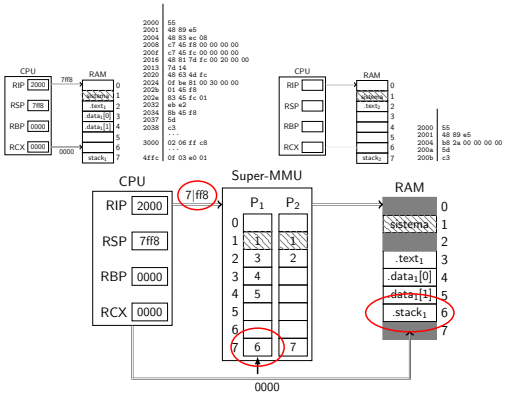
L'accesso viene completato e la CPU fisica riceve `pushq %rbp`, esattamente come la virtuale, e inizia ad eseguirla.



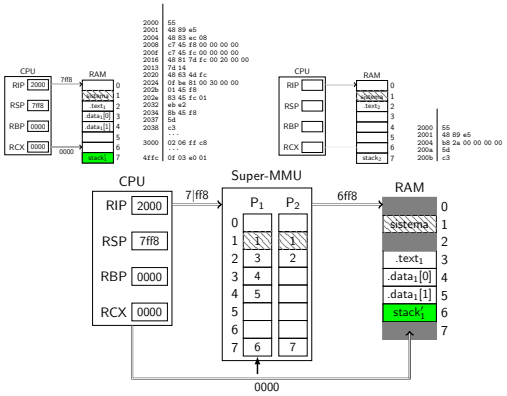
Le due CPU, fisica e virtuale, fanno esattamente le stesse cose. Come primo passo decrementano RSP di 8.



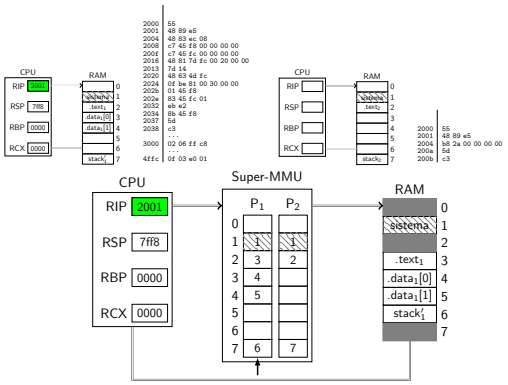
Come secondo passo voglino scrivere il contenuto di RBP all'indirizzo contenuto in RSP. Le due CPU iniziano quindi una operazione di scrittura all'indirizzo 7ff8.



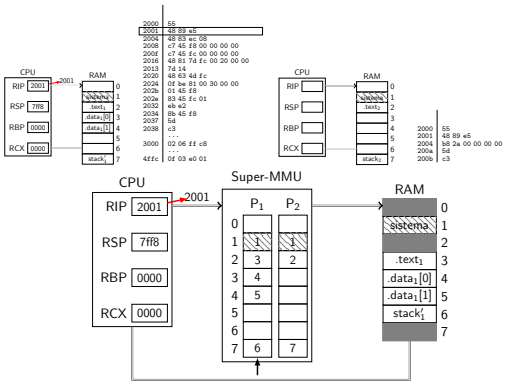
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (7) e offset (ff8). Consulta quindi l'entrata numero 7 della tabella di corrispondenza e trova il numero di frame (6)



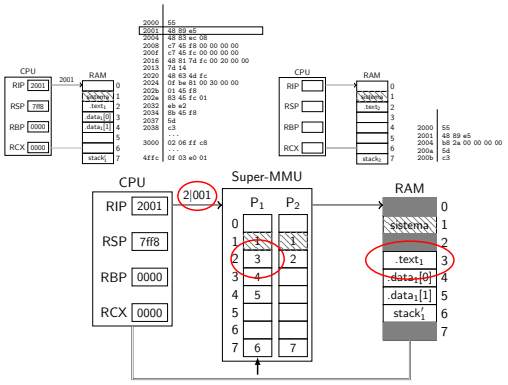
L'operazione di scrittura viene completata, dopo aver trasformato l'indirizzo. Il contenuto del frame 6 del sistema fisico continua ad essere uguale a quello della pagina 7 del sistema virtuale.



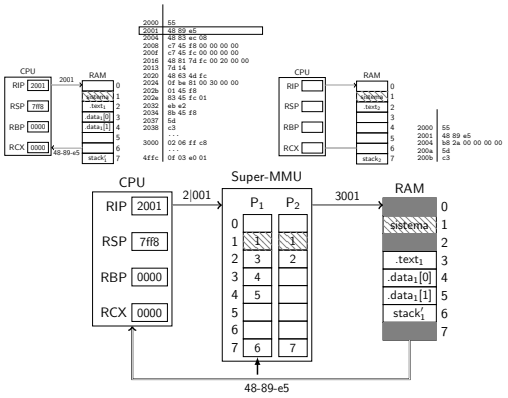
Le due CPU hannoo completato l'esecuzione dell'istruzione `pushq %rbp` e possono passare alla successiva. Incrementano RIP di 1 (che è la dimensione dell'istruzione appena terminata).



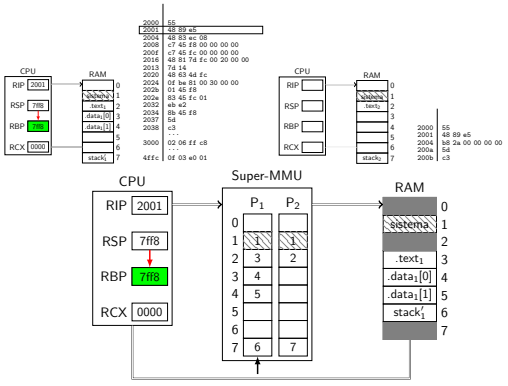
Nuovo ciclo: le CPU tentano di prelevare l'istruzione all'indirizzo contenuto in RIP. Inizia una operazione di lettura in memoria all'indirizzo 2001.



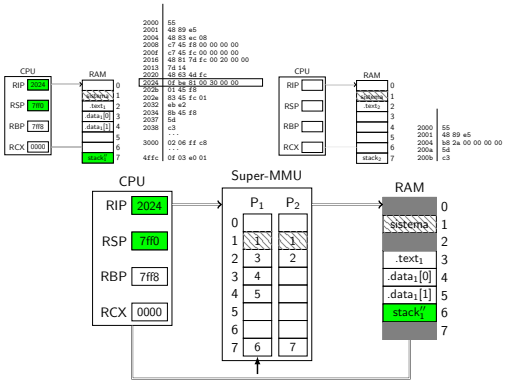
Come al solito, la MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina virtuale (2) e offset (001).



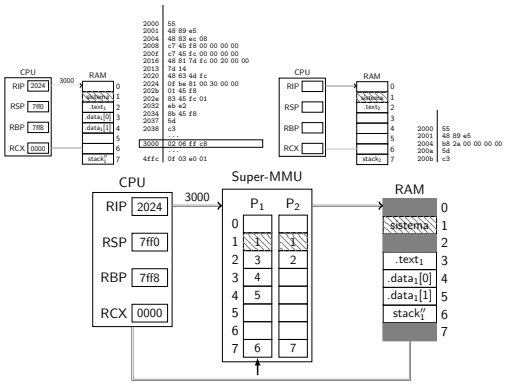
L'accesso viene completato, ovviamente dopo aver trasformato l'indirizzo. Entrambe le CPU ricevono l'istruzione `movq %rsp, %rbp`.



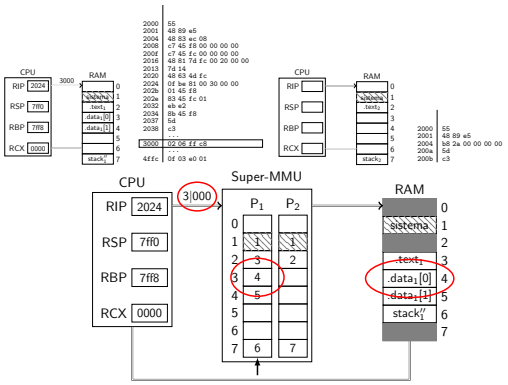
Per eseguirla copiano il contenuto di RSP in RBP. Dovremmo a questo punto convincerci che gli stati fisico e virtuale procedono di pari passo.



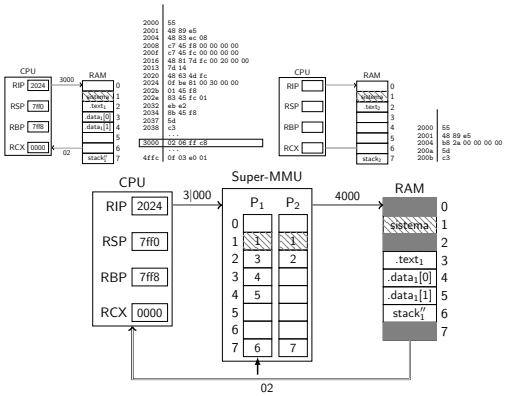
Dopo alcune istruzioni, l'esecuzione del programma arriva all'istruzione `movsbl buf(%rcx), %rax` che si trova all'indirizzo (virtuale) 2024.



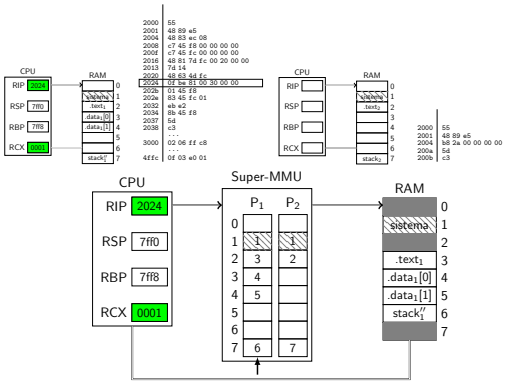
Entrambe le CPU sommano il contenuto di RCX e la costante 3000 (buf nel sorgente), ottenendo l'indirizzo 3000. Iniziano dunque una operazione di lettura a questo indirizzo.



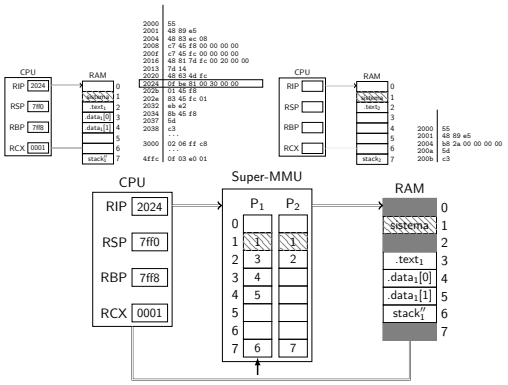
Ancora una volta la MMU scompone l'indirizzo in numero di pagina (3) e offset (000). L'entrata numero 3 della tabella di corrispondenza dice che la pagina 3 si trova nel frame 4.



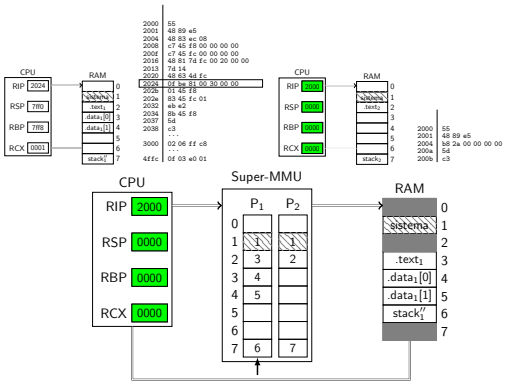
Viene completato l'accesso, sempre dopo aver trasformato l'indirizzo. Ancora una volta entrambe le CPU ricevono lo stesso valore e proseguono di pari passo.



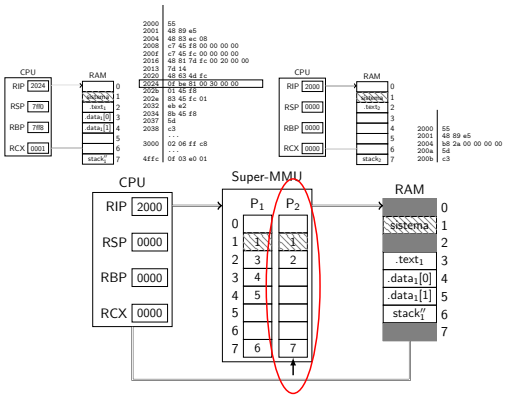
L'esecuzione prosegue aggiungendo al totale il valore precedentemente letto e incrementando `%rcx`, fino a quando si ritorna all'indirizzo 2024.



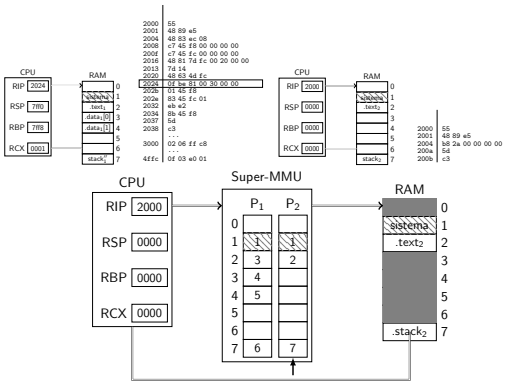
Supponiamo che a questo punto, prima di eseguire l'istruzione, il sistema fisico accetti una interruzione con cambio di processo e vada in esecuzione P_2



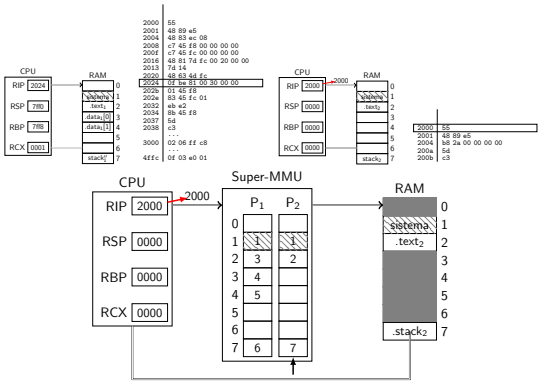
Il sistema carica i registri di P_2...



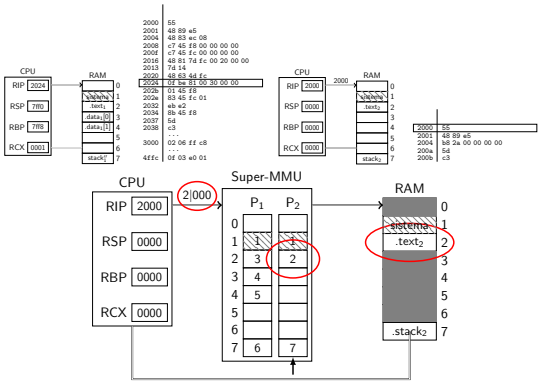
. . . e attiva la tabella P_2



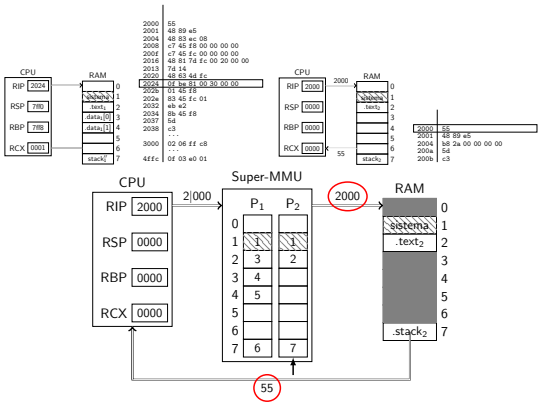
Diventano accessibili le pagine nel codominio di P_2 e non accessibili tutte le altre. Il sistema virtuale di P_1 è "congelato".



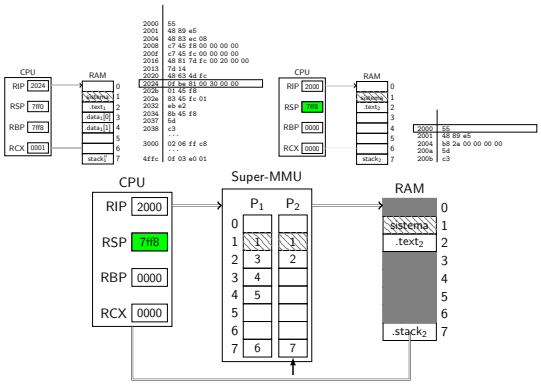
P_2 comincia la sua esecuzione. La CPU fisica e la CPU virtuale di P_2 eseguono una lettura all'indirizzo 2000.



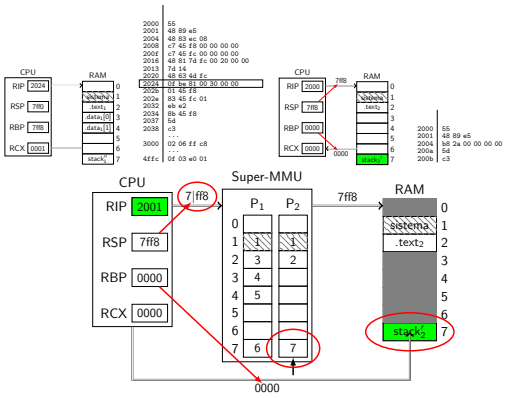
La MMU intercetta l'operazione e scompone l'indirizzo in numero di pagina (2) e offset (000). Consulta quindi l'entrata numero 2 della tabella di corrispondenza e trova il corrispondente numero di frame (2)



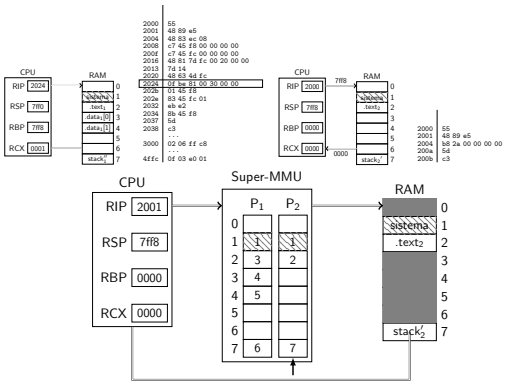
L'accesso viene completato dopo aver tradotto l'indirizzo. La prima istruzione di P_2 è una `pushq %rbp`.



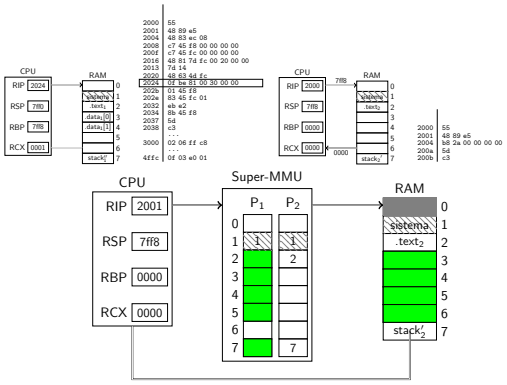
Per eseguirla, entrambe le CPU decrementano RSP di 8 . . .



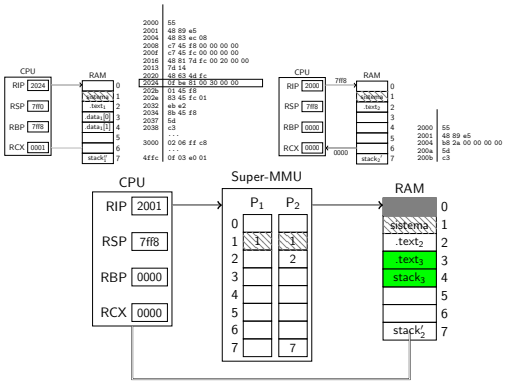
. . . e scrivono il contenuto di RBP all'indirizzo contenuto in RSP. La MMU traduce opportunamente l'indirizzo (in questo caso resta invariato)



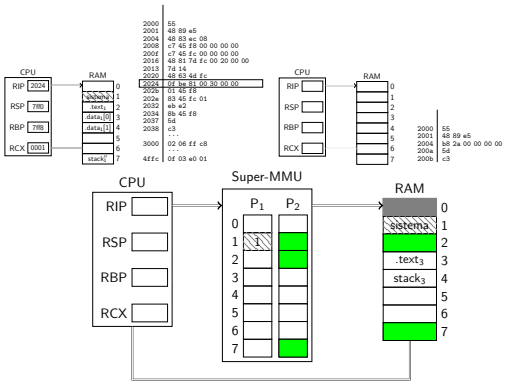
Supponiamo che ora che il sistema fisico venga interrotto nuovamente decida di caricare un altro processo, P_3.



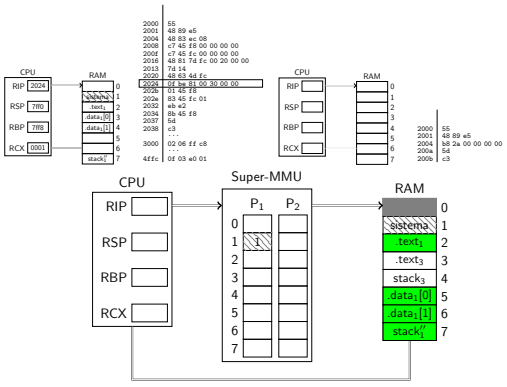
Per fare spazio, il sistema rimuove le pagine di P_1 dopo averle copiate nello swap (operazione di *swap-out*.



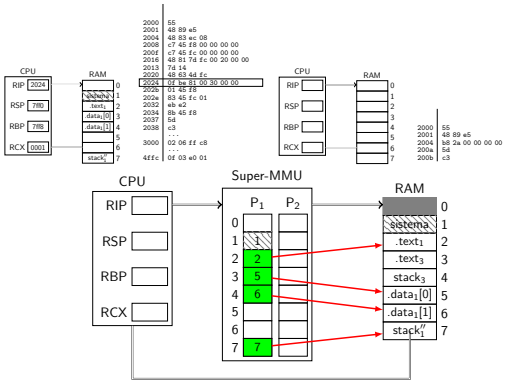
Quindi carica P_3 e inizializza opportunamente la sua tabella di corrispondenza (non mostrata).



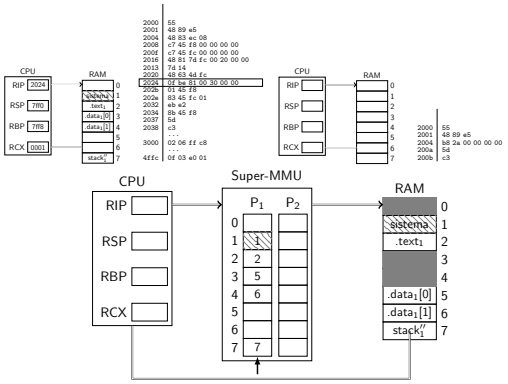
Successivamente, P_2 termina. Il sistema libera tutte le sue pagine.



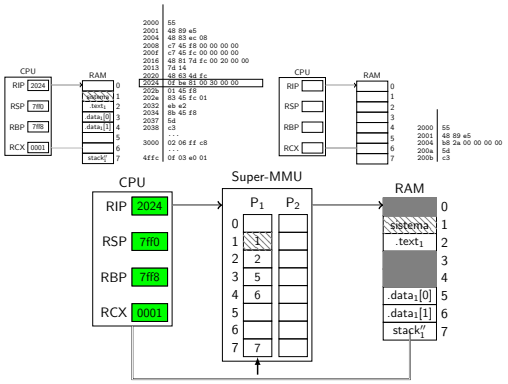
Ancora dopo, il sistema decide di ricaricare P_1 per rimetterlo in esecuzione. Le pagine di P_1 non occupano più i frame che occupavano in precedenza.



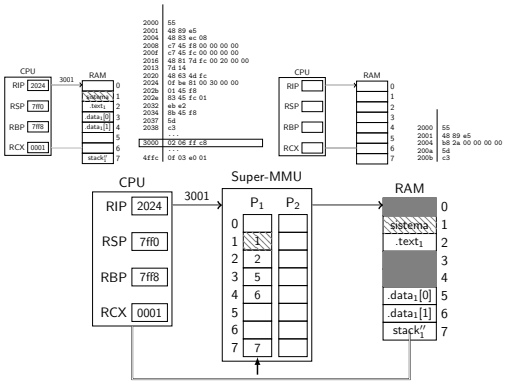
Il sistema inizializza la tabella di corrispondenza di P_1 con i nuovi numeri di frame.



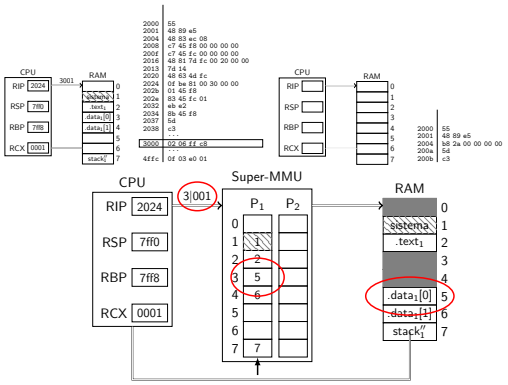
Attiva la tabella di P_1. Le pagine di P_3 diventano inaccessibili.



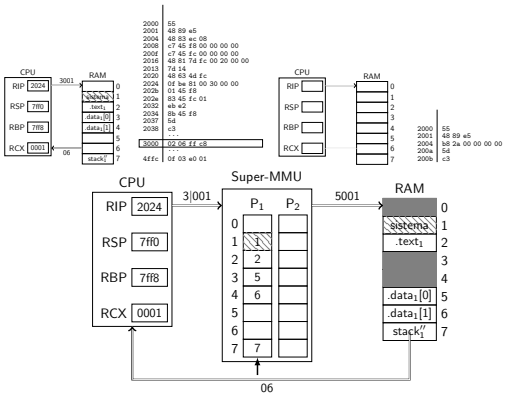
Infine, ricarica i registri con l'ultimo stato salvato di P_1 e gli cede il controllo. Ricordiamo che stava per eseguire `movsbl buf(%rcx), %eax`.



La CPU fisica e la virtuale (di P_1) sommano il contenuto di RCX e la costante 3000, ottenendo l'indirizzo 3001. Iniziano dunque una operazione di lettura a questo indirizzo.



Ancora una volta la MMU scompone l'indirizzo in numero di pagina (3) e offset (001). L'entrata numero 3 della tabella di corrispondenza dice che la pagina 3 si trova nel frame 5.



L'accesso viene completato, sempre dopo aver trasformato l'indirizzo. Entrambe le CPU ricevono lo stesso valore e continuano a procedere di pari passo, nonostante il fatto che le pagine di P_1 siano state spostate.

9.5 Tabelle di corrispondenza multilivello (*bitwise trie*)

Quanto è grande una tabella di corrispondenza? Supponiamo di avere uno spazio di 2^{48} byte: considerato che ogni pagina è grande 2^{12} byte otteniamo il seguente numero di pagine

$$\frac{2^{48}}{2^{12}} = 2^{36}$$

La dimensione di ogni entrata è di 8 byte: segue la dimensione della tabella di corrispondenza

$$2^{36} \times 8\,\text{byte} = 512\,\text{GiB}$$

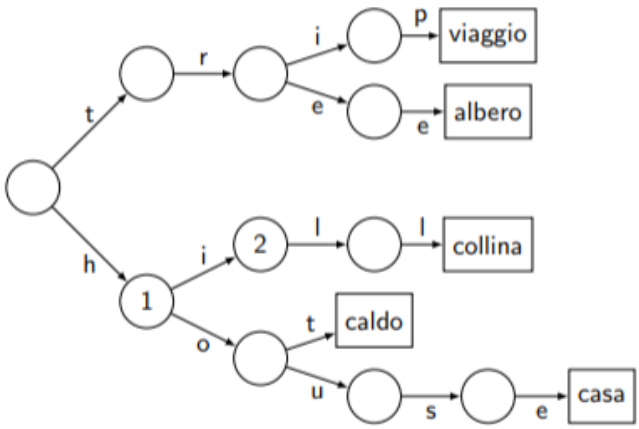
Morale della favola: è difficile avere un dispositivo che mi possa contenere una sola di queste tabelle. Ci serve una struttura dati che mi permetta di associare una chiave a un valore

Chiave (Numero di pagina) $\longrightarrow$ Valore (Numero di frame)

Struttura dati La struttura dati immaginata è una variante del *trie* : il *bitwise trie*. Con *trie* intendiamo strutture dati ad albero con cui mappiamo chiavi di tipo stringa. Supponiamo di avere le seguenti associazioni

trip $\longrightarrow$ viaggio
tree $\longrightarrow$ albero
hill $\longrightarrow$ collina
hot $\longrightarrow$ caldo
house $\longrightarrow$ casa

Otteniamo il seguente albero:



- I caratteri guidano i nostri movimenti all'interno dell'albero: gli archi sono marcati con i caratteri delle chiavi e il valore associato ad ogni chiave si trova nella foglia che si raggiunge partendo dalla radice e seguendo il percorso indicato dalla chiave.
- L'inserimento di una nuova associazione chiave-valore comporta una visita nell'albero come in una ricerca, con la creazione di eventuali nodi mancanti fino alla foglia che deve contenere il valore.

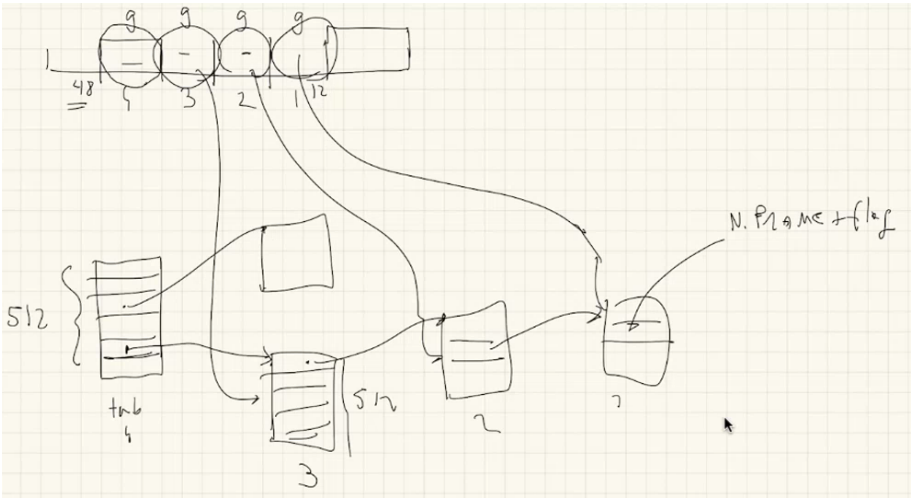
Tenendo conto che il trie si basa sulla codifica ASCII possiamo immaginarci un'implementazione dove ogni nodo consiste in un array di 128 entrate: ogni elemento dell'array è nullo o contiene un puntatore ad un altro nodo (quindi ad un altro array).

bitwise trie Il bitwise trie è una variante che prevede al posto dei caratteri gruppi di bit della chiave. L'indirizzo della radice è posto nel registro *CR3* della MMU.

- Il numero di pagina è composto da 36 bit, che raggruppiamo in quattro gruppi da 9 bit ciascuno. Nello scorrimento dell'albero consideriamo i gruppi **da quelli con le cifre più significative a quelli con le meno significative**.
- Ogni nodo del bitwise trie risulterà essere una tabella di $2^9 = 512$ entrate. Ciascuna entrata è un puntatore che può essere nullo, o rimandare a un nodo successivo.
- Sappiamo che ogni entrata ha dimensione di 8 byte, dunque otteniamo che ogni tabella ha dimensione

$$512 \cdot 8\,\text{byte} = 4096\,\text{byte}$$

- L'albero ha al più 4 livelli, numerati dalla radice in senso decrescente (da 4 ad 1)



  Se noi mettiamo insieme tutti i nodi, tutte le tabelle, otteniamo la tabella iniziale, ma a dimensione maggiore. Cio è valido solo se realizziamo tutti i sotto-alberi possibili (non abbiamo alberi dove il processo non ha mappato niente).
- Le foglie hanno la stessa struttura dei nodi, l'unica differenza sta nel significato delle entrate (che contengono non un puntatore, ma il numero di frame).
- Ogni riga delle tabelle ha la stessa struttura. Differenza rispetto alle tabelle viste all'inizio introducendo la MMU è la presenza di un maggior numero di bit: questo a causa dell'allineamento naturale delle tabelle (4096 byte), che impone i primi 12 bit meno significativi dell'indirizzo di una tabella uguali a zero. Alcuni bit non sono presenti (per esempio D), mentre altri sono presenti ma hanno un significato radicalmente diverso.
- il *page fault* viene generato non appena si individua una riga con $P = 0$.
- **Regola per le scritture**. La scrittura è permessa solo se è permessa da tutti i livelli.
- **Regola per l'accesso da livello utente**. L'accesso a livello utente è consentito solo se consentito da tutti i livelli.

Recap

- Abbiamo definito un architettura con la MMU posta tra CPU e Cache, distinguendo un "mondo fisico" con indirizzi fisici da un "mondo virtuale" con indirizzi virtuali (CPU e la MMU, dunque il software).
- La MMU, in ogni istante, avrà una traduzione attiva: un albero trie con cui associo un indirizzo fisico a un indirizzo virtuale (ottengo l'indirizzo fisico sostituendo, nell'indirizzo virtuale, il numero di pagina col numero di frame). La CPU contiene un registro, detto CR3, col numero di frame dove si trova la radice dell'albero attivo (l'albero si trova nella RAM, attenzione).
- Percorro l'albero utilizzando le varie parti che caratterizzano il numero di pagina: nella foglia troverà il numero di frame. Ricordiamoci che oltre al numero di frame abbiamo altre informazioni associate al numero di pagina.
- La struttura ci permette di evitare di associare un frame a qualunque pagina: dove non si può andare l'albero non prosegue, con $P = 0$.

Domanda sugli indirizzi della struttura dati Questa è una struttura dati preparata dal sistema e utilizzata dall'hardware (MMU). Nelle varie tabelle sono presenti puntatori ad altre tabelle nella RAM. Ci siamo chiesti: questi puntatori sono indirizzi virtuali o fisici?

- Fisici, altrimenti si genererebbe una sorta di loop (come faccio a muovermi tra le varie tabelle-nodo se ogni puntatore deve essere a sua volta tradotto attraverso le tabelle stesse?). **Emerge il problema di gestire questi passaggi**.

Come fa il software ad accedere a qualunque cosa? Per accedere a una qualunque entità abbiamo bisogno delle seguenti cose:

1. un indirizzo fisico per l'entità;
2. un indirizzo virtuale associato all'indirizzo fisico dell'entità;
3. la conoscenza da parte del software dell'indirizzo virtuale (all'utente non serve conoscere l'indirizzo fisico, il software lavora solo con indirizzi virtuali)

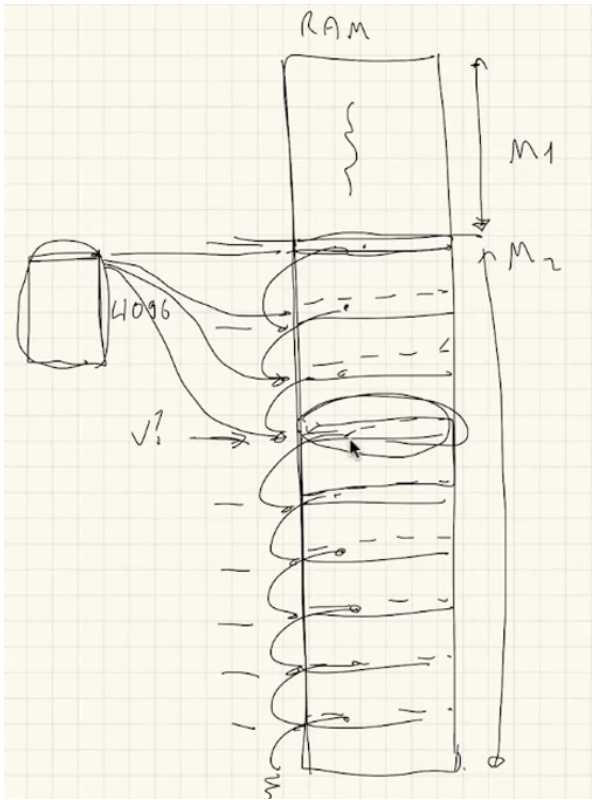
9.5.1 Utente e indirizzi virtuali e fisici

Per l'utente la vita è facile: conosce solo l'indirizzo virtuale. In un programma utente si dichiara una variabile globale, e il collegato sceglie un indirizzo virtuale per questa variabile. La variabile acquisisce un indirizzo fisico perchè il sistema caricherà la sezione data in una pagina, quindi in qualche frame. Il sistema, interpretando il file ELF, vedrà che il collegatore voleva he quella variabile si trovasse a un certo indirizzo virtuale: si crea così una corrispondenza tra indirizzo fisico e indirizzo virtuale.

stack Per quanto riguarda lo stack non ci sono problemi: creiamo la corrispondenza tra indirizzo fisico e indirizzo virtuale, a quel punto rendo conoscibile l'indirizzo virtuale mettendolo nel registro RSP.

9.5.2 Sistemi e indirizzi virtuali e fisici

Per il sistema le cose sono più difficili: deve essere consapevole in modo costante della differenza tra indirizzi fisici e indirizzi virtuali. In alcune situazioni gli indirizzi fisici sono irrinunciabili: le comunicazioni col bus mastering, l'aggiornamento delle tabelle di corrispondenza. Per memorizzare l'albero ci servono dei frame



in ciascun frame si ha, immaginiamo, un puntatore al frame successivo. Risulta immediato che dobbiamo sistemare qualcosa per poter inizializzare questa lista via software (devo scrivere in indirizzi fisici, mi servirebbe un indirizzo virtuale, ma come faccio se la corrispondenza si ottiene attraverso la tabella che stiamo inizializzando?)

Soluzione Prendiamo lo spazio di indirizzamento di un qualunque processo e lo dividiamo a metà.

- Una delle due metà viene riservata al sistema (in modo tale che guardando i bit più significativi gli indirizzi siano riconoscibili). Nel nostro caso l'utente andrà a considerare, per le sue cose, solo gli indirizzi che iniziano con 111111 (cifre più significative).
- Ci rimangono indirizzi virtuali a disposizione del sistema: l'idea è di mapparli in tutta la RAM, per fare in modo che tutti gli indirizzi fisici della RAM abbiano un corrispondente

indirizzo virtuale, a priori. Questa cosa permetterà al sistema di accedere a tutta la memoria fisica. In contemporanea dobbiamo negare questa possibilità ai processi utente.

A questo punto ogni indirizzo fisico ha un corrispondente indirizzo virtuale

- **Come garantiamo al sistema la conoscenza degli indirizzi virtuali?** Ponendo il numero di frame uguale al numero di pagina, cioè poniamo indirizzi virtuali uguali ad indirizzi fisici! A un certo punto modificheremo CR3: a quel punto la MMU inizierà ad usare un nuovo albero di traduzione. Se vogliamo garantire continuità nel passaggio da un processo a un altro dobbiamo fare in modo che la traduzione degli indirizzi relativi all'area per il sistema siano uguali in entrambi gli alberi. Questa cosa semplifica l'inizializzazione del sistema: la paginazione inizialmente è disattivata, fino a quando non modificheremo un flag nel registro CR0.

Sorpresa

Fino ad ora abbiamo sempre lavorato con la paginazione attiva

La AMD, per come ha realizzato il processore, ha fatto in modo che la modalità a 64 bit sia in realtà una sotto-modalità della paginazione. Al momento dell'avvio il bootstrap parte come un 8086, viene portato a 32 bit abilitando la protezione, e infine si passa a 64 bit con l'attivazione della paginazione

$$\text{8086 a 16 bit} \longrightarrow \text{32 bit con abilitaz. protezione} \longrightarrow \text{64 bit con abilitaz. paginazione}$$

Questi passaggi avvengono attraverso la modifica di appositi flag.

Registri Nel processore sono presenti i seguenti registri:

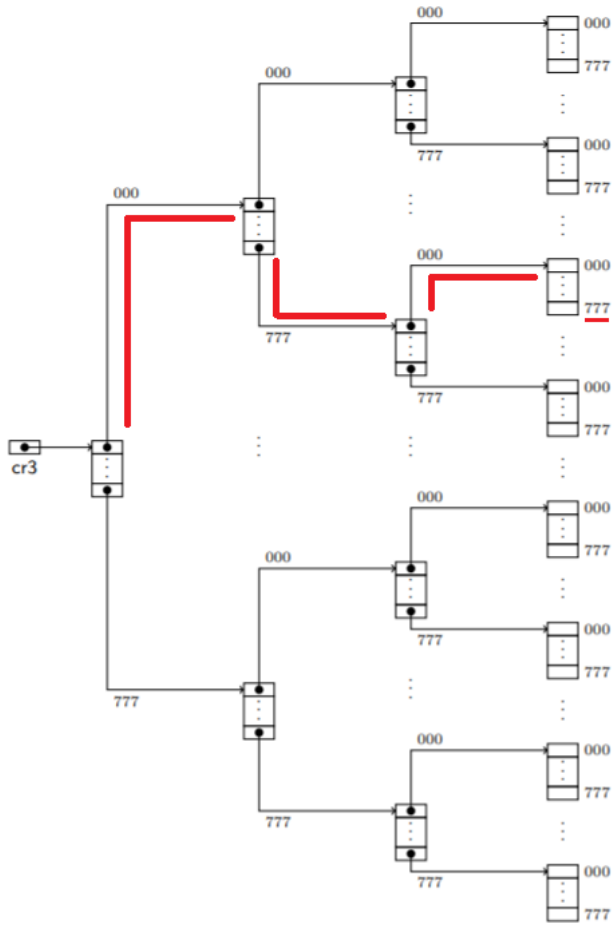
- CR0, che ha un flag con cui abilitare/disattivare la paginazione;
- CR1, mai implementato;
- CR2, che contiene in caso di *page fault* l'indirizzo che la MMU ha cercato di tradurre;
- CR3, che contiene il numero di frame relativo alla radice del *trie* attualmente utilizzato (si tenga a mente che è richiesto un allineamento ben preciso).

Chiaramente devo avere un albero già posto in CR3 prima di attivare la paginazione attraverso CR0.

Rappresentazione dei numeri in ottave Può essere utile rappresentare i numeri di pagina in base 8. Si applicano le stesse regole per la conversione da base due a base esadecimale (e viceversa), ma si hanno gruppi di tre cifre. Supponiamo di dover tradurre il seguente indirizzo virtuale

$$v = (000\,777\,000\,777\,1234)_8 \qquad \text{Il numero di pagina è } (000\,777\,000\,777)_8.$$

- Prendiamo i bit $(000)_8$: passiamo alla prima tabella di livello tre attraverso la prima entrata di *tab4*.
- Prendiamo i bit $(777)_8$: in questo caso prendiamo l'ultima entrata della tabella e passiamo alla seconda tabella di livello due in immagine.
- Prendiamo i bit $(000)_8$: in questo caso prendiamo la prima entrata della tabella e passiamo alla terza tabella di livello uno in immagine.
- Prendiamo i bit $(777)_8$: l'ultima uscita della terza tabella di livello uno in immagine contiene il numero di record che ci interessa.



9.5.3 Esempio non vitale: creazione di un albero di traduzione in Assembler

Proviamo a creare un albero di traduzione (non per usarlo, solo per capire come si ragiona all'avvio): vogliamo mappare . Lo facciamo in Assembler.

9.5.3.1 Codice Assembler

```
# vogliamo mappare i primi 8 MiB in se stessi
#                           12
# +--------------------------+-----------------+
# |    n. frame               |          DAwdSWP|
# +--------------------------+-----------------+

.global setup_vm
setup_vm:
    # 0 -> 0
    #  4   3   2   1   off
    # 000 000 000 000 xywz -> 000 000 000 000 xywz
    movq $tab3, tab4
    movb $0b00000111, tab4

     movq $tab2, tab3
    movb $0b00000111, tab3

    movabs $tab2, %rsi
    movabs $tab1_0, %rdi
    xor %rax, %rax

.Loop0:
    mov %rdi, (%rsi)
    movb $0b00000111, (%rsi)
    xor %rcx, %rcx
.Loop1:
    mov %rax, (%rdi, %rcx, 8)
    movb $0b00000111, (%rdi, %rcx, 8)
    add $0x1000, %rax
    cmp $511, %rcx
    jge .Lnext
    inc %rcx
    jmp .Loop1
.Lnext:
    cmp $tab1_3, %rdi
    je .Lend
    add $4096, %rdi
    add $8, %rsi
    jmp .Loop0
.Lend:
     movabs $tab4, %rax
     mov %rax, %cr3

     ret

.global a_page_fault
a_page_fault:
    pop %rdi
    mov (%rsp), %rsi
    mov %cr2, %rdx
    call c_page_fault
    hlt

.data
.global tab4, tab3, tab2, tab1_0, tab1_1, tab1_2, tab1_3, tab1_4
.balign 4096
tab4:
    .space 4096, 0
tab3:
    .space 4096, 0
tab2:
    .space 4096, 0
tab1_0:
    .space 4096, 0
tab1_1:
    .space 4096, 0
tab1_2:
    .space 4096, 0
tab1_3:
    .space 4096, 0
tab2_3:
    .space 4096, 0
tab1_4:
    .space 4096, 0
```

- **Allocazione dello spazio per l'albero**.
  - Ci serve spazio per un albero di traduzione.
  - La tabella non può essere allocata ovunque, deve essere posta a un indirizzo che è multiplo di 4096. Facciamo questo con la seguente istruzione

    `.balign 4096`

  - **Quante tabelle di livello quattro ci servono?**
    Una, visto che abbiamo solo la radice dell'albero.
  - **Quante tabelle di livello tre ci servono?** Sempre una.
    * Ogni riga copre 1 GB.
    * Se moltiplico per il numero di righe otteniamo

      $$512 \cdot 1\,\text{GB} = 512\,\text{GB}$$

      per 8 MB basta una sola tabella.

- **Quante tabelle di livello due ci servono?** Sempre una.
  - Ogni riga copre 2 MB.
  - Se moltiplico per il numero di righe ottengo

    $$2\,\text{MB} \cdot 512 = 1\,\text{GB}$$

    Anche in questo caso una sola tabella è più che sufficiente.
- **Quante tabelle di livello 1 ci servono?** Quattro.
  - Ogni riga copre 4 KB
  - Se moltiplico per il numero di righe ottengo

    $$4\,\text{KB} \cdot 512 = 2048\,\text{KB}$$

    Questa volta non ci basta una sola tabella, ne servono 4.

Codice con cui allochiamo lo spazio

```
.data
.global tab4, tab3, tab2, tab1_0,
tab1_1, tab1_2, tab1_3, tab1_4
.balign 4096
tab4:
.space 4096, 0
tab3:
.space 4096, 0
tab2:
.space 4096, 0
tab1_0:
.space 4096, 0
tab1_1:
.space 4096, 0
tab1_2:
.space 4096, 0
tab1_3:
.space 4096, 0
tab2_3:
.space 4096, 0
tab1_4:
.space 4096, 0
```

- **Prepariamo l'albero**.
  - Dobbiamo pensare a come la MMU lo percorrerà (prende il numero di pagina, 0, tralasciando l'offset).
  - Dobbiamo considerare, nei vari passaggi, la presenza di altre informazioni oltre al numero di frame: P, R/W, U/S, PCD, PWT, A, D. Alcuni di questi bit devono essere gestiti per forza, altrimenti non possiamo fare le nostre operazioni: settiamo i primi tre bit meno significativi (P, R/W, U/S). Lo facciamo aggiornando, di 64 bit, i 16 meno significativi

    ```
    movb $0b00000111, tabX
    ```
  - La prima riga di *tab4* deve puntare *tab3*, la prima riga di *tab3* deve puntare a *tab2*.

    ```
    movq $tab3, tab4
    movb $0b00000111, tab4

    movq $tab2, tab3
    movb $0b00000111, tab3
    ```
  - Le righe di *tab2* devono puntare alle tabelle di livello 1. La cosa più conveniente è aggiornare le tabelle di livello 1 attraverso dei cicli.
    - Le tabelle di livello 1 sono una dopo l'altra, dunque non è necessario gestirle in modo distinto: si parte dalla prima riga della prima tabella di livello 1 e si scorre tutto insieme.
    - I parametri in ingresso sono due:
      - registro RSI, indirizzo della tabella di livello 2 *tab2*;

        - registro RDI, indirizzo della prima riga della prima tabella di livello 1.
    - Abbiamo due cicli annidati.

- **Aggiornamento di CR3**.

  Dopo aver riempito tutte le entrate di tutte le tabelle di livello 1 possiamo mettere l'indirizzo dell'unica tabella di livello 4, *tab4*, nel registro CR3. Da questo punto in poi non stiamo più utilizzando la traduzione preparata dal bootloader, ma la nostra.

  ```
  movabs $tab4, %rax
  mov %rax, %cr3
  ```

9.5.3.2 Codice C++

```
#include <libce.h>

int var;

extern "C" void setup_vm();
extern "C" void a_page_fault();
extern "C" natq tab4[], tab3[], tab2[], tab1_0[], tab1_1[], tab1_2[], tab1_3[], tab1_4[];
extern "C" void c_page_fault(natq errore, natq rip, natq addr) {
    printf("errore %x, rip %x, addr %x\n", errore, rip, addr);
}

int main() {
     gate_init(14, a_page_fault);
     var = 1;
     printf("var = %d\n", var);
     pause();
     setup_vm();

     int *p = (int *)((natq)&var - 0x20f000 + 0x400000);
     printf("tab4[0] = %x\n", tab4[0]);
     printf("tab3[0] = %x\n", tab3[0]);
     printf("tab2[1] = %x\n", tab2[1]);
     printf("tab1_1[15] = %x\n", tab1_1[15]);

     printf("percorso alternativo:\n");
     printf("tab2[2] = %x\n", tab2[2]);
     printf("tab1_2[0] = %x\n", tab1_2[0]);
     *p = 2;
     printf("p %x, var = %d\n", p, var);
     printf("tab4[0] = %x\n", tab4[0]);
     printf("tab3[0] = %x\n", tab3[0]);
     printf("tab2[1] = %x\n", tab2[1]);
     printf("tab1_1[15] = %x\n", tab1_1[15]);

     printf("percorso alternativo:\n");
     printf("tab2[2] = %x\n", tab2[2]);
     printf("tab1_2[0] = %x\n", tab1_2[0]);

     pause();
}
```

- **Eccezione *page_fault*.**

  Immaginiamo di avere il seguente main

```
int main() {
    var = 1;
    printf("var = %d\n", var);
    pause();
    setup_vm();
    pause();
}
```

  Cosa succede dopo il secondo *pause*? Si ha l'eccezione di tipo 14 (*page_fault*), lanciata dalla MMU che non è riuscita a completare la traduzione. I motivi sono vari: incontra un bit $P = 0$, scritture fallite perchè non è permessa la scrittura, o accessi ad aree di sistema in modalità utente... Questa eccezione lascia in pila delle informazioni ulteriore: delle 4 quadword ci interessa quella in cima.

  – Il bit meno significativo dice se l'errore è dovuto a una traduzione non valida (0) o a un errore di protezione (1).
  – Il secondo bit meno significativo se l'eccezione è legata o meno a un'operazione di scrittura (si dice solo il tipo di operazione, non se era vietata la scrittura).
  – Il terzo bit meno significativo segnala il livello di privilegio a cui si trovava il processore al momento dell'errore (0, sistema, 1, utente).

  Nel caso nostro abbiamo

```
INF     -       entry point 00200120
WRN     -       Eccezione 14, err=0000000000000002, EIP=000000000020056e
qemu-system-x86_64: terminating on signal 2
studenti@debian-ce-2:~/vm$ addr2line -f 0x20056e
_Z8apic_outhj
/home/studenti/libce-2.5/apic_out.cpp:8
```

  L'errore è di traduzione, l'operazione era una scrittura in memoria

```
mov %edx, (%rax)
```

- **Registro CR2**.

  La MMU, quando non riesce a tradurre un indirizzo, scrive l'indirizzo che stava cercando di tradurre nel registro CR2.

- **Funzione da associare all'eccezione di tipo** 14.

  Abbiamo scritto una routine da lanciare in caso di *page_fault*, come al solito divisa in due parti. La prima si trova nel codice Assembler visto prima e raccoglie i parametri in ingresso a partire dal contenuto della pila (ricordiamo che sono state poste delle informazioni in pila).

  – RDI: quadword *errore* (il codice identificativo dell'errore, che si trova in cima alla pila)

```
pop %rdi
```

  – RSI: quadword *rip*, il rip dell'istruzione che il processore stava eseguendo quando si è manifestato il *page fault*.

```
mov (%rsp), %rsi
```

  – RDX: quadword *addr*, contenuto del registro CR2 (l'indirizzo che la MMU non è riuscita a tradurre)

```
mov %cr2, %rdx
```

La seconda parte, in C++, richiede la stampa delle informazioni sull'eccezione

```
extern "C" void c_page_fault(natq errore, natq rip, natq addr) {
    printf("errore %x, rip %x, addr %x\n", errore, rip, addr);
}
```

Associo la funzione al tipo 14 con la *gate_init*

```
gate_init(14, a_page_fault);
```

L'output ottenuto è il seguente

L'indirizzo a cui l'istruzione stava tentando di accedere è *fec00000*. La traduzione di questo indirizzo virtuale non l'abbiamo creata: facciamolo!

- **Aggiunta di una nuova traduzione di indirizzo**.

  Traduciamolo in se stesso: la *libce* pensava di utilizzare indirizzi fisici, la sua intenzione era di scrivere proprio a quell'indirizzo. Il codice Assembler viene aggiunto in *setup_vm*, dopo l'etichetta *Lend*.

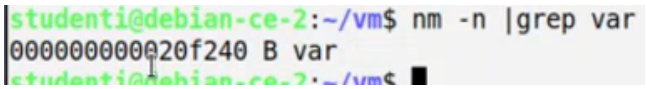
```
# tab4                                  #
# 0: -> tab3                            # fec00000 -> fec00000
#       0: -> tab2                      #
#             0: tab1_0                 #   numero di pagina
#             1: tab1_1                 #
#             2: tab1_2                 # 000000000 000/000/011 111/110/110 000/000/000
#             3: tab1_3                 # 000       003         766         000
#       3: -> tab2_3                    #
#             766: -> tab1_4            #

movl $tab2_3, tab3 + 8*3
movb $0b00000111, tab3 + 8*3

movl $tab1_4, tab2_3 + 0766 * 8 <--- 0766 interpretato in base 8
movb $0b00000111, tab2_3 + 0766 * 8 <--- 0766 interpretato in base 8

movl $0xfec00000, tab1_4
movb $0b00001111, tab1_4
```

  - Scomponendo l'indirizzo in ottave capiamo meglio come muoverci.
  - Siamo in *tab4*. Passiamo a *tab3* attraverso la prima entrata dell'unica tabella.
  - Siamo in *tab3*: l'entrata che ci interessa è la terza, dunque abbiamo una nuova tabella da creare rispetto a prima (*tab2_3*).
  - Nella tabella appena creata ci interessa la 502−esima entrata. Attraverso questa si passa alla tabella *tab1_4* (ulteriore tabella in più).

– Le istruzioni mov per aggiornare il contenuto dell'albero le abbiamo già viste nella preparazione dell'albero. Si tenga solo conto che avendo l'APIC a quell'indirizzo conviene disattivare la cache: segue il settaggio del quarto bit meno significativo nelle varie *movb*.

- **Creazione di un nuovo indirizzo virtuale per la variabile *var*.**

  Creiamo una variabile *var* nel nostro programma.



  Siamo sicuri, per quello che abbiamo fatto prima, che già esista un indirizzo virtuale. Mappiamo un altro indirizzo virtuale, completamente diverso, nello stesso indirizzo fisico. Il codice Assembler viene aggiunto in *setup_vm*, dopo l'etichetta *Lend*.

```
# 0x400-000 -> 0x20f-000
#
#  numero di pagina
#
# 000000000 000000000 000000010 000000000
# 000       000       002       000

movl $0x20f000, tab1_2
movb $0b0000111, tab1_2
```

  A livello di tabelle abbiamo già tutto, non ci serve fare altro. Entrata 0 in *tab4*, Ci interessano le seguenti entrate: entrata 0 in *tab3*, entrata 2 in *tab2*, entrata 0 in *tab1_2*.

  A questo punto possiamo vedere, nel *main*, l'utilizzo di questo nuovo indirizzo virtuale: la variabile *var* viene modificata attraverso il puntatore *p* (che contiene il nuovo indirizzo virtuale)

```
int *p = (int *)((natq)&var - 0x20f000 + 0x400000);
*p = 2;
printf("p %x, var = %d\n", p, var);
```

- **Modifica delle tabelle dell'albero in C++.**

  Possiamo notare come nel *main* si recuperi diverse volte il contenuto delle tabelle dell'albero. Possiamo agire sulla tabella a partire dal C++, dopo aver definito il tipo. Nella lezione successiva è presente un esempio di scrittura (col cosiddetto *Page Size Flag*).

9.6 Pagine di dimensione diversa (*Page Size* flag)

Abbiamo introdotto il trie, che ci offre il vantaggio di non avere sottoalberi se relativi indirizzi non sono utilizzati. Altra cosa utile è la possibilità di creare **pagine di dimensione diversa**.

Struttura indirizzo virtuale Abbiamo:

- 12 bit di offset (bit meno significativi);
- numero di pagina spezzato in quattro indici.

Più la pagina è grande, più è grande l'offset e piccolo il numero di pagina.

Proposta Se la dimensione delle pagine è variabile allora si potrebbe aumentarne la dimensione, arrivando a ridurre di 1 i livelli dell'albero (velocità della traduzione - tre accessi invece di quattro - e minore spazio occupato dalla struttura dati). La cosa solitamente non succede perchè pagine di dimensioni più elevate possono comportare un maggiore spreco di memoria (si consideri che la pagina è unità di condivisione tra progetti, e di condivisione).

***Page Size* flag** Nonostante questo l'albero ci permette di non dover decidere a priori la grandezza, possiamo decidere in modo flessibile. Questo perchè all'interno di ogni entrata abbiamo un flag **Page Size** (PS) che dica alla MMU che in questo percorso (supponiamo di settare il flag al livello 2) la pagina è più grande (si indica che il percorso finisce lì, nell'esempio detto abbiamo una dimensione di 2 MB). Il set o meno di questo flag determina il numero di bit del numero di pagina all'interno dell'indirizzo virtuale.

- **Vantaggio**: ci servono meno tabelle per fare la traduzione identità (quella dove indirizzi virtuali corrispondono a indirizzi fisici, nella parte di RAM riservata al sistema).

9.6.1 Esempio di utilizzo del flag nell'ultimo esempio di esercizio

Riprendiamo l'esercizio della scorsa lezione, abbiamo visto che possiamo leggere le tabelle dell'albero dal C++, dopo aver definito il tipo. Avevamo creato una traduzione (identità) solo fino a 8 MB, questo significa che se dichiaro qualcosa fuori da questa regione viene lanciata un'eccezione (per $P = 0$).

Cosa vogliamo fare? Adesso vogliamo creare una traduzione identità per altri 2 MB: senza le cose appena introdotte risulterebbe necessario introdurre una nuova tabella. Ci limitiamo a fare

```
tab2[4] = 0x800000 | 0x87;
```

settando il bit più significativo del byte di accesso, il bit PS. In questo modo diciamo ad MMU che la traduzione è già finita quando si arriva a questa entrata.

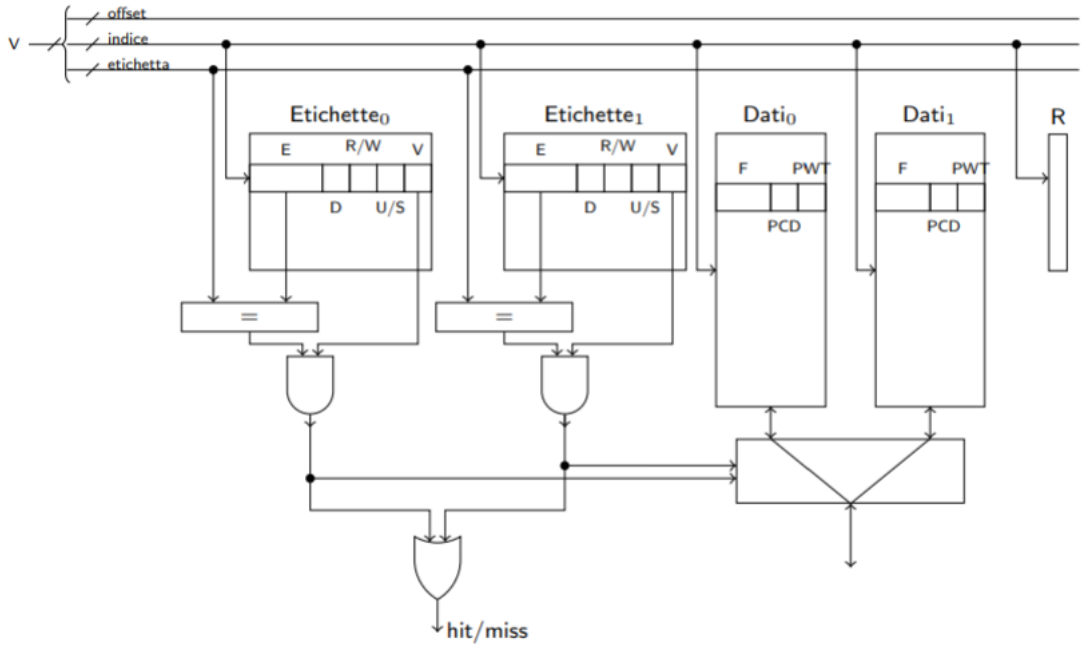
9.7 Cache *Translation Lookaside Buffer* (TLB)

Dal punto di vista dell'hardware ci resta un'unica cosa. L'implementazione via trie è comoda ma presenta anche degli svantaggi: pensiamo al numero elevato di accessi (con questi alberi il numero di accessi moltiplica ogni volta[3]). Anche accedere in cache ha un costo: il singolo accesso non occupa centinaia di clock, ma rispetto a prima abbiamo un numero maggiore di accessi in cache.

Soluzione La soluzione è la *Translation Lookaside Buffer* (TLB), una cache dedicata alla MMU (una cache delle traduzioni, entro qua per non consultare il trie).

[3] La cosa può avere grande impatto sul tempo di esecuzione di un programma: prelievo delle istruzioni, prelevo o scrittura degli operandi che si trovano in memoria...

Cache Limitiamoci a parlare delle cache a 4 KB: la cache conterrà le entrate delle tabelle di livello 1. La TBL è una letteralmente una cache a 8/16 vie. Lo schema interno è quasi del tutto identico a quello già visto.



Viene memorizzata l'associazione tra numero di pagina e numero di frame, oltre alle informazioni poste nel descrittore di pagina.

Osservazione su alcuni bit

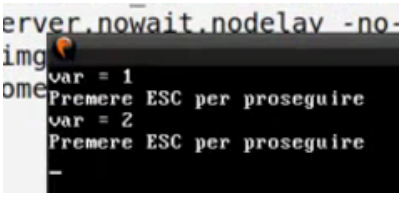
- Abbiamo detto che una pagina è riservata al sistema se tutti i bit U/S (uno per livello) sono settati. Che informazione metto nel TLB (1 bit) affinchè il controllo possa essere equivalente a quello fatto sul trie (4 bit))? La AND dei vari bit incontrati.
- Il bit A indica che c'è stato un accesso: non è riportato nel TLB perchè se l'entrata è presente in cache è logico che c'è stato un accesso.
- La cosa è più complessa col bit D: può accadere che su una data pagina ho inizialmente una lettura e poi una scrittura. In queste condizioni non ce ne accorgiamo passando dalla cache: D rimane a zero nonostante la scrittura. Risolviamo così: in caso di hit con operazione di scrittura e D a zero non si concede immediatamente la scrittura (*miss*), ma si ripercorre il trie.
- Il bit P non è presente: se abbiamo riportato l'entrata nella cache significa che abbiamo trovato tutti i $P = 1$.
- Se $V = 0$ la riga non è significativa (è il bit già incontrato quando abbiamo spiegato la cache, non ha a che vedere con le entrate delle tabelle del trie)

9.7.1 Riprendiamo l'esempio di esercizio

Abbiamo visto che è possibile modificare la variabile *var* a partire da due indirizzi virtuali completamente diversi. Immaginiamo di avere il seguente main

```
#include <libce.h>
#include <vm.h> <----  novita'
int var;

extern "C" void setup_vm();
extern "C" void a_page_fault();
extern "C" natq tab4[], tab3[], tab2[], tab1_0[], tab1_1[], tab1_2[], tab1_3[], tab1_4[];
extern "C" void c_page_fault(natq errore, natq rip, natq addr) {
    printf("errore %x, rip %x, addr %x\n", errore, rip, addr);
}

int main() {
    gate_init(14, a_page_fault);
    var = 1;
    prinft(var = %d\n", var);
    pause();
    setup_vm();
    int *p = (int*)((natq)&var - 0x20f000 + 0x400000);
    tab1_2[0] = 0x20f000 | 0x07;
    *p = 2;
    printf("var = %d\n", var)
    pause();
}
```

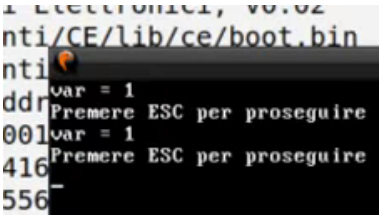
Se compiliamo vediamo di nuovo lo scrivere in *var* attraverso un indirizzo virtuale alternativo.



Aggiungiamo la seguente riga prima della scrittura in *tab1_2*.

```
*p = 3;
```

Stavolta *var* non viene modificato, l'output da 1.



Come mai?

- Inizialmente l'indirizzo *0x400000* è mappato su se stesso.
- Con la prima modifica

  ```
  *p = 3
  ```

  agiamo sull'indirizzo fisico *0x400000*. Segue che non modificheremo *var* con questa istruzione.
- Con la seconda modifica, che avviene dopo l'aggiornamento della riga di *tab1_2*

  ```
  *p = 2;
  ```

  La traduzione nel trie è ancora quella vecchia: abbiamo agito sempre su *0x400000*, e non su *var*.
- Non è la cache della memoria il problema, **ma la traduzione che ha usato la MMU**.

Questo problema è dovuto alla cache TLB. Il problema è il solito: mantenere la consistenza a seguito di modifiche. Quando modifico una traduzione il TLB quasi sicuramente non se ne accorge. Dobbiamo invalidarlo in modo esplicito! Possiamo farlo in due modi.

1. **Istruzione mov con registro CR3 operando destinatario**

   ```
   movq %rax, %cr3
   ```

   L'istruzione può potenzialmente alterare tutta la tabella di livello 4, quindi tutte le informazioni contenute nel TLD non possono considerarsi più valide.

2. **Istruzione Assembler *invlpg* (*invalid page*)**

   ```
   invlpg operando_in_memoria
   ```

   L'istruzione permette di invalidare la traduzione relativa all'indirizzo passato come operando. La libreria *libce*, *vm.h*, offre una funzione già pronta per poter eseguire l'istruzione da C++: **invalida_entrata_TLB**.

```
File  Edit  Tabs  Help
// dato un indirizzo virtuale (come parametro) usa l'istruzione invlpg per
// eliminare la corrispondente traduzione dal TLB
        .global invalida_entrata_TLB //
invalida_entrata_TLB:
        .cfi_startproc
        invlpg (%rdi)
        ret
        .cfi_endproc
```

A questo punto torniamo sul *main* e scriviamolo così

```
int main() {
     gate_init(14, a_page_fault);
     var = 1;
     prinft(var = %d\n", var);
     pause();
     setup_vm();
     int *p = (int*)((natq)&var - 0x20f000 + 0x400000);

     *p = 3; <--- riga in piu'

     tab1_2[0] = 0x20f000 | 0x07;

     invalida_entrata_TLB(p); <--- riga in piu'

     *p = 2;
     printf("var = %d\n", var)
     pause();
}
```

L'output torna e la seconda modifica del contenuto puntato da p ha agito su *var*.

Capitolo 10

Implementazione della paginazione

10.1 Extra: *typedef* per indirizzi fisici e virtuali

Nel file *tipo.h* della *libce* sono presenti due ulteriori typedef per distinguere indirizzi fisici da indirizzi virtuali (solo comodità visiva, sono entrambi *unsigned long*)!

- `typedef unsigned long vaddr;`
  Indirizzi virtuali
- `typedef unsigned long paddr;`
  Indirizzi fisici

Sono molto utilizzati nelle funzioni di *vm.h*. In quest'ultimo file troviamo anche

- `typedef natq tab_entry;`
  che usiamo per le entrate delle tabelle delle pagine.

10.2 Costanti in *vm.h* della *libce*

10.2.1 Numero massimo di livelli del *trie*

```
static const int MAX_LIV = 4;
```

Numero di livelli massimo. Nel nostro codice abbiamo 4, ma con processori a 57 bit è possibile avere un livello in più.

10.2.2 Maschere

- `static const natq BIT_SEGNO = (1ULL << (12 + 9* MAX_LIV -1)); <- ULL unsigned long long`
  Calcoliamo il bit dove si trova il segno (l'ultimo bit più significativo che ci interessa). Si pone ULL per evitare problemi con lo shift. La maschera è utile nella funzione per la normalizzazione degli indirizzi. Nel caso nostro avremo:

  ```
  BIT_SEGNO = (1ULL << 47);
  ```

- `static const natq MASCHERA_MODULO = BIT_SEGNO - 1;`
  Maschera con cui recupero il modulo.

10.2.3 Costanti per la manipolazione dei descrittori di pagina e di tabella

```
const natq BIT_P    = 1U << 0; // il bit di presenza
const natq BIT_RW   = 1U << 1; // il bit di lettura/scrittura
const natq BIT_US   = 1U << 2; // il bit utente/sistema
const natq BIT_PWT  = 1U << 3; // il bit Page Wright Through
const natq BIT_PCD  = 1U << 4; // il bit Page Cache Disable
const natq BIT_A    = 1U << 5; // il bit di accesso
const natq BIT_D    = 1U << 6; // il bit "dirty"
const natq BIT_PS   = 1U << 7; // il bit "page size"

const natq ACCB_MASK  = 0x00000000000000FF; // maschera per il byte di accesso
const natq ADDR_MASK  = 0x7FFFFFFFFFFFF000; // maschera per l'indirizzo
```

10.3 Funzioni in *vm.h* della *libce*

Il file è incluso attraverso apposita direttiva in *sistema.cpp*, precisamente nella sezione dedicata alla paginazione

```
#include <vm.h>
```

10.3.1 Normalizzazione dell'indirizzo con *norm*

```
static inline vaddr norm(vaddr a) {
    return (a & BIT_SEGNO) ? (a | ~MASCHERA_MODULO) : (a & MASCHERA_MODULO);
}
```

- Dato un indirizzo virtuale la funzione restituisce la corrispondente versione normalizzata (i 16 bit più significativi sono uguali al $47-$esimo bit dell'indirizzo).
- Step:
  - Controlliamo il bit di segno.
  - Se è uguale ad 1 metto i bit più significativi rimanenti uguali ad 1
  - Se è uguale a 0 metto i bit più significativi rimanenti uguali a 0.

10.3.2 Grandezza di una regione con *dim_region*

```
static inline constexpr natq dim_region(int liv) {
    natq v = 1ULL << (liv * 9 + 12);
    return v;
}
```

Definizione Con *regione di livello liv* intendiamo l'intervallo di indirizzi coperti da una singola entrata di una tabella di livello $i + 1$. Questo significa che:

- una regione di livello 0 è grande 4096 byte;
- una regione di livello 1 è grande 2 MiB;
- una regione di livello 3 è grande 1 GiB.

Scopo della funzione La funzione ci restituisce la grandezza di una regione di livello *liv*: prendo 12 (numero di bit dell'offset) e mi ricordo che ogni livello è rappresentato da 9 bit, quindi sommo a 12 il prodotto tra 9 e *liv*. Concludiamo traslando

$$1 \cdot 2^{\text{liv}\cdot 9+12}$$

10.3.3 Funzioni *base* e *limit* per l'indirizzo di base di una regione

```
static inline vaddr base(vaddr v, int liv) {
    natq mask = dim_region(liv) - 1;
    return v & ~mask;
}

static inline vaddr limit(vaddr e, int liv) {
    natq dr = dim_region(liv);
    natq mask = dr - 1;
    return (e + dr - 1) & ~mask;
}
```

Dato un indirizzo virtuale v le due funzioni restituiscono un altro indirizzo virtuale. Precisamente:

- la funzione *base* restituisce la base della pagina di livello *liv* in cui cade l'indirizzo v;
  - Otteniamo il numero di regione mascherando i bit meno significativi dell'indirizzo. Vediamo un esempio semplificato

    $$0b1000 - 0b0001 = 0b0111 \longrightarrow !(0b0111) = 0b1000$$

    Se uso l'operatore AND spariscono i tre bit meno significativi.
- la funzione *limit* restituisce la base della prima regione di livello *liv* a destra dell'intervallo (quello della regione dove si trova l'indirizzo v).
  - Recupero la dimensione di una regione di livello *liv*.
  - Decremento per ottenere una maschera (bit più significativo uguale a 0, tutti gli altri uguali ad 1).
  - Sommo all'indirizzo e la dimensione della regione e decremento di 1. A quel punto faccio AND con la negazione della maschera *mask* (mi sbarazzo dei bit meno significativi).

10.3.4 Funzioni per lavorare sul *trie*

10.3.4.1 Estrazione dell'indirizzo fisico da un'entrata (*extr_IND_FISICO*)

```
static inline paddr extr_IND_FISICO(tab_entry descrittore) {
    return descrittore & ADDR_MASK;
}
```

La funzione mi permette di ottenere l'indirizzo fisico presente nella *tab_entry* indicata nei parametri di ingresso. Nel caso della foglia non restituiremo solo un indirizzo fisico, ma l'indirizzo della base del frame.

Cosa faccio Mi ricordo dove è collocato l'indirizzo fisico all'interno del descrittore, tenendo conto che l'offset (i primi 12 bit) è sempre una sequenza di zeri. Utilizzo la costante *ADDR_MASK*, già vista

```
const natq ADDR_MASK  = 0x7FFFFFFFFFFFF000;
```

10.3.4.2 Settaggio dell'indirizzo fisico in un'entrata (*set_IND_FISICO*)

```
static inline void set_IND_FISICO(tab_entry& descrittore, paddr ind_fisico) {
    descrittore &= ~ADDR_MASK;
    descrittore != ind_fisico & ADDR_MASK;
}
```

Setto l'indirizzo fisico nel descrittore con due step:

- resetto tutti i bit relativi all'indirizzo fisico;
- imposto i bit ponendo il nuovo indirizzo fisico (metto l'indirizzo nel descrittore con un OR, dopo aver rimosso l'offset con l'AND).

Anche qua utilizziamo la costante *ADDR_MASK*

```
const natq ADDR_MASK  = 0x7FFFFFFFFFFFF000;
```

10.3.4.3 Indice dell'entrata di un vaddr in una tabella di livello *liv* (*i_tab*)

```
static inline int i_tab(vaddr ind_virt, int liv) {
    int shift = 12 + (liv - 1) * 9;
    natq mask = 0x1ffULL << shift;
    return (ind_virt & mask) >> shift;
}
```

Dato un indirizzo virtuale *ind_virt*, estraggo dall'indirizzo virtuale l'indice del descrittore relativo, in una tabella di livello *liv*.

- Prendo una maschera di 9 bit, che shifto in modo da posizionarla sotto l'indice che mi interessa.
- Applico la maschera con l'operatore AND e shifto nuovamente per portare l'indice sulle cifre meno significative.

10.3.4.4 Indirizzo di un'entrata di indice *index* da una tabella *tab* (*get_entry*)

```
static inline tab_entry& get_entry(paddr tab, natl index) {
    tab_entry *pd = reinterpret_cast<tab_entry*>(tab);
    return pd[index];
}
```

Dato l'indirizzo fisico di una tabella *tab* e un indice *index*, la funzione restituisce il riferimento a una certa entrata.

10.3.4.5 Classe *tab_iter* per la visione del trie (iteratore)

Una cosa che si deve fare molto spesso è visitare l'albero. Uno, per esempio, potrebbe essere interessato a visitare tutte le entrate che si occupano della traduzione degli indirizzi in un certo intervallo di indirizzi. Le strade possibili sono due:

1. scrivere una funzione ricorsiva;
2. realizzare un *iteratore* (cioè un qualcosa che si ricorda in che punto siamo arrivati nell'albero, e rimane lì finchè qualcuno non gli dice di spsotarsi).

La scelta adottata è la seconda.

Codice

```
class tab_iter {
    struct stack {
        vaddr cur, end;
        paddr tab;
    } s[MAX_LIV + 1];

    int l;

    stack *sp() { return &s[l - 1]; }
    stack const *sp() const { return &s[l - 1]; }
    stack *sp(int lvl) { return &s[lvl - 1]; }
    stack *pp() { return &s[MAX_LIV]; }
    bool done() const { return !sp()->tab; }

    public:
        static bool valid_interval(vaddr beg, natq dim) {
            vaddr end = beg + dim - 1;
            return !dim || (
            // no wrap-around
            !(end < beg)
            // non inizia nel buco
            &&   norm(beg) == beg
            // non termina nel buco
            &&   norm(end) == end
            // non attraversa il buco
            &&   !((beg & BIT_SEGNO) != (end & BIT_SEGNO)));
        }

        tab_iter(paddr tab, vaddr beg, natq dim = 1, int liv = MAX_LIV);

        vaddr get_v() const {
            return sp()->cur;
        }

        [... vedere le spiegazioni successive]
        bool down();
        bool up();
        bool right();
```

```
    void post();
    void next_post();
};
```

- L'iteratore viene costruito indicando:
  - l'indirizzo fisico della tabella *tab* (normalmente una tabella di livello 4),
  - un indirizzo virtuale di partenza *beg* (di cui si vuole visitare la traduzione),
  - quanto è grande l'intervallo di indirizzi virt. che si vuole visitare (*dim*, di default 1),
  - il livello di partenza (di default *MAX_LIV*).

  ```
  tab_iter(paddr tab, vaddr beg, natq dim = 1, int liv = MAX_LIV);
  ```

- L'operatore booleano indica "se la visita è finita", cioè se tutte le entrate coinvolte nella traduzione che va da *beg* a *beg + DIM* (quest'ultimo escluso) sono state visitate.

  ```
  bool done() const { return !sp()->tab; }
  operator bool() const {
  return !done();
  }
  ```

- Con la funzione *next* indichiamo all'iteratore di spostarsi sulla prossima entrata da visitare. Il tipo di visita è *anticipata*.

  ```
  void next();
  ```

- Quando siamo fermi su un'entrata possiamo chiedere varie informazioni:
  - a che livello siamo

    ```
    int get_l() const {
        return l;
    }
    ```

  - se l'entrata è una foglia (ho $P = 0$, ho livello 1, oppure PS è settato)

    ```
    bool is_leaf() const {
        tab_entry e = get_e();
        return !(e & BIT_P) || (e & BIT_PS) || l == 1;
    }
    ```

  - il riferimento all'entrata (in modo da poterlo modificare)

    ```
    tab_entry& get_e() const {
        return get_entry(sp()->tab, i_tab(sp()->cur, l));
    }
    ```

  - l'indirizzo fisico della tabella che contiene l'entrata su cui l'iteratore è fermo

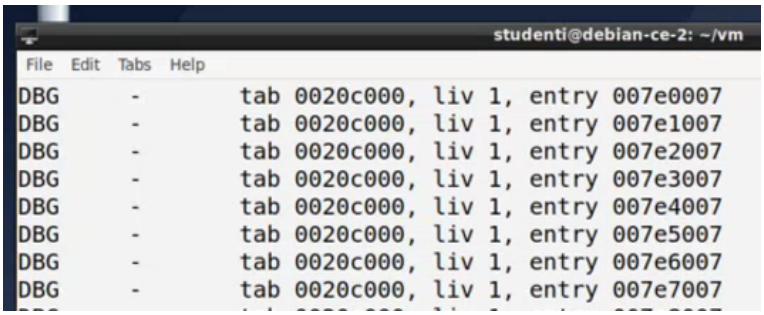
    ```
    paddr get_tab() const {
        return sp()->tab;
    }
    ```

10.3.4.6 Esempio di uso della classe *tab_iter*

Partiamo dalla radice del nostro albero, *tab4* (paddr), e scorre tutto l'intervallo che abbiamo mappato (8 MB, da 0 a $0x800000$).

```
for(tab_iter it((paddr)tab4, 0, 0x800000); it; it.next()) {
    flog(LOG_DEBUG, "tab %x, liv %d, entry %x", it.get_tab(), it.get_l(), it.get_e());
}
```

L'output è il seguente



Si includa anche un richiamo alla funzione *panic*, che non fa niente.

```
extern "C" void panic();
```

Recap. Il modulo sistema si deve preoccupare di gestire sia la memoria fisica che quella virtuale.

- **Memoria fisica**.

  Per quanto riguarda quella fisica (la RAM) abbiamo detto che una parte è ad uso esclusivo del modulo sistema (con sezioni *text, data, stack, heap* del sistema), mentre la parte rimanente è divisa in frame. Si tenga conto che tutto ciò che è presente in questa seconda area di memoria non è tutta accessibile all'utente: gli indirizzi devono essere tradotti.

- **Memoria virtuale**.

  Il sistema deve creare gli spazi di indirizzamento di ogni singolo processo: abbiamo una serie di indirizzi (col solito buco). A priori decidiamo che tutti gli indirizzi che vanno da 0 fino al buco appartengono al sistema (solo traduzioni con $U/S = 0$, quindi accessibili solo se ci troviamo a livello sistema). La parte rimanente è accessibile a livello utente, contiene traduzioni utilizzabili a livello utente (e anche a livello sistema).

10.4 Suddivisione dello spazio di indirizzamento

Sistema La parte accessibile a livello sistema si divide in tre parti, ciascuna dedicata a priori a una certa cosa:

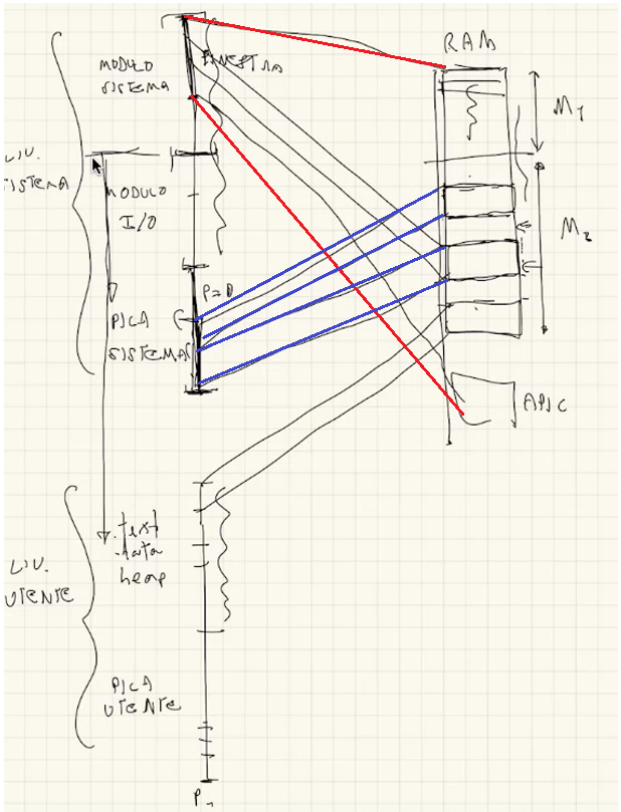
- una prima parte dedicata al *modulo sistema* (unica parte con traduzione identità);
- una dedicata al *modulo I/O*;
- una dedicata alla *pila sistema* (quella usata dal processore a livello sistema).

Nel modulo sistema garantiamo all'accesso a tutta la RAM, ma anche ad altre cose come l'APIC. Tutto il resto (modulo I/O, pila sistema e livello utente) conterrà traduzioni che portano ad $M2$.

Utente Il livello utente si divide in due parti:

- l'area con *text, data, heap*;
- pila utente.

Le pagine sono mappate sempre in $M2$.



Sono necessarie le traduzioni di modulo I/O e pila sistema?

- Teoricamente non ci sarebbe bisogno di fare queste traduzioni: potrei utilizzare gli indirizzi della finestra, cioè quelli del modulo sistema.

- Conviene avere questa seconda traduzione. Abbiamo frame allocati dinamicamente: se il sistema ha già lavorato per un po' diventa molto improbabile che i frame relativi a un processo siano tutti consecutivi. Il mapping per pila e periferiche I/O permette di creare indirizzi contigui (anche se i frame non lo sono).

Processi concorrenti, mappaggio comune Pensiamo a un altro processo, anche lui con la sua memoria virtuale. Alcune traduzioni saranno esattamente identiche a quelle di $P1$:

- modulo sistema
- modulo I/O
- *text, data, heap* del livello utente (gli indirizzi sono gli stessi, ma hanno significato diverso a seconda del processo attivo[1]).

Attenzione Alcune persone dicono all'esame che la MMU sceglie se usare la parte sopra o quella sotto in base al livello di privilegio. **Questa frase non ha alcun senso**: la MMU usa le traduzioni in base agli indirizzi ricevuti e controlla se il livello di privilegio è sufficiente per determinare l'accessibilità della traduzione. Non è un discorso di scegliere cosa usare.

10.5 Costanti dedicate alla memoria virtuale

Ritorniamo nel file *include/costanti.h* per vedere alcune costanti legati alla memoria virtuale. Abbiamo detto che la suddivisione dello spazio di indirizzamento in varie parti è fatta a priori. La memoria virtuale è allocata in unità di regioni di livello 4 (di 512 GB alla volta). In sostanza decidiamo quante entrate della tabella di livello 4 di ciascun processo sono dedicate a una cosa oppure a un'altra.

```
// tipo del driver del timer (prioritÃ  massima)
#define TIPO_TIMER 0xFF

// ( suddivisione della memoria virtuale
//   N    = Numero di entrate in root_tab
//   I   = Indice della prima entrata in root_tab
//   SIS  = SIStema
//   MIO  = Modulo IO
//   UTN  = modulo UTeNte
//   C    = Condiviso
//   P    = Privato
#define I_SIS_C 0
#define I_SIS_P 1
#define I_MIO_C 2
#define I_UTN_C       256
#define I_UTN_P       384

#define N_SIS_C 1
```

[1] Quindi a seconda della traduzione attiva.

```
#define N_SIS_P 1
#define N_MIO_C 1
#define N_UTN_C       128
#define N_UTN_P       128
// )
```

- Per ogni parte abbiamo la costante che indica l'entrata iniziale e quella che indica il numero di pagine/frame dedicati alla parte.
- **Esempi**:
  - `#define I_SIC_C    0`
    `#define N_SIC_C    1`
    L'entrata iniziale della parte sistema condivisa è la 0*, il numero di entrate dedicato alla parte è* 1
  - `#define I_UTN_C    256`
    `#define N_UTN_C    128`
    L'entrata iniziale della parte utente condivisa è la 256*, il numero di entrate dedicato alla parte è* 128
- Si consideri che con *parte sistema privata* si intende la *pila sistema*. Si osservi anche il salto che avviene tra la parte dedicata al modulo I/O e la *parte utente condivisa* (si vuole superare il buco, utilizzando questo per dividere ciò che è rivolto al sistema e ciò che è rivolto all'utente.)

10.6 Costanti dedicate alla memoria fisica

Rimaniamo nel file *include/costanti.h*: sono presenti delle costanti dedicate alla memoria fisica.

```
// ( varie dimensioni
#define KiB 1024UL
#define MiB (1024*KiB)
#define GiB (1024*MiB)
#define DIM_PAGINA 4096UL
#define DIM_BLOCK 512UL
// )

// ( limiti modificabili
// ...
// dimensione della memoria fisica
#define MEM_TOT (32*MiB)
// dimensione dello heap utente
#define DIM_USR_HEAP (1*MiB)
// dimensione degli stack utente
#define DIM_USR_STACK (64*KiB)
// dimensione dello heap del modulo I/O
#define DIM_IO_HEAP (1*MiB)
// dimensione degli stack sistema
#define DIM_SYS_STACK (4*KiB)
// ...
// )
```

- Per prima cosa abbiamo una serie di costanti con cui rappresentiamo le varie unità di misura: *KiB*, *MiB*, *GiB*, *DIM_PAGINA* e *DIM_BLOCK*.
- Attraverso altre costanti indichiamo:
  - la dimensione della memoria fisica (*MEM_TOT*);
  - la dimensione dello heap utente (*DIM_USR_HEAP*);
  - la dimensione delle pile utente (*DIM_USR_STACK*);
  - la dimensione dello heap del modulo I/O (*DIM_IO_HEAP*);
  - la dimensione delle pile sistema (*DIM_SYS_STACK*).
- Le cose indicate sono tutte dimensioni fisiche. Nello spazio fisico le modifiche hanno un costo: ogni cosa dovrà finire da qualche parte, e dovremo fare le corrispondenti traduzioni.

10.7 Descrittore di processo

Riprendiamo il descrittore di processo in *sistema/sistema.cpp*.

```
struct des_proc {
    natw id;
    natw livello;
    natl precedenza;
    vaddr punt_nucleo; /* Puntatore a pila sistema */
    natq contesto[N_REG];
    paddr cr3;

    des_proc *puntatore;
};
```

Ogni processo si ricorda l'indirizzo della sua tabella di livello 4 (*cr3*).

10.8 Aggiornamento del registro CR3 nella *carica_stato*

Una parte del codice della funzione *carica_stato*, che inizialmente abbiamo sorvolato, è dedita all'aggiornamento del registro CR3 e quindi al passaggio allo spazio di indirizzamento del processo entrante.

```
// carica nei registri del processore lo stato contenuto nel des_proc del
// processo puntato da esecuzione.  Questa funzione sporca tutti i registri.
carica_stato:
    movq esecuzione, %rbx

    popq %rcx   //ind di ritorno, va messo nella nuova pila

    // nuovo valore per cr3
    movq CR3(%rbx), %r10
    movq %cr3, %rax
    cmpq %rax, %r10
    je 1f // evitiamo di invalidare il TLB
    // se cr3 non cambia
```

```
    movq %r10, %rax
    movq %rax, %cr3 // il TLB viene invalidato
1:
    // anche se abbiamo cambiato cr3 siamo sicuri che l'esecuzione prosegue
    // da qui, perché ci troviamo dentro la finestra FM che è comune a
    // tutti i processi
    movq RSP(%rbx), %rsp    //cambiamo pila
    pushq %rcx              //rimettiamo l'indirizzo di ritorno
```

Nel codice evitiamo di sovrascrivere CR3 se già presente nel registro è uguale a quello che vogliamo porre. Lo facciamo per evitare di svuotare il TLB quando non necessario (ricordare cosa abbiamo detto sull'istruzione *mov*): svuotarlo è un problema dal punto di vista delle prestazioni (centinaia di clock in più per una mossa che poteva essere evitata).

10.9 Indirizzi virtuali e fisici nei file ELF

Modulo sistema

```
studenti@debian-ce-2:~/nucleo-6.5$ readelf -Wl build/sistema

Elf file type is EXEC (Executable file)
Entry point 0x200120
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0x0000000000200000 0x0000000000200000 0x00b800 0x00b800 R E 0x1000
  LOAD           0x00bfe0 0x000000000020cfe0 0x000000000020cfe0 0x000212 0x0134a8 RW  0x1000
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RWE 0x10
  GNU_RELRO      0x00bfe0 0x000000000020cfe0 0x000000000020cfe0 0x000020 0x000020 RW  0x8
```

Il modulo sistema viene collegato a indirizzi virtuali che sono effettivamente fisici. Il bootloader crea una traduzione identità, e gli cede il controllo. Grazie alla finestra sulla RAM continueremo ad utilizzare questi indirizzi.

Modulo I/O

```
studenti@debian-ce-2:~/nucleo-6.5$ readelf -Wl build/io

Elf file type is EXEC (Executable file)
Entry point 0x10000000120
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0x0000010000000000 0x0000010000000000 0x003c88 0x003c88 R E 0x1000
  LOAD           0x010fe0 0x000001000010fe0 0x0000010000010fe0 0x0000e8 0x00f108 RW  0x10000
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RWE 0x10
  GNU_RELRO      0x010fe0 0x0000010000010fe0 0x0000010000010fe0 0x000020 0x000020 RW  0x8
```

Il modulo I/O sta da tutt'altra parte. Gli indirizzi sono molto più grandi e sono collocati nella parte dedicata al modulo I/O.

Modulo utente

```
studenti@debian-ce-2:~/nucleo-6.5$ readelf -Wl build/utente

Elf file type is EXEC (Executable file)
Entry point 0xffff8000000001d3
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0xffff800000000000 0xffff800000000000 0x0016d0 0x0016d0 R E 0x1000
  LOAD           0x001fd8 0xffff800000002fd8 0xffff800000002fd8 0x000028 0x000090 RW  0x1000
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RWE 0x10
  GNU_RELRO      0x001fd8 0xffff800000002fd8 0xffff800000002fd8 0x000028 0x000028 RW  0x8
```

Il modulo utente utilizza gli indirizzi oltre il buco (attenzione a *ffff*).

10.10 Strutture dati per la gestione dei frame

10.10.1 Descrittore di frame

I frame sono descritti attraverso la struttura *des_frame*, il descrittore di frame.

```
struct des_frame {
    union {
        // numero di entrate valide (se il frame contiene una tabella)
        natw nvalide;
        // lista di frame liberi (se il frame e' libero)
        natl prossimo_libero;
    };
};
```

Uso della union nel descrittore di frame Per ridurre lo spazio richiesto dall'array dei frame usiamo una *union*: le informazioni nello struct non ci interessano mai in contemporanea.

- *nvalide* mi interessa quando il frame è occupato da una tabella, mi indica il numero di entrate valide in una tabella ($P = 1$). Possiamo deallocare una tabella solo se tutti i P sono uguali a zero (la cosa può servirmi al termine di un processo, devo verificare quali tabelle posso deallocare e quali no senza doverle scorrere per intero).
- *prossimo_libero* mi interessa solo quando il frame è libero, per sapere qual è il frame libero successivo.

10.10.2 Array dei descrittori di frame e numero totale dei frame

I descrittori di frame sono posti nel seguente array

```
des_frame vdf[N_FRAME];
```

dove *N_FRAME* è il numero di frame ($M1 + M2$), che può essere determinato a priori dividendo la dimensione della memoria per la dimensione della pagina.

```
natq const N_FRAME = MEM_TOT / DIM_PAGINA;
```

Le costanti usate sono quelle della *libce* introdotte qualche pagina indietro

10.10.3 Lista dei frame liberi: testa e contatore

Indice del primo frame libero (all'interno dell'array).

```
natq primo_frame_libero;
```

Con *prossimo_libero* andiamo a costruire una lista di frame liberi.

Numero di frame liberi Con la seguente variabile contiamo il numero di frame nella lista

```
natq num_frame_liberi;
```

10.10.4 Numero dei frame in $M1$ e in $M2$

Nelle seguenti variabili poniamo il numero di frame riservati per $M1$ e il numero di frame riservati per $M2$

```
natq N_M1;
natq N_M2;
```

10.11 Funzioni di utilità per il contatore *nvalide* nel *des_frame*

Le seguenti funzioni di utilità manipolano il contatore *nvalide* in un descrittore di frame.

```
void inc_ref(paddr f) {
    vdf[f / DIM_PAGINA].nvalide++;
}

void dec_ref(paddr f) {
    vdf[f / DIM_PAGINA].nvalide--;
}

natl get_ref(paddr f) {
    return vdf[f / DIM_PAGINA].nvalide;
}
```

Esse richiedono in ingresso un indirizzo fisico relativo al frame.

- *inc_ref* incrementa il contatore delle entrate valide.
- *dec_ref* decrementa il contatore delle entrate valide.
- *get_ref* restituisce il valore del contatore delle entrate valide.

10.12 Funzioni per la gestione della paginazione

10.12.1 Funzione per l'occupazione di un frame libero (*alloca_frame*)

```
// estrea un frame libero dalla lista, se non vuota
paddr alloca_frame() {
    if (!num_frame_liberi) {
        flog(LOG_ERR, "out of memory");
```

```
        return 0;
    }
    natq j = primo_frame_libero;
    primo_frame_libero = vdf[primo_frame_libero].prossimo_libero;
    vdf[j].prossimo_libero = 0;
    num_frame_liberi--;

    return j * DIM_PAGINA;
}
```

- Controllo se la memoria non sia tutta occupata attraverso il contatore *num_frame_liberi*. Se tutti i frame sono occupati segnalo con *flog* e mi fermo restituendo 0.
- Se la memoria non è occupata prendo la testa della lista dei frame liberi, *primo_frame_libero* e ne restituisco l'indirizzo fisico della base. L'indirizzo fisico si calcola moltiplicando l'indice della testa per *DIM_PAGINA*.
- Modifico il puntatore *primo_frame_libero*: abbiamo rimosso la testa della lista.
- Modifico anche il puntatore *prossimo_libero* nel descrittore stesso, azzerandolo (non fa più parte della lista).
- Decremento il numero di frame liberi (*num_frame_liberi*).

La funzione si presta bene come *getpaddr* nella funzione *map*, affrontata poco più avanti.

10.12.2 Funzione per il rilascio di un frame occupato (*rilascia_frame*)

```
// rende di nuovo libera il frame descritto da df
void rilascia_frame(paddr f) {
    natq j = f / DIM_PAGINA;
    if (j < N_M1) {
        panic("tentativo di rilasciare un frame di M1");
    }
    vdf[j].prossimo_libero = primo_frame_libero;
    primo_frame_libero = j;
    num_frame_liberi++;
}
```

- Quando vogliamo rilasciare un frame passiamo in ingresso l'indirizzo fisico della base del frame.
- Si calcola il numero di frame dividendo l'indirizzo per *DIM_PAGINA*.
- Se il frame risulta appartenente ad $M1$ blocco tutto chiamando la *panic*.
- Colloco il descrittore rilasciato nella lista dei frame liberi. Non ha senso mettere il descrittore in un posto specifico (come invece facevamo coi processi, *precedenza*): grazie alla MMU non ci fa nessuna differenza avere un frame o un altro. Il rilascio più semplice possibile consiste nel mettere in testa il descrittore del frame rilasciato.

La funzione si presta bene come *putpaddr* nella funzione *unmap* (l'opposto della *map*).

10.12.3 Allocazione della tabella con *alloca_tab* (usa *alloca_frame*)

```
// alloca un frame libero destinato a contenere una tabella
paddr alloca_tab() {
    paddr f = alloca_frame();
    if (f) {
        memset(reinterpret_cast<void*>(f), 0, DIM_PAGINA);
    }
    vdf[f / DIM_PAGINA].nvalide = 0;
    return f;
}
```

- Allocare una tabella significa allocare un frame. Quindi chiamo la *alloca_frame*.
- Il valore restituito dalla *alloca_frame* è 0 se non è stato possibile allocare il frame. Se l'allocazione ha successo chiamo la *memset*, con cui resetto il contenuto del frame (scrive dentro l'indirizzo il byte 0 per *DIM_PAGINA* volte).
- Pongo a 0 il contatore *nvalide*.
- Restituisco l'indirizzo indicato dalla *alloca_frame*.

10.12.4 Rilascio della tabella con *rilascia_tab* (usa *rilascia_frame*)

```
// dealloca un frame che contiene una tabella, controllando che non contenga entrate valide
void rilascia_tab(paddr f) {
    if (int n = get_ref(f)) {
        flog(LOG_ERR, "tentativo di deallocare la tabella %x con %d entrate valide", f, n);
        panic("errore interno");
    }
    rilascia_frame(f);
}
```

- Quando rilasciamo una tabella dobbiamo controllare che non sia utilizzata con la funzione di utilità *get_ref*.
- Se il controllo ci dice che la tabella non è utilizzata allora rilasciamo il frame chiamando la *rilascia_frame*.

10.12.5 Settaggio dell'entrata di una tabella con *set_entry*

```
// setta l'entrata j-esima della tabella 'tab' con il valore 'se'.
// Si preoccupa di aggiustare opporutinamente il contatore delle
// entrate valide.
void set_entry(paddr tab, natl j, tab_entry se) {
    tab_entry& de = get_entry(tab, j);
    if ((se & BIT_P) && !(de & BIT_P)) {
        inc_ref(tab);
    } else if (!(se & BIT_P) && (de & BIT_P)) {
        dec_ref(tab);
    }
    de = se;
}
```

- La funzione richiede l'indirizzo fisico della tabella *tab*, l'indice j dell'entrata, e l'entrata *se*.
- Recuperiamo il contenuto attuale dell'entrata j nella tabella *tab* con la funzione *get_entry*.
- Gestiamo l'incremento di *nvalide* nel descrittore di frame.
  - Se nella nuova c'è il bit $P = 1$ e nella vecchia $P = 0$ incrementiamo con *inc_ref*.
  - Se nella nuova c'è il bit $P = 0$ e nella vecchia $P = 1$ decrementiamo con *dec_ref*.
- Concludiamo sostituendo l'entrata vecchia con la nuova.

10.12.6 Copia di descrittori da una tabella a un'altra (*copy_des*)

```
// copia 'n' descrittori a partire da quello di indice 'i' dalla
// tabella di indirizzo 'src' in quella di indirizzo 'dst'
void copy_des(paddr src, paddr dst, natl i, natl n) {
    for (natl j = i; j < i + n && j < 512; j++) {
        tab_entry se = get_entry(src, j);
        set_entry(dst, j, se);
    }
}
```

- Con la funzione possiamo copiare entrate da una tabella *src* alla *dst*, ne copio n a partire dall'entrata $i-$esima.
- Faccio un for, ogni volta leggo l'entrata sorgente con *get_entry* e setto l'entrata destinataria con *set_entry*. L'ultima funzione mi gestisce anche l'aggiornamento del contatore *nvalide*.

10.12.7 Settaggio di descrittori uguali in una tabella (*set_des*)

```
// setta 'n' descrittori a partire da quello di indice 'i' nella
// tabella di indirizzo 'dst' con valore 'e'
void set_des(paddr dst, natl i, natl n, tab_entry e) {
    for (natl j = i; j < i + n && j < 512; j++) {
        set_entry(dst, j, e);
    }
}
```

- Imposto n entrate della tabella *dst*, tutte uguali ad e, a partire dalla $i-$esima.
- Nel for utilizziamo solo la funzione di utilità *set_entry*, con cui gestisco anche l'aggiornamento del contatore *nvalide*.

10.13 Funzione *map*

La funzione più utile di tutte (cit.).

```
template<typename T>
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T getpaddr, int ps_lvl = 1);
```

Scopo La funzione crea una traduzione per tutti gli indirizzi dell'intervallo [begin, end[nel trie di radice *tab*. Ogni traduzione dovrà avere i flag *flags* indicati. Opzionalmente possiamo settare il *page size* uguale ad 1 in un livello diverso dal livello 1: indicando come *page size level* 2 posso creare pagine da 2 MB, ad esempio. Con *getpaddr* indichiamo l'indirizzo fisico corrispondente.

Vediamo

- **Il *typename* e il parametro di ingresso *getpaddr*.**

  Vogliamo creare una traduzione per un intervallo di indirizzi. Molte delle cose da fare non dipendono da cosa vogliamo far corrispondere a questi indirizzi. La funzione è stata resa generica, in modo da utilizzarla in più situazioni possibili: non sa lei direttamente quali indirizzi fisici associare a quelli virtuale. Per fare questa associazione ricorriamo all'oggetto *getpaddr*, il cui tipo T è stabilito dal *typename* (studiato ad *Algoritmi e Strutture dati*).

  All'interno del codice della funzione troverò a un certo punto il seguente assegnamento

  ```
  new_f = getpaddr(v);
  ```

  esso mi restituisce, dato l'indirizzo virtuale v, il corrispondente indirizzo fisico. Dalla sintassi deduciamo che *getpaddr* funziona con:

  - puntatori a funzione;
  - classi dove ridefiniamo l'operatore parentesi (la classe può ricordarsi anche altre cose).

  In tutti gli altri casi il compilatore segnala errore.

- **Verifica dei parametri di ingresso**.

  - **Indirizzi *begin* e *end*.**

    La prima cosa che dobbiamo fare nel codice è verificare che *begin* ed *end* siano validi. Calcoliamo la dimensione della regione e verifichiamo che l'allineamento alle pagine

    ```
    natq dr = dim_region(ps_lvl - 1);
    ```

    * Verifico che *begin* sia allineato alle pagine di livello *ps_lvl*.
      `if(begin & (dr - 1)) ...`
      La condizione risulta vera se esiste almeno un bit tra i meno significativi che non è nullo (per avere l'allineamento rispetto a una certa dimensione i primi bit devono essere nulli).
    * Verifico che *end* sia allineato alle pagine di livello *ps_lvl*.
      `if(end & (dr - 1)) ...`
      Valgono gli stessi ragionamenti fatti con *begin*.

```
natq dr = dim_region(ps_lvl - 1);
if (begin & (dr - 1)) {
    flog(LOG_ERR, "begin=%p non allineato alle pagine di livello %d", begin, ps_lvl);
    panic("chiamata di map() non valida");
}
if (end & (dr - 1)) {
    flog(LOG_ERR, "end=%p non allineato alle pagine di livello %d", end, ps_lvl);
    panic("chiamata di map() non valida");
}
```

  - **Parametro di ingresso *flags***.
    Verifico che gli unici flag modificati siano tra RW, US, PWT e PCD. Prendo le relative maschere (già viste nelle costanti), le unisco con l'operatore OR, e nego il risultato. Ottengo una maschera dove sono uguali ad 1 tutti i bit tranne quelli di cui è consentita la modifica.

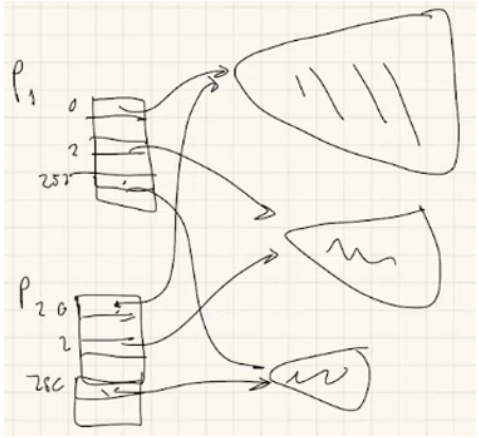
    ```
    if (flags & ~(BIT_RW|BIT_US|BIT_PWT|BIT_PCD)) {
        panic("flags contiene bit non validi (ammessi RW, US, PWT e PCD)");
    }
    ```

  - ***ps_lev***.
    Verifico che *ps_lvl* sia stato impostato correttamente:
    * Non deve essere minore di 1
    * Non deve essere maggiore del livello massimo *MAX_PS_LVL*

    ```
    if (ps_lvl < 1 || ps_lvl > MAX_PS_LVL) {
        flog(
            LOG_ERR,
            "ps_lvl %d non ammesso (deve essere compreso tra 2 e %d)",
            ps_lvl,
            MAX_PS_LVL
        );
        panic("chiamata di map() non valida");
    }
    ```

- **Uso della funzione**.
  Usiamo la funzione per creare tutte le parti dello spazio di indirizzamento.

  - Quelle condivise sono le stesse in ogni processo, dunque la loro traduzione può essere fatta all'avvio.



Abbiamo sotto-alberi condivisi tra più alberi. Li creiamo inizialmente per il processo *dummy* (processo che è sempre vivo).

– Ogni volta che creiamo un processo ci limitiamo a creare solo le parti di albero relative esclusivamente a quel processo. Per quanto riguarda le parti condivise ci limitiamo a porre i relativi indirizzi fisici nella tabella di livello 4 (non dobbiamo ricreare queste parti nuovamente).

All'avvio del nucleo, dopo l'inizializzazione dell'APIC, abbiamo

```
INF     -       APIC inizializzato
INF     -       Numero di frame: 545 (M1) 7647 (M2)
INF     -       sis/cond [0000000000000000, 0000008000000000)
INF     -       sis/priv [0000008000000000, 0000010000000000)
INF     -       io /cond [0000010000000000, 0000018000000000)
INF     -       usr/cond [ffff800000000000, ffffc00000000000)
INF     -       usr/priv [ffffc00000000000, 0000000000000000)
INF     -       Creata finestra sulla memoria centrale:  [0000000000001000, 0000000002000000)
INF     -       Creata finestra per memory-mapped-IO:    [00000000fec00000, 0000000100000000)
INF     -       mappo il modulo I/O:
INF     -        - segmento sistema read-only  mappato a [0000010000000000, 0000010000004000)
INF     -        - segmento sistema read/write mappato a [0000010000010000, 0000010000021000)
INF     -        - heap:                                 [0000010000021000, 0000010000121000)
INF     -        - entry point: start [io.s:8]
INF     -       mappo il modulo utente:
INF     -        - segmento utente  read-only  mappato a [ffff800000000000, ffff800000002000)
INF     -        - segmento utente  read/write mappato a [ffff800000002000, ffff800000004000)
INF     -        - heap:                                 [ffff800000004000, ffff800000104000)
INF     -        - entry point: start [utente.s:10]
INF     -       Create le traduzioni per le parti condivise
INF     -       Frame liberi: 7098 (M2)
```

– Indico il numero di frame in $M1$ e in $M2$ (numero stabilito a priori).

– Calcolo e stampo gli indirizzi virtuali estremi dei vari intervalli definiti nello spazio di indirizzamento: sis/cond, sis/priv, io/cond, usr/cond, usr/priv.

– Creo le varie parti condivise una volta per tutte:

  * la finestra sulla parte centrale (solo sulla RAM, di grandezza variabile),
  * la finestra sul *memory mapped I/O*,
  * il modulo I/O,
  * il modulo utente.

  Quando il controllo viene passato al modulo sistema questo interpreta il file ELF e vede a che indirizzi virtuali devono andare le varie parti del file ELF.

Capitolo 11

Funzione *main* del modulo sistema

Con le spiegazioni precedenti abbiamo tutte le nozioni necessarie per approfondire le prime funzioni del modulo sistema eseguite dall'emulatore.

11.1 Premessa: funzione *start* (indirizzi moduli, IDT, costruttori, chiamata main)

Prendiamo la funzione *start* in *sistema.s*, eseguita non appena il bootloader cede il controllo.

```
.globl  _start, start
_start: // entry point
start:
    pushq %rdi

    // inizializziamo la IDT
    call init_idt

    // Il C++ prevede che si possa eseguire del codice prima di main (per
    // es. nei costruttori degli oggetti globali). gcc 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 a main.  Poiche' abbiamo
    // compilato il modulo con -nostdlib dobbiamo provvedere da soli a
    // chiamare queste funzioni, altrimenti gli oggetti globali non saranno
    // inizializzati correttamente.
    movabs $__init_array_start, %rbx
1:  cmpq $__init_array_end, %rbx
    je 2f
    call *(%rbx)
    addq $8, %rbx
    jmp 1b

    // il resto dell'inizializzazione e' scritto in C++
2:  popq %rdi
    call main
    // se arrivamo qui c'e' stato un errore, fermiamo la macchina
    hlt
```

1. Pongo temporaneamente in pila il contenuto del registro RDI. Il contenuto consiste nel puntatore a una struttura *multiboot_info*, che contiene l'indirizzo dei moduli I/O e utente.
2. Chiamiamo *init_idt* per inizializzare la *Interrupt Descriptor Table* (nei modi visti prima).
3. Effettuo una serie di chiamate di costruttori relativi a oggetti globali.
4. Recupero dalla pila il contenuto del registro RDI.
5. Chiamata della funzione *main*, scritta in C++ e contenente il resto dell'inizializzazione. Il codice della funzione è in *sistema.cpp*. Si osservi che

   ```
   extern "C" void main(paddr mbi) ...
   ```

   RDI è parametro di ingresso della funzione *main*.

11.2 Primo processo, inizializzazione di GDT e APIC, associazione del tipo al piedino del timer

Nelle prime righe:

- definisco le caratteristiche del primo processo

  ```
  // anche se il primo processo non è completamente inizializzato,
  // gli diamo un identificatore, in modo che compaia nei log
  init.id = 0xFFFF;
  init.precedenza = MAX_PRIORITY;
  init.cr3 = readCR3();
  esecuzione = &init;
  ```

  dove *init* è un *des_proc* inizializzato nello stesso file

  ```
  // un primo des_proc, allocato staticamente, da usare durante l'inizializzazione
  des_proc init;
  ```

- Chiamo *init_gdt* per inizializzare la *Global Descriptor Table* (nei modi visti prima)

  ```
  flog(LOG_INFO, "Nucleo di Calcolatori Elettronici, v6.5");
  init_gdt();
  flog(LOG_INFO, "GDT inizializzata");
  ```

- Inizializzo l'APIC e associo il tipo 2 al piedino del timer.

  ```
  apic_init(); // in libce
  apic_reset(); // in libce
  apic_set_VECT(2, TIPO_TIMER);
  flog(LOG_INFO, "APIC inizializzato");
  ```

11.3 Inizializzazione di $M2$ e indirizzi virtuali dei vari intervalli

All'interno del codice si ha la chiamata della funzione *init_frame*, con cui inizializziamo $M2$. La parte $M1$ è già disponibile: è stata caricata dal *bootloader* e contiene il modulo sistema.

```
// inizializziamo la parte M2
init_frame();
flog(LOG_INFO, "Numero di frame: %d (M1) %d (M2)", N_M1, N_M2);

flog(LOG_INFO, "sis/cond [%p, %p)", ini_sis_c, fin_sis_c);
flog(LOG_INFO, "sis/priv [%p, %p)", ini_sis_p, fin_sis_p);
flog(LOG_INFO, "io /cond [%p, %p)", ini_mio_c, fin_mio_c);
flog(LOG_INFO, "usr/cond [%p, %p)", ini_utn_c, fin_utn_c);
flog(LOG_INFO, "usr/priv [%p, %p)", ini_utn_p, fin_utn_p);
```

11.3.1 Funzione *init_frame*

La funzione *init_frame* si occupa di inizializzare i frame relativi alla parte di memoria non ancora utilizzata dopo il caricamento del modulo sistema: $M2$.

```
void init_frame() {
    // primo frame di M2
    paddr fine_M1 = allinea(reinterpret_cast<paddr>(&end), DIM_PAGINA);
    // numero di frame in M1 e indice di f in vdf
    N_M1 = fine_M1 / DIM_PAGINA;
    // numero di frame in M2
    N_M2 = N_FRAME - N_M1;

    if (!N_M2)
        return;

    // creiamo la lista dei frame liberi, che inizialmente contiene tutti i  frame di M2
    primo_frame_libero = N_M1;
#ifndef N_STEP
    // alcuni esercizi definiscono N_STEP == 2 per creare mapping non
    // contigui in memoria virtale e controllare meglio alcuni possibili
    // bug
    #define N_STEP 1
#endif
    for (natq i = N_M1; i < N_FRAME - N_STEP; i++) {
        vdf[i].prossimo_libero = i + N_STEP;
        num_frame_liberi++;
    }
    vdf[N_FRAME - N_STEP].prossimo_libero = 0;
}
```

- **Cosa conosciamo?**

  Si consideri la variabile *end*, che contiene l'indirizzo del primo byte non occupato dal modulo sistema (calcolato dal collegatore). Conosciamo il numero di frame totali *N_FRAME*, calcolato dividendo la dimensione totale della memoria e la dimensione di ciascuna pagina.

- **Funzione *allinea*.**

  Trovo il primo frame libero di $M2$

```
paddr fine_M1 = allinea(reinterpret_cast<paddr>(&end), DIM_PAGINA);
```

utilizzando la funzione di utilità *allinea*, che calcola il primo indirizzo multiplo di *DIM_PAGINA* maggiore del primo argomento.

- A partire dal numero precedente otteniamo il numero di frame in $M1$ e quelli in $M2$

```
N_M1 = fine_M1 / DIM_PAGINA;
N_M2 = N_FRAME - N_M1;
```

- Se non ci sono frame disponibili in $M2$ mi fermo subito.

```
if(!N_M2)
    return;
```

- Se ci sono frame creiamo la lista dei frame liberi attraverso un for: facciamo in modo che ogni descrittore di frame presente nel vettore *vdf* punti al successivo. L'ultimo descrittore nel vettore non punta a niente, mentre il primo (la testa) viene puntato da *primo_frame_libero*.

11.3.2 Indirizzi virtuali iniziali e finali di tutti gli intervalli

Nelle seguenti variabili, poste nella sezione dedicata alla paginazione di *sistema.cpp*, andiamo a memorizzare degli indirizzi...

```
// indirizzo virtuale di partenza delle varie zone della memoria virtuale dei processi
const vaddr ini_sis_c = norm(I_SIS_C * dim_region(MAX_LIV - 1)); // sistema condivisa
const vaddr ini_sis_p = norm(I_SIS_P * dim_region(MAX_LIV - 1)); // sistema privata
const vaddr ini_mio_c = norm(I_MIO_C * dim_region(MAX_LIV - 1)); // modulo IO
const vaddr ini_utn_c = norm(I_UTN_C * dim_region(MAX_LIV - 1)); // utente condivisa
const vaddr ini_utn_p = norm(I_UTN_P * dim_region(MAX_LIV - 1)); // utente privata

// indirizzo del primo byte che non appartiene alla zona specificata
const vaddr fin_sis_c = ini_sis_c + dim_region(MAX_LIV - 1) * N_SIS_C;
const vaddr fin_sis_p = ini_sis_p + dim_region(MAX_LIV - 1) * N_SIS_P;
const vaddr fin_mio_c = ini_mio_c + dim_region(MAX_LIV - 1) * N_MIO_C;
const vaddr fin_utn_c = ini_utn_c + dim_region(MAX_LIV - 1) * N_UTN_C;
const vaddr fin_utn_p = ini_utn_p + dim_region(MAX_LIV - 1) * N_UTN_P;
```

- Nel codice vengono calcolati gli indirizzi iniziali e finali di tutti gli intervalli definiti a inizio lezione. Sono tutti *vaddr*.

- Per quanto riguarda le prime costanti devo trovare l'indirizzo di partenza dell'area. Moltiplico l'indice per la dimensione della regione di livello *MAX_LIV* - 1.

- Normalizzo con la funzione di utilità *norm* per sistemare i bit più significativi.

11.4 Allocazione di una tabella radice e creazione della finestra sulla memoria fisica

```
// creiamo le parti condivise della memoria virtuale di tutti i processi
// le parti sis/priv e usr/priv verranno create da crea_processo()
// ogni volta che si attiva un nuovo processo
paddr root_tab = alloca_tab();
if (!root_tab)
    goto error;

// finestra di memoria, che corrisponde alla parte sis/cond
if(!crea_finestra_FM(root_tab))
    goto error;
```

- Allochiamo una prima tabella radice con *alloca_tab*.
- Dobbiamo creare la finestra di memoria. In realtà il bootloader ne ha già creata una, ma non ci piace:
  - è accessibile a livello utente;
  - l'indirizzo 0 è mappabile.

  facciamo queste cose chiamando la funzione *crea_finestra_fm*.

11.4.1 Funzione *crea_finestra_fm*

La funzione *crea_finestra_fm* crea la finestra che dicevamo prima.

```
bool crea_finestra_FM(paddr root_tab) {
    auto identity_map = [] (vaddr v) -> paddr { return v; };
    // mappiamo tutta la memoria fisica:
    // - a livello sistema (bit U/S non settato)
    // - scrivibile (bit R/W settato)
    // - con pagine di grandi dimensioni (bit PS)
    //   (usiamo pagine di livello 2 che sono sicuramente disponibili)

    // vogliamo saltare la prima pagina per intercettare *NULL, e inoltre
    // vogliamo settare per il bit PWT per la pagina che contiene la memoria
    // video.  Per farlo dobbiamo rinunciare a settare PS per la prima regione
    natq first_reg = dim_region(1);
    if (map(root_tab, DIM_PAGINA, first_reg, BIT_RW, identity_map) != first_reg)
        return false;

    // settiamo il bit PWT per la pagina che contiene la memoria video.
    // Usiamo un tab_iter su una sola pagina, fermandoci sul descrittore
    // "foglia" che contiene la traduzione.
    tab_iter it(root_tab, 0xb8000, DIM_PAGINA);
    while (it.down())
        ;

    tab_entry& e = it.get_e();
    e |= BIT_PWT;
```

```
    // mappiamo il resto della memoria con PS settato
    if (MEM_TOT > first_reg) {
        if (map(root_tab, first_reg, MEM_TOT, BIT_RW, identity_map, 2) != MEM_TOT)
            return false;
    }

    flog(LOG_INFO, "Creata finestra sulla memoria centrale:  [%p, %p)", DIM_PAGINA, MEM_TOT);

    // Mappiamo gli ultimi 20MiB prima di 4GiB settando sia PWT che PCD.
    // Questa zona di indirizzi è utilizzata dall'APIC per mappare i propri registri.
    vaddr start_pci = 4*GiB - 20*MiB,
    end_pci = 4*GiB;
    if (map(root_tab, start_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map, 2) != end_pci)
        return false;

    flog(LOG_INFO, "Creata finestra per memory-mapped-IO:    [%p, %p)", start_pci, end_pci);
    return true;
}
```

Osservazione della funzione *map*. Osserviamo dall'uso della funzione *map* che questa restituisce l'ultimo indirizzo mappato. Quando chiamiamo la funzione indichiamo un estremo finale: se la funzione restituisce l'estremo finale della finestra significa che tutto è andato bene, altrimenti qualcosa è andato storto. In alcuni casi (anche in prove d'esame) può essere necessario chiamare la funzione *unmap*, per annullare il "mappaggio" fatto fino all'indirizzo restituito.

- **Indirizzo** 0.

  La prima pagina non viene mappata a causa dell'indirizzo 0, che vogliamo disabilitare e usare per *nullptr*. La cosa si risolve facilmente ignorando i primi indirizzi.

- **Pagina contenente la memoria video**.

  Usiamo *map* nella tabella radice *root_tab*, a partire da DIM_PAGINA fino alla fine della prima regione. Si pone il bit RW (ma non US) e la funzione *identity_map* (scritta con una notazione diversa dal solito) per restituire l'identità.

  ```
  // vogliamo saltare la prima pagina per intercettare *NULL, e inoltre
  // vogliamo settare per il bit PWT per la pagina che contiene la memoria
  // video.  Per farlo dobbiamo rinunciare a settare PS per la prima regione
  natq first_reg = dim_region(1);
  if (map(root_tab, DIM_PAGINA, first_reg, BIT_RW, identity_map) != first_reg)
      return false;
  ```

  Perchè mappiamo solo fino alla prima regione?

  Vogliamo mettere il bit PWT solo sulla memoria video (che sta in mezzo, dobbiamo lavorare su delle pagine in particolare). Lo facciamo utilizzando un iteratore.

```
// settiamo il bit PWT per la pagina che contiene la memoria video.
// Usiamo un tab_iter su una sola pagina, fermandoci sul descrittore
// "foglia" che contiene la traduzione.
tab_iter it(root_tab, 0xb8000, DIM_PAGINA);
while (it.down())
;

tab_entry& e = it.get_e();
e |= BIT_PWT;
```

Col while scorriamo l'albero arrivando a una foglia. Sappiamo che sicuramente la foglia avrà bit $P = 1$ (abbiamo appena sistemato le traduzioni con la *map*). Quando arriviamo alla foglia settiamo il flag con l'operatore OR. Non posso agire su aree di memorie molto specifiche, come in questo caso, se mappo gli indirizzi usando il *Page Size Flag* (una traduzione mi identificherebbe un'area di memoria troppo grande).

- **Pagine rimanenti**.

  A questo punto possiamo mappare il resto della memoria in un colpo solo, ricorrendo al *Page Size Flag*.

```
// mappiamo il resto della memoria con PS settato
if (MEM_TOT > first_reg) {
    if (map(root_tab, first_reg, MEM_TOT, BIT_RW, identity_map, 2) != MEM_TOT)
    return false;
}
```

- ***Memory-mapped I/O***.

  Concludiamo mappando in memoria gli indirizzi relativi alle periferiche (*Memory-mapped I/O*). In particolare è presente l'APIC, che utilizza indirizzi di memoria per i registri.

```
// Mappiamo gli ultimi 20MiB prima di 4GiB settando sia PWT che PCD.
// Questa zona di indirizzi è utilizzata dall'APIC per mappare i propri registri.
vaddr start_pci = 4*GiB - 20*MiB,
end_pci = 4*GiB;
if (map(root_tab, start_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map, 2) != end_pci)
    return false;
```

  Gli indirizzi si trovano negli ultimi 20 MiB, alla fine dei primi 4 GiB. Per il calcolo degli estremi utilizziamo le costanti *GiB* e *MiB*. Mettiamo tra i flag RW, PCD E PWT (la politica deve essere diversa per forza, in particolare è vitale disattivare la cache).

Osservazione da ricordare

Non abbiamo usato il bit U/S. La finestra si usa solo con livello di privilegio sistema.

11.5 Creazione dello spazio condiviso

Dove siamo arrivati Con lo step precedente abbiamo creato la finestra, quindi lo spazio di indirizzamento virtuale necessario per lavorare a livello sistema. Adesso ci rimane da creare e mappare lo spazio rimanente, riservato a modulo I/O e modulo utente.

Codice

```
// parti io/cond e usr/cond, che contengono i segmenti ELF dei
// moduli I/O e utente caricati dal boot loader
if (!crea_spazio_condiviso(root_tab, mbi))
    goto error;

flog(LOG_INFO, "Create le traduzioni per le parti condivise");
flog(LOG_INFO, "Frame liberi: %d (M2)", num_frame_liberi);
```

Nel codice viene chiamata la funzione *crea_spazio_condiviso*. Non approfondiremo fin troppo il contenuto (*troppo complesso*, cit), ricordiamoci che:

- il bootloader ci fornisce gli indirizzi di memoria fisica dove ha copiato i file relativi ai moduli (attraverso la struttura *multiboot_info*, è qua che utilizzeremo il contenuto del registro RDI);
- all'interno il codice relativo a un certo modulo è gestito con una funzione a parte, entrambe fanno ricorso alla funzione di utilità *carica_modulo* (funzione molto complicata, che **interpreta il file ELF**);
- i segmenti posti nel file ELF vengono mappati utilizzando la solita funzione *map*;
- lo spazio destinato allo heap del modulo viene mappato sempre usando la *map*. Ricordiamo che ci limitiamo a preparare uno spazio secondo la dimensione massima indicata con la costante *heap_size*: lo spazio effettivo cambia dinamicamente durante l'esecuzione. La funzione utilizzata per indicare la funzione è la *alloca_frame*: si scelgono i frame (non ci importa che siano consecutivi), ottenendo ogni volta il corrispondente indirizzo fisico.

Dopo l'esecuzione della funzione avremo tutte le entrate dello spazio condiviso in *root_tab*.

11.6 Registro CR3

Aggiorniamo il contenuto del registro CR3 ponendo l'indirizzo della *root_tab*

```
loadCR3(root_tab);
flog(LOG_INFO, "CR3 caricato")
```

Da questo punto in poi utilizzeremo le nostre traduzioni, non più quelle create dal bootloader.

11.7 Inizializzazione dello *heap*

Inizializziamo lo heap chiamando l'apposita funzione

```
// Assegna allo heap di sistema HEAP_SIZE byte nel secondo MiB
heap_init((void*)HEAP_START, HEAP_SIZE);
flog(LOG_INFO, "Heap di sistema: %x B @%x", HEAP_SIZE, HEAP_START);
```

Si consideri la presenza delle seguenti funzioni, nel modulo sistema, per gestire lo *heap*

```
void* operator new(size_t size) {
    return alloca(size);
}

void* operator new(size_t size, align_val_t align) {
    return alloc_aligned(size, align);
}

void operator delete(void *p) {
    dealloca(p);
}
```

11.8 Creazione processo sistema

Creiamo un nuovo processo che lanceremo dopo aver concluso con la funzione *main*.

```
// creazione del processo main_sistema
mid = crea_main_sistema(mbi);
if (mid == 0xFFFFFFFF)
    goto error;
flog(LOG_INFO, "Creato il processo main_sistema (id = %d)", mid);
```

Il codice della funzione è il seguente:

```
natl crea_main_sistema(natq mbi) {
    des_proc* m = crea_processo(main_sistema, mbi, MAX_PRIORITY, LIV_SISTEMA, false);
    if (m == 0) {
        flog(LOG_ERR, "Impossibile creare il processo main_sistema");
        return 0xFFFFFFFF;
    }
    inserimento_lista(pronti, m);
    processi++;
    return m->id;
}
```

- Per prima cosa si usa la *crea_processo* per creare effettivamente il processo.
  - La funzione associata al processo è la *main_sistema*. Il suo unico parametro di ingresso è mbi (registro RDI, indirizzo al *multiboot_info*).
  - La priorità è massima: *MAX_PRIORITY*.

  - Il processo è eseguito con livello di privilegio sistema.

- Dopo aver creato il processo lo pongo nella lista *pronti*: avendo posto priorità massima finirà sicuramente in testa.
- Incremento il numero di processi creati.
- Concludo restituendo l'ID del processo creato.

11.8.1 Funzione *crea_processo*, inizializzazione della pila sistema ed eventualmente della pila utente, inizializzazione del *punt_nucleo*

Ogni volta che viene creato un processo chiamiamo la *crea_processo* (anche nella primitiva *activate_p*): indichiamo funzione, parametri, priorità, livello di privilegio, se le interruzioni sono attive o disattivate.

- **Creazione del *des_proc*.**

  Alloco e resetto un *des_proc*, e ne riempio i campi.

```
// allocazione (e azzeramento preventivo) di un des_proc
p = new des_proc;
if (!p)
    goto errore1;
memset(p, 0, sizeof(des_proc));

// rimpiamo i campi di cui conosciamo già  i valori
p->precedenza = prio;
p->puntatore = nullptr;
// il registro RDI deve contenere il parametro da passare alla funzione f
p->contesto[I_RDI] = a;
```

- **Ricerca di un identificatore per il processo**.

  Viene allocato un identificatore numerico, se non esiste più finiamo subito.

```
// selezione di un identificatore
p->id = alloca_proc_id(p);
if (p->id == 0xFFFFFFFF)
    goto errore2;
```

  dove *alloca_proc_id* è una funzione di utilità

```
// alloca un id non utilizzato.
natl alloca_proc_id(des_proc *p) {
    static natl next = 0;

    // La funzione inizia la ricerca partendo  dall'id successivo
    // all'ultimo restituito (salvato nella variable statica 'next'),
    // saltando quelli che risultano in uso.
    natl scan = next, found = 0xFFFFFFFF;
    do {
        if (proc_table[scan] == nullptr) {
            found = scan;
```

```
            proc_table[found] = p;
        }
        scan = (scan + 1) % MAX_PROC;
    } while (found == 0xFFFFFFFF && scan != next);
    next = scan;
    return found;
}
```

Scorro l'array finchè non individuo un'entrata non utilizzata. Se individuo un'entrata disponibile ne restituisco l'indice, altrimenti restituisco 0xFFFFFFFF.

- **Creazione della tabella per il *trie*.**

  Creo una tabella che fungerà da radice del trie.

```
// creazione della tabella radice del processo
p->cr3 = alloca_tab();
if (p->cr3 == 0)
    goto errore3;
init_root_tab(p->cr3);
```

  Con la funzione *init_root_tab* inizializzo il *trie*.

```
// inizializza la tabella radice di un nuovo processo
void init_root_tab(paddr dest){
     paddr pdir = readCR3();

    copy_des(pdir, dest, I_SIS_C, N_SIS_C);
    copy_des(pdir, dest, I_MIO_C, N_MIO_C);
    copy_des(pdir, dest, I_UTN_C, N_UTN_C);
}
```

  - La funzione ha in ingresso un paddr *dest*, che consiste nel CR3 del contesto.
  - Per prima cosa si pone in *pdir* il contenuto del registro CR3, recuperando così l'indirizzo dell'attuale *trie*.
  - Copio dall'attuale *trie* le traduzioni relative alle parti condivise: modulo sistema, modulo utente e modulo I/O. Il risultato sono alberi di processi diversi che presentano sottoalberi comuni: la cosa permette di risparmiare memoria.

- **Creazione della pila sistema e funzione *crea_pila*.**

  L'unica cosa che ci manca da fare è creare la pila sistema.

```
// creazione della pila sistema
if (!crea_pila(p->cr3, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
    goto errore4;
```

  Lo facciamo con la funzione *crea_pila*.

```
// crea una pila processo (utente o sistema, in base a 'liv').  Creiamo una
// traduzione dagli indirizzi riservati alla pila verso frame allocati sul
// momento.
```

```
bool crea_pila(paddr root_tab, vaddr bottom, natq size, natl liv) {
    vaddr v = map(root_tab,
              bottom - size,
              bottom,
              BIT_RW | (liv == LIV_UTENTE ? BIT_US : 0),
              [](vaddr) { return alloca_frame(); }
              );
    if (v != bottom) {
        unmap(root_tab, bottom - size, v, [](vaddr p, int) { rilascia_frame(p); });
        return false;
    }
    return true;
}
```

  - Creiamo una pila nel *trie* relativo al processo, utilizzando la funzione *map*. Come funzione per indicare gli indirizzi utilizziamo, anche qua, la *alloca_frame*.
  - Se la *map* è fallita dobbiamo fare l'*unamp* di quanto fatto dalla *map* (possiamo farlo, abbiamo posto nella variabile v l'ultimo indirizzo mappato). Le tabelle vengono distrutte, ovviamente, ciò che è vuoto. Per rilasciare i vari frame utilizza la *rilascia_frame*.

- **Ottenimento dell'indirizzo fisico della pila**.

```
pila_sistema = trasforma(p->cr3, fin_sis_p - DIM_PAGINA) + DIM_PAGINA;
```

Gli indirizzi virtuali adottati per la pila sistema sono gli stessi in ogni processo, la cosa che cambia è il significato (la traduzione) attribuita all'indirizzo virtuale stesso. Il fatto è che se noi utilizziamo l'indirizzo senza porci troppi pensieri andiamo a manipolare l'albero del processo corrente, e non l'albero del processo che stiamo creando. Risolviamo la cosa utilizzando la funzione *trasforma*.

```
// restituisce l'indirizzo fisico che corrisponde a ind_virt nell'albero
// di traduzione con radice root_tab.
paddr trasforma(paddr root_tab, vaddr v) {
    // usiamo un tab_iter su una sola pagina fermandoci sul
    // descrittore foglia lungo il percorso di traduzione di 'v'
    tab_iter it(root_tab, v);
    while (it.down())
    ;

    // si noti che il percorso potrebbe essere incompleto.
    // Ce ne accorgiamo perchè il descrittore foglia ha il bit P a
    // zero. In quel caso restituiamo 0, che per noi non è un
    // indirizzo fisico valido.
    tab_entry e = it.get_e();
    if (!(e & BIT_P))
        return 0;

    // se il percorso è completo calcoliamo la traduzione corrispondente.
    // Si noti che non siamo necessariamente arrivati al livello 1, se
    // c'era un bit PS settato lungo il percorso.
    int l = it.get_l();
```

```
    natq mask = dim_region(l - 1) - 1;
    return (e & ~mask) | (v & mask);
}
```

- Si pone in ingresso l'indirizzo della radice del *trie*, *root_tab*, e l'indirizzo virtuale v (l'indirizzo che vogliamo tradurre).
- Si scorre l'albero usando l'iteratore *tab_iter* (stesso percorso della MMU, ma fatto in software). Arrivati alla foglia si verifica il valore del bit P:
  - se è uguale a 0 la traduzione non è valida e si restituisce 0;
  - altrimenti la traduzione è valida e restituiamo l'indirizzo fisico.
- Attenzione alle somme e differenze presenti

  ```
  pila_sistema = trasforma(p->cr3, fin_sis_p - DIM_PAGINA) + DIM_PAGINA
  ```

  Attenzione alla somma e alla differenza: voglio il *bottom fisico* della pila, se io non faccio la differenza ottengo l'indirizzo del frame successivo (ricordarsi l'indirizzo puntato sulla pila quando non c'è niente).

- **Contenuto della pila sistema**.

  La pila sistema viene creata in un qualunque processo, come già spiegato dobbiamo riempirla. Il contenuto differisce lievemente in base al livello di privilegio

```
// ----- PROCESSO DI LIVELLO UTENTE -----
pl[-5] = reinterpret_cast<natq>(f); // RIP (codice utente)
pl[-4] = SEL_CODICE_UTENTE;     // CS (codice utente)
pl[-3] = IF ? BIT_IF : 0;     // RFLAGS
pl[-2] = fin_utn_p - sizeof(natq);  // RSP
pl[-1] = SEL_DATI_UTENTE;     // SS (pila utente)
// eseguendo una IRET da questa situazione, il processo
// passera' ad eseguire la prima istruzione della funzione f,
// usando come pila la pila utente (al suo indirizzo virtuale)

// ----- PROCESSO DI LIVELLO SISTEMA -----
pl[-6] = reinterpret_cast<natq>(f);   // RIP (codice sistema)
pl[-5] = SEL_CODICE_SISTEMA;            // CS (codice sistema)
pl[-4] = IF ? BIT_IF : 0;           // RFLAGS
pl[-3] = fin_sis_p - sizeof(natq);      // RSP
pl[-2] = 0;         // SS
pl[-1] = 0;         // ind. rit. (non significativo)
// i processi esterni lavorano esclusivamente a livello
// sistema. Per questo motivo, prepariamo una sola pila (la
// pila sistema)
```

- **Eventuale creazione della pila utente**.

  Se il processo è di livello utente allora dobbiamo creare la relativa pila utente. La cosa non è invece necessaria se il processo è di livello sistema.

```
// creazione della pila utente
if (!crea_pila(p->cr3, fin_utn_p, DIM_USR_STACK, LIV_UTENTE)) {
    flog(LOG_WARN, "creazione pila utente fallita");
    goto errore5;
}
```

Anche in questo caso usiamo la funzione *crea_pila*.

- **Inizializzazione di RSP**.

  Un processo di livello utente si trova, inizialmente, a livello sistema (come se avessi eseguito un'istruzione INT): aggiorniamo RSP nel *contesto*, in modo tale che punti all'ultimo elemento della pila sistema.

  ```
  // inizialmente, il processo si trova a livello sistema, come
  // se avesse eseguito una istruzione INT, con la pila sistema
  // che contiene le 5 parole lunghe preparate precedentemente
  p->contesto[I_RSP] = fin_sis_p - 5 * sizeof(natq);
  ```

  L'inizializzazione di RSP si fa anche con un processo di livello sistema, ma ponendo in pila anche l'indirizzo di ritorno (che però non è significativo)

  ```
  // inizializziamo il descrittore di processo
  p->contesto[I_RSP] = fin_sis_p - 6 * sizeof(natq);
  ```

- **Inizializzazione del *punt_nucleo***.

  Se stiamo gestendo un processo di livello utente dobbiamo inizializzare *punt_nucleo* (in un processo di livello sistema non è necessario, avremo solo la pila sistema)

  ```
  // dal momento che usiamo traduzioni diverse per le parti sistema/private
  // di tutti i processi, possiamo inizializzare p->punt_nucleo con un
  // indirizzo (virtuale) uguale per tutti i processi
  p->punt_nucleo = fin_sis_p;
  ```

  L'indirizzo posto verrà copiato in TSS, in modo che il meccanismo delle interruzioni utilizzi questo indirizzo per il cambio pila.

  Fermi tutti: se abbiamo un unico indirizzo virtuale potremo evitare ogni volta la modifica dell'indirizzo nel TSS, basta solo cambiarne l'interpretazione nella traduzione usata dalla MMU.

11.9 Creazione processo dummy

Prima di passare la parola al processo creato con la *crea_main_sistema* dobbiamo creare il processo dummy con la funzione *crea_dummy* (ricordare le motivazioni per cui lo facciamo).

```
// creazione del processo dummy
dummy_id = crea_dummy();
if (dummy_id == 0xFFFFFFFF)
    goto error;
flog(LOG_INFO, "Creato il processo dummy (id = %d)", dummy_id);
```

11.10 Schedulatore

Chiamiamo lo schedulatore

```
schedulatore();
```

La funzione pone nel puntatore *esecuzione* il processo creato in *crea_main_sistema* (a cui abbiamo dato massima priorità, troveremo in testa per forza quello).

11.11 Funzione *salta_a_main*

Il codice della funzione si conclude con la chiamata della funzione *salta_a_main*.

```
// esegue CALL carica_stato; IRETQ (vedi "sistema.s"). Il resto
// dell'inizializzazione prosegue più comodamente nel processo
// main_sistema(), che può essere interrotto e può sospendersi.
salta_a_main();
```

La sua implementazione è in *sistema.s*

```
    .global salta_a_main
salta_a_main:
    call carica_stato
    iretq
```

Con la *carica_stato* e la *iretq* avviene il passaggio al processo creato con la *crea_main_sistema*, con cui viene eseguita (a livello sistema) la funzione *main_sistema*.

11.12 Funzione *main_sistema*

Concludiamo con la funzione *main_sistema*, ultimo step prima del primo processo utente.

```
void (*io_entry)(natq);
void (*user_entry)(natq);

void main_sistema(natq mbi) {
    natl sync_io;
    natl id;

    // occupiamo a_p[2] (in modo che non possa essere sovrascritta
    // per errore tramite activate_pe()) e smascheriamo il piedino
    // 2 dell'APIC
    a_p[2] = ESTERN_BUSY;
    apic_set_MIRQ(2, false);
    // attiviamo il timer, in modo che i processi di inizializzazione
    // possano usare anche delay(), se ne hanno bisogno.
    attiva_timer(DELAY);
    flog(LOG_INFO, "Timer attivato (DELAY=%d)", DELAY);

    // inizializzazione del modulo di io
    // Creiamo un processo che esegua la procedura cmain del modulo I/O.
    // Usiamo un semaforo di sincronizzazione per sapere quando
    // l'inizializzazione è terminata.
    sync_io = sem_ini(0);
    if (sync_io == 0xFFFFFFFF) {
        flog(LOG_ERR, "Impossibile allocare il semaforo di sincr per l'IO");
        goto error;
    }
    id = activate_p(io_entry, sync_io, MAX_PRIORITY, LIV_SISTEMA);
    if (id == 0xFFFFFFFF) {
        flog(LOG_ERR, "impossibile creare il processo main I/O");
        goto error;
```

```
    }
    flog(LOG_INFO, "Creato il processo main I/O (id = %d)", id);
    flog(LOG_INFO, "attendo inizializzazione modulo I/O...");
    sem_wait(sync_io);
    flog(LOG_INFO, "... inizializzazione modulo I/O terminata");

    // creazione del processo start_utente
    id = activate_p(user_entry, 0, MAX_PRIORITY, LIV_UTENTE);
    if (id == 0xFFFFFFFF) {
        flog(LOG_ERR, "impossibile creare il processo main utente");
        goto error;
    }
    flog(LOG_INFO, "Creato il processo start_utente (id = %d)", id);

    // terminazione
    flog(LOG_INFO, "passo il controllo al processo utente...");
    terminate_p();

error: panic("Errore di inizializzazione");
}
```

- Attenzione all'array *a_p* (andiamo a fare l'assegnamento per evitare confusione con la funzione per l'attivazione dei processi esterni, che vedremo più avanti). Con la *apic_set_MIRQ* attiviamo l'ascolto delle interruzioni sul piedino. Attiviamo il timer con la funzione *attiva_timer*: ricordarsi che l'implementazione del timer stesso è posta nel modulo sistema e non nel modulo I/O (per permettere l'implementazione, ad esempio, della funzione *delay*).
- Creo il semaforo *sync_io*, creo un processo con funzione quella puntata da *io_entry* e passo come parametro l'identificativo del semaforo. Questa cosa permetterà di eseguire le inizializzazioni del modulo I/O, affrontate nel capitolo del codice del Modulo I/O.

  ```
  sync_io = sem_ini(0);
  [...]
  id = activate_p(io_entry, sync_io, MAX_PRIORITY, LIV_SISTEMA);
  [...]
  flog(LOG_INFO, "Creato il processo main I/O (id = %d)", id);
  flog(LOG_INFO, "attendo inizializzazione modulo I/O...");
  sem_wait(sync_io);
  flog(LOG_INFO, "... inizializzazione modulo I/O terminata");
  ```

  In particolare si osservi che non ritorneremo su questo codice finchè non verrà rilasciato il gettone *sync_io*. Questa cosa avviene nella funzione *main* del Modulo I/O, dopo aver fatto le inizializzazioni necessarie (si veda il felativo capitolo per approfondire).
- Creo il primo processo utente e termino l'attuale processo.

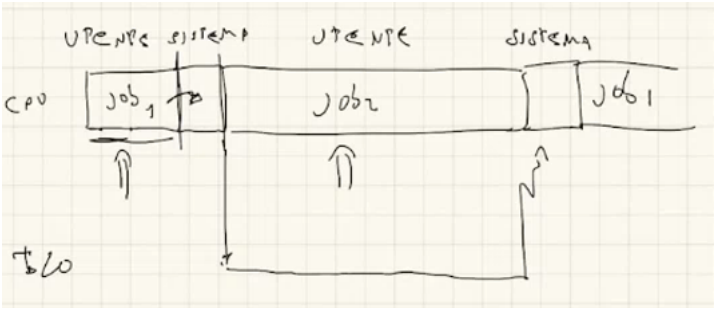
  ```
  // creazione del processo start_utente
  id = activate_p(user_entry, 0, MAX_PRIORITY, LIV_UTENTE);
  [...]
  flog(LOG_INFO, "Creato il processo start_utente (id = %d)", id);
  // terminazione
  flog(LOG_INFO, "passo il controllo al processo utente...");
  terminate_p();
  ```

Capitolo 12

Periferiche

12.1 I/O nel nucleo

Esempio Ripartiamo dall'esempio visto quando abbiamo introdotto le interruzioni.



- Siamo in un sistema batch, caratterizzato da *job*. Avevamo preso come riferimento la *Calcolatrice Elettronica Pisana*.
- **Abbiamo un job che a un certo punto deve fare un'operazione di I/O**. Invece di tenere occupata la CPU fino al termine dell'operazione chiamiamo una primitiva: questa esegue una routine di sistema che avvia l'operazione di I/O e cede la CPU a un altro *job*. L'elemento fondamentale è che non serve la CPU (salvo casi rari) per portare a terminare l'operazione di I/O.
- Quando l'operazione di I/O finisce viene lanciata un'interruzione. Il *job* in esecuzione viene interrotto e una routine di sistema ci riporta al primo *job*, che adesso ha gli elementi per poter proseguire.

Dal punto di vista del *job* l'operazione di I/O è istantanea: quello che abbiamo fatto è sospendere il *job* e recuperarlo solo dopo aver completato l'operazione di I/O. Quello che è successo tra l'avvio e la conclusione dell'operazione è come se non fosse mai avvenuto (ripeto, dal punto di vista del job).

Nel nostro caso

Job = Processo

12.1.1 Limitazioni sull'utente

Come sempre non possiamo fidarci dell'utente, quindi

1. impediamo all'utente l'accesso diretto alle periferiche, e
2. forniamo all'utente delle primitive per svolgere le operazioni di I/O sotto il controllo del sistema

Il punto (1) lo otteniamo intervenendo sull'IOPL, nel registro dei flag. Questa cosa limita l'accesso solo alle periferiche con indirizzi nello spazio di I/O. Per quanto riguarda le periferiche con indirizzi nello spazio di memoria interveniamo con la MMU: facciamo in modo che lo spazio di indirizzamento virtuale non permetta di raggiungere, agli utenti, i frame relativi alle periferiche (poniamo il bit U/S a zero).

12.1.2 Questioni/problemi delegati alle primitive

- **Problema di *mutua esclusione*.**

  L'utente non si preoccupa che la periferica con cui vuole dialogare non sia già occupata da qualcun altro. Supponiamo che il *job* lanciato dalla routine dopo il primo *job* voglia anch'esso rivolgersi alla periferica: siamo obbligati ad attendere che la tastiera abbia finito.

- **Problema di *sincronizzazione*.**

  Il job iniziale deve essere "risvegliato" solo dopo aver concluso l'operazione di I/O.

12.1.3 Schema di una primitiva di I/O (IN read, OUT write)

La schematizzazione di un'operazione di I/O è un trasferimento di un certo numero di byte da e verso una periferica

Primitiva per periferica IN Possiamo immaginarci una primitiva del seguente tipo

```
read_n(int id, char* buf, natq quanti)
```

l'utente che invoca la primitiva dovrà indicare:

- un qualcosa che identifichi la periferica da cui vuole leggere,
- dove i byte devono essere trasferiti,
- quanti byte devono essere trasferiti.

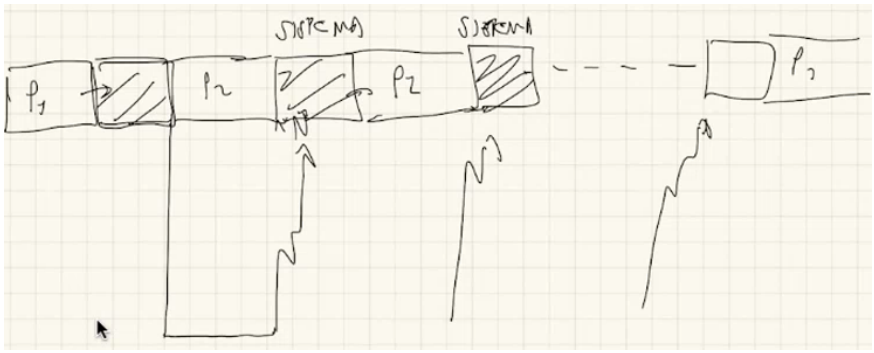
La primitiva avvia un'operazione di ingresso dalla periferica ID: i *quanti* byte letti dovranno trovarsi in memoria a partire dall'indirizzo *buf* (nel buffer).

Primitiva per periferica OUT Lo schema non è molto diverso da quello già visto.

```
write_n(int id, const char* buf, natq quanti)
```

- Passiamo l'indirizzo del primo byte che si vuole scrivere sulla periferica (solo lettura, quindi può essere *const*) e il numero di byte da trasferire.
- La primitiva avvia un'operazione di uscita: poniamo il contenuto del buffer (cioè i *quanti* byte dall'indirizzo *buf*) nella periferica *id*.

12.1.4 Necessità di più interruzioni, primitive e driver



Non è detto che un'operazione di I/O si risolva con una sola interruzione. Supponiamo di voler leggere un certo numero di byte dalla periferica: questa è in grado di inviarci un solo byte alla volta. Questo significa che dovremo attendere per ogni byte, ricevendo più interruzioni (una ogni volta che un byte viene reso disponibile).

- Nel processo $P1$ lanciamo una routine di sistema con l'istruzione *INT*. Questa avvia l'operazione di I/O e lascia la palla al processo $P2$.
- Il processo $P2$ continua finchè non riceve un'interruzione esterna: a quel punto si ritorna nella routine di sistema, si legge il byte e lo si pone da qualche parte.
- Dopo la lettura non ritorno subito nel processo $P1$, ma continuo ad eseguire il processo $P2$ (o un altro processo nel caso $P2$ sia concluso).

Quello che andremo a fare è un'operazione eseguita in parte da una **primitiva** e in parte da un **driver**.

- La *primitiva* avvia l'operazione di I/O e blocca il processo.
- Il *driver* si occupa di trasferire effettivamente i byte e sbloccare il processo quando l'operazione si è conclusa.

Nell'implementazione dobbiamo risolvere la mutua esclusione e la sincronizzazione: lo facciamo **utilizzando i semafori**.

- Per la mutua esclusione un semaforo inizialmente con un gettone, associato a certa periferica (*mutex*).
- Per la sincronizzazione un semaforo inizialmente senza gettoni (*sync*).

Assunzioni

- Sono presenti almeno due registri nelle periferiche: il RBR e il CTR (che ha un flag che permette l'attivazione/disattivazione delle interruzioni). Nel caso di una prova d'esame il professore indica i registri presenti con i relativi indirizzi.

- Si suppone la presenza di un handshake dove la periferica non invia una nuova richiesta di interruzione finchè non ha ricevuto risposta sulla precedente (la periferica riceve la risposta con la lettura del registro RBR).

12.1.5 Implementazione delle primitive e differenze con le classiche primitive

La primitiva dovrà fare le seguenti cose:

```
sem_wait(mutex)
AVVIO OPERAZIONE
sem_wait(sync)
sem_signal(mutex)
```

- Prende il gettone *mutex* relativo alla periferica. Altri processi che proveranno ad interagire con la periferica non troveranno il gettone, dunque si porranno in attesa.
- Svolge le istruzioni necessarie per avviare l'operazione di I/O.
- Si pone in attesa tentando di prendere un gettone *sync* non presente.
- Rilascia il gettone preso all'inizio in *mutex*: adesso la periferica è nuovamente disponibile per svolgere operazioni.

Attenzione Abbiamo detto *si pone in attesa...*

L'uso delle primitive semaforiche comporta la perdita dell'atomicità.

Le primitive semaforiche spezzano l'atomicità: si pensi a cosa succede quando non si hanno gettoni disponibili.

12.1.5.1 Prototipo di descrittore di periferica e array di descrittori

Per una qualunque periferica abbiamo bisogno di raccogliere informazioni e gestirle in una struttura. Il prototipo di struttura è il descrittore di periferica *des_io*

```
des_io {
    ioaddr iRBR, iCTR;
    char* buf;
    natq quanti;
    natl mutex;
    natl syncr;
};
```

In questo prototipo abbiamo:

- l'indirizzo dei due registri (RBR e CTR) che abbiamo supposto sempre presenti;
- l'indirizzo del buffer *buf*;
- il numero di byte che ci interessa;
- gli identificatori dei semafori *mutex* e *sync*.

La struttura dati permetterà alla primitiva di comunicare col driver.

Array di descrittori di periferica Abbiamo due situazioni possibili:

1. gestione di un'unica periferica di un certo tipo;
2. gestione di più periferiche.

Nel secondo caso è necessario introdurre un array di descrittori di periferica

```
des_io array_des_io[MAX_DES_IO];
```

12.1.5.2 Assembler

La parte Assembler della primitiva si implementa come al solito, ma con una differenza fondamentale:

```
.global read_n
read_n:
    int $IO_TIPO_RN
    ret

.global a_read_n
a_read_n:
    call c_read_n
    iretq
```

non abbiamo le chiamate di *salva_stato* e *carica_stato*. In sostanza:

- cambia il livello di privilegio (si passa a livello sistema con l'attraversamento del gate);
- non cambia il processo (rimane quello che ha invocato l'interruzione).

Se noi eseguiamo la *sem_wait* viene memorizzato lo stato, e si sospende qualora non ci siano gettoni: quando riprenderemo l'esecuzione lo faremo da un punto intermedio della primitiva (cosa impensabile fino ad ora).

12.1.5.3 C++

Per quanto riguarda la funzione C++ scriveremo

```
void c_read_n(natl id, natb* buf, natl quanti) {
    # [..] vedere sez. sul cavallo di troia

    des_io* d = &array_des_io[id]; // <-- recupero il descrittore

    sem_wait(d->mutex); // <-- mutua esclusione
    d->buf = buf;
    d->quanti = quanti;
    outputb(1, d->iCTL);
    sem_wait(d->sync); // <-- sincronizzazione
    sem_signal(d->mutex); // <-- mutua esclusione
}
```

La funzione contiene tutto quello che ci serve: inizializza il contenuto della struttura dati e lavora con le primitive semaforiche.

- Il semaforo *mutex* garantisce la mutua esclusione: se un altro processo prova ad accedere alla periferica questo verrà sospeso quando chiama la *sem_wait* (ecco perché la primitiva non può essere atomica).
- Il semaforo *sync* garantisce la sincronizzazione: con zero gettoni all'inizio il processo rimane bloccato all'interno dell'area di mutua esclusione. La relativa *sem_signal* per *sync* sarà eseguita nel driver quando le operazioni di lettura/scrittura dei byte risulteranno concluse.

Con la *outputb*, inoltre, abilitiamo le interruzioni scrivendo nel registro CTL.

12.1.6 Implementazione del driver

Il driver viene lanciato a seguito di richiesta di interruzione esterna (in contrasto con la primitiva lanciata con una INT). Ricordare come l'Ave Maria:

Questa interruzione non è un processo.

12.1.6.1 Assembler

```
a_driver_i:
    call salva_stato
    mov $i, %rdi
    call c_driver
    call apic_send_EOI
    call carica_stato

    iretq
```

Inviamo il segnale di EOI per avvertire che la CPU ha finito di gestire l'interruzione.

12.1.6.2 C++

Il driver non può non essere atomico: questo perchè a un certo punto chiamerà la funzione *sem_signal*, che comporta il ritorno al processo iniziale (quello che ha invocato la primitiva) e quindi **manipolazioni delle code dei processi**. Facciamo questa cosa in C++:

```
void c_driver_i(int i) {
    des_io* d = array_des_io[i];
    d->quanti--;
    if(d->quanti == 0) {
        outputb(0, d->iCTL);
        c_sem_signal(d->sync);
    }
    char c = inputb(d->iRBR);
    *d->buf = c;
    d->buf++;
}
```

- Punto al descrittore di I/O della periferica i

  ```
  des_io* d = array_des_io[i];
  ```

- Decremento *quanti*, visto che abbiamo considerato un byte.

  ```
  d->quanti--;
  ```

- Quando *quanti* arriva a 0 dobbiamo svegliare il processo che ha lanciato la primitiva. Oltre a questo disattivo le interruzioni per quella periferica.

  ```
  if(d->quanti == 0) {
      outputb(0, d->iCTL);
      c_sem_signal(d->sync);
  }
  ```

- Imposto il buffer

  ```
  *d->buf = c
  ```

- Incremento buf, in modo che alla prossima istanza il prossimo carattere venga scritto nella posizione successiva nel buffer

  ```
  d->buf++;
  ```

Osservazioni

- **Le interruzioni vengono disattivate prima della lettura dell'ultimo byte**.

  ```
  if(d->quanti == 0) {
      outputb(0, d->iCTL);
      c_sem_signal(d->sync);
  }
  char c = inputb(d->iRBR);
  ```

  La cosa è conseguenza del fatto che la lettura del byte consiste nella risposta alla periferiche (è parte dell'handshake). Se non disattivo le interruzioni l'interfaccia potrebbe generare una nuova interruzione.

- **Chiamata diretta della *c_sem_signal***.

  ```
  c_sem_signal(d->sync);
  ```

  La cosa è conseguenza del fatto che *sem_signal* salva e ripristina lo stato, ma il driver non è un processo e non ha un suo descrittore di processo. **Se io eseguo quelle istruzioni salvo lo stato nel processo attivo**. In aggiunta la funzione manipola le code dei processi, pertanto deve essere eseguita con le interruzioni disattivate: **ATOMICITA'**!

- **L'istruzione con *buf***.

  Si consideri questa istruzione

```
*d->buf = c;
```

l'indirizzo indicato è un *indirizzo virtuale*. Il buffer relativo alla primitiva era stato allocato nel processo che ha chiamato la primitiva, mentre qua siamo in un altro processo. La soluzione è porre il buffer nella sezione *data*, che è globale (quindi dobbiamo rendere il buffer condiviso): se non faccio questa cosa gli indirizzi virtuali saranno interpretati diversamente, andando a scrivere nei posti sbagliati.

12.2 Il problema del cavallo di Troia, primitiva *access*

Non possiamo fidarci a priori del buffer passato dall'utente: non è solo passare una cosa locale e non globale, ma cose molto più maligne. Potrei passare un indirizzo di sistema: l'utente non può manipolarlo, ma il sistema sì. Questa cosa è il problema del cavallo di Troia:

- l'indirizzo attraversa il gate senza problemi (il cavallo che entra nella città di Troia);
- l'uso dell'indirizzo da parte del sistema (che crede sia un buffer) provoca problemi (chi è dentro il cavallo esce e mette a fuoco la città).

Dobbiamo controllare i parametri di ingresso *buf* e *quanti*. Lo facciamo con la primitiva *access*.

```
// primitiva utilizzata dal modulo I/O per controllare che i buffer passati dal
// livello utente siano accessibili dal livello utente (problema del Cavallo di
// Troia) e non possano causare page fault nel modulo I/O (bit P tutti a 1 e
// scrittura permessa quando necessario)
extern "C" bool c_access(vaddr begin, natq dim, bool writeable) {
    if (!tab_iter::valid_interval(begin, dim))
        return false;

    // usiamo un tab_iter per percorrere tutto il sottoalbero relativo
    // alla traduzione degli indirizzi nell'intervallo [begin, begin+dim).
    for (tab_iter it(esecuzione->cr3, begin, dim); it; it.next()) {
        tab_entry e = it.get_e();

        // interrompiamo il ciclo non appena troviamo qualcosa che non va
        if (!(e & BIT_P) || !(e & BIT_US) || (writeable && !(e & BIT_RW)))
            return false;
    }
    return true;
}
```

- Prendo un indirizzo di partenza *begin* e una dimensione *dim*, indicando con un booleano se la parte deve essere scrivibile o meno (se non posso scrivere si genera un eccezione in pieno driver, e noi non vogliamo).
- Chiamo la funzione *valid_interval* per controllare che l'intervallo sia tutto dalla stessa parte rispetto al buco.
- Purtroppo dobbiamo percorrere tutta la traduzione con l'iteratore *tab_iter*, nell'intervallo che ci interessa. Ogni volta prendiamo l'entrata e restituiamo *false* se troviamo qualcosa che non va. Intendiamo:

- qualcosa di non mappato;

  `!(e & BIT_P)`

- qualcosa di non accessibile a livello utente;

  `!(e & BIT_US)`

- qualcosa di non scrivibile (se *writeable* è posto a true).

  `(writeable && !(e & BIT_RW))`

Concludiamo ponendo in cima al corpo di *c_read_n* le seguenti righe

```
if(!c_access(begin, quanti, true)) {
    flog(LOG_WARN, "buf non valido");
    abort_p();
}
```

Recap

- Abbiamo visto che un processo che vuole svolgere un'operazione di I/O invoca una primitiva collocata (per ora) nel modulo sistema, con la particolarità dell'atomicità della primitiva. **Le interruzioni sono disabilitate, le eccezioni sono vietate (se vogliamo), ma le INT ammesse**.
- Le deroghe rispetto alla regola sono necessarie per utilizzare le primitive semaforiche e implementare in modo più semplice mutua esclusione (gestire l'uso di una periferica da parte di più processi) e sincronizzazione (sospendere il processo in attesa che l'operazione di I/O sia terminata). L'operazione di I/O è gestita da un *driver*.
- Si ricordi che, teoricamente, possiamo definire la primitiva non atomica come eseguita dal processo che l'ha lanciata (non abbiamo chiamato la salva stato, dunque l'unica cosa che abbiamo fatto è innalzare il livello di privilegio).
- Il *driver* è un po' noioso: deve essere atomico (non ha un posto per salvare il proprio stato, visto che non è un processo, **e deve agire sulle code dei processi**), sta prendendo in prestito le risorse del processo che ha interrotto.

12.3 Introduzione del Modulo I/O: perché e vantaggi

Problema Il fatto che un driver sia atomico è un requisito troppo stringente in sistemi complessi, per esempio quelli che gestiscono la rete e/o i file system.

Soluzione

- Il fatto è che **nel modulo sistema le interruzioni NON possono essere abilitate a causa della manipolazione delle code dei processi**. Un'idea base potrebbe essere utilizzare CLI ed STI per proteggere le strutture dati da manipolazioni erronee, ma la cosa complica il codice.
- **Soluzione definitiva**.

  La soluzione è introdurre un nuovo modulo: il modulo I/O. Questo conterrà tutto ciò che è legato alle primitive di I/O, incluse le primitive stesse e le loro strutture dati. I problemi precedenti si risolvono perchè lanceremo, dal modulo I/O, una primitiva per alterare le strutture dati necessarie (solo a quel punto mi sposto nel modulo sistema, e ho le interruzioni disabilitate SOLO quando devo manipolare le strutture dati).

Cosa ci guadagnamo?

- Memorizzare le primitive in un modulo separato permette di ricevere un aiuto da compilatore e collegatore: in assenza di collegamento tra modulo I/O e modulo sistema non è possibile accedere alle strutture dati. **Il fatto che il nucleo ci fornisca solo due livelli di privilegio complica le cose**: il nostro interesse è eseguire un qualcosa che abbia maggiori privilegi rispetto all'utente e meno privilegi rispetto al sistema. Nel nucleo abbiamo scelto di implementare il modulo I/O con livello di *privilegio sistema*.
- **Perchè non a livello utente?** Potremo modificare l'IOPL per permettere al modulo utente di eseguire le istruzioni outputX (non posso chiamare una primitiva ogni volta che devo lavorare con quelle istruzioni, è troppo dispendioso). Il fatto è che questa modifica rende utilizzabili anche CLD ed STD, e la cosa è inaccettabile.
- Il modulo I/O gira a interruzioni abilitate (contrariamente a prima), e utilizza delle primitive di sistema per accedere alle strutture dati del modulo sistema. A questo punto è possibile rendere la primitiva non atomica (permettendo tutte le cose che normalmente non sono permesse nella primitiva).

Le scelte adottate ci permettono di proteggere le strutture dati del sistema da errori involontari.

12.4 Processi esterni

Problema Il driver non è un processo: non ha risorse sue, non può essere salvato o caricato da qualche parte. La gestione delle interruzioni in questo modo è poco efficiente.

Soluzione La soluzione è renderlo un processo: diamo la possibilità al modulo I/O di creare dei processi speciali, detti **processi esterni**[1] (privilegiati, non come quelli utente). Li creiamo con una primitiva simile all'*activate_p*.

[1] Esterni rispetto al modulo sistema, pur facendone parte.

12.4.1 Creazione del processo esterno

12.4.1.1 Primitiva *activate_pe*

```
natq activate_pe(void f(natq), natq a, natl prio, natl liv int irq);
```

La primitiva *activate_pe*, implementata nel modulo sistema, presenta gli stessi parametri della classica *activate_p*, ma con in più il parametro *irq*. Con *irq* indico l'identificativo di un piedino dell'APIC: l'idea è che questo processo esterno si deve svegliare ogni volta che viene lanciata un'interruzione da questo piedino.

```
extern "C" void c_activate_pe(void f(natq), natq a, natl prio, natl liv, natb irq) {
    des_proc *p; // des_proc per il nuovo processo
    natw tipo;

    if (prio < MIN_EXT_PRIO || prio > MAX_EXT_PRIO) {
        flog(LOG_WARN, "priorita' non valida: %d", prio);
        c_abort_p();
        return;
    }

    p = crea_processo(f, a, prio, liv, true);
    if (p == 0)
        goto error1;

    tipo = prio - MIN_EXT_PRIO;
    if (!aggiungi_pe(p, tipo, irq))
        goto error2;

    flog(LOG_INFO, "estern=%d entry=%p(%d) prio=%d (tipo=%2x) liv=%d irq=%d",
    p->id, f, a, prio, tipo, liv, irq);

    esecuzione->contesto[I_RAX] = p->id;
    return;

error2: distruggi_processo(p);
error1: esecuzione->contesto[I_RAX] = 0xFFFFFFFF;
    return;
}
```

- Verifico che la priorità inserita sia corretta.
- Chiamo la funzione *crea_processo*. Si osservi che le strutture dati relative ai processi esterni sono le stesse dei processi tradizionali.
- Calcolo il tipo sottraendo al livello di priorità posto in ingresso la minima priorità di un processo esterno. Chiamo la funzione *aggiungi_pe* per gestire *irq*. Passo in ingresso l'indirizzo del descrittore di processo creato, il tipo appena calcolato e *irq*.
- Restituisco l'identificativo del processo appena creato

```
    esecuzione->contesto[I_RAX] = p->id;
```

12.4.1.2 Array di puntatori a descrittori di processo esterni

I descrittori di processo vengono messi in un array di puntatori a descrittori di processo. Si parla di puntatori perchè nella primitiva viene chiamata la *crea_processo*, dunque i descrittori vengono creati e posti nell'array che già conosciamo.

```
// Registrazione processi esterni
const natl MAX_IRQ  = 24; // <-- Nulla di strano, 24 piedini dell'APIC
des_proc *a_p[MAX_IRQ];
```

Questo secondo array è un ulteriore supporto, che utilizzeremo nei cosiddetti *handler*.

12.4.1.3 Funzione *aggiungi_pe*

La funzione *aggiungi_pe* viene chiamata nella primitiva e si occupa, in sostanza, di gestire il parametro di ingresso aggiuntivo *irq*.

```
extern "C" bool load_handler(natq tipo, natq irq);
// associa il processo esterno puntato da "p" all'interrupt "irq".
// Fallisce se un processo esterno era già  stato associato a quello stesso interrupt
bool aggiungi_pe(des_proc *p, natw tipo, natb irq) {
    if (irq >= MAX_IRQ) {
        flog(LOG_WARN, "irq %d non valido (max %d)", irq, MAX_IRQ);
        return false;
    }
    if (a_p[irq]) {
        flog(LOG_WARN, "irq %d occupato", irq);
        return false;
    }
    if (!load_handler(tipo, irq)) {
        flog(LOG_WARN, "tipo %x occupato", tipo);
        return false;
    }

    a_p[irq] = p;
    apic_set_VECT(irq, tipo);
    apic_set_MIRQ(irq, false);
    apic_set_TRGM(irq, false);
    return true;
}
```

- Verifico che *irq* sia un valore accettabile (abbiamo una struttura dati con un massimo di processi esterni, il numero di piedini dell'APIC).
- Verifico che il piedino *irq* indicato non sia già stato associato a un processo esterno.
- Chiamo la funzione *load_handler*, implementata in Assembler, con cui associo il tipo al piedino *irq* nel gate nella *Interrupt Descriptor Table* (viene anche calcolato l'indirizzo).

```
    .global load_handler
// La funzione si aspetta un <tipo> in %rdi e un <irq> in %rsi.
```

```
// Provvede quindi a caricare il gate <tipo> della ITD in modo che punti a handler_<irq>.
load_handler:
    movq %rsi, %rax
    // visto che gli handler sono tutti della stessa dimensione,
    // calcoliamo l'indirizzo dell'handler che ci interessa usando
    // la formula "handler_0 + <dim_handler> * <irq>"
    // dove <dim_handler> si può ottenere sottraendo gli indirizzi
    // di due handler consecutivi qualunque.
    movq $(handler_1 - handler_0), %rcx
    mulq %rcx
    movq $handler_0, %rsi
    addq %rax, %rsi
    // ora %rsi contiene l'indirizzo dell'handler, mentre %rdi
    // contiene ancora il tipo
    movq $LIV_SISTEMA, %rdx
    xorq %rcx, %rcx // tipo interrupt
    call init_gate
    ret
```

- Pongo il puntatore al descrittore di processo nell'entrata *irq* dell'array

  ```
  a_p[irq] = p;
  ```

- Lavoro sull'APIC: associo il tipo all'*irq* dell'APIC (*vect*), attivo le interruzioni sul piedino (*mirq*) e imposto come riconoscimento delle interruzioni quello sul fronte (convenzione).

  ```
  apic_set_VECT(irq, tipo);
  apic_set_MIRQ(irq, false);
  apic_set_TRGM(irq, false);
  ```

12.4.2 Struttura del processo esterno e routine *handler*

Il corpo del processo esterno sarà strutturato con un ciclo infinito, dove si fanno una serie di cose e alla fine dell'iterazione si chiama la primitiva wfi (*wait for interrupt*), che lo sospende in attesa della prossima richiesta di interruzione.

```
for(;;) {
    [...]
    wfi();
}
```

Ogni volta che c'è una richiesta di interruzione dobbiamo lanciare **per forza** qualcosa che assomigli a un driver (una qualche routine che temporaneamente usa le risorse del processo interrotto). La cosa è l'**handler**.

handler L'handler è una routine atomica (**gira nel modulo sistema**) che

- sospende il processo in esecuzione, e
- risveglia il processo esterno precedentemente fermato con la wfi.

```
handler_i:
    call salva_stato
    call inspronti

    // esecuzione = a_p[i]  <--- imposto il processo esterno, quello col ciclo infinito
    movq $i, %rcx
    movq a_p(, %rcx, 8), %rax
    movq %rax, esecuzione

    call carica_stato
    iretq
```

L'handler interrompe un processo che non ha a che vedere con l'I/O. Assumiamo che i processi esterni abbiano tutti una priorità maggiore rispetto ai processi utente: salvo lo stato del processo interrotto e faccio una schedulazione forzata (so che tutti i processi esterni hanno priorità maggiore). Si osservi l'array *a_p*, che abbiamo appena introdotto.

Dove ricomincia il processo esterno dopo la IRETQ nell'handler? Il processo esterno ricomincia dall'inizio o dalla *wfi*, non è possibile che la IRETQ faccia ripartire il processo esterno in punti diversi.

- L'handler è partito perchè è stata lanciata una richiesta di interruzione avente un certo tipo. Chiaramente l'APIC ha fatto passare la richiesta (l'ha trasmessa al processore).
- Se l'APIC ha fatto passare la richiesta di interruzione significa che precedentemente c'è stato un EOI.
- **L'unica cosa che fa passare la EOI è la wfi**. ■

12.4.3 Primitiva *wfi*

La primitiva *wfi* (*wait for interrupt*) viene chiamata dal processo esterno quando ha finito una richiesta di interruzione e vuole sospendersi in attesa della prossima richiesta di interruzione.

```
wfi:
    int $TIPO_WFI
    ret

a_wfi:
    call salva_stato
    call apic_send_EOI
    call schedulatore
    call carica_stato
    iretq
```

Dobbiamo scegliere un altro processo, dove mettiamo quello esterno? Da nessuna parte, il processo esterno viene messo in esecuzione da un *handler* solo quando arriva un certo interrupt. Ci prepariamo una "tabellina", un array di processi, indicizzata dall'*Interrupt Request Number* *i*. In ogni entrata avremo il puntatore al corrispondente *des_proc* esterno.

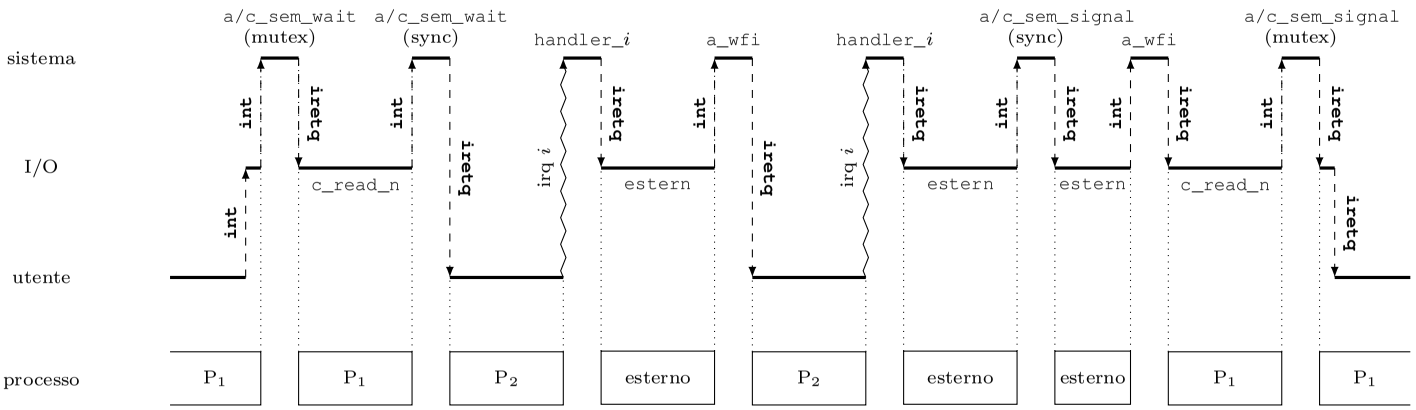
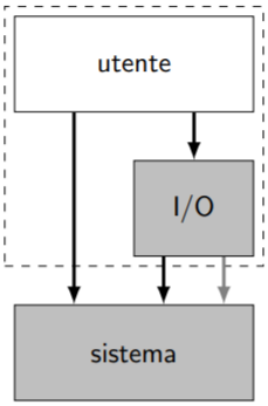


Figura 6: Esempio di esecuzione temporale in seguito all'invocazione di `read_n()` da parte di un processo P_1 (trasferimento di due byte). Le tre righe in alto mostrano il modulo che contiene il codice attualmente in esecuzione. La riga in basso mostra quale processo sta eseguendo il codice. In particolare, quando siamo nel modulo sistema tutti i processi sono fermi e il codice è eseguito atomicamente, mentre il codice del modulo I/O è sempre eseguito da un processo (quello che ha invocato la primitiva oppure un processo esterno).

12.4.5 Recap

Evidenzio in scuro i moduli eseguiti a livello sistema, mentre tratteggio ciò che viene eseguito con interruzioni abilitate.

- Il modulo I/O è non atomico, ma eseguito a livello sistema.
- Il modulo I/O usa primitive fornite dal modulo sistema e fornisce primitive al modulo utente (per esempio quelle di read e write, che fino ad ora abbiamo sempre posto nel modulo sistema).

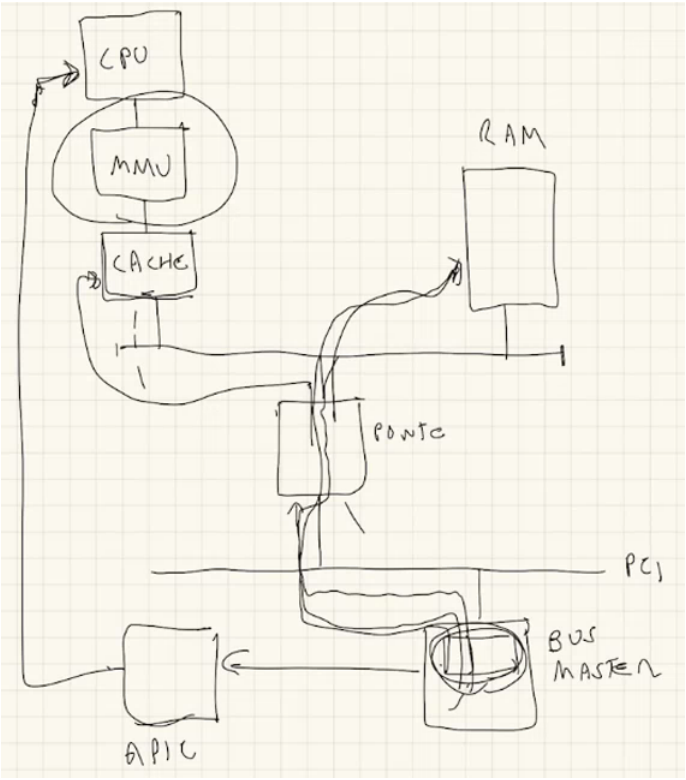


Altre primitive sono destinate esclusivamente al modulo I/O Possiamo impedirne l'uso da parte del modulo utente impostando la CPL, nelle relative entrate della IDT. La cosa è risolvibile in modo semplice grazie all'esecuzione del modulo I/O a livello di sistema. Le primitive che ci servono nel modulo I/O sono:

- *activate_pe*;
- *wfi*;
- *gate_init* (per un inserimento controllato nel gate).

12.5 Primitive per operazioni di bus mastering

Il nostro sistema offre agli utenti di programmare periferiche bus master.



Supponiamo di avere primitive di sistema apposite

```
dmaread_m(int id, char* buf, natl quanti)
```

L'utente specifica una periferica, il buffer, e quanti byte vuole trasferire. L'idea è la stessa dell'operazione non in DMA: In caso di lettura si alloca un buffer, indicando la periferica dalla quale si vuole trasferire. Si tiene conto, anche in questo caso della mutua esclusione

```
wait mutex
avvia operazione
wait sync
rel mutex
```

Dopo aver avviato l'operazione avremo tutte le cose che abbiamo visto succedere nel bus master, tutto questo mentre la CPU lavora su altro. Quando tutta l'operazione si è conclusa il bus master invia una richiesta di interruzione con l'APIC, segnalando la conclusione dell'intera operazione: a quel punto viene eseguito l'handler. Il processo esterno, grazie alla DMA, ha poco da fare:

- segnala al bus master che la richiesta di interruzione è stata gestita (*aknowledge*, lettura di un registro);
- rilascia un gettone a sync, risvegliando il processo all'inizio.

Queste cose, che potevamo già immaginare, dobbiamo pensarle con la presenza della MMU.

12.5.1 Problemi col bus master

- **Il bus master non ha a che vedere con la MMU**.
  - Per comunicare con la RAM è necessario conoscere l'indirizzo fisico, non c'è storia.
  - Dobbiamo scorrere l'albero di traduzione e scoprire quale indirizzo fisico è associato all'indirizzo virtuale *buf*.
  - **Primitiva *trasforma*, già vista** Da un punto di vista architetturale siamo nel modulo I/O, quindi possiamo eseguire la primitiva *trasforma*:

    ```
    paddr trasforma(vaddr v)
    ```

    passo un indirizzo virtuale e ne ottengo uno fisico.

- **I frame potrebbero essere non contigui**.
  - Supponiamo di voler lavorare su un buffer di grandi dimensioni: i frame potrebbero non essere contigui, quindi potrei finire in un frame che non ha nulla a che vedere col buffer.
  - La soluzione dipende dalla periferica. Potrei creare in memoria una tabella dove si eseguono vari trasferimenti: indico alla periferica dove si trova la periferica (indirizzo fisico), si legge le entrate e ognuna è un indirizzo di trasferimento diverso (con un numero di byte). Indico anche quale sia l'ultima operazioni attraverso un flag.
  - Se la periferica non è troppo intelligente allora eseguiamo la cosa in più passi: ogni volta programmo la periferica solo per la parte di trasferimento di un frame. Quando finisce riceviamo un'interruzione e programmiamo un'altra operazione. Ad ogni cambio di pagina chiameremo la *trasforma* per ottenere il corrispondente indirizzo fisico.
  - **Requisito opposto rispetto ai trasferimenti non in DMA**.

    Nei trasferimenti non in DMA abbiamo detto che è obbligatorio avere il buffer nella parte condivisa. **Vale la stessa cosa anche con i trasferimenti in DMA?** L'indirizzo del buffer passato alla primitiva non viene utilizzato direttamente, ma tradotto in fisico con la primitiva *trasforma*. A questo punto non è più necessario avere il buffer nella parte condivisa.

Capitolo 13

Codice del Modulo I/O

13.1 Contenuto della cartella *io*

Nella cartella *io* troviamo i soliti file:

- *io.cpp*,
- *io.s*.

All'interno abbiamo l'implementazione di primitive e processi esterni per le periferiche che abbiamo già visto, in particolare tastiera, memoria video e hard disk.

Timer Il *timer* è l'unica periferica implementata nel modulo sistema. Serve al sistema per implementare la *delay*.

13.2 Ritorniamo nella cartella *include*

In include abbiamo tre file contenenti le dichiarazioni delle primitive offerte.

13.2.1 *sys.h*

sys.h contiene le primitive offerte dal modulo sistema, quelle che possono essere usate sia nel modulo utente che nel modulo sistema.

```
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 delay(natl n);

extern "C" void do_log(log_sev sev, const char* buf, natl quanti);

[...]
```

13.2.2 *io.h*

io.h contiene le primitive implementate dal modulo I/O e rivolte al modulo utente.

```
extern "C" void iniconsole(natb cc);
extern "C" natq readconsole(char* buff, natq quanti);
extern "C" void writeconsole(const char* buff, natq quanti);

extern "C" void readhd_n(void* vetti, natl primo, natb quanti);
extern "C" void writehd_n(const void* vetto, natl primo, natb quanti);
extern "C" void dmareadhd_n(void* vetti, natl primo, natb quanti);
extern "C" void dmawritehd_n(const void* vetto, natl primo, natb quanti);

[...]
```

- Le prime tre sono per tastiera e video.
- Le rimanenti sono per leggere e scrivere nell'hard disk. Si ha la versione in DMA e quella non in DMA per entrambe le operazioni (lettura / scrittura).

13.2.3 *sysio.h*

sysio.h contiene le primitive che il modulo sistema implementa **SOLO** per il modulo I/O.

```
extern "C" natl activate_pe(void f(int), int a, natl prio, natl liv, natb type);
extern "C" void wfi();
extern "C" void abort_p();
extern "C" void io_panic();
extern "C" paddr trasforma(void* ff);
extern "C" bool access(const void* start, natq dim, bool writeable);
extern "C" void fill_gate(natl tipo, vaddr f);
```

- **activate_pe** e *wfi* (per le cose dette prima).
- **abort_p**. Si permette di abortire i processi: la primitiva fa i controlli sui parametri passati dall'utente, se questi non vanno bene si deve abortire. **Idealmente solo il modulo sistema può abortire i processi**: per questo si fornisce una primitiva.
- **io_panic** equivalente della *panic* di sistema, se c'è qualche errore che impedisce di andare avanti.
- **fill_gate**, chiamata dal modulo I/O per associare una propria funzione a una certa entrata della IDT (e quindi a un certo tipo).
- **access**: primitiva per le verifiche atte a risolvere il problema del cavallo di Troia (nel modulo sistema chiamiamo direttamente *c_access*, qua invochiamo la primitiva - negli esami facciamo la stessa cosa con la *abort*).
- **trasforma**: primitiva per ottenere, dato un indirizzo fisico e un albero di traduzione, il corrispondente indirizzo virtuale.

13.3 File *io.s*

13.3.1 Solite inizializzazioni

Come nel modulo sistema dobbiamo fare le solite inizializzazioni di costruttori di oggetti globali.

```
    .text
    .global _start, start
_start:
start:
    // chiamiamo eventuali costruttori di oggetti globali
    movabs $__init_array_start, %rbx
    movabs $__init_array_end, %r12
1:  cmpq %r12, %rbx
    je 2f
    call *(%rbx)
    addq $8, %rbx
    jmp 1b
    // il resto dell'inizializzazione e' scritto in C++
2:  jmp main
```

Dopo aver fatto questo ci spostiamo nella funzione *main*, implementata in C++ e contenente il resto delle inizializzazioni.

13.3.2 Interfacce

Il modulo presenta le interfacce per chiamare le primitive di sistema (*chiamate di sistema generiche*), esattamente come il modulo sistema.

```
.global getmeminfo
getmeminfo:
    int $TIPO_GMI
    ret

    .global activate_p
activate_p:
    int $TIPO_A
    ret

    .global terminate_p
terminate_p:
    int $TIPO_T
    ret

    .global sem_ini
sem_ini:
    int $TIPO_SI
    ret
```

```
    .global sem_wait
sem_wait:
    int $TIPO_W
    ret

    .global sem_signal
sem_signal:
    int $TIPO_S
    ret

    .global delay
delay:
    int $TIPO_D
    ret

    .global do_log
do_log:
    int $TIPO_L
    ret
```

Oltre a queste presenta le interfacce per le primitive da lui implementate (*chiamate di sistema specifiche per il modulo I/O*).

```
    .global activate_pe
activate_pe:
    int $TIPO_APE
    ret

    .global wfi
wfi:
    int $TIPO_WFI
    ret

    .global abort_p
abort_p:
    int $TIPO_AB
    ret
```

```
    .global io_panic
io_panic:
    int $TIPO_IOP
    ret

    .global fill_gate
fill_gate:
    int $TIPO_FG
    ret

    .global trasforma
trasforma:
    int $TIPO_TRA
    ret

    .global access
access:
    int $TIPO_ACC
    ret
```

13.3.3 Gate per le primitive di I/O

Come già anticipato la *Interrupt Descriptor Table* posta nel modulo sistema non è completa. Attraverso il seguente codice inizializziamo i gate rimanenti, necessari per l'esecuzione delle primitive.

```
// Chiama fill_gate con i parametri specificati
.macro fill_io_gate gate off
    movq $\gate, %rdi
    movabs $\off, %rax
    movq %rax, %rsi
    movq $LIV_UTENTE, %rdx
    call fill_gate
.endm

// Inizializzazione dei gate per le primitive di IO
    .global fill_io_gates
fill_io_gates:
    pushq %rbp
    movq %rsp, %rbp

    fill_io_gate IO_TIPO_RCON a_readconsole
    fill_io_gate IO_TIPO_WCON a_writeconsole
    fill_io_gate IO_TIPO_INIC a_iniconsole
    fill_io_gate IO_TIPO_HDR a_readhd_n
    fill_io_gate IO_TIPO_HDW a_writehd_n
    fill_io_gate IO_TIPO_DMAHDR a_dmareadhd_n
    fill_io_gate IO_TIPO_DMAHDW a_dmawritehd_n
    fill_io_gate IO_TIPO_GMI a_getiomeminfo

    leave
    ret
```

Chiamo la *fill_gate* e associo i tipi delle primitive alle funzioni di ingresso *a*.

13.3.4 Primitive di I/O

Concludiamo ponendo la parte Assembler delle primitive implementate nel modulo I/O:

```
     .extern c_readconsole
a_readconsole:
     call c_readconsole
     iretq

     .extern c_writeconsole
a_writeconsole:
     call c_writeconsole
     iretq

     .extern c_iniconsole
a_iniconsole:
     call c_iniconsole
     iretq

     .extern c_readhd_n
a_readhd_n:
     call c_readhd_n
     iretq
```

```
     .extern c_writehd_n
a_writehd_n:
     call c_writehd_n
     iretq

     .extern c_dmareadhd_n
a_dmareadhd_n:
     call c_dmareadhd_n
     iretq

    .extern c_dmawritehd_n
a_dmawritehd_n:
    call c_dmawritehd_n
    iretq

    .extern c_getiomeminfo
a_getiomeminfo:
    call c_getiomeminfo
    iretq
```

Si osservi che non sono presenti la *carica_stato* e la *salva_stato*:

- non dobbiamo usarle perchè le primitive non sono atomiche;
- non possiamo usarle neanche per sbaglio, visto che le due funzioni non sono definite nel modulo I/O.

Si ha un semplice innalzamento di privilegio, al loro interno possono sospendersi e riprendersi utilizzando primitive di sistema.

13.4 File *io.cpp*

13.4.1 Funzione *main*

La funzione *main* viene chiamata dalla *start* vista in *io.s*:

```
extern "C" void fill_io_gates();
// eseguita in fase di inizializzazione
extern "C" void main(int sem_io) {
    fill_io_gates();
    mem_mutex = sem_ini(1);
    if (mem_mutex == 0xFFFFFFFF) {
        panic("impossible creare semaforo mem_mutex");
    }
    vaddr end_ = reinterpret_cast<vaddr>(&end);
    end_ = (end_ + DIM_PAGINA - 1) & ~(DIM_PAGINA - 1);
    heap_init((void *)end_, DIM_IO_HEAP);
    if (!console_init())
```

```
        panic("inizializzazione console fallita");
    if (!hd_init())
        panic("inizializzazione hard disk fallita");
    sem_signal(sem_io);
    terminate_p();
}
```

- Chiama *fill_io_gates* per porre nella *Interrupt Descriptor Table* i gate per il modulo I/O.
- Crea il semaforo *mem_mutex* con la primitiva *sem_ini*. Lo abbiamo trovato prima nelle funzioni per lo heap.
- Inizializza lo *heap* con la funzione *heap_init*.
- Inizializza la console con *console_init*
- Inizializza l'hard disk con *hd_init*.
- La *sem_signal* è il rilascio di un gettone relativo alla sincronizzazione tra questo processo esterno e il processo sistema che sta inizializzando il modulo sistema. Abbiamo trovato la *sem_wait* nella funzione *main_sistema* del modulo sistema.
- Terminiamo il processo con *terminate_p*.

13.4.2 Heap del modulo I/O

Anche il modulo I/O ha il suo heap. Definiamo gli operator *new* e *delete*.

```
natl mem_mutex;

void* operator new(size_t s) {
void *p;

sem_wait(mem_mutex);
p = alloca(s);
sem_signal(mem_mutex);

return p;
}

void operator delete(void* p) {
sem_wait(mem_mutex);
dealloca(p);
sem_signal(mem_mutex);
}

[...]
```

La differenza rispetto al modulo sistema è il ricorso alle primitive semaforiche. Dobbiamo mantenere la mutua esclusione negli operatori (nel modulo sistema non abbiamo questo problema perchè lì le primitive sono atomiche per definizione).

Parte II

Modello di prova d'esame - Semaforo *mutex*

Un `mutex` è un tipo particolare di semaforo che può assumere soltanto due stati: libero oppure occupato. Inoltre, un mutex ricorda l'identità del processo che lo ha occupato. È un errore se lo stesso processo tenta di occupare nuovamente il mutex prima di averlo liberato. Inoltre, è un errore se il mutex viene liberato da un processo diverso da quello che lo aveva occupato.

Per realizzare un mutex definiamo la seguente struttura (file `sistema.cpp`):

```
struct des_mutex {
    natl owner;
    des_proc* waiting;
};
```

Se il mutex è occupato, il campo `owner` contiene l'id del processo che lo ha occupato, altrimenti `owner` vale 0. Il campo `waiting` serve a realizzare una lista di processi in attesa di acquisire il `mutex`.

Le seguenti primitive, accessibili dal livello utente, operano sui `mutex`:

- `natl mutex_ini()` (già realizzata): inizializza un nuovo mutex, con i campi `owner` e `waiting` entrambi a 0, e ne restituisce l'identificatore. Se non è possibile creare un nuovo mutex restituisce `0xFFFFFFFF`.
- `void mutex_wait(natl mux)`: tenta di occupare il mutex di identificatore `mux`. Se il mutex è già occupato sospende il processo in attesa che il mutex venga prima liberato. Abortisce il processo in caso di errore.
- `void mutex_signal(natl mux)`: libera il mutex di identificatore `mux`. Se qualche altro processo era in attesa, lo risveglia e gli cede il mutex (che in questo caso resta occupato). Gestisce una eventuale *preemption*. Abortisce il processo in caso di errore.

Modificare i file `sistema.cpp` e `sistema.s` in modo da realizzare le primitive mancanti.

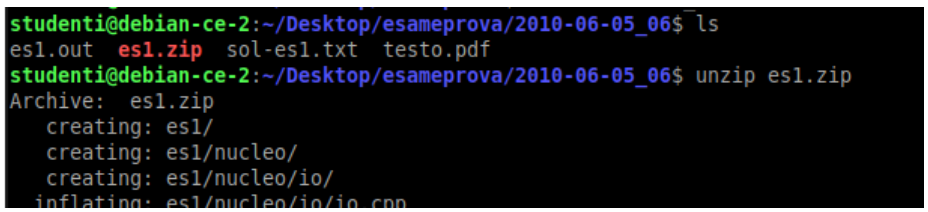
- I semafori sono primitive generiche che permettono di risolvere una grande varietà di problemi. L'esercizio introduce una nuova tipologia di semaforo, specializzato su alcuni aspetti rispetto ai classici semafori (per esempio potrei concentrarmi sulla questione della mutua esclusione): il *mutex*.
- Il semaforo *mutex* presenta solo due stati: libero e occupato. Le proprietà del *mutex* sono indicate nella consegna.

Struttura di una prova d'esame

- **Unzip della prova d'esame**.

  La prima cosa che facciamo è l'unzip del file contenente il codice del nucleo

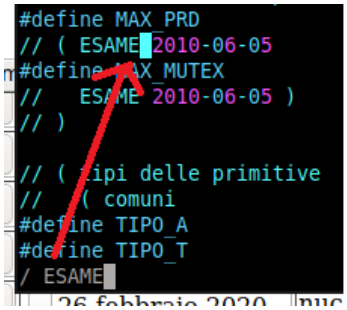
  `unzip es1.zip`



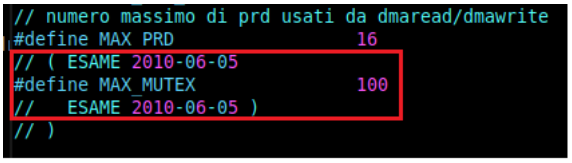
verrà creata una cartella nella directory in cui ci troviamo. La versione del nucleo offerta è modificata apposta per l'esercizio.

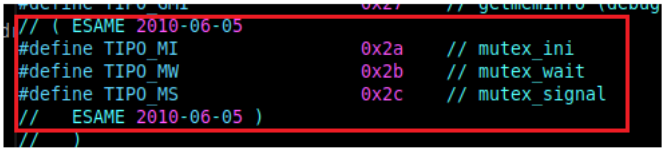
- **Modifiche del nucleo introdotte per la prova**.

  Le parti aggiunte, così come le aree dove dobbiamo scrivere qualcosa, vengono segnalate con del testo. Ricordarsi che per ricercare testo all'interno dell'editor basta premere lo slash e scrivere quanto desiderato (quando si trova il punto premiamo invio, la ricerca si chiude e possiamo lavorare nel luogo desiderato).

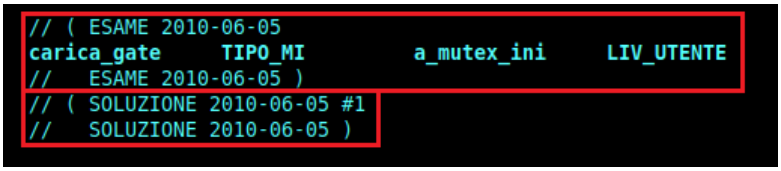


- In questo esercizio abbiamo
  - Aggiunta di costanti in *include/costanti.h*

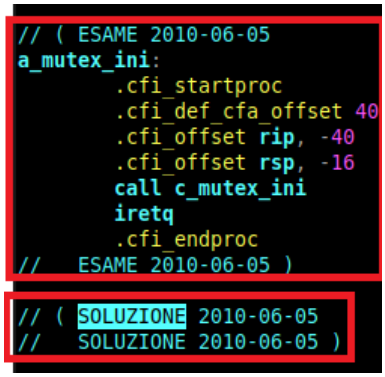




  - Creazione di un gate in *sistema/sistema.s*



    Abbiamo anche la parte relativa all'assembler dell'unica primitiva implementata.



    Si osservino le scritte *SOLUZIONE*: **tra le due scritte deve essere posto del codice, per forza**.
  - **Strutture dati e implementazione della *mutex_ini***.

    In *sistema.cpp* abbiamo l'implementazione delle strutture dati necessarie e dell'unica primitiva implementata

```
// ( ESAME 2010-06-05
struct des_mutex {
        natl owner;
        des_proc* waiting;
};

extern des_mutex array_desm[MAX_MUTEX];

des_mutex array_desm[MAX_MUTEX];
natl mutex_allocati = 0;
natl alloca_mutex()
{
        natl i;

        if (mutex_allocati >= MAX_MUTEX)
                return 0xFFFFFFFF;

        i = mutex_allocati;
        mutex_allocati++;
        return i;
}
bool mutex_valido(natl mux)
{
        return mux < mutex_allocati;
}

// parte "C++" della primitiva sem_ini [4.11]
extern "C" natl c_mutex_ini()
{
        natl i = alloca_mutex();

        return i;
}
// ESAME 2010-06-05 )

// ( SOLUZIONE 2010-06-05
// SOLUZIONE 2010-06-05 )
```

Per quanto riguarda questo esercizio sono state introdotte delle costanti in *costanti.h*, posto il gate relativo all'unica primitiva realizzata in *sistema.s*.

- **Programma per la verifica della correttezza del codice**.

  Nella cartella *prog*, in *utente*, è presente un programma che prova ad usare le primitive richieste. Questo programma stampa l'output che ogni prova deve rispettare nel portale di autocorrezione.

```
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo$ ls
build  debug  include  io  Makefile  run  sistema  utente  util
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo$ cd utente
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo/utente$ ls
all.h  examples  lib.cpp  lib.h  prog  utente.cpp  utente.s
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo/utente$ cd prog
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo/utente/prog$ ls
pmutex.in
studenti@debian-ce-2:~/Desktop/esameprova/2010-06-05_06/es1/nucleo/utente/prog$ vi pmut
```

Solitamente il programma cerca sempre di utilizzare le primitive in modo sbagliato, per verificare se abbiamo posto nel codice tutti i controlli richiesti dalla consegna. Si consideri che l'output corretto non garantisce al 100% che abbiamo svolto l'esercizio correttamente.

Soluzione

Analizziamo la soluzione

- **Per quanto riguarda la parte assembler in *sistema.s* abbiamo**

```
// ( SOLUZIONE 2010-06-05 #1
carica_gate TIPO_MW a_mutex_wait LIV_UTENTE
carica_gate TIPO_MS a_mutex_signal LIV_UTENTE
//   SOLUZIONE 2010-06-05 )

// ( SOLUZIONE 2010-06-05
a_mutex_wait:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_mutex_wait
    call carica_stato
    iretq
    .cfi_endproc

a_mutex_signal:
    .cfi_startproc
     .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_mutex_signal
    call carica_stato
    iretq
    .cfi_endproc
//   SOLUZIONE 2010-06-05 )
```

  - Poniamo i gate per le primitive che la consegna ci richiede di implementare. Per fare ciò ricorriamo alle nuove costanti poste in *costanti.h* (*TIPO_MW* e *TIPO_MS*). Il livello di privilegio richiesto per attraversare il gate non può che essere *LIV_UTENTE* (l'utente non potrebbe utilizzare le primitive con livello di privilegio maggiore).
  - Il codice rimanente consiste nell'implementazione della parte Assembler delle primitive. La struttura può essere copiata dalle altre primitive comuni senza grossi problemi (attenzione a cosa stiamo implementando, in esercizi di tipologia diversa il contenuto di questa parte potrebbe cambiare).

    Attenzione a una cosa: non si rimuovano le righe con *cfi*, sono necessarie per il corretto funzionamento del debugger.

- **Per quanto riguarda la parte C++ in *sistema.cpp* abbiamo**

```
// ( SOLUZIONE 2010-06-05

bool mutex_valido(natl mux);
extern "C" void c_mutex_wait(natl mux) {
    des_mutex *m;

    if (!mutex_valido(mux)) {
        flog(LOG_WARN, "mutex errato: %d", mux);
        c_abort_p();
        return;
    }

    m = &array_desm[mux];

    if (m->owner == esecuzione->id) {
        flog(LOG_WARN, "mutex_wait ricorsiva");
        c_abort_p();
        return;
    }

    if (m->owner == 0) {
        m->owner = esecuzione->id;
    } else {
        inserimento_lista(m->waiting, esecuzione);
        schedulatore();
    }
}
```

- La prima cosa che dobbiamo fare è verificare la validità del parametro di ingresso *mux* (ripetere come l'ave maria, non possiamo fidarci dell'utente): usiamo la funzione *mutex_valido*, fornita dal docente nel nucleo modificato.
- La consegna definisce errore l'occupazione di un mutex da parte di un processo che lo ha già occupato e che non lo ha ancora liberato. Segue la necessità di controllare che il processo che ha lanciato la primitiva sia o no lo stesso indicato nella struttura dati *des_mutex*.
- Se superiamo il controllo precedente allora l'unica cosa che ci manca da controllare è lo stato del *mutex*: è occupato da un altro processo o è libero? A tal proposito andiamo a vedere il valore di *owner*.
  - Se *owner* è 0 il semaforo è libero e aggiorno la variabile ponendo l'identificativo del processo in esecuzione.
  - Se *owner* è diverso da 0 il semaforo non è libero, dunque pongo il processo attualmente in esecuzione in attesa (inserisco il processo attualmente in esecuzione nella lista *waiting* e lancio lo *schedulatore*).

  Ricordarsi cosa fa la funzione *schedulatore*

  ```
  extern "C" void schedulatore(void) {
  ```

```
        esecuzione = rimozione_lista(pronti);
    }

extern "C" void c_mutex_signal(natl mux) {
    des_mutex *m;

    if (!mutex_valido(mux)) {
        flog(LOG_WARN, "mutex errato: %d", mux);
        c_abort_p(); return;
    }

    m = &array_desm[mux];

    if (m->owner != esecuzione->id) {
        flog(LOG_WARN, "mutex_signal su mutex errato");
        c_abort_p(); return;
    }

    if (m->waiting != 0) {
        des_proc *lavoro = rimozione_lista(m->waiting);
        m->owner = lavoro->id;
        // possibile preemption
        inspronti();
        inserimento_lista(pronti, lavoro);
        schedulatore();
    } else { m->owner = 0; }
}

// SOLUZIONE 2010-06-05 )
```

- La prima cosa che dobbiamo fare è verificare la validità del parametro di ingresso *mux* (ripetere come l'ave maria, non possiamo fidarci dell'utente): usiamo la funzione *mutex_valido*, fornita dal docente nel nucleo modificato.
- La consegna definisce errore la liberazione del mutex da parte di un processo diverso da quello che lo ha occupato. Segue la necessità di controllare che l'identificativo del processo in esecuzione sia lo stesso posto in *owner*.
- Se superiamo il controllo precedente l'unica cosa che ci manca è controllare che non ci siano già altri processi in attesa del *mutex*. Lo faccio osservando il valore di *waiting*.
  - Se *waiting* è uguale a 0 non ci sono processi in attesa e pongo *owner* uguale a 0
  - Se *waiting* è diverso da 0 ci sono processi in attesa, riportiamo in vita il processo in attesa con priorità maggiore gestendo la *prelazione* (la cosa è richiesta in modo esplicito dalla consegna).

- Ricordarsi che prelazione significa gestire processi che passano da *esecuzione* a *pronto*. Il processo attualmente in esecuzione passerà da *esecuzione* a *pronto*.

Abbiamo già detto che i processi in attesa sono trattati col criterio FIFO: il primo che arriva è il primo ad uscire. Non vogliamo interrompere il processo attualmente in esecuzione solo perchè è presente un altro processo in *pronti* avente la stessa priorità, dunque:

* rimuovo la testa dalla lista dei processi in attesa del *mutex*;

```
des_proc *lavoro = rimozione_lista(m->waiting);
```

* inserisco forzatamente in testa alla lista *pronti* il processo in esecuzione;

```
extern "C" void inspronti() {
    esecuzione->puntatore = pronti;
    pronti = esecuzione;
}
```

* inserisco il processo rimosso dalla lista dei processi in attesa nella lista *pronti*, con la funzione *inserimento_lista* (che inserisce il processo sulla base delle precedenze);

```
inserimento_lista(pronti, lavoro);
```

* lancio lo schedulatore.

```
schedulatore();

extern "C" void schedulatore(void) {
    esecuzione = rimozione_lista(pronti);
}
```

Ricordarsi come l'ave maria che non si ha il cambio di processo con la chiamata della funzione schedulatore. **SIAMO IN MODALITA' SISTEMA, dunque la primitiva che stiamo eseguendo è ATOMICA**!