

CALCOLATORI ELETTRONICI

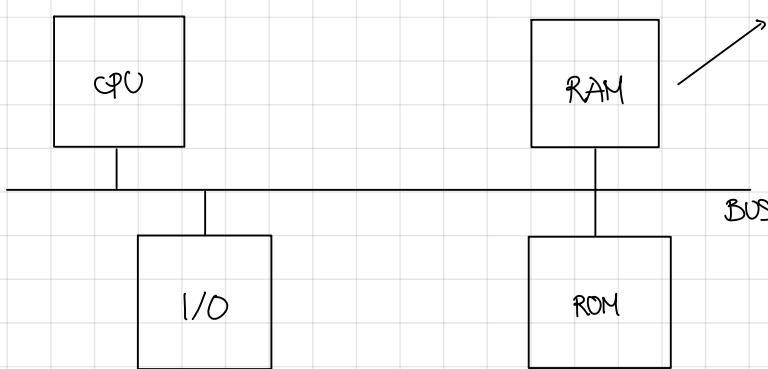
Architettura del calcolatore moderno

INTRODUZIONE → Chi fa cosa?

Il computer può eseguire ogni calcolo richiesto da qualunque programma. Il computer è quindi comandato dal software → da CPU preleva istruzione dalla RAM e la esegue. Anzi ve lo il contenuto dei registri ma NON ricorda ciò che ha eseguito precedentemente e NON sa cosa eseguirà in futuro. Da RAM è solo un CONTENITORE del software in esecuzione. Il sistema operativo comanda ad un livello più alto e CEDE il controllo al software in esecuzione → viene eseguito UN SOFTWARE per VOLTA. Una volta terminata l'esecuzione, il controllo ritorna al software precedente.

IN GENERALE: tutta la terminologia si basa sulle istruzioni aritmetiche in cui abbiamo 2 operandi ed un risultato

CALCOLATORI MODERNI



⚠ ATT. non c'è scatto cos'è una particolare istruzione né vi sono scatti gli indirizzi

In tutte le strutture dati è sufficiente conoscere l'indirizzo base e per accedere agli altri elementi il calcolo del loro indirizzo è veloce

memoria

Affinano la convergenza di 2 indirizzi: permettono di accedere sia all'intera parola (32 bit) oppure ai byte singoli.

Queste modalità di lettura e scrittura sono poi state estese anche allo spazio di I/O

spazio di I/O

Il software ha il controllo dell'I/O e quindi per interagire con i computer moderni è sempre necessario un software → All'avvio viene caricato un programma presente in ROM: bootstrap

Tutto l'I/O viene ridotto a lettura / scrittura: ogni periferica ha un'interfaccia con una serie di registri → Leggere o scrivere in una periferica ha degli effetti collaterali tipicamente irreversibili. Anche la lettura è distruttiva (registro F1).

Spazio di indirizzamento: serie di indirizzi che possono essere indistintamente di memoria e I/O → Viene fatta una distinzione netta tra indirizzi e memoria

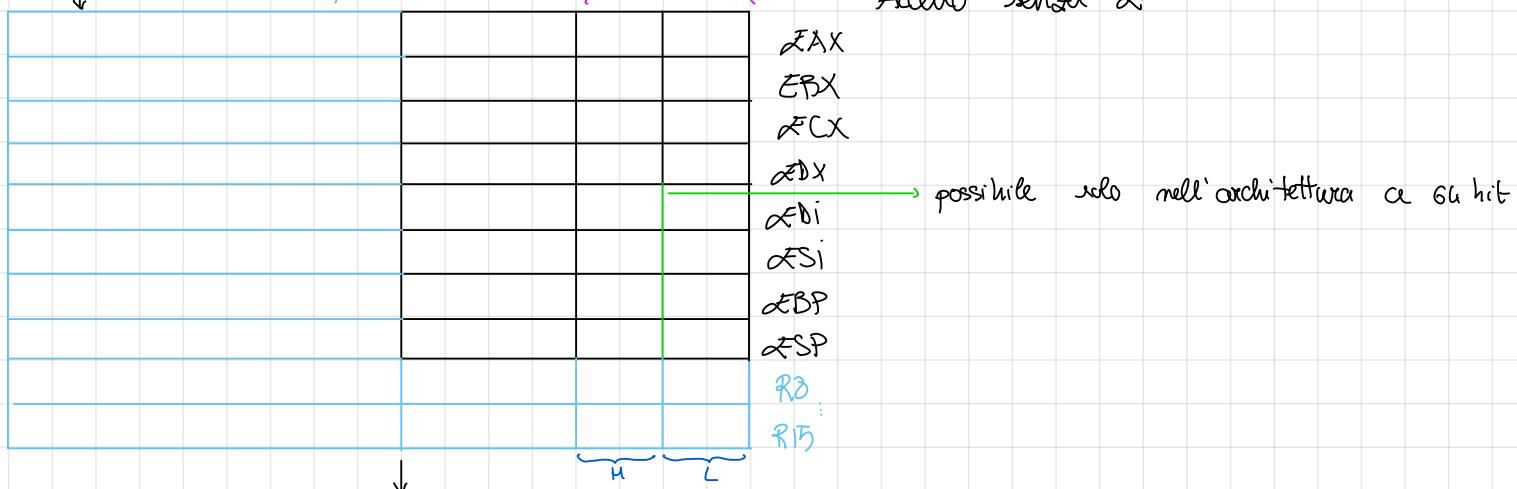
Oggi è presente RAM, periferiche e memoria video. Assumiamo che, in caso noi leggiamo un indirizzo non allocato, venga restituito un valore casuale.

Lo spazio di I/O viene utilizzato dalla CPU solo in caso di I/O OUT, altrimenti viene utilizzata la memoria

Per utilizzare una periferica comune al bus è necessario che questa abbia un indirizzo non riservato ad altre periferiche → accesso tramite indirizzo CPU

Il ciclo fondamentale prevede il registro IP che contiene la prossima istruzione da eseguire. Essa preleva l'istruzione dalla memoria, la decodifica e la esegue ed incrementa IP.

Analizziamo l'architettura AMD a 64 bit: estende l'architettura a 32 bit amplificandone alcuni aspetti e permette di eseguire anche programmi a 32 bit. Vengono estesi i registri a 64 bit e vengono raddoppiati in numero.



è possibile adesso avere entrambi gli operandi in memoria e rimangono le stesse 3 tipologie di operandi: immediato, registro, memoria con lo stesso sintassi già vista precedentemente in Assembly. Abbiamo poi una nuova tipologia data da offset (% RIP)

Specifica quanto dista l'istruzione da RIP che già contiene la prossima istruzione

Poiché quasi tutte le istruzioni gli immediati e gli offset rimangono a 32 bit poiché non allungare tutte le istruzioni a 64 bit. L'unico offset rimangono a istruzione a supportare questa modalità è MOVABS

Dobbiamo sapere TRIMA dove è posizionato il programma in quanto è richiesta una sequenza differente di istruzioni a secondi che queste siano vicine o lontane ↳ l'assemblatore definisce il contenuto della memoria a partire da un programma

In direzionando un dato tramite offset e RIP, la sezione DATA e TEXT non fanno più di 2 GB. Questo vale anche per le istruzioni di CALL

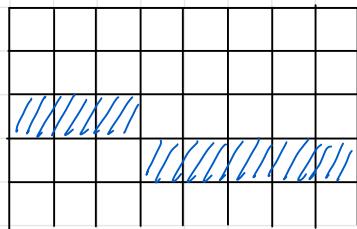
In teoria è a 64 bit, in pratica non tutti gli indirizzi sono accettabili perché i processori più comuni implementano 48 bit e quindi possono accedere ad al massimo 256 TB

Divisi in 2 tronconi con "buco" nel mezzo. Gli indirizzi devono essere morniali fissati: 0 16 bit mancanti devono essere uguali al bit 47 quindi

o tutti 0 o tutti 1 → importanti per la normalizzazione

oggetti allineati: Indirizzo di partenza multiplo di una certa potenza di 2 di un certo indirizzo dato

Immaginiamo la memoria a 64 bit come composta da tutte righe di 8 byte



→ il processore in un' unica operazione in memoria può leggere una riga di 8 byte o tre otto-byte. Sono invece necessarie più operazioni di memoria per leggere più righe

Se una variabile si trova nel modo indicato in blu sono richieste due operazioni in memoria ed in più il ricordo dei posti che si trovano nella parte incagliata del bus. Alcune architetture (ARM) non accettano proprio le variabili poste in questo modo, mentre Intel supporta questa cosa

Se l'oggetto è più grande di un byte (0x1122). Possiamo avere

2 possibili tipi

Big Endian

Little Endian

0x11 22

→ utilizzata da internet
Le macchine Intel sono Little Endian

Utilizziamo questa memoria: 7 6 5 4 3 2 1 0 → Byte numerati da dx

7	6	5	4	3	2	1	0

0 a nx

8

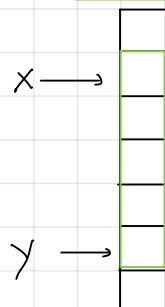
16

NOMENCLATURA degli indirizzi → Cogni byte è rappresentato da 2 cifre eadeciadi
In generale, è importante ricordare:

$$2^a = 1 \underbrace{000 \dots 000}_{a}$$

$$2^a - 1 = 1 \underbrace{M \dots M}_{a}$$

offset: distanza tra 2 indirizzi:



Q: quanti indirizzi dobbiamo notare

Tra x ed y c'è Ø

intervalli: $[x, y)$ per convenzione e perché la dimensione dell'intervolo è $y - x$

Possiamo comporre gli intervalli: $[x, y) + [y, z) : z - x$ con y contata una volta

circoscrivere:

Gli indirizzi sono circolari poiché vogliamo l'aritmetica modulare: $1 \mod 2^n$. Questo accade in quanto abbiamo soltanto n fili. Sommando infatti 1 al massimo rappresentabile, si ottiene un numero composto da n zero ed 1 all'ultima posizione.

Se l'offset è realizzato su n bit non importa il fatto che sia positivo o negativo. Se l'offset è realizzato su meno di n bit ci interessa invece sapere se è positivo o negativo → vengono estesi tenendo conto del segno. Consideriamo inoltre sempre intervalli che non finiscono overflow dell'altra parte

confini / boundaries: lo spazio di indirizzi viene diviso in pezzi di una determinata dimensione (potenze di 2). Gli indirizzi che si trovano all'inizio di ciascuna di queste parti vengono chiamati boundaries. Se

guardando un indirizzo al confine di una porzione di 2^b bit noto che corrisponde a b bit a 0 (quelli meno significativi). Una regione naturale di 2^b è l'insieme degli indirizzi che sta tra i confini di 2^b .

→ Possiamo creare una geografia assegnando a ciascuna regione un numero che identifica tutti gli indirizzi che cadono all'interno di ciascuna regione naturale.

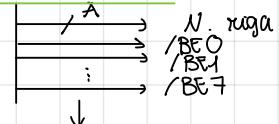
Il 2^b compare in tutti gli indirizzi. In particolare, i b bit meno significativi cominciano, mentre gli $m-b$ bit più significativi sono costanti e uguali a 2^b . I meno significativi variano da 0 a 2^{b-1} e rappresentano l'offset all'interno della regione.

Applicazione:

	7	6	5	4	3	2	1	0
0								
1								
2								
:	:	:	:	:	:	:	:	:

→ L'indirizzo di ciascuna riga è composto nel suo interno rispetto ad essa.

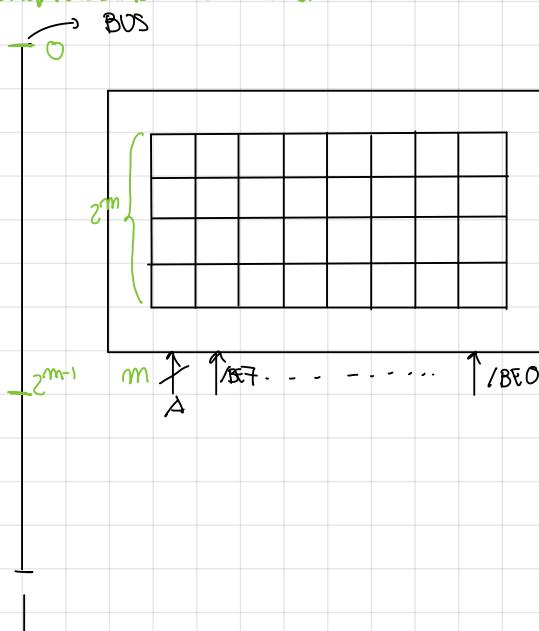
Quando la CPU emette un indirizzo, essa fornisce solo il numero di riga ed 8 byte enable di modo da dire quali byte della riga si interessano:



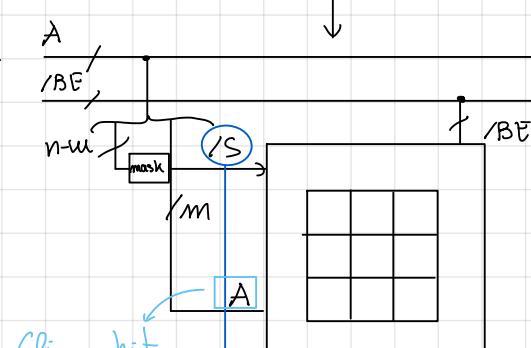
È più flessibile perché può leggere byte sparpagliati ma è limitante perché può leggere una riga per volta → MOV 7, %EAX : 3 operazioni:

- a. A=0 e /BE7=0
- b. A=1 e /BE0 = /BE1 = /BE2 = /BE3 = 0
- c. Shift

ORGANIZZAZIONE MEMORIA



Come fa a capire che l'indirizzo è rivolto a lui?

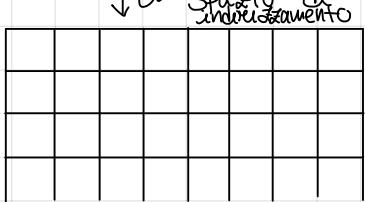


Gli m bit meno significativi specificano l'offset. Gli $m-n$ bit più significativi identificano il numero della riga.

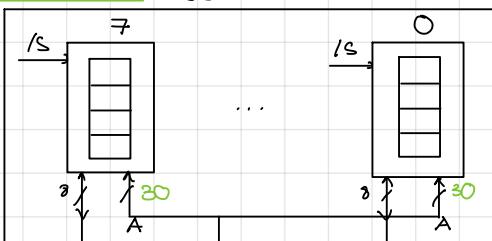
Ogni cosa che occupa indirizzi è un oggetto. Si parla di indirizzo dell'oggetto riferendosi al primo byte che occupa. L'allineamento riguarda le proprietà di tale indirizzo.

Si dice che un oggetto è allineato a 2^b se il suo indirizzo è un confine di 2^b . Si dice che un oggetto è allineato ad un altro oggetto quando l'altro

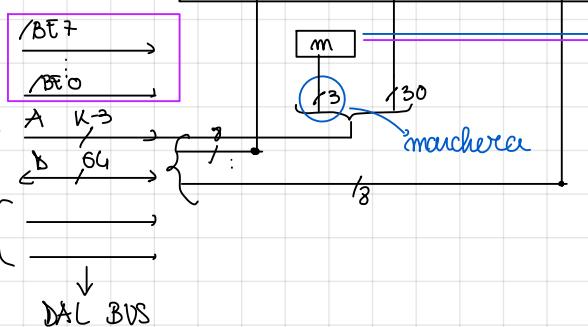
ha una dimensione che è potenza di 2 e che il primo è allineato a quella potenza di 2 → avere allineato ad un long significa avere allineati al 3 byte. Si parla di **allineamento matriciale** quando la dimensione di un oggetto è una potenza di 2 ed è allineato a se stesso.



per rispondere a richieste

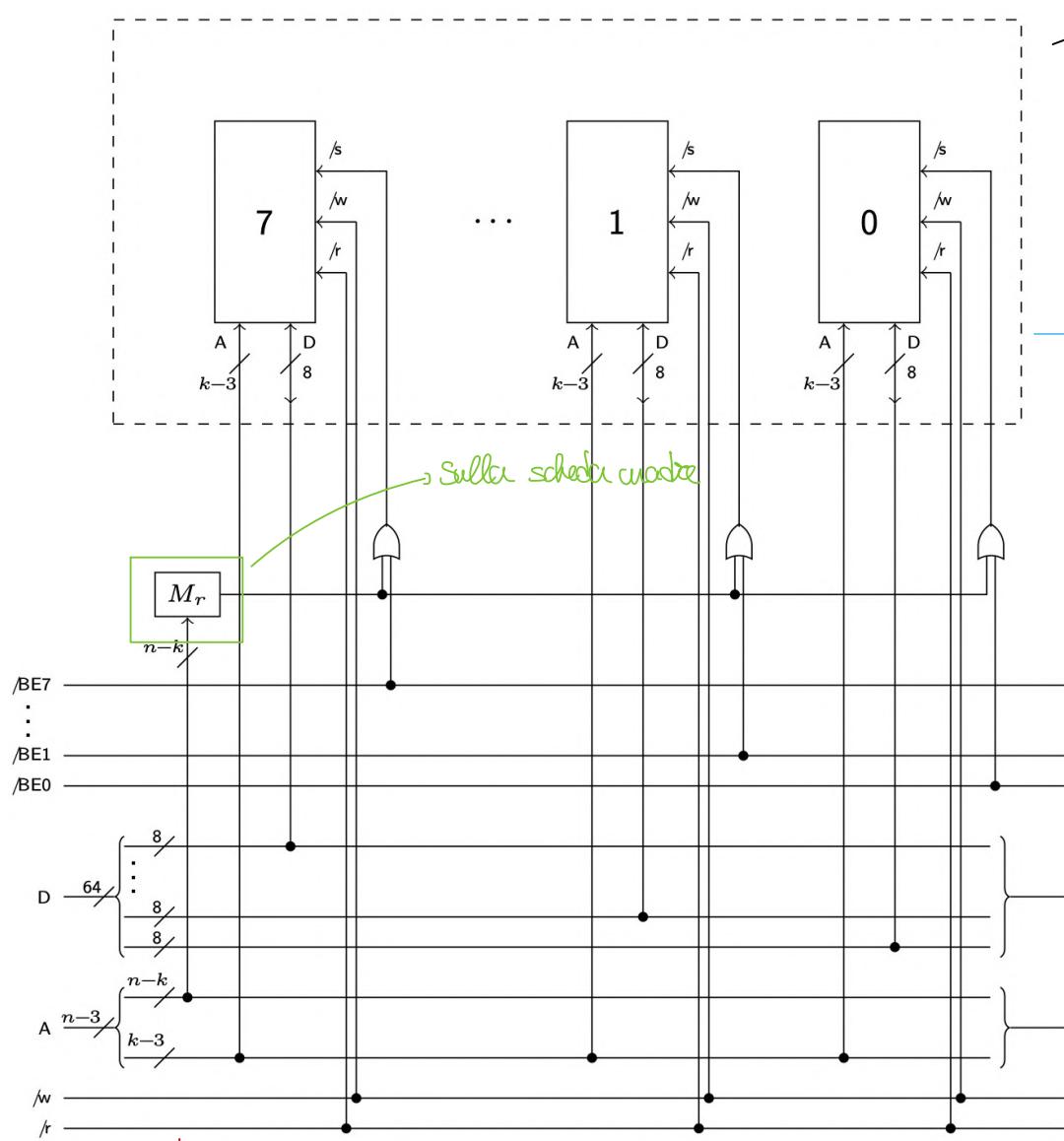


utilizziamo 8 chip di RAM da 1 byte ciascuno



Deve essere verificata via la condizione dei fili di indirizzo che quella dei byte enable

Supponiamo che la memoria sia da 8 GB : Abbiamo $k = 36$
↓ Scritto meglio



Se vogliamo fare un accesso indiretto dobbiamo fare le due operazioni già dette in precedenza

Parte tratteggiata:
si può scrivere
a parte

⚠ ATT: i valori possono essere 0/1 ma non è mai vero che non contengono niente, al massimo valori non significativi

⚠ ATT: possono fare un'operazione per volta

Il compilatore

Utilizziamo lo standard System V che definisce le seguenti convenzioni (su Linux)

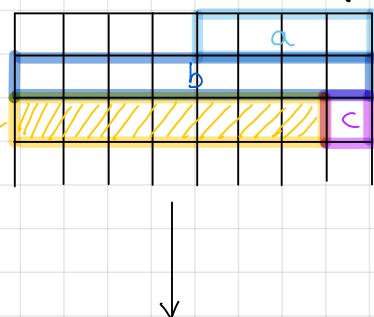
- ↳ char → 1 byte
- ↳ short → 2 byte
- ↳ int → 4 byte
- ↳ long → 8 byte
- ↓ Tipi derivati: i campi all'interno rispettano l'allineamento dei tipi base
- array: int a [...] ;
- ↓ Esempio

	a[0]		
	a[1]	a[2]	
		a[3]	
		a[4]	

} Allineamento matrice: ciascuno colonna propria dimensione
Sovra a semplificare i calcoli
campi all'interno rispettano l'allineamento dei tipi base
Viene mantenuto l'allineamento a 4 byte a partire da
○ (primo elemento)
→ array di 4 elementi

structs: struct {
 int a;
 long b;
 char c;
};

specifici



→ La struttura inizia all'inizio di una riga e i campi vanno allocati in ordine

Mettiamo int a al primo offset che rispetta l'allineamento
↳ Regola generale

multiplo dell'allineamento

La dimensione dell'oggetto deve essere un multiplo dell'allineamento

↓ cosa fa il compilatore?

```
int a=1
long b = 2;
char c[10] = {0};
```

```
int main()
{
    b=3;
}
```

data: se all'assemblatore non si dice niente, mette mai corretto

a:

long 1 → 4 byte: dimensione di un int

b:

↓ quad 2 → 8 byte: dimensione di un long
Quando viene incontrata un'etichetta l'assemblatore si regna il valore del contatore a partire dal cambio del suo valore

.balign: incrementa il contatore fino ad arrivare ad 8



C=4

→ Con balign: vengono specificati di bit in modo da allineare ad 8

ci pensa il compilatore a mettere la direttiva in quanto l'assemblatore fa altranto cosa c'è scritto

Il contatore di b in questo caso vale 8
d'entry point, del collegatore è start: definito in un file oggetto automaticamente collegato quando si esegue un file C++. Questo avviene quando si esegue il programma e si preoccupa di dire l'intervento operativo che il programma è terminato

↓ in Assembly

main : → deve essere dichiarata in modo global: .global main

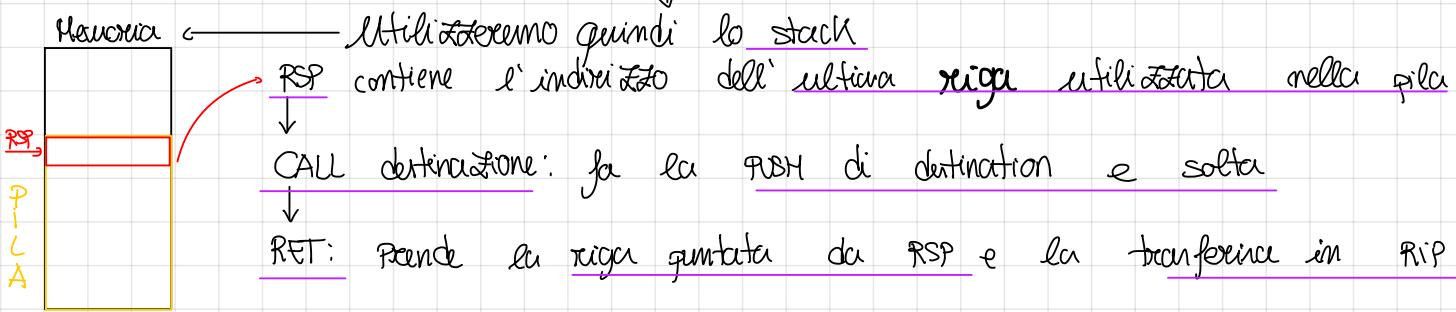
ret

è un'etichetta come le altre poste in TEXT. Le etichette locali vengono ignorate dal collegatore, mentre vengono considerate solo quelle globali

GESTIONE DI FONZIONI

Una funzione è un codice che viene chiamato da un'altra funzione. La stessa funzione può essere chiamata da più punti diversi e deve sapere dove ritornare.

f() → f: → utilizziamo le istruzioni call, ret e lo stack oppure un registro → **Svantaggio:** va sempre salvato



Dobbiamo inoltre gestire parametri, variabili locali e valori di ritorno

a. VARIABILI LOCALI

Per permettere la ricorrenza, le variabili locali vengono messe in pila. La funzione viene chiamata e per prima cosa riserva spazio nella pila.

Allocazione automatica: le variabili vengono definite quando la funzione termina.

⚠ problema: le variabili locali si trovano in punti diversi in ciascuna esecuzione della funzione

b. Utilizziamo un registro di riferimento: %RSP

chi fa cosa
chiamante chiamato

o meglio chi fa cosa

compilazione separata

chiamante

type (type, ..., type);

f(expr, expr...)

chiamato
type f (type, ..., type);
{ ...
:
return exp;
}

Importante perché è possibile ricompilare soltanto un file, utile per programmi molto grandi

Cosa sa il CHIAMANTE? Numero degli argomenti, tipo di argomento, tipo di ritorno, nome della funzione

Dove conosce queste informazioni prima della chiamata, quindi la funzione deve essere prima dichiarata → è sufficiente per tradurla

↳ Già inclusa di default in C++

COSA SA IL CHIAMATO? Per il chiamato è utile solo sapere type, typen: vengono passati dal chiamante al chiamato

↳ È responsabilità del chiamante passare espressioni del tipo corretto

Tra di esso vengono passati in pila, mentre nell'architettura a 64 bit vengono utilizzati i registri → vanno utilizzati in ordine: %RDI, %RSI, %RDX, %RCX, %R8, %R9

Così argomento ne prende un numero intero quindi non si possono mettere più argomenti nello stesso registro → per i tipi base e puntatori ci serve prendere un singolo registro e per gli array abbiamo che questi decadono a puntatori e quindi viene passato il puntatore all'indirizzo del primo elemento.

f → g → h: in questo caso può accadere che il contenuto dei registri vada sovrascritto oltretutto
↓ Se non ottimi ottieno
Si ha sempre salvataggio in memoria

c. VALORI DI RITORNO

Utilizziamo registri: %RAX, %RDX

per raddoppiare la dimensione di %RAX

CONVENZIONI sui REGISTRI

Alcuni registri sono chiamati scratch: il chiamante non scrive niente perché ↓ è compito del chiamato

%RDI, %RSI, %RDX, %RCX, %R8, %R9 + %RAX, %RDX + %RIO, %R11

Altri sono preservati: il chiamato deve preoccuparsi di salvare prima di sovraccoprire

↓
%RBP, %RDX

chiamato ↓ inizio funzione

PROLOGO

push %RBP (preservato)

MOV %RSP, %RBP

SUB \$x, %RSP → lo sposta in modo da poter contenere le variabili locali

SALVATAGGIO registri preservati
COPIA parametri

Prepara base pointer e alloca spazio per variabili locali

lo sposta in modo da poter contenere le variabili locali

EPILOGO [RIPRISTINO registri - salvati]

LEAVE

quivale a ↓ equivale a

MOV %RBP, %RSP
POP %RBP

[RET

Spazio per parametri
e variabili locali

← %RSP

%RSP →

vecchio %RBP

ind. RIT

← %RSP

← %RSP

ESEMPIO

```
studenti@ce: ~/as
File Edit Tabs Help
extern "C" int f(int, int);
int main()
{
    int b = 2;
    int a = f(b+3, b);
    return a;
}
int f(int a1, int a2)
{
    return a1+a2;
}
```

sarebbe che la funzione sia dichiarata
↓ corpo di f



.global f → Altrimenti non viene associata ad f nel main
f:

push %rbp
mov %rsp, %rbp
sub \$16, %rsp

→ vogliamo trovare posto per 2 interi
CONVENZIONE SYSTEM5: stack allineato a 16
a1 in %rdi
a2 in %rsi

mov %edi, -8(%rbp)
mov %esi, -4(%rbp)

{ versione non ottimizzata: tutto in memoria }

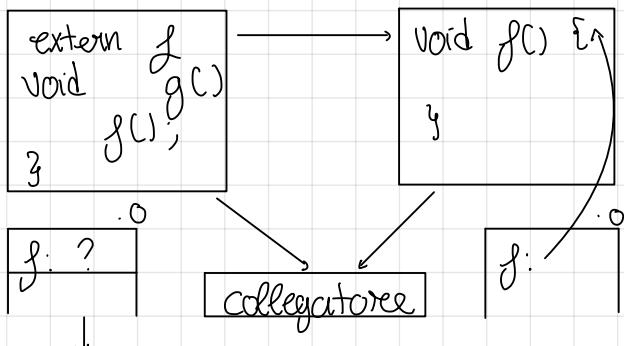
fine prologo

mov -8(%rbp), %eax
add -4(%rbp), %eax
epilogo
leave
ret

extern "C" void f();

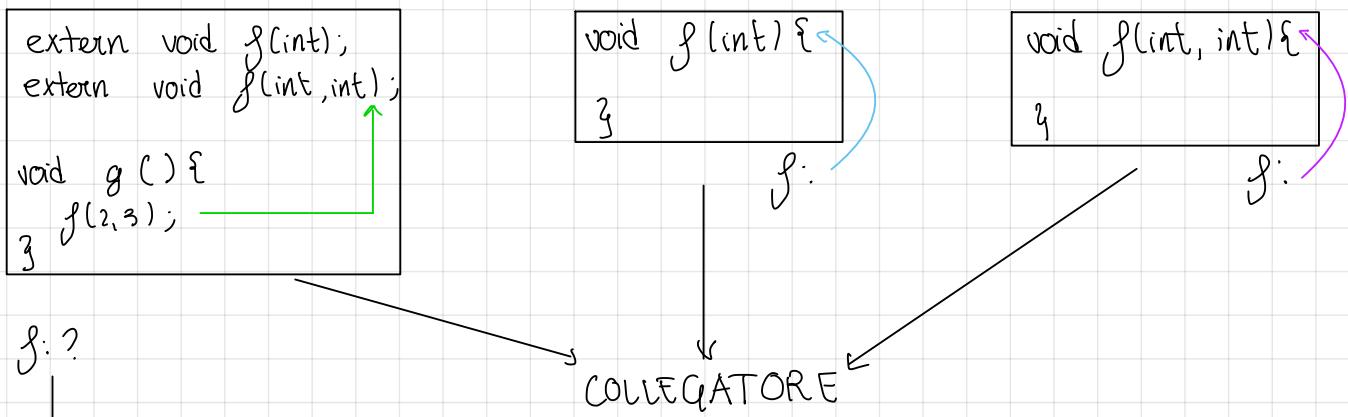
→ mi dice che la funzione rispetta lo standard del C per la chiamata

Se metterei C++ specifiche diverse: non è sufficiente mettere il nome della funzione come etichetta per permettere overloading → f(int), f(int,int), f(int*)



Nel file .o c'è una tabella dei simboli

Per ogni file ove sono marcate come indefinite esso cerca un file dove queste sono definite → confronto tra stringhe



Le funzioni vengono tradotte codificando le loro interne numeri e tipo di parametri: la prima si chiama f_1 , la seconda f_2

REGOLE: fun(type1, ..., typen) →



tabella dei tipi: int → i

char → c

long → l

type* → Ptype

void f(int*) → -Z1fPi

void f(int**) → -Z1fPPI

struct miscstruct {

}

si riferisce all'etichetta di struct

void f(misctruct) → -Z1fPmisctruct

void f(misctruct*) → -Z1fPmisctruct

void f(int&, const char&) → -Z1R0R0 tipo riferito

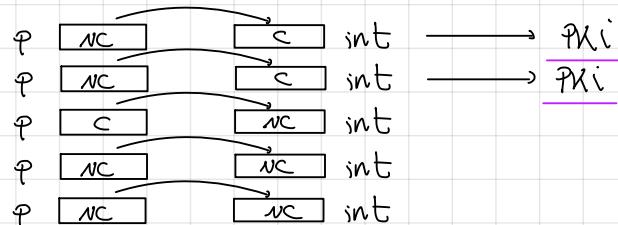
array:

void f(int a[100]), void f(int a[100]), void f(int a[]), -Z1fPi

void f(int*a) → Sono tutte uguali: -Z1fPi

const:

const int *
int const *
int * const
const int * const
int const * const



void f(int const) → Sono uguali: il punzecchio è per valore e quindi la funzione non può modificare i parametri traduttore zj, fi: la stringa non può terminare con K

LE CLASSI

class miaclasse {
};

→ Differenza tra classe e struttura: visibilità di default

public:

miaclasse();
miaclasse (int);
~miaclasse();
fun1();
fun2 (int);
fun1 (miaclasse *);
;

};

Non viene tradotto in quanto se non vengono rispettate le regole non viene creato nemmeno file Autocompiler.

→ void f () {

miaclasse c; → da forma dipende solo delle variabili membri e non delle funzioni
c.fun1();
} ↓
chiamata a funzione membro che equivale a scrivere miaclasse.fun1 (&c,...)

Nella traduzione abbiamo una normale traduzione come le altre funzioni scendo &c con %ORH

Aggiungiamo all'overloading

class cc {

void f (int);
void f (int, int);
void f (cc);
void f (cc &);

il fatto che la funzione sta in una classe

nested

End

-ZN2CC1fEi
-ZN2CC1fEii
-ZN2CC1fES_

La stringa è già presente prima (2CC)

-ZN2CC1fERS-

Spazi di nomi per arrivare a funzione

};

+

costruttori → C1

caso precedente

-ZN2CC1fEV void: senza argomenti

distruttori → D1

caso precedente

-ZN2CC1fEV

costruttore di copia → cc (const

cc &);

-ZN2CC1fERS -

Quando il tipo di ritorno è una struttura / classe: architetturalmente grande → struct ss

class cc

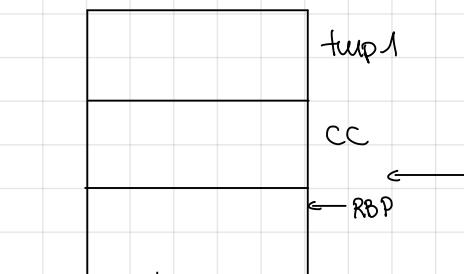
ss f();
cc f();

void f(ss)

↓

Se entra in 16 byte allora la poniamo in %RAX, %RCX a partito che non rice definito un costruttore di copia (anche se ponuta per valore). Quando è più grande di 16 byte avremo

Ch.: cc f()
void g() {
 cc m1oc;
 m1oc = f(); } → Oggetti temporanei che devono esistere ai fini della traduzione



↓ Avrò quindi
lea temp1(%RBP), %RDI
call -2(%RBP) # azioni di f
return

Il chiamante deve trovare lo spazio perché restino a lui

Allocò lo spazio e pone l'indirizzo ove era allocato lo spazio

Il parametro macro è sempre il primo oggetto, quindi sta in %RDI

PREPROCESSORE

```
#include "lib.h"

long var1 = 8;
long var2 = 4;

#ifdef DEBUG
void debug(const char *msg) {}
#else
#define debug(msg)
#endif

int main()
{
    // un commento
    var1 = foo(var2);
    debug("questo è un messaggio di debug");
    return var1;
}
```

main.cpp

```
long foo(long);
void bar();
lib.h
```

```
.data
foovar1:
    .quad 5
foovar2:
    .quad 6
.bss
foovar3:
    .quad 0
.text
.global _Z3foo
_Z3foo:
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, %rax
    movabsq foovar3, %rax
    addq foovar1, %rax
    addq foovar2(%rip), %rax
    leave
    ret
```

foo.s

Il preprocessore fa delle elaborazioni testuali ed i comandi che si riferiscono a lui iniziano per `#`. Questi si chiamano direttive per il preprocessore. Il compilatore non le vedrà mai perché verranno trasformate in altro dal preprocessore.

#include: Trova il file: " " → prima nella directory di preseparazione e poi in posti standard
 < > → soltanto in posti standard

Una volta che lui trova il file sostituisce la linea con il codice del file trovato

#define: utilizzato per definire macro compilazione condizionale: `#ifdef / ifndef`

```
#ifdef DEBUG
void debug(const char *msg) {}
#else
#define debug(msg)
#endif
```

Se la macro DEBUG è definita
 da definisce se non è già definita → Sostituisce la chiamata

Output del preprocessore:

`g++ -E main.cpp`

Fermo al preprocessore

```

# 1 "main.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.cpp"
# 1 "lib.h" 1
long foo(long);
void bar();
# 2 "main.cpp" 2
long var1 = 8;
long var2 = 4;
int main()
{
    var1 = foo(var2); → Toglie i commenti ed uniforma gli spazi
    ;
    return var1;
}

```

File che arriva al compilatore

`g++ -D DEBUG -E main.cpp`

definisco la macro debug

```

# 1 "main.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.cpp"
# 1 "lib.h" 1
long foo(long);
void bar();
# 2 "main.cpp" 2
long var1 = 8;
long var2 = 4;
void debug(const char *msg) {}
int main()
{
    var1 = foo(var2);
    debug("questo_e'_un_messaggio_di_debug");
    return var1;
}

```

↓
la macro debug era stata definita

da compilazione condizionale viene utilizzata nel caso di compilazione su sistemi operativi diversi esempio, linux: non può esistere un'altra variabile di nome linux → per vedere: `-dM`

COMPILATORE

Dopo aver preprocessato il file questo va in passo al compilatore. Questo produce un file assembly che va avvenuto

```

.data
.global var1, var2
var1:
    .quad 8
var2:
    .quad 4
.text
.global _start
_start:
    pushq %rbp
    movq %rsp, %rbp
    movq var2(%rip), %rax
    movq %rax, %rdi
    call _Z3foo
    movq %rax, var1(%rip)
    movq var1(%rip), %rax
    popq %rbp
    ret

```

→ il programma non può girare

```

.data
foovar1:
    .quad 5
foovar2:
    .quad 6
.bss
foovar3:
    .quad 0
.text
.global _Z3foo
_Z3foo:
    pushq %rbp
    movq %rsp, %rbp
    movabsq foovar3, %rax
    addq foovar1, %rax
    addq foovar2(%rip), %rax
    leave
    ret

```

file main.s

file fo.o

ASSEMBLATORI

Produce i byte che vanno in memoria e può contenere più sezioni tra cui DATA, TEXT ed anche altre. L'assemblatore legge il file due volte completamente:

- Scopre dove e quali sono le etichette
- Assembla effettivamente

È possibile in questo modo fare un salto in avanti. Per ogni sezione tiene un contatore che conta quanti byte occupa questa sezione. Nel primo passaggio scrive da una parte, per ogni etichetta, [etichetta, contatore, sezione]: tabella dei simboli. Una viene incaricata nel secondo passaggio. Se deve essere utilizzata un'etichetta non vista nel primo passaggio viene aggiunta in entrata alla tabella dei simboli per dire che l'etichetta è indeterminata e vengono aggiunte le istruzioni di rilocazione al file di output, utili per il collegatore che sostituisce l'etichetta che non concorda con la funzione effettiva.

formato ELF (file oggetto, eseguibili, librerie)

hexdump -C

```
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00 |.>.....
00000020 00 00 00 00 00 00 00 00 00 00 e0 01 00 00 00 00 |.....
00000030 00 00 00 00 40 00 00 00 00 00 00 40 00 08 00 07 00 |...@....@...
00000040 55 48 89 e5 48 89 f8 48 a1 00 00 00 00 00 00 00 |UH..H..H...
00000050 00 48 03 04 25 00 00 00 00 48 03 05 00 00 00 00 |H..%....H...
00000060 c9 c3 05 00 00 00 00 00 00 00 06 00 00 00 00 00 |.....
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000090 00 00 00 00 03 00 01 00 00 00 00 00 00 00 00 00 |.....
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 03 00 03 00 |.....
000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000000c0 00 00 00 00 03 00 04 00 00 00 00 00 00 00 00 00 |.....
000000d0 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 03 00 |.....
000000e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000000f0 09 00 00 00 00 00 03 00 08 00 00 00 00 00 00 00 |.....
00000100 00 00 00 00 00 00 00 00 11 00 00 00 00 00 04 00 |.....
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000120 19 00 00 00 10 00 01 00 00 00 00 00 00 00 00 00 |.....
00000130 00 00 00 00 00 00 00 00 00 66 6f 6f 76 61 72 31 |.....
00000140 00 66 6f 6f 76 61 72 32 00 66 6f 6f 76 61 72 33 |.foovar2.foovar3|
00000150 00 5f 5a 33 66 6f 6f 6c 00 00 00 00 00 00 00 00 |..Z3foo1...
00000160 09 00 00 00 00 00 00 00 01 00 00 03 00 00 00 00 |.....
00000170 00 00 00 00 00 00 00 00 15 00 00 00 00 00 00 00 |.....
00000180 0b 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 |.....
00000190 1c 00 00 00 00 00 00 00 02 00 00 02 00 00 00 00 |.....
000001a0 04 00 00 00 00 00 00 00 00 2e 73 79 6d 74 61 62 |.....symptr|
000001b0 00 2e 73 74 72 74 61 62 00 2e 73 68 73 74 72 74 |..strtab.shstrt|
000001c0 61 62 00 2e 72 65 6c 61 2e 74 65 78 74 00 2e 64 |ab..rela.text..d|
000001d0 61 74 61 00 2e 62 73 73 00 00 00 00 00 00 00 00 |ata.bss.....
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
*
00000220 20 00 00 01 00 00 00 06 00 00 00 00 00 00 00 00 |.....
00000230 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 |.....
00000240 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |".....
00000250 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000260 1b 00 00 00 04 00 00 40 00 00 00 00 00 00 00 00 |.....
00000270 00 00 00 00 00 00 00 60 01 00 00 00 00 00 00 00 |.....
00000280 48 00 00 00 00 00 00 05 00 00 01 00 00 00 00 00 |H.....
00000290 08 00 00 00 00 00 00 18 00 00 00 00 00 00 00 00 |.....
000002a0 26 00 00 01 00 00 03 00 00 00 00 00 00 00 00 00 |&.....
000002b0 00 00 00 00 00 00 00 62 00 00 00 00 00 00 00 00 |.....
000002c0 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000002d0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000002e0 2c 00 00 08 00 00 03 00 00 00 00 00 00 00 00 00 |.....
000002f0 00 00 00 00 00 00 00 72 00 00 00 00 00 00 00 00 |.....
00000300 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000310 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000320 01 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000330 00 00 00 00 00 00 00 78 00 00 00 00 00 00 00 00 |.....
00000340 c0 00 00 00 00 00 00 06 00 00 07 00 00 00 00 00 |.....
00000350 08 00 00 00 00 00 00 18 00 00 00 00 00 00 00 00 |.....
00000360 09 00 00 03 00 00 00 00 00 00 00 00 00 00 00 00 |.....
00000370 00 00 00 00 00 00 00 38 01 00 00 00 00 00 00 00 |.....
00000380 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |!.....
00000390 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000003a0 11 00 00 03 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000003b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000003c0 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |!.....
000003d0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
000003e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....
```

readelf -h: decodifica l'inizio →
del file che contiene info sul
contenuto del resto del file

```
ELF Header:
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: REL (Relocatable file)
Machine: Advanced Micro Devices X86-64
Version:
Entry point address:
Start of program headers: 0x1
Start of section headers: 0x0
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 64 (bytes)
Number of section headers: 8
Section header string table index: 7
```

Negli eseguibili: per caricare programma in memoria
tabella delle sezioni

There are 8 section headers, starting at offset 0x1f0:

→ readelf -WS

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	PROGBITS	0000000000000000	000000	000000	00	0	0	0	0
[1]	.text	PROGBITS	0000000000000000	000040	000023	00	AX	0	0	1
[2]	.rela.text	RELA	0000000000000000	000158	000060	18	I	5	1	8
[3]	.data	PROGBITS	0000000000000000	000063	000010	00	WA	0	0	1
[4]	.bss	NOBITS	0000000000000000	000073	000000	00	WA	0	0	1
[5]	.symtab	SYMTAB	0000000000000000	000078	0000c0	18		6	4	8
[6]	.strtab	STRTAB	0000000000000000	000138	00001a	00		0	0	1
[7]	.shstrtab	STRTAB	0000000000000000	0001b8	000031	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
l (large), p (processor specific)

↓
readelf -s:

(mostra tabella simboli)

Symbol table '.symtab' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	non definita
1:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1	
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3	
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4	
4:	0000000000000000	DATA	NOTYPE	GLOBAL	DEFAULT	3	var1
5:	0000000000000008	0	NOTYPE	GLOBAL	DEFAULT	3	var2
6:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	1	_start
7:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	_Z3fool

readelf -r:

(mostra tabella rilocazioni)

Relocation section '.rela.text' at offset 0x158 contains 4 entries:

Offset	Info	Symbol	Type	Sym. Value	Sym. Name	+ Addend
0000000000000007	000500000002	R_X86_64_PC32	0000000000000008	var2 - 4	non conosco di preciso offset ma lo faccio per lo scavo	qui
000000000000000f	000700000004	R_X86_64_PLT32	0000000000000000	_Z3fool - 4	e lo scrivo	dico dove inizia interazione successiva (var2 - 4 - 7)
0000000000000016	000400000002	R_X86_64_PC32	0000000000000000	var1 - 4		
000000000000001d	000400000002	R_X86_64_PC32	0000000000000000	var1 - 4		

COLLEGATORE

Quando chiamiamo il collegatore gli giuiamo la lista dei file da collegare e quindi vede il programma per intero → Comando ld lista file. In ordine di file, emme le sezioni TEXT e DATA e così via. Fatto questo ero obbligato a tenere nota dei simboli non definiti e restringere un errore se un simbolo è definito più volte in file diversi. A questo punto anche gli indirizzi alle sezioni e riceva tutti gli offset. Deve adesso gestire i simboli non definiti. Se non vengono aggiustati tutti da errore. Infine applica le rilocazioni. A questo punto il programma può eseguire: File ELF con tabella di programmi

ELF Header:

Magic:	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF64
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Advanced Micro Devices X86-64
Version:	0x1
Entry point address:	0x4000b0
Start of program headers:	64 (bytes into file)
Start of section headers:	720 (bytes into file)
Flags:	0x0
Size of this header:	64 (bytes)
Size of program headers:	56 (bytes)
Number of program headers:	2
Size of section headers:	64 (bytes)
Number of section headers:	8
Section header string table index:	7

readelf -wl:
(tavelli di progettazione)

Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x0000000	0x0000000000400000	0x0000000000400000	0x0000f5	0x0000f5	R E	0x1000
LOAD	0x0000f5	0x00000000004010f5	0x00000000004010f5	0x000020	0x000020	RW	0x1000

Section to Segment mapping:

Segment Sections...

00 .text

01 .data .bss

PERMESSI
Read
Write
Eexecute

LIBRERIE STATICHE

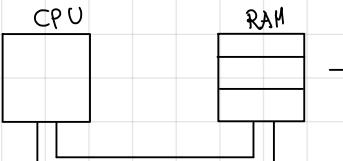
Collezioni di file oggetto → Comando ar (archive) o nmehih.a lista-fil.
↓

Vantaggio: Quando il collegatore vede una libreria prende da essa soltanto i file oggetto che contengono simboli non definiti

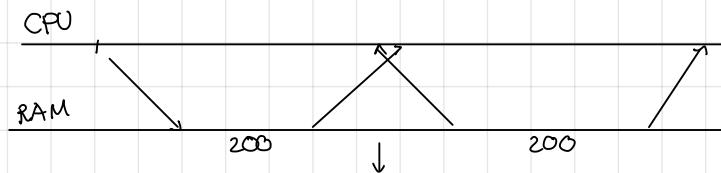
⚠ ATT: Lo fa nel punto in cui è scritta la libreria → tipicamente in fondo
ricerca libreria: Vanno nelle im posti standard (/usr/local/lib) e ai nomi devono iniziare per lib → Posso specificare solo la parte intermedia del nome preceduta da -l

include non è né necessario né sufficiente perché copia tutto il file. È infatti sufficiente la dichiarazione di simboli (ad esempio cout). Non è nemmeno sufficiente la include perché la dichiarazione viene fatta nella libreria e poi vedendo dove collegarla → Non è compito del proprio codice una del collegatore che deve collegare la libreria standard di C++. Non ce ne accorgiamo perché nel caso di g++ la libreria standard la collega sempre. Questo non accade per librerie "strane".

da cache



→ impiega 200 - 300 cicli di clock per rispondere. La CPU è invece molto più veloce e lo è diventata nel tempo.



Il tempo di esecuzione è dettato dalla velocità della RAM in questo caso.

tipologie di RAM

Venne utilizzata la tecnica della memoria dinamica che prevede di memorizzare i hit in piccoli condensatori che in realtà sono capacità parziali dei transistori. Questo permette di condurre tanti hit in poco spazio anche se la velocità di lettura e scrittura è piuttosto bassa. La costruzione invece dura poco e deve essere recarica. Attraverso poi la tecnologia della RAM statica che prevede l'utilizzo di scatole e quindi molti transistori. Queste memorie sono molto più veloci rispetto alle memorie dinamiche ma costano molto di più e consumano anche molta più energia elettrica.

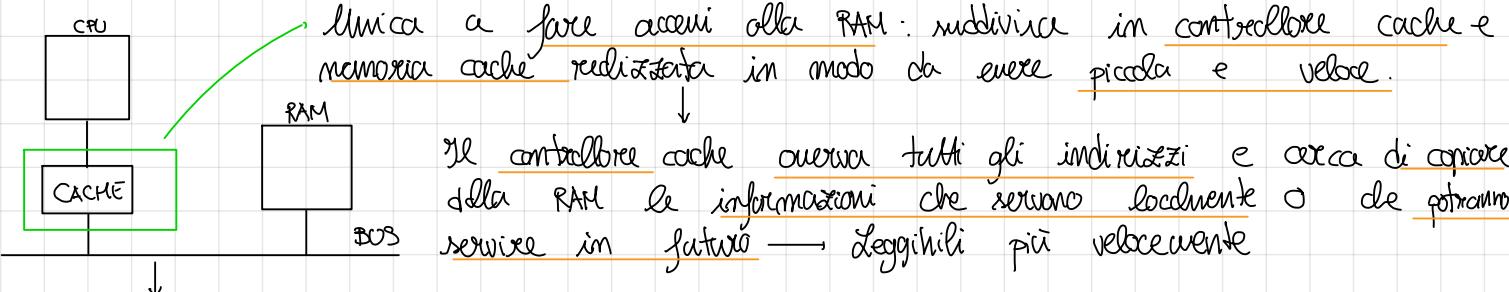
↓ Geiamo l'illusione di avere una RAM grande e veloce

Princípio di località temporale: se un programma ha fatto un'operazione di lettura o scrittura ad un certo indirizzo è molto probabile che rifiora un'operazione di lettura o scrittura a quello stesso indirizzo

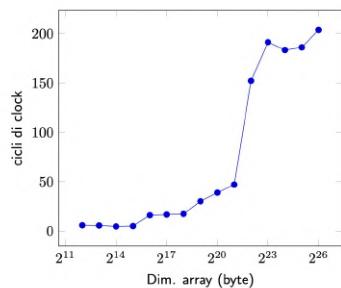
princípio di località spaziale: se un programma ha fatto un'operazione di lettura o scrittura ad un certo indirizzo è molto probabile che in breve tempo faccia operazioni di lettura e scrittura ad indirizzi vicini

Questo accade ad esempio il codice viene memorizzato in località spaziale e facendo un ciclo preleva sempre le stesse informazioni. Perché il codice nell'incremento di una variabile ed anche perché è facendo un ciclo preleva sempre le stesse informazioni del codice

↓ Modello



Soluzione hardware: fa tutto il controllore cache perché il software funziona indipendentemente dalla presenza della cache. Il controllore è quindi trasparente al software.



→ Il tempo aumenta in quanto l'array non entra più nella cache e quindi è necessario un accesso in memoria. Nel caso migliore il tempo di accesso è 4 (cache di primo livello) → Comunque fare le ottimizzazioni con la cache

realizzazione

Utilizzano il principio di località temporale conservando i dati fino al riempimento. Si pone quindi il problema del riempimento tramite il quale il controllore deve tenere i dati più utili → deve avere rimpiastata l'informazione che verrà richiesta più in avanti nel futuro → A proposito che il futuro sarà simile al passato, viene rimpiastata l'informazione che non è stata richiesta da più tempo: Least recently used.

↓
Non ha senso parlare di cache per le operazioni di ingresso ed uscita in quanto per la cache abbiamo posto che la RAM non cambia il proprio contenuto da sola. Le operazioni di lettura e scrittura in I/O hanno inoltre effetti collaterali che verrebbero azionati in modo errato da parte della cache. Questo deve quindi essere escluso in caso di operazioni in I/O. Questo è però complicato in quanto anche i registri dell'I/O prendono parte allo spazio di indirizzamento della RAM sostitutiva in memoria.

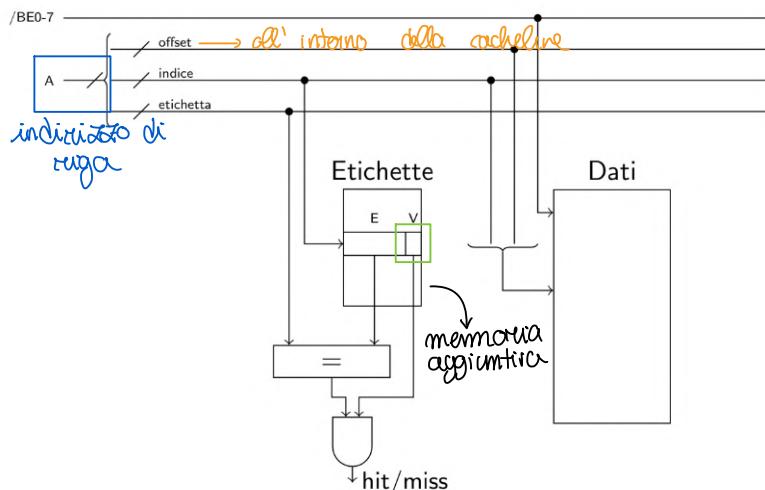
Il controllore copia l'indirizzo a cui deve essere fatta la scrittura e poi la scrittura può essere fatta soltanto sulla copia oppure sia sulla copia che sulla memoria. Si ha miss quando l'indirizzo richiesto dal processore non è presente in cache altrimenti si ha hit. Nel caso di miss la cache può adottare una politica di write allocate o write no allocate. Nel secondo caso si procede direttamente alla scrittura in RAM mentre nel primo caso, prima di fare la scrittura, viene copiato il contenuto dell'indirizzo a cui deve essere fatta la scrittura in cache. In questo caso la scrittura oppure scorrere in entrambi in cache anche se prima o poi ci dovrà essere anche la scrittura in RAM per non perdere le modifiche. Non c'è quindi un guadagno sostanziale sotto questo aspetto perché la scrittura in RAM deve comunque esserci. È conveniente quando un indirizzo viene scritto più volte perché in tal caso si ha un solo riempimento. Il secondo vantaggio si ha perché nel corso di un riempimento la CPU può fare altre operazioni.

↓ Ma come facciamo ad evitare scritture inutili?

Quando c'è una scrittura si accende un flag. Quando non c'è questo flag non c'è stata modifica e quindi non è necessaria una modifica in memoria.

A basso livello

Il controllore deve mappare velocemente un indirizzo di memoria in cache e vedere se a quell'indirizzo c'è quello che il processore sta cercando



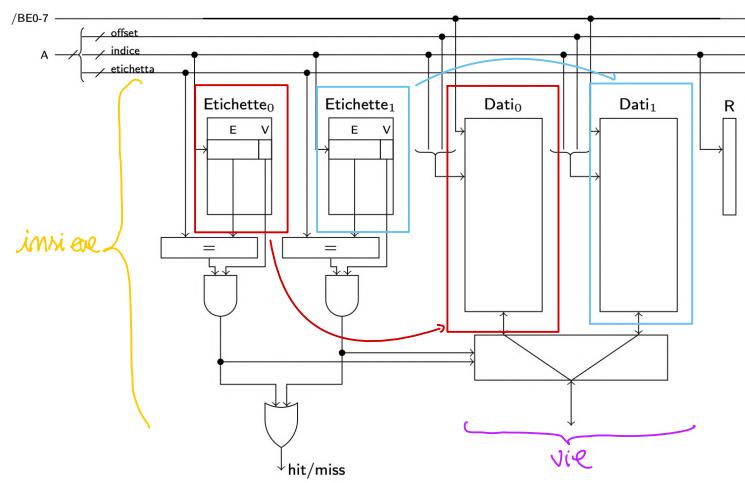
Organizzata come la memoria RAM

↓ L'indirizzo di cache viene mappato prendendo i meno significativi dell'indirizzo del processore tanti quanti ne servono per indirizzare la memoria cache

↓ indirizzamento diretto: l'indirizzo del processore fornisce direttamente l'indirizzo di cache

Ma come faccio a vedere se il contenuto è giusto? con i hit più significativi. Questa parte si chiama etichetta. Abbiamo quindi bisogno di una seconda RAM detta RAM delle etichette con la quale confronto l'etichetta data dal processore. c'è poi bisogno di sapere se l'etichetta è valida. Se entrambe le condizioni sono verificate si ha un hit. Ma perché accediamo alla memoria delle etichette solo con l'indice? (indirizzo cache - offset + indice). La memoria delle etichette è over-head e quindi questa deve essere più piccola possibile. Invece di prendere delle righe della RAM vengono prese regioni più grandi riducendo la memoria delle etichette rinchiuso però di leggere cose inutili della RAM. Possiamo poco utilizzare il secondo principio di località per avere un numero di queste effettive di unità di trasferimento non è la riga ma la cacheline (8 righe / 64 byte typ), allineata naturalmente. Per quanto riguarda l'indirizzo, i 3 bit meno significativi sono l'offset all'interno della cacheline ed il resto sono il numero di cacheline. Per sapere se il dato c'è o non c'è il processore guarda solo l'indice. La cacheline di solito è grande su byte. Quando il processore preleva una cacheline della memoria (miss) lei copre milioni di posizioni corrispondenti all'indice, indipendentemente dal fatto che ci siano altre posizioni libere. Con la memorizzazione detta, tutte le cacheline con lo stesso indice devono essere nella stessa posizione della memoria perché è utilizzato solo l'indice per effettuare l'accesso. → **PROBLEMA:** Da cache-line in posizione e posizione + dim. cache hanno lo stesso indice →

È possibile fare corrispondere più dati allo stesso indice

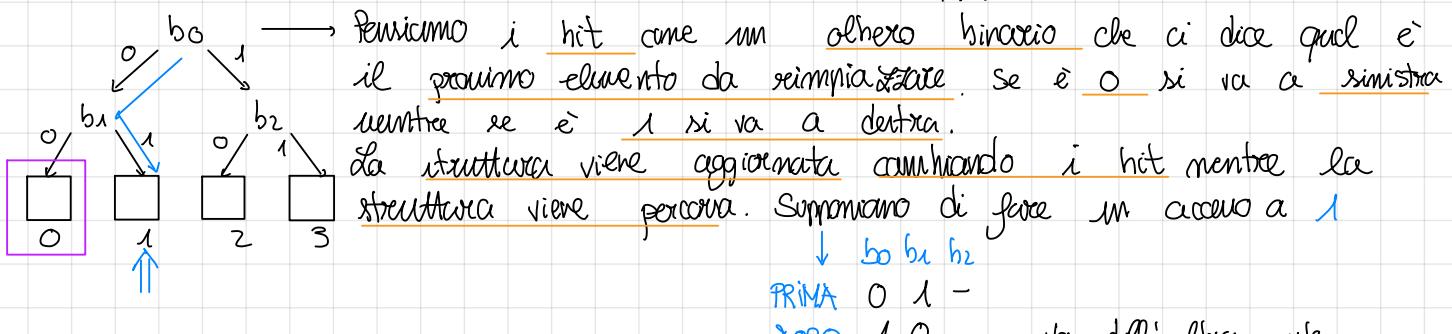


→ Mettiamo in parallelo più memorie etichette e dati. L'indirizzo del processore viene ricomposto allo stesso modo dell'indirizzamento diretto e la ricerca dell'etichetta viene fatta parallelamente in entrambe le memorie etichette. Questa viene trovata se sta almeno in una delle due memorie etichette → non può stare in entrambe perché il controllore non la cerca più volte.

Una volta trovata viene prelevata dalla memoria dati corretta

⚠ Poco bene la convenienza di cacheline con lo stesso indice

Può essere estesa anche a più vie di cui una regola è un inviere. Cioè un indice adesso identifica un inviato. Il controllore adesso può anche scegliere cosa rimpiazzare. Con 2 vie abbiamo un'ulteriore memoria di indice quel è quella in cui è stato fatto l'ultimo accesso. Con 6 vie utilizzano un algoritmo che utilizza 3 bit, chiamato pseudo-LRU b0, b1, b2



L'aggiornamento si può fare con una scrittura di una costante fino a seconda della scorsa che ritorna rimpiantando. Questo algoritmo schiglia in quanto viene riceve un "bonus" andando in cima alla lista per non avendo ricevuto alcuna scrittura ↓

Troppo eserci entro a più vie aggiungendo più candidati per non farlo diventare troppo costoso

comando lscpu --cache

NAME	ONE-SIZE	ALL-SIZE	WAYS	TYPE	LEVEL
L1d : dati	32K	128K	8	Data	1
L1i : istruzioni	32K	128K	8	Instruction	1
L2	256K	1M	8	Unified	2
L3	8M	8M	16	Unified	3

Periferiche

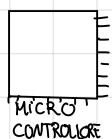
• LIMITAZIONI DEL SISTEMA OPERATIVO

Non possiamo ricevere sempre programmi che accedono a periferiche. I programmi sono infatti soggetti a limitazioni date dal sistema operativo e NON dal processore. Non vengono forniti cicli infiniti e non vengono fornite le istruzioni I/O. Non è inoltre possibile accedere in memoria ad un indirizzo casuale non compreso nella tabella di caricamento del programma → In questo caso dobbiamo guardare i permessi di RWE. Vengono utilizzati i meccanismi di interazione, protezione e memoria virtuale per limitare le funzionalità del programma.

↓

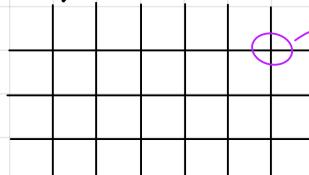
Utilizziamo una macchina virtuale che ci elimina tali meccanismi in modo da poter vedere il funzionamento delle periferiche → Macchina QEMU con un vantaggio: può caricare nella memoria del programma un eseguibile creato dalla macchina ospite. Allo stesso tempo inoltre ci permette di semplificare la gestione initializzando il processore e permettendo di ricevere input da tastiera e far vedere cose a video. Non è possibile utilizzare la libreria standard ed in particolare ioctl() in quanto richiede un kernel Linux per fare determinate azioni. Lo script compile permette di utilizzare ioctl() e dice dove collegare i file.

• TASTIERA



→ Piccolo chip che funziona come un computer: ha un processore, una RAM, una ROM e può pilotare dei piedini utilizzandoli come I/O.

↓ sotto: tasti



Collegamenti elettrici collegati ai piedini del microcontrollore collettati elettricamente alla riga di un tasto. Il microcontrollore invia un impulso ad ogni colonna ciclicamente molto velocemente e poi legge tutte le sue righe dai piedini. Se un tasto è premuto l'impulso non torna indietro altrimenti capisce che è premuto il tasto all'interruzione tra riga e colonna.

↓

Il microcontrollore invia un codice a ciascun tasto e poi invia l'informazione sul PC che dialoga con il bus attraverso il registro RBR che contiene il codice del tasto premuto (codice di scansione). Il microcontrollore si ricorda inoltre i tasti che sono premuti e quando un tasto viene rilasciato invia un altro codice che segnala il rilascio del tasto. Abbiamo quindi 2 codici per ciascun tasto che nel caso delle tastiere compatibili IBM si differenziano per il bit più significativo. Supponendo che il tasto è stato rilasciato ci permette di continuare a ricevere lo stesso carattere più volte sul computer e di utilizzare le combinazioni di tasti. La comunicazione è inoltre bidirezionale e quindi è presente un TBR. Questo permette di utilizzare i LED e ad impostare alcuni parametri → è molto più complicato di quanto si pensi perché è implementata come nell'"hardware".

↓

Il microcontrollore sul computer non è più finora presente ma è emulato da un altro processore di hardware che emula anche altre cose. I PC IBM inoltre dovevano avere pochi pin per essere economici → la interfaccia utilizzano l'indirizzo interno attraverso un unico piedino di indirizzo. Per denominare tra i 4 registri si utilizza quindi il tipo di accesso (R/W). I registri sono visti accoppiati (RTSR, RTBR) agli indirizzi 60 e 64.

ESEMPIO

```
#include <libce.h>
const ioaddr iSTR = 0x64;
const ioaddr iRBR = 0x60;
```

```
natb get_code()
{
```

```
    natb c;
    do
        c = inputb(iSTR);
    while (!(c & 0x01));
```

return inputb(iRBR);

Vogliamo leggere i codici di scansione
e mostrarli a video

ci serve per vedere il flag
Fi (vedi commenti rati)

```
int main() : chiamato da start
{
    natb c;
    for (;;) {
        c = get_code(); → codice di scansione di esc
        if (c == 0x01) →
            break; // make code di ESC
        for (int i = 0; i < 8; i++) {
            if (c & 0x80) → bit piu' significativo di c
                char_write('1'); → stampa a video
            else
                char_write('0');
            c <<= 1;
        }
        char_write('\n');
    }
}
```

La cache non rispetta
compatibilità: leggerà
sempre il primo valore

```
.text
.global inputb
inputb:
.cfi_startproc
movw %di, %dx
inb %dx, %al
ret indirizzo > 256
.cfi_endproc
```

ESEMPIO: eco da tastiera

```
#include <libce.h>
```

```
namespace kbd {
```

```
const natl MAX_CODE = 29;
bool shift = false; → Si ricorda se shift è premuto
natb tab[MAX_CODE] = { // tasti lettere (26), spazio, enter, esc
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
    0x1E, 0x1F, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26,
    0x2C, 0x2D, 0x2E, 0x2F, 0x30, 0x31, 0x32, 0x39, 0x1C, 0x01
};
```

```
natb tabmin[MAX_CODE] = {
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p',
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l',
    'z', 'x', 'c', 'v', 'b', 'n', 'm', ' ', '\n', 0x1B
};
```

```
natb tabmai[MAX_CODE] = {
    'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P',
    'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L',
    'Z', 'X', 'C', 'V', 'B', 'N', 'M', ' ', '\r', 0x1B
};
```

```
char charread()
```

```
{
```

```
    natb c;
    char a;
```

```
    do {
```

```
        c = get_code(); → codice premere shift
```

```
        if (c == 0x2A) // left shift make code
```

```
            shift = true;
```

```
        else if (c == 0xAA) // left shift break code
```

```
            break; → shift = false;
```

```
    } while ((c >= 0x80) || c == 0x2A); // make code;
```

```
    a = conv(c); // conv() puo' restituire 0
```

```
    return a; // 0 se tasto non riconosciuto
```

→ Associai al codice di scansione quando shift non è premuto

→ Associai quando shift è premuto

→ Restituisce il codice ASCII corrispondente al codice di scansione

```

char conv(natb c)
{
    natb cc;
    natl pos = 0;

    while (pos < MAX_CODE && tab[pos] != c)
        pos++;
    if (pos == MAX_CODE)
        return 0; → Cerco nella
    if (shift) tabella
        cc = tabmai[pos]; } Utilizzo
    else cc = tabmin[pos]; } tabella conv.
    return cc;
}

```

```

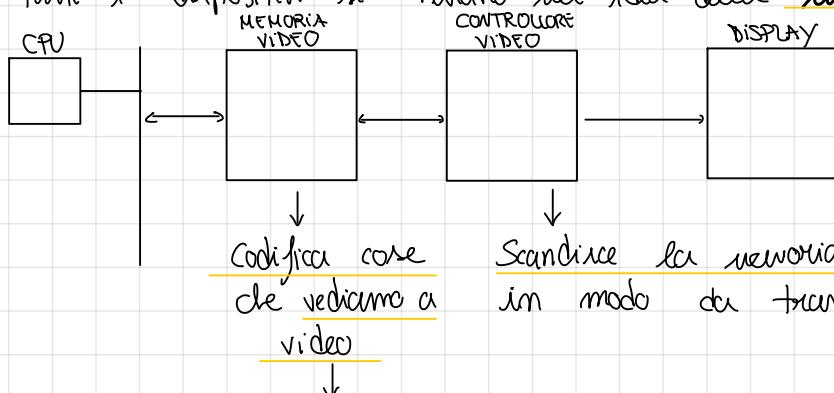
int main()
{
    char c;

    for (;;) {
        c = char_read(); // puo' restituire 0
        if (c == 0x1B) // carattere ASCII esc
            break;
        char_write(c); // non effettua azioni se c vale 0
    }
    return 0; → fa eco
}

```

• VIDEO

Tutti i dispositivi si basano sull'idea della memoria video



Codifica core che vediamo a video

Scandisce la memoria video ed interpreta il contenuto in modo da trasformarlo in segnali per il display.

È presente all'interno dello spazio di indirizzamento sotto forma di indirizzi dove il suo può scrivere → a. MONO TESTO: la memoria video contiene i codici ASCII + altre informazioni

b. MONO GRAFICO: vengono codificati i colori dei singoli pixel nel display
Come distinguere ↓

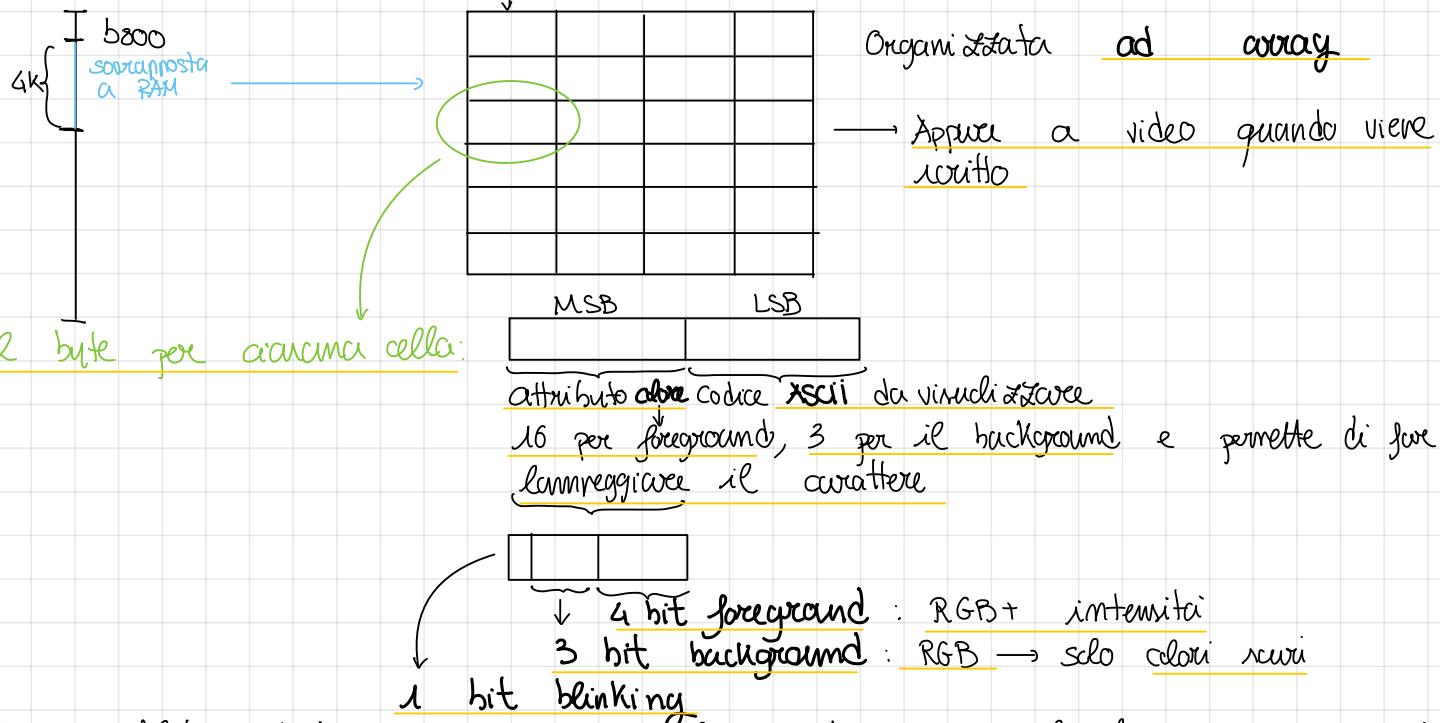
Nel modo testo abbiamo il numero di colonne e righe che vogliamo che vengano visualizzate. Nel modo grafico abbiamo risoluzione e profondità (numero di colori per pixel)

→ Utilizzano gli standard VGA e SVGA

Il controllore video è collegato nello spazio di I/O con alcuni registeri, alcuni per sapere dove far apparire il cuore in modalità testo. Da fuori si vedono solo due registeri mentre agli altri si accede indirettamente. Si scrive in uno dei due registeri il numero del registro interno con il quale vogliamo parlare ed il secondo diventa una "finestra" sul registro relazionato.

dispositivi compatibili VGA

Partono in modalità testo. Per far vedere qualcosa a schermo basta sapere dove si trova la memoria video e cosa scrivere nelle celle.



In modalità testo c'è una ROM che contiene i pixel che vanno accesi a seconda del carattere

ESEMPIO: riempimento memoria video

```
#include <libce.h>

namespace vid {
    volatile natw* video = reinterpret_cast<natw*>(0xb8000); // array di 2000
    word; // array di unsigned short
} // inizio memoria video

using namespace vid;

int main()
{
    for(int i = 0; i < 2000; i++)
        video[i] = 0x4B00 | 'A'; // sfondo rosso (0100), simbolo
    // azzurro chiaro (1011) giallo foreground codice ASCII di A
    for (;;) {
        char c = char_read();
        if (c == 0x1B)
            break; // carattere ASCII esc
    }
    return 0; // termina programma con ESC
}
```

Tradotti con delle MOV
vedi codici RGB

azurro chiaro (1011) giallo foreground codice ASCII di A

return 0; termina programma con ESC

↓ OUTPUT



Ma come facciamo a far muovere il cursore?

```
void char_write(natb c)
{
    switch (c) {
        case 0: → Non fa nulla
            break;
        case '\n': case '\r':
            x = 0;
            y++;
            if (y >= ROWS)
                scroll();
            break;
        default:
            video[y * COLS + x] = attr | c; → Posizione matrice [x][y]
            ← x++; → attributo corrente
            if (x >= COLS) { } → colore
                x = 0; } → ultima colonna
                y++; } → avanzata x e poi y a capo
            if (y >= ROWS)
                scroll(); → riportare endline
            break;
    }
    cursore();
}

void scroll()
{
    for (int i = 0; i < VIDEO_SIZE - COLS; i++)
        video[i] = video[i + COLS]; → copia in pos.
    for (int i = 0; i < COLS; i++)
        video[VIDEO_SIZE - COLS + i] = attr | ' '; → quello che c'è in
} y--; → ultima riga riempie ultima riga con spazi
```

politura alberino

```
void clear_screen(natb col)
{
    attr = static_cast<natw>(col) << 8;
    for (int i = 0; i < VIDEO_SIZE; i++)
        video[i] = attr | ' ';
    x = 0;
    y = 0;
    cursore();
}
```

posizione cursore

```
using namespace vid;

void cursore()
{
    natw pos = COLS * y + x; → 2 byte
    scritto in 2 registri { outputb(CUR_HIGH, IND);
    outputb(pos >> 0x8, DAT); ↓
    outputb(CUR_LOW, IND); ↓
    outputb(pos, DAT); } finestra su CUR HIGH
}
```

back space

```
case '\b':  
    if (x > 0) {  
        x--;  
    } else if (y > 0) {  
        y--;  
        x = COLS - 1; // cancella il  
                      // vecchio carattere  
    }  
    video[y * COLS + x] = attr | ' ';  
    break;
```

modalità grafica (SVGA)

La memoria diventa più grande pur essendo sempre organizzata a matrice e ciascun elemento della matrice è il colore di un pixel

⚠ È complicato passare in modalità grafica → Le schede video lavorano con una ROM per impostare le modalità grafiche

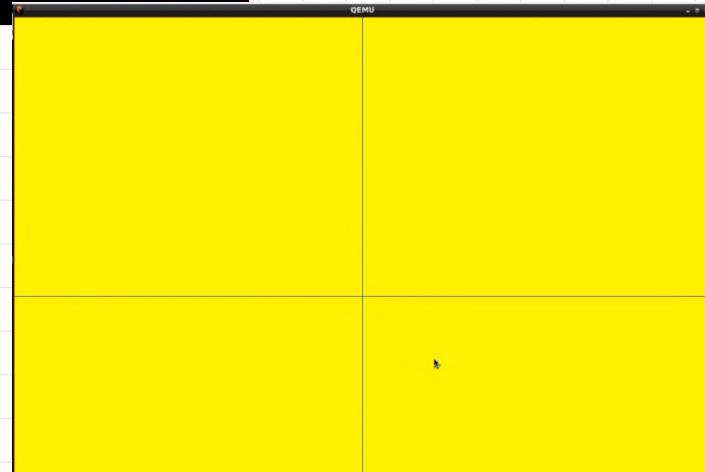
↓ Emulazione

```
#include <libce.h>  
  
const natl COLS = 1280;  
const natl ROWS = 1024;  
  
volatile natb* framebuffer;  
  
int main()  
{  
    framebuffer = bochsvga_config(COLS,ROWS);  
  
    for (int i=0; i<COLS; i++)  
        for (int j=0; j<ROWS; j++)  
            framebuffer[j*COLS + i] = 0x04; // rosso scuro (vga 0100)  
  
    char c;  
    do  
        c = char_read();  
    while (c != 0x1B); // carattere ASCII esc  
    return 0;  
}
```

→ La memoria grafica è una matrice di colori

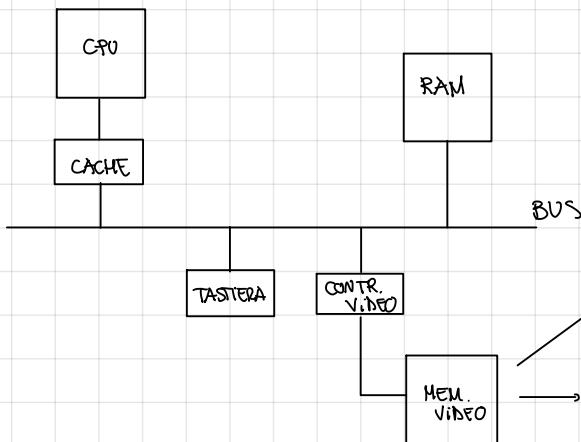
ESEMPIO

```
void rettangolo(natb c)  
{  
    for (int i = 0; i < COLS; i++)  
        framebuffer[i] = c; // linea in alto  
    for (int j = 0; j < ROWS; j++)  
        framebuffer[COLS * j + (COLS - 1)] = c; // linea a destra  
    for (int i = 0; i < COLS; i++)  
        framebuffer[COLS * (ROWS - 1) + i] = c; // linea in basso  
    for (int j = 0; j < ROWS; j++)  
        framebuffer[COLS * j] = c; // linea a sinistra  
}  
void assi(natb c)  
{  
    for (int i = 0; i < COLS; i++)  
        framebuffer[COLS * y0 + i] = c; // asse cartesiano x  
    for (int j = 0; j < ROWS; j++)  
        framebuffer[COLS * j + x0] = c; // asse cartesiano y  
}
```



ATT: gli indirizzi partono dall'otto a xx e y va in mano

ATT2: Il controllore cache da l'autoriso solo se utilizza la politica write back in quanto potrebbe non aggiornare la memoria video



Nello spazio di memoria: gli accini rimano

Fuò anche essere rotta: in tal caso il controllore cache ha senso perché essa non cambia da sola

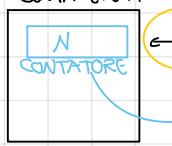
• TIMER

Servono per poter avere una misura del tempo che funca. Un tempo si poteva usare il numero di istruzioni perché esse venivano eseguite in un tempo prestabilito. Non è più vero nei processori moderni e pensando da una generazione ed un'altra di processore

↓ potrei pensare, per fare passare tempo

for (int i=0; i < 1000000; i++) ma non posso farlo perché non ho un tempo di esecuzione fino → il processore esegue le operazioni in parallelo ed il compilatore ottimizza (in questo caso toglie il ciclo perché non fa niente)

Allora ho bisogno di qualche che misura il tempo → interfaccia di conteggio che contiene gli eventi:



Pulsino di ingresso: se periodico contiene il tempo
Registro: viene decrementato a gestire del trigger
che può essere hardware o software

Può generare impulsi in determinati eventi oppure onde quadre e può anche riportare dal valore iniziale quando il registro arriva a 0 → segnale periodico

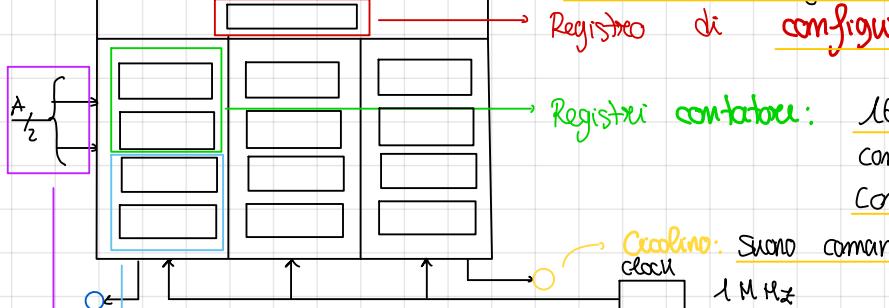
↓ Nei PC IBM

0 1 2 → 3 contatori configurabili

Registro di configurazione:

da ordine di memoria effettuare

o può scegliere cosa fare
tutti (3 per registro) che
potrà attivare la porta bus del



Registri contatore: 16 bit

contengono

parte alta

e parte bassa del

contatore

Cicloni: sono comandati con onde quadre

Controllore delle interruzioni: serve per leggere il valore corrente del contatore perché se il programma

legge gli altri questi potranno varicare nel mentre

indirizzamento: Un indirizzo è riservato al registro come e si dà un indirizzo a ciascun contatore. Questi partono da 0. Per indirizzare i 4 registri di ciascun contatore si utilizza il trucco della tastiera avendo 2 registri nei quali è

possibile scrivere e gli altri in cui è possibile soltanto leggere. Infine, per fare la scrittura nei 2 registri contatore, prima viene scritta la parte bassa e poi la parte alta → lo fa automaticamente il chip

```
#include <libce.h>
namespace tim {
    const ioaddr iSPR = 0x61; → Controlla ciclino
    const ioaddr iCWR = 0x43; → Control word register
    const ioaddr iCTR2_LSB = 0x42; → Registrì contatore 2
    const ioaddr iCTR2_MSB = 0x42; ↓
                                Problema con cache write back
};

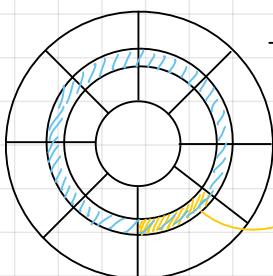
using namespace tim;

void avvia_timer(natw N) → Il veloce serve per dire che vogliono configurare il contatore 2
{
    outputb(0xB6, iCWR);           // contatore 2, modo 3
    outputb(N, iCTR2_LSB);         ↓ genera onda
    outputb(N >> 8, iCTR2_MSB);   ↓ quadra
}

int main()
{
    avvia_timer(190); → da qui in poi si sente il nome          // costante 1000 Hz
    outputb(3, iSPR);
    pause(); → Registro collegato a ciclino
    outputb(0, iSPR); → che permette di silenziarlo, contano i due LSB. Uno è collegato al circuito del timer e ferma il contatore e l'altro è posto in porta AND con il ciclino → Per farlo funzionare si deve mettere 1
    return 0;
}
```

o HARD DISK

Ci occupiamo degli hard disk meccanici (con i dischi che girano) → Soluzione più economica per memorizzare grandi quantità di dati



→ Dischi che contengono materiale ferromagnetico organizzato in tracce e settori → parti di dati memorizzati su una traccia

SETTORE: 512 byte → Dati memorizzati cambiando la polarizzazione del materiale ferromagnetico

Unità di trasferimento: può ricevere / trasferire un intero settore

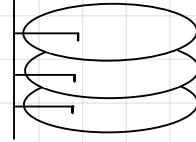
→ programmi vengono eseguiti dalla RAM mentre l'hard disk serve solo per la memorizzazione

→ una periferica → non è possibile accedervi direttamente da un programma ma con un'interfaccia

Vi sono 2/3 dischi memorizzati in entrambe le facce con delle testine montate sullo stesso braccio che ponono reettore come un giradischi →

Le testine si muovono tutte insieme sulle facce dei dischi che sono tenuti in rotazione costante da 5000 a 5000 rpm.

Per leggere / scrivere un settore si dà spazio in quale faccia si trova il settore



e quindi con quale testina leggere, la traccia dove si trova per ruotare il braccio delle testine ed infine il setore → numerati negli header

↓

Il controllore fa partire l'operazione che comporta un seek + latency + accesso di quanto deve ruotare il braccio delle testine per posizionarle sulla traccia richiesta

↓

Dipende da dove era il braccio prima: millisecondi

dopo che il setore giri sotto la testina

↓

MS: dipende

Ora facendo rotazione il setore, dipende dalla grandezza delle rotazioni al minuto per giro del setore completo: millisecondi

↓

moltò più lento per aumento del processo

migliorato

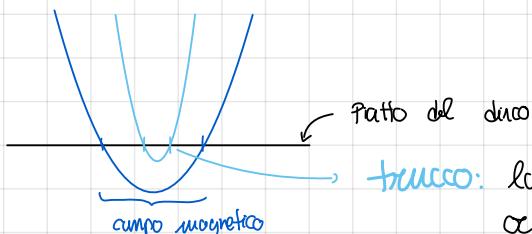
per aumento

capacità

tempi troppo lunghi: depositivi imbarazzante spazio per accedere caricate poi disertate sequenziali per la convenienza in termini di tempo

più dati in uno spazio sempre più ristretto?

Ma come facciamo a memorizzare



trucco: la testina viene avvicinata di più in modo da coprire zone più piccole

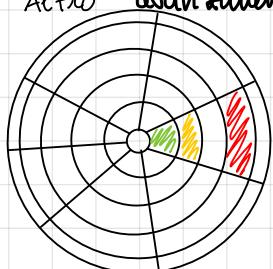
↓ come?

La testina si tiene sollevata dal disco e non tocca mai il disco

↓

Il tempo di accesso migliora: è l'unico tempo abboccato perché è sufficiente memorizzare in sequenza per annullare gli effetti dei tempi

Altro avanzamento tecnologico



→ c'è processo di radice: i settori interni sono più piccoli di quelli esterni ma contengono sempre la stessa informazione

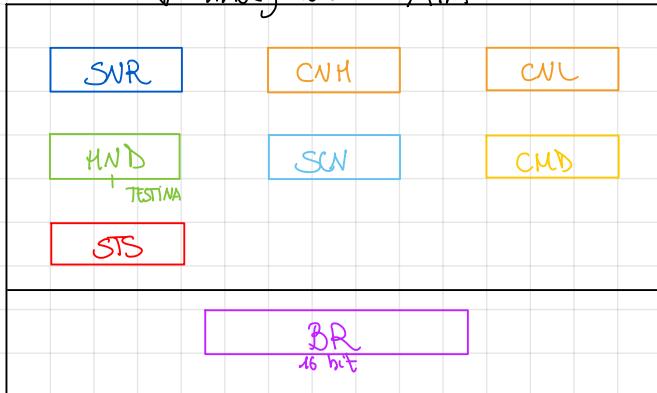
↓

Il disco viene suddiviso in zone in modo che i settori abbraccino tutti la stessa dimensione aumentando il numero di settori verso l'esterno

↓ In questo tempo di lettura e scrittura

la velocità per leggere/norare un settore dall'esterno è maggiore di quella impiegata per settori all'interno ($v = w \cdot r$). → I SO riempiono gli hard disk partendo dall'esterno e quindi rallentano quando il disco si riempie

Sul punto di vista del programmatore
 ↳ uttori hanno solo un'indicazione logica (numerati da 0 al massimo)
 ↓ interfaccia ATA



SNR (sector number): indica il numero del settore sulla traccia

CNLH (cylinder number): cilindro = insieme delle tracce su cui sull'oltre

SCN (sector count number): permette di trasferire più di un settore

CHD (comando): Specifica l'azione da fare

STS (status)

BR: permette di leggere/scrivere dati

Per fare una lettura: scrivo le coordinate del settore ed il comando di lettura. A questo punto il controllore fa tutte le azioni necessarie e mette il contenuto del settore in una memoria interna e fa uscire tramite il bit data ready in STS che il dato è presente. A questo punto il software deve fare 256 letture di BR che restituisce ogni volta 16 bit successivi del buffer.

Per fare una scrittura: richiamo direi in quale settore scrivere, dove il comando di scrittura e scrivere tutti i 512 byte nel buffer interno facendo 256 scritture in BR. A questo punto il controllore fa la scrittura

Adesso le coordinate sono costituite dal logical block address (numero logico del settore) → in questo modo un hard disk può avere grande al massimo 120 GB ($2^{30} \cdot 2^9 = 2^{37}$ byte): modificando con la stessa tecnica di accesso del timer

ESEMPIO

```
void hd_set_lba(natl lba)
{
    natb lba_0 = lba,           ↴ logical block address
    lba_1 = lba >> 8,
    lba_2 = lba >> 16,
    lba_3 = lba >> 24;

    outputb(lba_0, iSNR);
    outputb(lba_1, iCNL);
    outputb(lba_2, iCNH);
    natb hnd = (lba_3 & 0x0F) | 0xE0; natb buff[8*512];
    outputb(hnd, iHND);       ↴ altre funzioni
                                ↴ (abilità logica
                                ↴ numbering)
}
```

```
void hd_start_cmd(natl lba, natb quanti, natb cmd)
{
    hd_set_lba(lba);
    outputb(quanti, iSCR);
    outputb(cmd, iCMD);      ↴ numero di set
                                ↴ comando di sc
}
```

```
natl lba = 1;
natb quanti = 2;           ↴ quanti settori
for (int i = 0; i < quanti*512; i++)
    buff[i] = 'f';          ↴ buffer con dati
                            ↴ comando scrittura
                            ↴ hd_start_cmd(lba, quanti, WRITE_SECT);
    for (int i = 0; i < quanti; i++)
        hd_output_sect(&buff[i * 512]);
```

```
str_write("OK\n");
pause();
// ...
return 0;
```

```
void hd_wait_for_br()
{
    natb s;
    do
        s = inputbw(iSTS); // attende che HD sia pronto
    while (s & 0x88 != 0x08);

    void hd_output_sect(natb vett[])
    {
        hd_wait_for_br(); // attende in 256 volte
        outputbw(reinterpret_cast<natw*>(vett), 256, iBR);
    }
}

// scrive una sequenza di parole in una porta di I/O
.global outputbw
outputbw:
    .cfi_startproc
    movq %rsi, %rcx
    movq %rdi, %rsi
    cld
    rep
    outsw
    ret
    .cfi_endproc
```

Le interruzioni

o APIC per interruzioni singole

Ahhiamo visto in precedenza che vi sono alcuni casi in cui la lettura della tastiera o la lettura / scrittura dell'hard disk che richiedono cicli di attesa per avere alcuni bit nel valore desiderato. Questo accade perché abbiamo un'unica CPU con un unico flusso di controllo. Questi bit vanno riletti ogni volta per capire il loro valore attuale.



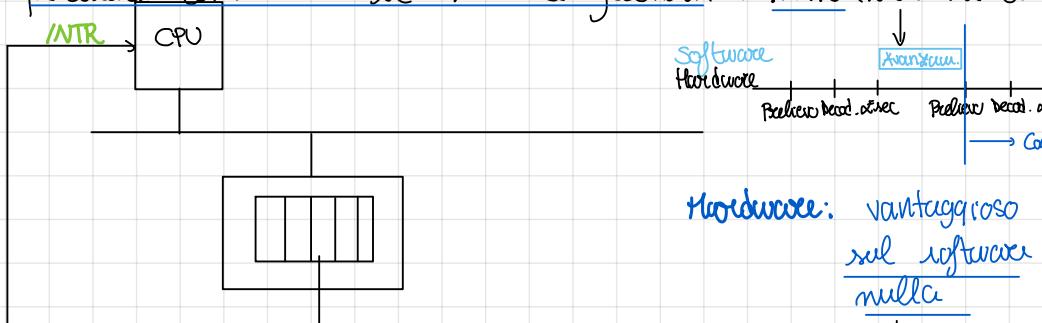
Le interruzioni nascono negli anni '60 quando abbiamo una telerivente. Per collegarla al computer è facile stampare i caratteri da computer chiamando porte degli elettromagneti sotto ai tasti. Nel computer è sufficiente avere un registro con il codice del tasto da premere ed un decoder collegato a ciascun elettromagnete. Per ricevere una nuova lettera il programma deve attendere che l'operazione precedente sia terminata. È necessario un feedback per sapere che l'operazione è finita. Sappiamo invece di voler ricevere un'applicazione che prenderà una tabella

X	f(x)
0.000	vol 0
0.001	vol 1
0.002	vol 3
...	...

→ L'applicazione deve ric算olare periodicamente i valori della funzione e poi eventualmente stampare il valore nel dispositivo precedente. Questo significa stampare un carattere per volta e attendere il termine dell'operazione. Vorremo, mentre si stampa, iniziare i calcoli per la riga successiva



Modifichiamo il programma e le periferiche in modo da portare il hit di feedback dei dispositivi al processore direttamente. Inoltre, il programma nel tempo fa il ciclo relitto di fetch-execute. Tra la fine dell'esecuzione dell'istruzione ed il prelievo della successiva il processore controlla il hit di feedback: INTR (richiesta di interruzione).



Hardware: vantaggioso perché il test non grava sul software e mem costi quindi nulla



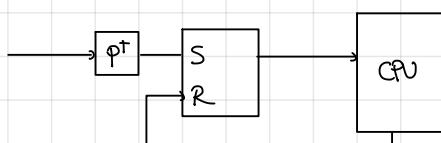
È il programma che gestisce il caso in cui INTR=1 e commissi cosa fare al processore (salvo). → L'indirizzo della routine è di default e quella zona di RAM deve contenere l'azione da fare in caso di interruzione. Inoltre, è necessario inserire un'altra routine per fornire al programma principale. Questa, nei processori Intel, comincia nel salvataggio in pillole di IP. Al termine dell'interruzione torna quindi sufficiente fare una POP. → Gestore asimmetrico: la routine di interruzione deve sapere di dover restituire il controllo al programma principale nel minor tempo possibile.

⚠ Il stato della macchina può essere determinato dalle routine di interruzione.

↓ Necesarie alcune correzioni

- a. La CPU NON deve vedere la stessa interruzione più volte

↓ Metodo



Il P+ permette di settare il latch SR in caso di interruzione e questo viene rese fatto dopo la sua esecuzione

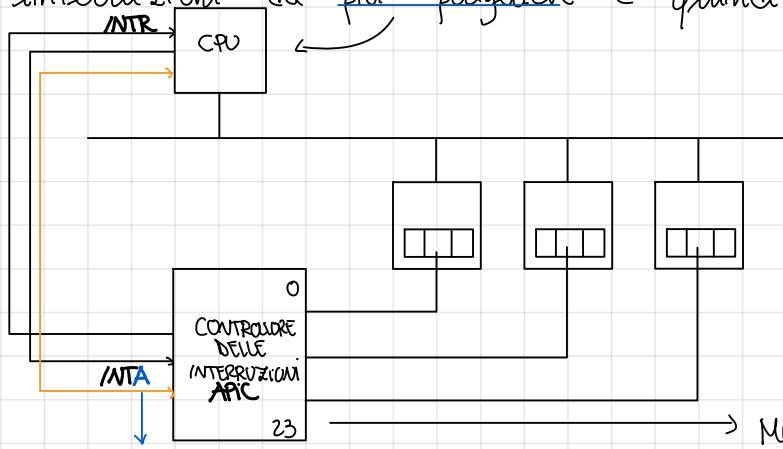
b. Le richieste di interruzione NON fanno utilizzo dei registri senza salvare e riportarli → i compilatori C/C++ ignorano le interruzioni e quindi le utilizzano come funzioni standard. Si utilizza quindi l'Assembly:

a - routine:
 // soluce - registri
 call i_routine
 // carica - registri
 RET

Il programmatore può stabilire di non permettere interruzioni in alcune parti del programma. Per permettere questo, il programma utilizza il bit 9 del registro dei flag (FLAGS). Se è = 1 le interruzioni sono ammesse, altrimenti non lo sono. Il flag può avere uno di 0 con CLI e ad 1 con STI. Dopo aver accettato la richiesta di interruzione inoltre il programma azzerà il flag dopo aver salvato il valore del flag in pila. → IRETQ

Preluce IP ed il valore del registro del flag. Q si usa per motivi di compatibilità. Senza Q viene usata la IRET a 32 bit.

Limitazione: unica periferica → In situazioni normali però è necessario accettare interruzioni da più periferiche e quindi da più interfacce.



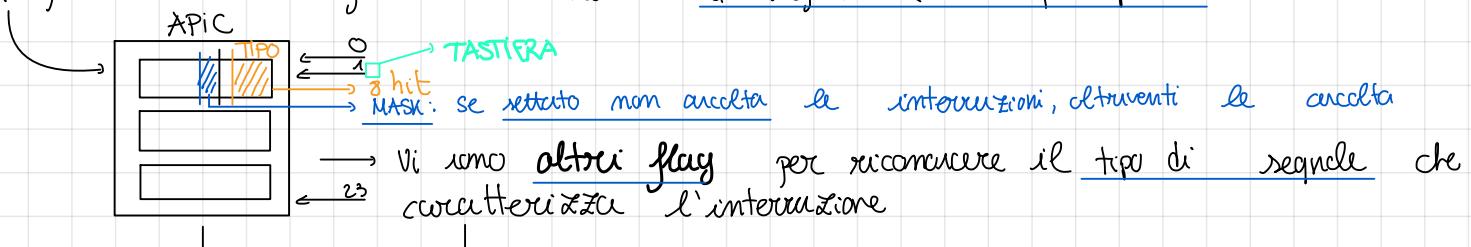
acknowledge: permette l'handshake

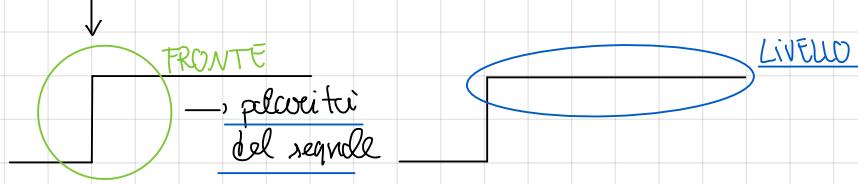
Monitora lo stato dei piedini di ingresso e puoi le richieste una per volta alla CPU: 24 piedini che vengono costantemente controllati

Le interruzioni sono "richieste di attenzione" da parte dei dinossiti e si caratterizzano per essere renestate → APIC memorizza la richiesta in un registro interno, le prioritizza e le invia al processore. Il controllore associa un numero a ciascuna interruzione, detta tipo dell'interruzione, che viene passato alla CPU attraverso un bus dedicato. Il processore dopo aver ricevuto la richiesta consulta la IDT (interrupt descriptor table). L'indirizzo della tabella si trova in un registro del processore scelto dal programmatore dopo aver compilato la tabella. Questa ha un'entrata per ogni tipo di interruzione. Per ogni entrata abbiamo l'indirizzo a cui saltare → intervallazione vettoriale. IDT ha inoltre un flag per sapere se abilitare o no le interruzioni.

Ma come fa APIC ad associare un tipo ad ogni piedino?

Programmabile via software: Vi sono 24 registri (uno per piedino)





ed anche per determinare il livello attivo, che può essere \emptyset o 1

Il registro end of interrupt permette di gestire l'hardware con il processore che comunica ad APIC che l'interruzione è stata gestita. Questo permette di gestire le interruzioni di livello.

ESEMPIO

```
int main()
{
    char spinner[] = { '|', '/', '-', '\\' };

    clear_screen(0x0f); → tastiera a predino 1
    apic_set_VECT(1, KBD_VECT); → APIC
    gate_init(KBD_VECT, tastiera);
    apic_set_TRGM(1, false);
    apic_set_MIRQ(1, false);
    enable_intr_kbd();

    natq spinpos = 0;
    while (!fine) { → buanca che ruota
        video[12*80+40] = attr | spinner[spinpos % 4];
        spinpos++;
    }
    return 0;
}
```

stabilito tipo interruzione →

const natb KBD_VECT = 0x40;

Errori: il programma non risponde più di un tasto perché non comuniciamo la fine dell'interruzione ad APIC e manca lo IRETQ poiché manca in C++. Non viene quindi aggiornato IF

d'APIC ha anche 2 registri a 256 bit: ISR e IRR. Quando c'è una richiesta riconosce il tipo e mette ad 1 il bit corrispondente al tipo di IRR: quelle non ancora inselbrate al processore. ISR ha che NON hanno ancora ricevuto un tasto e sono state riconosciute una nuova richiesta dovranno attendere end of interrupt per quelle di livello

↓ Nel debug (programma già visto precedentemente)

intr: abilitati : il bit a di DEFALUS è abilitato

Scrivendo apic posso uno vedere il suo stato:

ISR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000
IRR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000

16 bit per volta

apic_set scrive in memoria in quanto questo ha i registri mappati in memoria

→ [1]: polarity=high mode=level vector=40 (masked)
ISR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000
IRR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000

gate_init scrive in entrata della interrupt table → (gdb) idt 0x40

20017f (:tastiera)

apic_set : 2 funzioni → TRGM: cambia modalità riconoscimento

MIRQ: permette di sincronizzare interruzioni

enable_intr_kbd(); → Abilita interruzioni: tipicamente recensio solo 3 volte. Devono infatti essere abilitate nell'APIC, nell'interfaccia e nel processore

```

using namespace vid;
using namespace kbd;

void enable_intr_kbd()
{
    outputb(0x60, iCMR);
    outputb(0x61, iTBR);
}

void disable_intr_kbd()

```

} in questo modo la tastiera manda le interruzioni

Quando si riceve un'interruzione:

```
(gdb) apic
[1]: polarity=high mode=edge vector=40
ISR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000
IRR: 0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000:0000
bit 40: interruzioni da tastiera
```

+
interrupt dinabilitati del programma

+
indirizzo di ritorno in pila e valore dei registri

Alla fine per tornare al programma principale viene fatta una RET

ATTENZIONE: non viene ripristinato il valore dei registri dei flag e quindi se interruzioni restano dinabilite → non hanno avere IRETQ e poi fare quanto facciamo dell'assembly raggiungendo soltanto le istruzioni necessarie a far funzionare l'interruzione

Filo aggiunto:

```
.global a_tastiera
a_tastiera:
    call c_tastiera
    iretq
```

Funzione c_tastiera:

```
extern "C" void a_tastiera();
extern "C" void c_tastiera()
{
    natb c = inputb(iRBR);
    if (c == 0x01) // make code di ESC
        fine = true;
    for (int i = 0; i < 8; i++) {
        char write('0' + !(c & 0x80));
        c <<= 1;
    }
    char_write('\n');
```

All'uscita adesso gli interrupt vengono di nuovo abilitati correttamente
abbiamo poi dire all'APIC che l'interruzione è stata gestita

→ aggiungo apic send EOI(); alla funzione c_tastiera

Notiamo adesso che l'azione "torna indietro". → Stiamo mettendo chiamate alla funzione c_tastiera in mezzo al programma che il compilatore non vede.
da funzione suo quindi nuovi registri senza ridurli e scrisserci. Utilizziamo di nuovo l'assembly ed in particolare alcune macro della libce →

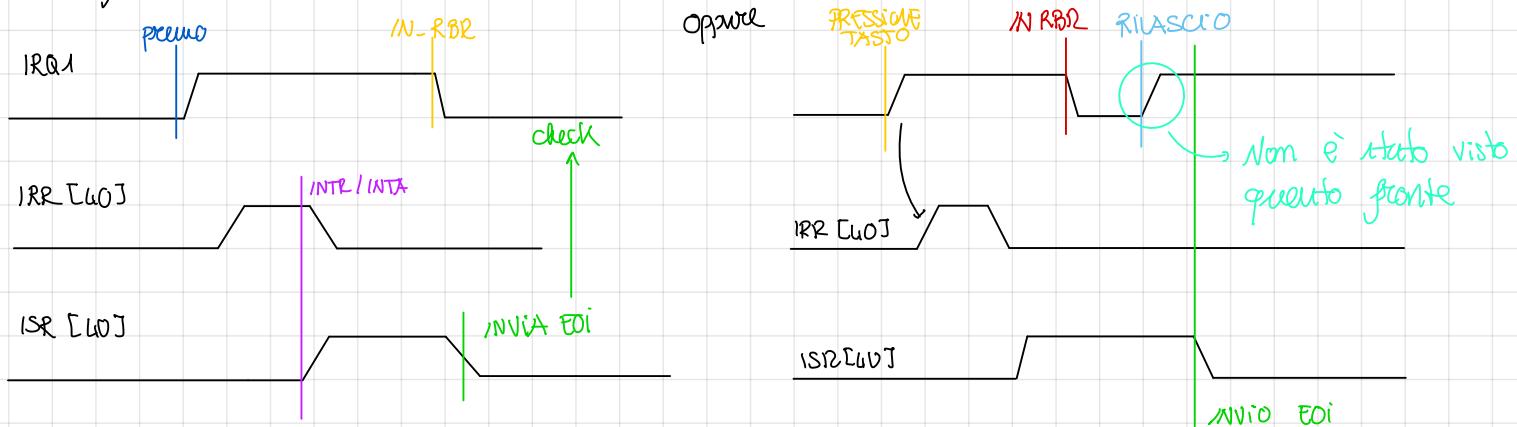
```
#include "libce.s"
.global a_tastiera
a_tastiera:
    salva_registri
    call c_tastiera
    carica_registri
    iretq
```

Soltanto adesso permette di uscire dal programma con la premessa del tutto ESC → ottimizzando, la variabile fine non viene considerata perché il suo valore viene cambiato dalle interruzioni non viste dal compilatore → si aggiunge la keyword volatile nella dichiarazione della variabile.

ATT: avere variabili condivise tra programma principale ed interruzioni può portare alla nascita di errori tempo-vocianti.

Ma cosa accade se avremo nello stesso livello anche

sul fronte?



La cosa funziona poiché la tastiera supporta un handshake. Se IRQ1 rimane ad 1 dopo l'EOI viene riconosciuta una nuova interruzione. L'handshake consiste nella lettura di RBR da parte del software che abilita IRQ1.

Il timer ha un handshake. Le richieste devono essere necessariamente gestite sul fronte

```
volatile natq counter = 0;
extern "C" void c_timer()
{
    counter++;
    apic_send_EOI();
}
```

```
void attiva_timer(natw N)
{
    outputb(0b00110111, iCWR); // timer 0, modo 3
    outputb(N, iCTR0_LOW);
    outputb(N >> 8, iCTR0_HIGH);
}
```

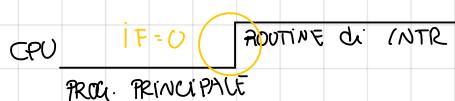
```
extern "C" void a_timer();
const natb TIM_VECT = 0x50;
int main()
{
    apic_set_VECT(2, TIM_VECT);
    gate_init(TIM_VECT, a_timer);
    attiva_timer(50000); // periodo di circa 50ms
    apic_set_TRGM(2, true); // false: fronte, true: livello
    apic_set_MIRQ(2, false); // SBAGLIATO
    for (volatile int i = 0; i < 100000000; i++)
        ;
    printf("counter = %d\n", counter);
    pause();
}
```

```
#include <libce.s>

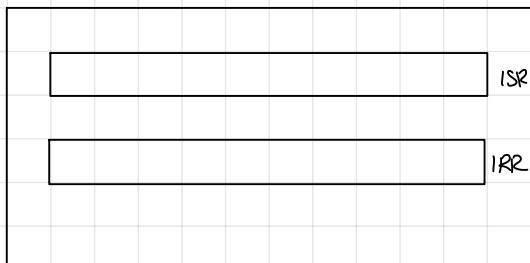
.global a_timer
a_timer:
    salva_registri
    call c_timer
    carica_registri
    iretq
```

• GESTIONE DI PIÙ RICHIESTE di INTERRUZIONE

Cosa succede se vogliamo gestire più richieste di interruzione contemporaneamente? Apic le può gestire in base alla priorità



→ importante per prioritizzare diverse richieste (timer)



È possibile non far resettoce IF: è possibile annidare le interruzioni che devono chiedere a stack

→ La priorità della richiesta è associata al tipo: ogni 16 tipi hanno una clava di priorità diversa (2° cifra esadecimale diversa)

Se avanza una richiesta con clava superiore a quella attualmente in servizio la richiesta viene accettata ed il processore viene di nuovo interrotto. Se invece la richiesta ha clava di priorità inferiore a quella in servizio deve aspettare. Al momento di end of interrupt APIC aspetta ISR e guarda IRR. Se c'è un'interruzione con priorità superiore alla seconda presente in ISR questa viene eseguita, altrimenti attende

ESERCIZIO

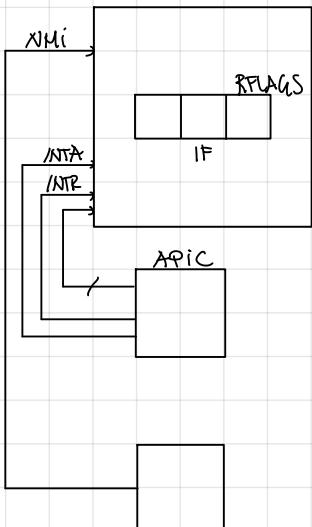
```
int main()
{
    clear_screen(0x0f);

    apic_set_VECT(1, KBD_VECT);
    trap_gate_init(KBD_VECT, a_tastiera);
    apic_set_MIRQ(1, false);
    enable_intr_kbd();
    permette anni durento
    trap_gate_init(TIM_VECT, a_timer);
    attiva_timer(50000); // periodo di circa 50ms
    apic_set_TRGM(2, false); // false: fronte, true: livello
    apic_set_MIRQ(2, false);

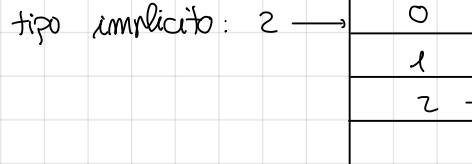
    natq spinpos = 0;
    while (!fine) {
        video[12*80+40] = attr | spinner[spinpos % 4];
        spinpos++;
    }
    return 0;
}
```

```
extern "C" void c_timer()
{
    static natq timer_spinpos = 0;
    video[12*80+60] = attr | spinner[timer_spinpos % 4];
    timer_spinpos++;
    apic_send_EOI();
```

- o INTERRUZIONI NON NASCHERABILI



NMi: Non-Maskable Interrupts → avviano forzatamente il ciclo di esecuzione da eseguire indipendentemente dallo stato del processore



→ salta alla routine sempre
utilizzato ad esempio quando salta la corrente

Eccezioni

Possiamo replicare il vecchissimo delle intuizioni a cose che accadono internamente al procuratore (ad esempio la derivazione per 0) → il procuratore potrebbe essere intercettato a sapere il verificarsi di queste situazioni eccezionali in modo da potere gestire (ad esempio quando il procuratore non può eseguire alcune intuizioni)

Deve essere interrotto il programma per eseguire un altro deciso dal programmatore. In particolare abbiamo un'interruzione proveniente dall'interno. Viene utilizzato, nei processori Intel, lo stesso meccanismo delle interruzioni esterne eccetto che il tipo di interruzione è implicito ed è codificato all'interno del microprogramma stesso. Il processore riconosce 32 diversi tipi di queste interruzioni.

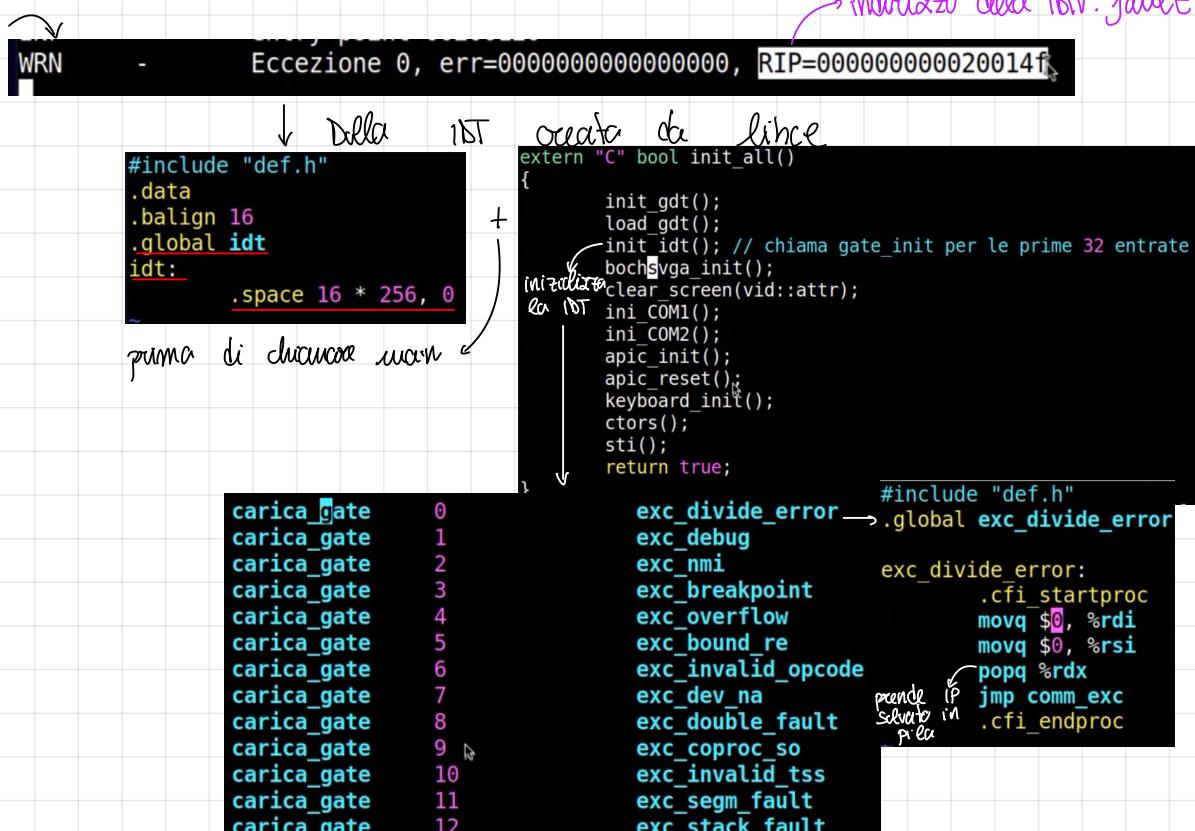
de intervazioni esterne possono arrivare in qualsiasi momento ma il procedimento de raccomande solo dopo la fine di esecuzione dell'intruzione perché il bit IF viene generato solo dopo l'esecuzione dell'intruzione. Nelle intervazioni interne invece è possibile che la fine di esecuzione non venga portata a termine. Altre possono avere invece rimandate. Nel primo caso viene salvato l'IP dell'intruzione che ha generato l'intruzione mentre nel secondo caso viene salvato l'IP dell'intruzione successiva. Nel primo caso tipicamente l'errore può essere remediatob. Le prune intruzioni sono di tipo fault mentre le raccomande sono di tipo trap. Esiste un terzo tipo detto abort che viene usato per erori gravi che bloccano tutto.

ESEMPIO

```
#include <libce.h>

int main()
{
    int x = 3, y = 0;

    return x / y;
}
```



Se volessimo gestirle in modo diverso

```
#include <libce.h>

void div()
{
    printf("divisione per zero!!!");
}

int main()
{
    int x = 3, y = 0;
    gate_init(0, div);

    return x / y;
}
```

~

x volte perché
fa RET allo stesso
indirizzo

⚠ trap: eccezione → l'IP salvato è quello dell'instruzione successiva
entrafa INT → tipo interrupt / trap: interruzioni dichiarative/abilitate

Eccezione di tipo TRAP: tipicamente generata per tale scopo (int 3)

```
#include <libc.h>

int main()
{
    int x = 0;

    x++;
    asm("int3");
    x++;
}
```

Eccezione 3, err=0000000000000000, RIP=00000000000200149

istruzione successiva

Sovr per il debugger
in quanto funziona come
interruzione e poi il
controllo ritorna al programma
quando si fa continue

```
200144: 83 45 fc 01      addl $0x1,-0x4(%rbp)
200148: cc                int3
200149: 83 45 fc 01    addl $0x1,-0x4(%rbp)
```

Viene messo cc qui quando c'è un break point e quindi quando il
processore sente di questa causa un'eccezione. Per poter proseguire poi deve
avere reimpintata l'istruzione originale e poi deve essere decrementato l'IP in
pila di 1. Infine, va rimesso il break point al punto di successivo

Eccezione debug

```
0]: sys intr      201a85 (:exc_divide_error)
1]: sys intr      201a5d (:exc_debug) →
2]: sys intr      201b05 (:exc_nmi)
3]: sys intr      201a35 (:exc_breakpoint)
4]: sys intr      201b19 (:exc_overflow)
5]: sys intr      201a21 (:exc_bound_re)
6]: sys intr      201acf (:exc_invalid_opcode)
7]: sys intr      201a71 (:exc_dev_na)
8]: sys intr      201a99 (:exc_double_fault)
9]: sys intr      201a49 (:exc_coproc_so)
a]: sys intr      201ae3 (:exc_invalid_tss)
b]: sys intr      201b3b (:exc_segm_fault)
c]: sys intr      201b5d (:exc_stack_fault)
d]: sys intr      201b2d (:exc_prot_fault)
e]: sys intr      201ac1 (:exc_int_tipo_pf)
f]: sys intr      201b6b (:exc_unknown_exc)
10]: sys intr     201aad (:exc_fp)
11]: sys intr     201a0d (:exc_ac)
12]: sys intr     201af1 (:exc_mc)
```

Viene emulata in modalità single step: viene generata un'eccezione dopo
l'esecuzione di ogni istruzione

⚠ trap flag: il processore genera
una eccezione alla fine di
ogni istruzione

dichilitato ad ogni entrata
della INT. Questo verrà reabilitato
alla fine dell'eccezione

Come gestiscono il trap flag?

abilitano

```
.global enable_single_step
enable_single_step:
    pushf   → setta ie bit 10
    orw $0x100, (%rsp)
    popf   → contiene l'indirizzo del registro da modificare
    ret
```

dichilitano

```
.global disable_single_step
disable_single_step:
    pushf
    andw $0xFEFF, (%rsp)
    popf
    ret
```

```
#include <libce.h>
```

```
extern "C" void enable_single_step();
extern "C" void disable_single_step();
int main()
{
    int x;
    gate_init(1, a_debug); // per eseguire la nostra
    enable_single_step();   routine
    x++;
    x++;
    x++;
    x++;
    x++;
    x++;
    disable_single_step();
}
```

→ Nella libce la routine stampa l'eccezione e si ferma

↓ abbandona

```
.global a_debug
a_debug:
    salva_registri
    call c_debug
    carica_registri
    iretq
```

```
+  
extern "C" void a_debug();  
extern "C" void c_debug()  
{  
    printf("debug\n");  
}
```

risultato

```
modo addrs 0x0000000000000000 QEMU  
debug  
Premere ESC per proseguire  
  
/home/studenti/CE/lib/ce/boot.bin'
```

Se volevamo stampare l'indirizzo dell'interruzione in pillo

```
.global a_debug
a_debug:
    salva_registri
    mov i20(%rsp), %rdi
    call c_debug
    carica_registri
    iretq
```

```
extern "C" void a_debug();
extern "C" void c_debug(natq rip)
{
    printf("debug rip=%p\n", rip);
}
```

↓ output

```
debug rip=000000000000200176
debug rip=00000000000020017a
debug rip=00000000000020017e
debug rip=000000000000200182
debug rip=000000000000200186
debug rip=00000000000020018a
debug rip=00000000000020018e
debug rip=0000000000002001a4
debug rip=0000000000002001a5
debug rip=0000000000002001ab
debug rip=0000000000002001ac
Premere ESC per proseguire
```

→ attivata nella POP di enable-single-step: vede trap flag ad 1

} tutte nel main

→ da qui in due single step

→ La pop q disattiva il flag
ma lo fa in ritardo e quindi viene di nuovo generato

Il breakpoint può essere quindi rimesso al suo posto ad ogni esecuzione togliendo il trap flag prima di cedere il controllo e facendo eseguire al programma l'interruzione per poi settarlo di nuovo ad 1.

step

L'istruzione step deve fare lo step di una singola istruzione in linguaggio assembly. Per fare questo il compilatore costruisce una tabella di corrispondenze tra le regole del linguaggio e gli IR. Questo tabella se il compilatore ottimizza e per fare questo viene utilizzato il single step.

Protezione

SISTEMISTI: coloro che mettono i programmi nella macchina



: Si suppone che lo stato del programma 1 venga relato per poter eseguire contemporaneamente il programma 2

L'utente 1 non ha nessun interesse a vedere la CPU dell'utente 2 e l'utente 2 non ha interesse a restituire il controllo all'utente 1 con le interruzioni e quindi potrebbe ad esempio disabilitare le interruzioni. Vorremmo avere quindi dei modi per far avere ai programmi determinati comportamenti → i linguaggi di programmazione sono troppo esplicativi e non è sufficiente in nessun caso analizzare il programma staticamente per capire il suo comportamento

È necessario l'aiuto del procedore: se quelle interruzioni sta eseguendo in ogni caso → deve sapere chi gli sta chiedendo di fare una determinata operazione. Aggiungiamo un contesto al testo del programma e modifichiamo il procedore in modo da riconoscerlo

- sistema: eseguite tutte le interruzioni
- utente: alcune interruzioni NON vengono eseguite (STI, CLI, IN, OUT)

È necessario fornire al utente le interruzioni necessarie per dialogare con la macchina in modalità sistema → forniamo delle routine in contesto sistema. Abbiamo però bisogno di un'interruzione che serva sia il contesto da utente a sistema. → Meccanismo strettamente correlato alle interruzioni:

- /INT \$tipo → fa la stessa cosa di un'interruzione esterna
- o di un'eccezione



→ Routine: primitive di interruzione

→ Campo livello: indica al procedore a quale livello portarsi nell'esecuzione della routine

Non viene utilizzata la CALL perché l'utente potrebbe saltare in punti casuali della routine di sistema saltando alcuni controlli. Con l'utilizzo di /INT invece l'indirizzo della routine sta in una tabella a cui l'utente non può accedere. Inoltre, vi sono delle zone di memoria a cui l'utente non può accedere in cui sono presenti strutture di interruzione o la IDT ad esempio. Se l'utente prova a ricevere su questa memoria viene sollevata un'eccezione con conseguente esecuzione di una routine

In tutti i casi (eccezioni, interruzioni, INT) si salta ad una routine speciale decisa dalla IDT e si pone sempre a livello utenza. Tutti e 3 i meccanismi convergono ai gate della IDT che eseguiranno sempre dei controlli

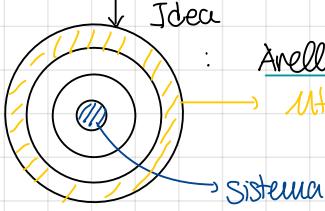
Vi sono alcuni controlli

CODICE
DATI (INT)

- M1: riservata al sistema → il processore parte nella modalità sistema e carica il codice ed i dati tra cui la INT
- : il programma prima di lanciare il controllo comunica questo indirizzo e poi mette il processore in modalità utente
- M2: Utente (il più possibile)
- ↓
- Gli utenti non possono usare le periferiche e le interruzioni esterne. Quando ci sono eccezioni o interruzioni il controllo passa al sistema
- Vogliono proteggere gli utenti fra di loro per non creare conflitti nell'esecuzione di programmi → per adesso esistono che non sia possibile avere programmi di utenti diversi in memoria

Nel processore intel

→ meccanismi di protezione sono stati aggiunti a partire dal 286 (16 bit).



: Anelli di protezione → 4 implementati in hardware

Utente: nei livelli più privilegiati sono eseguite più istruzioni mentre nel livello utente non ne possono essere eseguite alcune come:

LIDTR

HLT

Per I/O, CLI/SI vi è più flessibilità anche a causa dei livelli intermedii

↓

Nel registro RFLAGS c'è un flag (IOPL) che specifica a quale livello posso eseguire tali istruzioni → contenendo anche IF questo registro non può essere modificato perché in tal modo potrebbe concederti dei nuovi privilegi. Per questo la T0FF è stata modificata in modo da ignorare le modifiche al livello utente senza però generare alcuna eccezione

↓

La modalità corrente del processore si trova nel registro CS



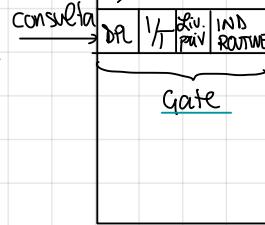
↓ 2 bit meno significativi

Questo, insieme ad altri registri, verrà utilizzato per la segmentazione e poi, a partire dal 386 (32 bit), per la paginazione. Per questa vengono utilizzati solo il livello utente ed utente.

↓ **IMPORTANTE: funzionamento**

INT

eccezione → in tutti e 3 i casi contiene un tipo

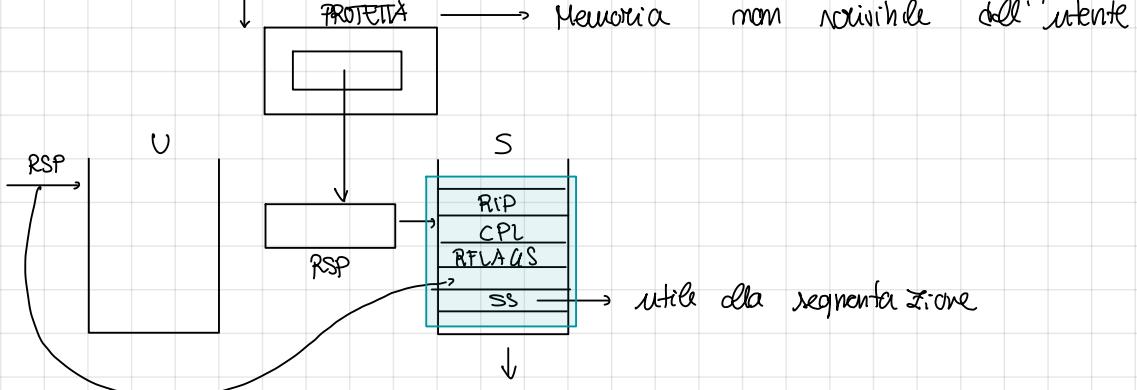


livello di privilegio che bisogna avere prima di eseguire l'interruzione

Vale solo per INT mentre ha senso per le interruzioni esterne → Si assume che provenga dal sistema (eccetto per INT3)

Su quali gate è necessario mettere un DPL sistema? Le eccezioni devono verificarsi solo uccidendo il verificarsi effettivo dell' evento che le ha generate. Questo vale anche per le intervazioni esterne.
 Al livello di privilegio inoltre, attraversando un gate, non può diminuire il motivo e de non possano fidarsi del codice utente
 ↓ Attraversamento del gate

Se il privilegio aumenta il processore cambia pila il cui indirizzo è scritto all'interno del processore e la zona non può essere scritta dall'utente



Se non c'è incremento di privilegio questi elementi vengono salvati nella pila corrente senza cambiare.

Ma perché cambiano pila e non facciamo decidere l'indirizzo all'utente?cio che è scritto in RSP è deciso dall'utente e quindi potrebbe puntare alla memoria di utente. Il processore quindi comprenderebbe la memoria di utente. Inoltre, ci permette di garantire l'affidabilità delle informazioni in memoria sistema.

↓ Restituzione controllo

IRETQ: legge da RSP attuale 5 parole lunghe → dare ritorno, a che livello di privilegio, come rimettere i flag e come ripristinare la pila. Unico meccanismo per ascendere di privilegio e non può ridire: eccezione perché l'utente potrebbe scrivere CPL a processore

```
#include "def.h"
.data
.balign 16
.global sys_tss
.global gdt → serve per la segmentazione: parti di memoria contenenti un tipo
.global gdt_pointer descritte nella gdt tramite descruttore per ciascun segmento
gdt:
    .quad 0           //segmento nullo
    code_sys_seg: → segmento codice sistema
        .word 0b0      //limit[15:0] not used
        .word 0b0      //base[15:0] not used
        .byte 0b0       //base[23:16] not used
        ATTRIB. byte 0b10011010 //DPL[1|1|C|R|A] DPL=0=sistema
        byte 0b00100000 //G|D|L|------| L=1 long mode
    code_usr_seg: → segmento codice utente
        .word 0b0      //limit[15:0] not used
        .word 0b0      //base[15:0] not used
        .byte 0b0       //base[23:16] not used
        byte 0b11111010 //DPL[1|1|C|R|A] DPL=11=utente
        byte 0b00100000 //G|D|L|------| L=1 long mode
    data_usr_seg:
        .word 0b0      //limit[15:0] not used
        .word 0b0      //base[15:0] not used
```

Al ogni istante il processore ha un segmento codice, un segmento dati ed un segmento pila: registri CS, DS, SS che contengono l'identificatore del segmento corrispondente → identificati da indirizzi e dimensione → eccezione se c'è fuori da segmento

Scambi da utilizzare

Per la segmentazione il livello di privilegio del processore è pari a quello del segmento codice corrente. Per avere più livelli di privilegio è necessario definire segmenti di codice con privilegio differente. → Per cambiare CS è necessario uscire da

+

```

data_usr_seg: → il registro SS deve avere un valore valido (dovuto a livello utente)
    .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
des_tss: → indexi 2to nuova pila quando il privilegio del processore cambia
    .quad 0
    .quad 0
end_gdt:
.balign 16

```

ESempio

```

.global liv_sistema
liv_sistema:
    movq $0x8, 8(%rsp)
    movq $0, 32(%rsp)
    iretq

.data
.balign 16
.global sys_tss, sys_tss_end
sys_tss:
    .fill 1, 4, 0
    .quad sys_stack_end → bottom pila per passaggio a livello sistema
    .fill 11, 8, 0
    .fill 1, 2, 0
    .word sys_tss_end - sys_tss
sys_tss_end:

.bss ← . align 16
sys_stack:
    .space 4096, 0
sys_stack_end:
\<sys_stack_end\>

```

```

#include <libce.h>
void foo()
{
    natb c;
    printf("Provo a leggere RBR\n");
    c = inputb(0x60);
    printf("Ho letto RBR: %2x\n", c);
}

void foo2()
{
    printf("provo ad eseguire hlt\n");
    asm("hlt");
}

void foo3()
{
    printf("disabilito le interruzioni\n");
    asm("cli");
    printf("riabilito le interruzioni\n");
    asm("sti");
}

```

```

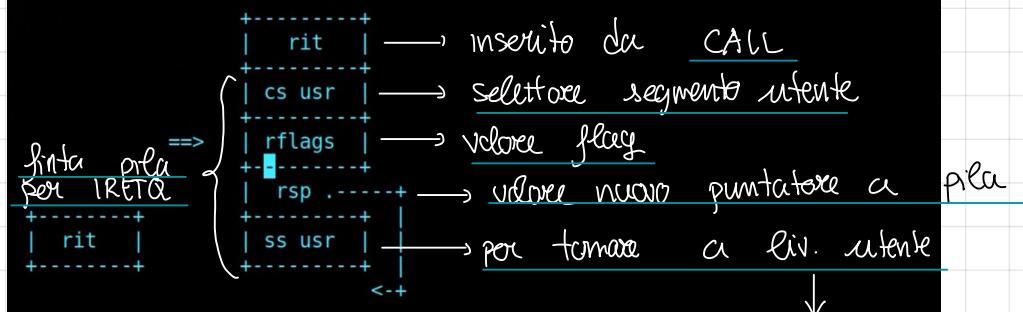
extern "C" void liv_utente();
extern "C" void liv_sistema();
int main()
{
    gate_init(0x70, liv_sistema);

    // proviamo ad eseguire foo() prima a livello sistema e poi a livello
    // utente
    foo();
    // portiamo il processore a livello utente
    liv_utente();
    // ora siamo a livello utente
    foo();
    // [provare anche con foo2() e foo3()]

```

Vogliamo una funzione che ci permetta di passare al livello utente: conosciamo solo la IRET per fare questo

modifichiamo la pila in modo da poter eseguire una iretq che porti il processore a livello utente. Vogliamo tornare comunque al chiamante e senza passare ad una nuova pila, quindi effettuiamo la seguente trasformazione:



```
.global liv_utente
liv_utente:
    popq %rax      # rit
    movq %rsp, %rdi
    pushq $0x1b    # selettore dati utente (ss usr)
    pushq %rdi     # rsp salvato in precedenza
    pushq %rdi     # rflags con IF=1 e IOPL=sistema
    pushq $0x0200  # provare anche con
                   # $0x3200 (IF=1 e IOPL=utente)
    pushq $0x13    # selettore codice utente (cs usr)
    pushq %rax    # rit salvato in precedenza
    iretq
```

Output

```
Provo a leggere RBR
Ho letto RBR: fa
Provo a leggere RBR
Ho letto RBR: fa
```

debbug modo: sistema intr: abilitati io: permesso cr3: 0x00001000
 dobbiamo generare IOPL ed il modo attuale: in IOPL c'è scritto quando sono abilitate le intenzioni di I/O
 ↓ dopo liv utente

↳ La printf funziona perché scrive in memoria ma non può spostare il cursor perché dovrebbe usare OUT

imprinth esegue istruzione di I/O: eccitare 13 (general protection fault)

[d] sys_intr 201d0c (:exc_prot_fault)

gate porta a livello sistema

verde: dal livello di privilegio attuale si può attraversare il gate

chiamando foo2 e foo3: eccitare 13
 impostando IOPL ad utente: pushq \$0x3200

↓ eseguo foo

```
Provo a leggere RBR
Ho letto RBR: fa
Provo a leggere RBR
Ho letto RBR: fa
Premere ESC per proseguire
```

Proviamo a fornire a livello sistema con IRETQ

```
.global liv_sistema_bad
liv_sistema_bad:
    popq %rax      # rit
    movq %rsp, %rdi
    pushq $0x0      # selettore dati utente (ss usr)
    pushq %rdi     # rsp salvato in precedenza
    pushq $0x0200   # rflags con IF=1 e IOPL=sistema
    # provare anche con
    # $0x3200 (IF=1 e IOPL=utente)
    # selettore codice utente (cs usr)
    pushq $0x8      # rit salvato in precedenza
    pushq %rax
    iretq
```

→ Ottengo eccezione B

Proviamo invece ad alzare il livello tramite il pulaggio da gate sistema al gate e

```
.global liv_sistema
liv_sistema:
    movq $0x8, 8(%rsp)
    movq $0, 32(%rsp)
    iretq
```

→ in questo per tornare al livello conosciuto a questa funzione

Modifichiamo la funzione componi-gate:

```
#include "def.h"
.global componi_gate_utente
componi_gate_utente:
    .cfi_startproc
    movq %rsi, %rax          // offset della routine

    movw %ax, (%rdi)          // primi 16 bit dell'offset
    movw $SEL_CODICE_UTENTE 2(%rdi)
    a cose normali è sistema
    movw $0, %ax
    movb $0b1101110, %ah      // byte di accesso
                                // (presente, 32bit, tipo interrupt)

    andq $0x1, %rdx
    orb %dl, %ah
    movb $0, %al
    movl %eax, 4(%rdi)        // 16 bit piu' sign. dell'offset
                                // e byte di accesso
    shrq $32, %rax
    movl %eax, 8(%rdi)
    movl $0, 12(%rdi)          //riservato

    ret
    .cfi_endproc
```

+

```
void gate_init_usr(natl num, void routine())
{
    gate gg;
    componi_gate_utente(gg, routine, false /* interrupt */);
    idt[num] = gg;
}

void trap_init_usr(natl num, void routine())
{
    gate gg;
    componi_gate_utente(gg, routine, true /* trap */);
    idt[num] = gg;
}
```

↓

```
void gate_init_usr(natl, void());
int main()
{
    gate_init_usr(0x70, liv_sistema);

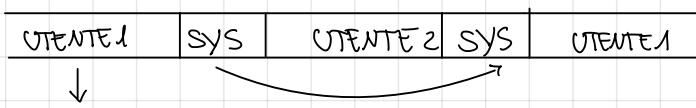
    // proviamo ad eseguire foo() prima a livello sistema e poi a livello
    // utente
    foo();
    // portiamo il processore a livello utente
    liv_utente();
    // ora siamo a livello utente
    //foo3();
    // [provare anche con foo2() e foo3()]

    // nel caso in cui riusciamo ad arrivare fin qui (per esempio con foo()
    // o foo3() se IOPL=utente) dobbiamo poi ritornare alla funzione start
    // della libce. Questa vorrà eseguire alcune operazioni privilegiate e,
    // se torniamo mentre siamo ancora a livello utente, si genererà
    // un'eccezione di protezione. Per tornare a livello sistema abbiamo
    // predisposto un gate della IDT (numero 0x70) ed eseguiamo una INT per
    // attraversarlo
    asm("int $0x70");
    pause();
}
```

eccezione B

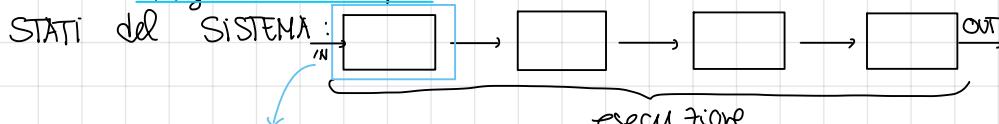
Multiprogrammazione

Il sistema multiprogrammato tenta di eseguire più programmi contemporaneamente.



processo: programma in esecuzione su un'unità nel tempo.

im prende il processo può eseguire più programmi tra di loro precisi. È qualcosa che si



Contiene tutto il memoria per far progredire l'esecuzione: risultati del processo, memoria del utente → Ciascuno stato rappresenta un contenimento di questi elementi e permette di stabilire il successivo (c'è IP e codice) → permette di eseguire da un processo ad un altro: il sistema sa che esempio sul disco la foto del processo da cui salta e carica in memoria una foto di un altro processo presente sul disco e fa avanzare il processo. Con la stessa metodologia si può tornare al processo precedente.

tutti i personaggi richiedono il tramito attraverso il sistema perché è l'unico che ha l'accesso al disco ed alle foto → il sistema quindi gestisce i processi.

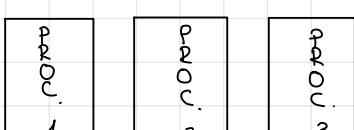
• PRIMITIVA (CHIAMATA DI SISTEMA / SYSTEM CALL)

Moltissimi fondamentali forniti dal sistema non comprensibili o modificabili. Nei sistemi operativi viene detta nucleo o kernel e sta sempre in memoria e sono primitive le funzioni che offre ai programmi o all'utente. Nei sistemi multi programmati permettono di creare o terminare processi e sono implementate come sus call perché utilizzano il meccanismo della protezione per immobilizzare il livello del processo al livello sistema in modo da avere accesso a determinate strutture dati →

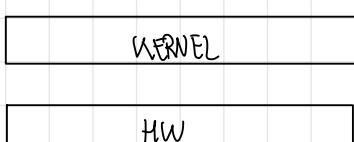


va in esecuzione per primo all'avvio quando il processo è ancora al livello sistema. Imposta la idt, la gdt ed altro in modo tale da fornire i gate necessari e poi cede il controllo all'utente portando il processo a livello utente. Non rimane in memoria ed acquista il controllo quando viene attraversato un gate (esercizio, interruttore, INT).

⚠ ATT: il processo fa una cosa per volta



→ Un processo può accedere all'hardware solo tramite il kernel: i processi ottengono la CPU del kernel e controlla così sia il processo tramite le eccezioni comunque "immaginate" del kernel.



Il kernel protegge i processi l'uno dell'altro → Riprendiamo il concetto di contesto: ADD %RAX, 1000 contesto: in un sistema multiprogrammato il significato dipende da quale processo viene eseguita. Se lo eseguono P1 e P2 il contenuto di RAX e di 1000 dipendono dal processo stesso. Ogni processo è quindi associato ad un contesto. Il kernel quindi gestisce i contesti dei vari

processi

nella macchina virtuale

```
studenti@ce:~$ cd nucleo-6.6.1/  
studenti@ce:~/nucleo-6.6.1$ ls  
build debug include io Makefile run
```

fintorni di I/O

sistema utente util

→ modello utente: modilità utente con INT abilitato

```
studenti@ce:~/nucleo-6.6.1$ ls sistema/  
sistema.cpp sistema.s
```

↓
modelita atomica di loro
(livello sistema e INT disabilitati)

la funzione log specificando la severità del log (LOG -> INFO / WARNING / ERROR) e poi il messaggio

Nel debugger:

```
processi: 1
esecuzione: [0xFFFF, MAX_PRIO]
pronti: [1, MAX_PRIO] → [0, DUMMY]
p_sospesi: nessuna priorità minima prioritaria
```

Per ogni processo viene fornito un identificatore e la sua priorità

Vengono gestiti dalla funzione Schedulatore (): fra tutti i processi pronti sceglie il primo da eseguire (in ordine di priorità). Nella struttura inoltre sono elencate in verde le primitive concerne del sistema e del modello I/O.

sistemi

```
[20]: sys intr      20070c (sistema:a activate_p)
[21]: sys intr      20071d (sistema:a terminate_p)
[22]: sys intr      20072e (sistema:a sem_ini)
[23]: sys intr      20073f (sistema:a sem_wait)
[24]: sys intr      200750 (sistema:a sem_signal)
[25]: sys intr      200761 (sistema:a delay)
[26]: sys intr      200772 (sistema:a do_log)
[27]: sys intr      200783 (sistema:a getmeminfo)
```

```
[40]: sys trap      10000000272 (io:a_readhd_n)
[41]: sys trap      10000000279 (io:a_writehd_n)
[42]: sys trap      10000000280 (io:a_dmareadhd_n)
[43]: sys trap      10000000287 (io:a_dmawritehd_n)
[44]: sys trap      1000000025d (io:a_readconsole)
[45]: sys trap      10000000264 (io:a_writeconsole)
[46]: sys trap      1000000026b (io:a_iniconsole)
[47]: sys trap      1000000028e (io:a_getiomeminfo)
```

Organizzazione memoria

--sistema/condiviso---

0000000000000000	S-000-000-000-000: unmapped
0000000000010000	S-000-000-000-001: S W
00000000000b8000	S-000-000-000-270: S W PWT A D
00000000000b9000	S-000-000-000-271: S W
0000000000100000	S-000-000-000-400: S W A D
0000000000101000	S-000-000-000-401: S W
0000000000107000	S-000-000-000-407: S W A D
0000000000108000	S-000-000-000-410: S W
0000000000200000	S-000-000-001-000: S W PS A D
0000000000600000	S-000-000-003-000: S W PS
0000000000200000	S-000-000-020-000: unmapped
00000000fec00000	S-000-003-766-000: S W PWT PCD PS A D
00000000ff000000	S-000-003-770-000: S W PWT PCD PS
0000000100000000	S-000-004-000-000: unmapped

menu video
meccaniche primitive

Rosso: acceso relativ

Giallo: acceso in lettura

Verde: acceso in lettura e scrittura

codice / dati

scrittura

pila {

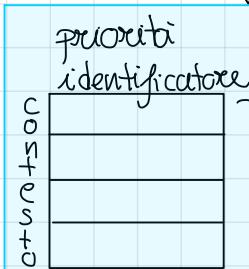
--sistema/privato---

000000fffffff000	S-001-777-777-777: S W A D
---IO/condiviso---	
0000010000000000	S-002-000-000-000: S R A
000001000004000	S-002-000-000-004: unmapped
0000010000010000	S-002-000-000-020: S W A
0000010000011000	S-002-000-000-021: S W A D
0000010000012000	S-002-000-000-022: S W
0000010000020000	S-002-000-000-040: S W A D
0000010000022000	S-002-000-000-042: S W
0000010000012100	S-002-000-000-441: unmapped
---utente/condiviso---	
ffff800000000000	U-400-000-000-000: U R A
ffff800000002000	U-400-000-000-002: U W A
ffff800000003000	U-400-000-000-003: U W A D
ffff800000005000	U-400-000-000-005: U W
ffff8000000104000	U-400-000-000-404: unmapped
---utente/privato---	
fffffffffffff0000	U-777-777-777-760: U W
fffffffffffff0000	U-777-777-777-777: U W A D

Gestione dei processi

Un processo è una struttura dati: il kernel ha un descrittore che contiene tutte le informazioni utili a descriverlo

↓ descrittore

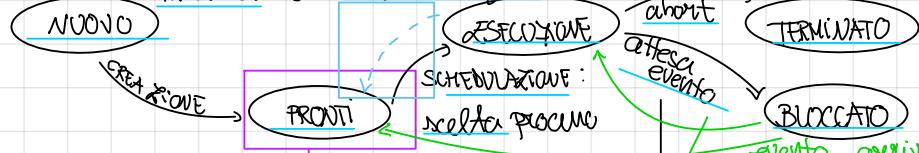


Registri del processo
(copia parziale registri
CPU)

→ I descrittori sono posizionati nella "area rossa" della memoria e sono organizzati in liste: stato del processo

↓ in quali stati può trovarsi

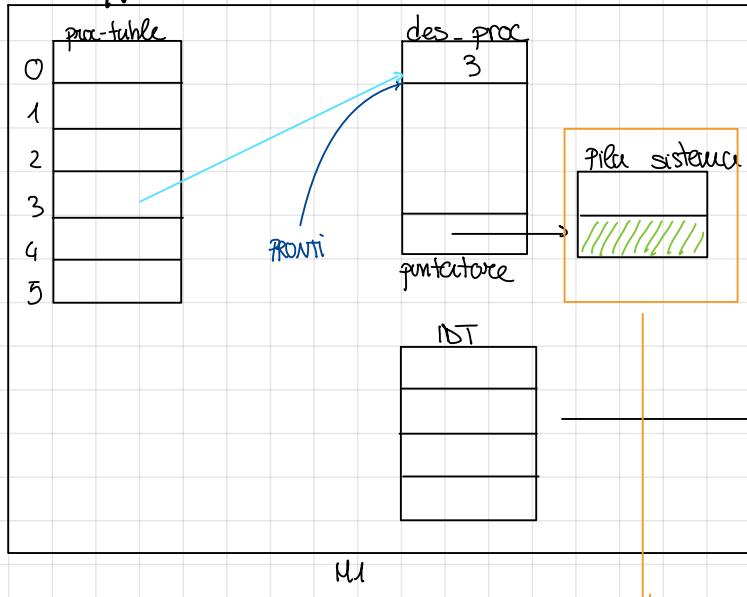
PREEMPTION: è il intervallo che toglie (ecc, int.) temporaneamente



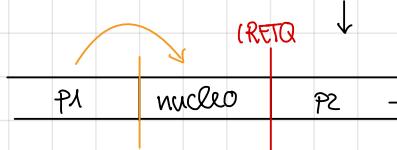
evento cancellato

Potremmo progettare se avremo a disposizione le code sono dive per eventi
una CPU.

○ PASSAGGIO di PROCESSI



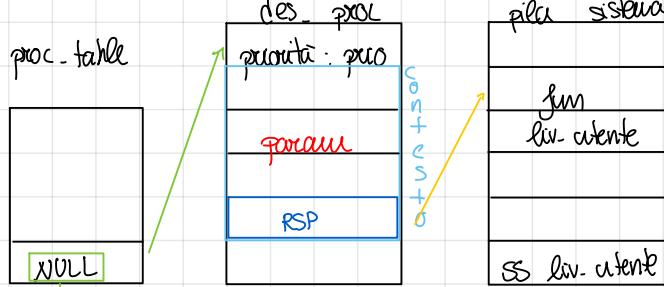
d'E necessaria una pila virtuale diversa per ogni processo. Quando un processo entra nel nucleo vogliamo isolare le informazioni: id, livello, precedenza, punt-nucleo, contesto e puntatore. Queste sono m'agrendere alla pila virtuale. Avendo più pile di virtuale possiamo ordinare a piacere i processi. La pila viene decisa automaticamente in quanto viene utilizzato RSP che viene salvato e caricato ogni volta



ma niente sicuri che la pila utilizzata da P2 contenga tutte le informazioni necessarie ad eseguirlo?

Chiediamo ci la storia fatta di P. Supponiamo che non è in esecuzione e quindi è possibile che non ci sia mai stato oppure è possibile che non già molto. Nel secondo caso deve esserci stata un'eccezione, un'interruzione o una INT. In tutti e 3 i casi ha attivato la INT e quindi sicuramente avrà solvuto tutte le informazioni necessarie. Nel primo caso è stato creato dal activate-p. È necessario immediatamente lo stato di tale processo in modo da renderlo compatibile con la INTQ ed in particolare lo si rende simile ad un processo appena entrato nel nucleo.

↓ activate- φ (fun, param, proc, env)

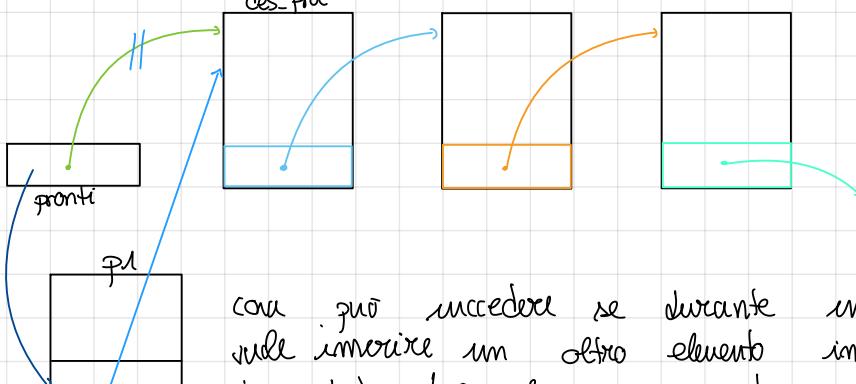


Scelto sempre quello a null

← RIP → vogliono che
← CS venga eseguita
← IF=1, iOP= sistema
← puntatore pila utente

fun (porcana)
scritto in RDI all'interno
di RDI nel contesto

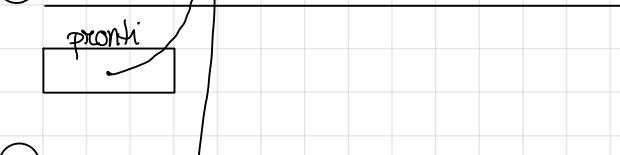
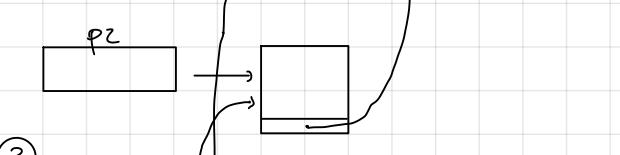
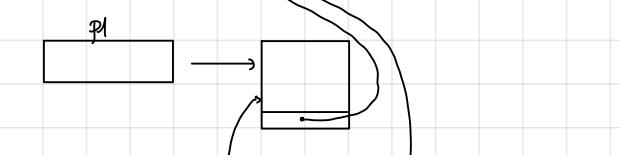
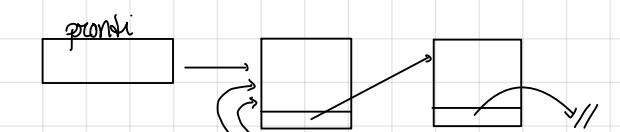
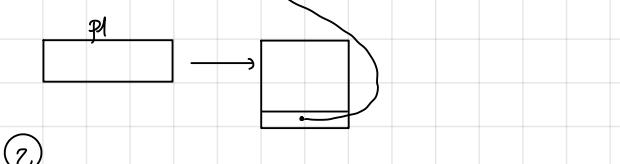
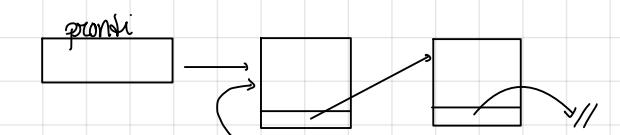
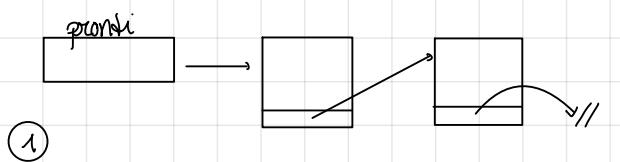
Ma perché affanno l'atomicità?



Cosa può succedere se durante un intervento in testa emetto routine
vulle immovere un altro elemento in testa nello stesso posto? Per nonere
in testa dovo fare $p1 \rightarrow next = pronti$ e $pronti = p1$. Niente vreca che un'inter-
-ruzione corriu in mezzo alle due operazioni

1. $p1 \rightarrow next = pronti$
4. $pronti = p1$

1. $p2 \rightarrow next = pronti$
3. $pronti = p2$



Non va bene garantisce atomicità però p2

Impediamo quindi che vi possano avere interruzioni.
Nel passaggio fra stati comuniti della lista si passa da
stati inconsistenti ma all'inizio dell'esecuzione delle funzioni

la struttura deve essere in uno stato correttivo. Per questo il codice deve essere eseguito in modo atomico

d'atomicità "trasforma" una serie di più istruzioni in una sola istruzione di linguaggio macchina: si vede soltanto il risultato finale

codice modello sistema

Si ha subito l'inizializzazione della IST e di alcune parti di memoria

```
#include <costanti.h>
#include <tipo.h>
#include <libce.h>
#include <sys.h>
#include <sysio.h>

// indirizzo virtuale e indirizzo fisico nello spazio di memoria
typedef unsigned short ioaddr;
// indirizzo (fisico) nello spazio di I/O
typedef unsigned char natb;
typedef unsigned short natw;
typedef unsigned int natl;
#if defined(__x86_64__) || defined(CE_UTILS)
typedef unsigned long size_t;
typedef unsigned long natq;           → per utilizzare
                                         il hautloader
                                         a 32 bit
typedef unsigned long vaddr;
typedef unsigned long paddr;
#else
typedef unsigned int size_t;
typedef unsigned long long natq;
typedef unsigned long long vaddr;
typedef unsigned long long paddr;
#endif
```

Processi:

```
struct des_proc {
    natw id;           → descrizione di processo
    natw livello;      → indice tabella processi
    natl precedenza;
    vaddr punt_nucleo; → indirizzo della pila virtuale del processo Attno: va
    natq contesto[N_REG]; → copiato nel TSS
    paddr cr3;          → Array di natq con tante entrate quanti sono i registri
    des_proc *puntatore; → per fare le liste di processi
    // per debugging
    void (*corpo)(natq); } debugging
    natq parametro;
};
```

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

volatile natl processi;

→ numero di processi attivi: a 0 si fa lo shutdown

```
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 };
```

→ costanti indici nell'array contesto

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

```

// inserimento ordinato (per priorità) di p_elem in p_lista
void inserimento_lista(des_proc *&p_lista, des_proc *p_elem)
--- 20 lines: {---}

// rimuove da p_lista il processo a più alta priorità e
// lo restituisce
des_proc *rimozione_lista(des_proc *&p_lista)
--- 12 lines: {---}

```

→ inserimento mantenendo
ordine di precedenza
(vedi codice da nucleo)

→ Estrazione della testa

```

// inserisce esecuzione in testa alla lista pronti
extern "C" void inspronti()
{
    esecuzione->puntatore = pronti;
    pronti = esecuzione;
}

```

→ inserisce remove in
testa. Divenuta della
precedente in quanto
l'altra nella condizione
specificava \geq e
quindi a partire
da priorità inseriva nella
posizione successiva

```

// sceglie il prossimo processo da mettere in esecuzione
extern "C" void schedulatore(void)
{
    // poiché la lista è già ordinata in base alla priorità,
    // è sufficiente estrarre l'elemento in testa
    esecuzione = rimozione_lista(pronti); → C'è sempre almeno un
    }                                         processo: dummy
                                                ↓
// 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 variabile statica 'next'),
    // saltando quelli che risultano in uso.
    natl scan = next, found = 0xFFFFFFFF;
    do { → cerca di partire dall'ultimo
        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;
}

```

void dummy(natl i)
1 {
2 while (processi) → guarda le
 halt(); → interruzioni
 flag(LOG_INFO, "Shutdown");
 end_program();
3
4
5
end_program:
6 call reboot
7 cli
8 hlt

Bruttifica activate → entry point nella IDT inserito della corrispondente

```

.macro carica_gate num routine dpl
    → sostituisce il testo
    movq $num, %rdi
    movq $routine, %rsi
    movq $dpl, %rdx
    xorq %rcx, %rcx
    call init_gate
.endm

```

sopra con le definizioni sotto



	indice	routine	dpl
carica_gate	0	exc_div_error	LIV.SISTEMA
carica_gate	1	exc_debug	LIV.SISTEMA
carica_gate	2	exc_nmi	LIV.SISTEMA
carica_gate	3	exc_breakpoint	LIV.SISTEMA
carica_gate	4	exc_overflow	LIV.SISTEMA
carica_gate	5	exc_bound_re	LIV.SISTEMA
carica_gate	6	exc_inv_opcode	LIV.SISTEMA
carica_gate	7	exc_dev_no	LIV.SISTEMA
carica_gate	8	exc_dbl_fault	LIV.SISTEMA
carica_gate	9	exc_coproc_so	LIV.SISTEMA
carica_gate	10	exc_inv_ts	LIV.SISTEMA
carica_gate	11	exc_segm_fault	LIV.SISTEMA
carica_gate	12	exc_stack_fault	LIV.SISTEMA
carica_gate	13	exc_prot_fault	LIV.SISTEMA
carica_gate	14	exc_page_fault	LIV.SISTEMA

activate - ↗

```
a_activate_p:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    - call salva_stato
    call c_activate_p
    call carica_stato
    iretq
    .cfi_endproc

    .extern c_terminate_p
```

Riparte dal punto dove il processo si era bloccato
salva - stato

```
// offset dei vari registri all'interno di des_proc
.set PUNTNUCLEO, 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
```

+

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.
.cfi_startproc
.cfi_def_cfa_offset 8
pushq %rbx
.cfi_adjust_cfa_offset 8
.cfi_offset rbx, -16
pushq %rax
.cfi_adjust_cfa_offset 8
.cfi_offset rax, -24
```

```
movq esecuzione, %rbx
movq %rbx, esecuzione_precedente
```

l'indirizzo è minore di 4GB e quindi non serve uscire %RIP

```
// 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)
```

// 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_do_log LIV_UTENTE
carica_gate	TIPO_GMI	a_getmeminfo LIV_UTENTE

→ offset all'interno di un descrittore di processo

Copiamo i registri del processo nel descrittore

```

// 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)] un attivo dopo l'avvenuto del gate: questa è l'RIP relativo del processo
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)

```

→ Vogliamo salvare lo stato giusto
 ↓
 La CALL salva l'indirizzo di ritorno e quindi %rsp punta alla cella riserva a quella del processo

```

popq %rax
.cfi_adjust_cfa_offset -8
.cfi_restore rax
popq %rbx
.cfi_adjust_cfa_offset -8
.cfi_restore rbx

ret
.cfi_endproc

```

} Rispettina registri

carica_stato:

```

.carica_stato:
    .cfi_startproc
    .cfi_offset rbp, 8
    movq %rbx, %rcx → fa il dispatch: la riportare l'esecuzione
    popq %rcx // ind di ritorno, va messo nella nuova pila
    .cfi_adjust_cfa_offset -8
    .cfi_register rip, rcx nuovo pila Sistema

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

```

etichette utili al debugger
 esempio: funzione X chiama Y che chiama Z (record di attivazione)
 in fondo può funzionare e può la CALL salva stato

```

    movq %r10, %rax
    movq %rax, %cr3 // il TLB viene invalidato

```

```

    // 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
    mova RSP(%rbx), %rsp // cambiamo pila: avvichiamo il nuovo valore di %RSP
    pushq %rcx // rimettiamo l'indirizzo di ritorno
    .cfi_offset rip, -8

```

```

    // 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.
    cmpa $0, ultimo_terminato
    je 1f → salvo ad etichetta 1 davanti: sintesi utile
    call distruggi_pila_precedente per le macro

```

→ Distruggere tutte le strutture dati di un processo terminato → unico terminante - p

```
// aggiorniamo il puntatore alla pila sistema usata dal meccanismo
// delle interruzioni
movq PUNT_NUCLEO(%rbx), %rcx
movq %rcx, tss_punt_nucleo
```

Aggiorna puntatore a pila sistema nel TSS: trasferisce le 5 informazioni importanti nella pila del nuovo processo

```
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
```

→ caricamento registri

retq

funzione activate_p

```
_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
    1: controllo tutti i flag che possono essere fissati
    // non possiamo accettare una priorità minore di quella di dummy
    // o maggiore di quella del processo chiamante
    if (prio < MIN_PRIORITY || prio > esecuzione->precedenza) { : controllo
        flag(LOG_WARN, "priorita' non valida: %d", prio);
        c_abort_p(); // distrugge precedente corrente
        return;
    }

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

priorità

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

ci serve rispettare il livello di privilegio di chi chiama
 sono una funzione perché qui vorrei creare dinamicamente → vedere il
 livello di precedenza della pila
 (cs)

```

// restituisce il livello a cui si trovava il processore al momento
// in cui è stata invocata la primitiva. Attenzione: funziona solo
// se è stata chiamata salva_stato.
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;
}

```

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

```

if (p != nullptr) {
    inserimento_list(a, 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;`

Non posso fare return id
perché vorrei salvato in %RAX
che viene cancellato

Ora - processo:

```

des_proc* crea_processo(void f(natq), natq a, int prio, char liv)
{
    des_proc* p; // des_proc per il nuovo processo
    paddr pila_sistema; // pila_sistema del processo
    natq* pl; // pila sistema come array di natq
    natl id; // id del nuovo processo

    // allocazione (e azzeramento preventivo) di un des_proc
    p = new des_proc; // new chiama una funzione moreniente
    if (!p) // fornita dalla libreria standard di C
        goto err_out; // e restituisce un errore

```

`memset(p, 0, sizeof(des_proc));`

Mette a 0 l'area di memoria
puntata da p per la grandezza
di der_proc.

```

// riempiamo 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;

```

inizializzo i campi
noti di der_proc

`// selezione di un identificatore`

`id = alloca_proc_id(p);`

`if (id == 0xFFFFFFFF)`

`goto err_del_p;`

`p->id = id;` risponda a richiesta

`definire dopo`

`1`

```

14 err_del_stack: distruggi_pila(p->cr3, fin_sis_p, DIM_SYS_STACK);
15 err_rel_tab: clear_root_tab(p->cr3);
16 err_rel_id: rilascia_tab(p->cr3);
17 err_del_p: delete p;
18 err_out: return nullptr;
19 }
20

```

errore se identificatore non trovato

} diritti a cose in modo inverso

```

// creazione della pila sistema
if (!crea_pila(p->cr3, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
    goto err_rel_tab;
// otteniamo un puntatore al fondo della pila appena creata. Si noti
// che non possiamo accedervi tramite l'indirizzo virtuale 'fin_sis_p',
// che verrebbe tradotto seguendo l'albero del processo corrente, e non
// di quello che stiamo creando. Per questo motivo usiamo trasforma()
// per ottenere il corrispondente indirizzo fisico. In questo modo
// accediamo alla nuova pila tramite la finestra FM.
pila_sistema = trasforma(p->cr3, fin_sis_p - DIM_PAGINA) + DIM_PAGINA;

// convertiamo ao puntatore a natq, per accedervi più comodamente
pl = reinterpret_cast<natq*>(pila_sistema); ----- abbiamo similece una INT

```

```

if (liv == LIV_UTENTE) {
    // inizializziamo la pila sistema.
    pl[-5] = reinterpret_cast<natq>(f); // RIP (codice utente)
    pl[-4] = SEL_CODICE_UTENTE; // CS (codice utente)
    pl[-3] = BIT_IF; // 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
    // passerà ad eseguire la prima istruzione della funzione f,

```

```

// 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 err_del_sstack;
}

// 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);

p->livello = LIV_UTENTE;

// 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;

```

Come funzionano le primitive?

```

#include <all.h>

void mioproc(natq a)
{
    terminate_p();
}

void main()
{
    activate_p(mioproc, 10, 10, LIV_UTENTE);
    pause();
    terminate_p();
}

```

Converti questo in una CALL che non è
ceduta per attraversare i gate della INT.
Forniamo quindi un programma Amembix che
fa una INT -----

```

2 activate_p:
    .cfi_startproc
    int $TIPO_A
    ret
    .cfi_endproc

    .global terminate_p

```

Fornire del programma
sistema per facilitare

La funzione Amembler funziona come già visto, mentre per la funzione C è importante notare che i parametri ricevuti sono quelli passati dall'utente.

c_activate_p(void f(natq), natq a, natl prio, natl liv)

→ inseriti nei registri usuali per R
funziona di parametri

Vogliamo scrivere una primitiva che restituisce la precedenza del primo processo in coda pronti
↓ non posso fare

```
void main()
{
    activate_p(mioproc, 10, 10, LIV_UTENTE);
    printf("la precedenza del primo processo in coda pronti e': %d\n", pronti->precedenza);
    pause();
    terminate_p();
}
```

extern des_proc *pronti;

```
struct des_proc {
    natw id;
    natw livello;
    natl precedenza;
    vaddr punt_nucleo;
    natq contesto[N_REG];
    paddr cr3;

    des_proc *puntatore;

    // per debugging
    void (*corpo)(natq);
    natq parametro;
};
```

→ il linker non può collegare perché non conosce pronti
↓ Risolvilo

des_proc *pronti = reinterpret_cast<des_proc*>(0x210230);

↪ Metto l'indirizzo del file

Adesso il file utente viene compilato ma la CPU farà la sua esecuzione in quanto questa memoria è protetta dal hardware

L'utente scriveva una primitiva

Passo 1. Sceglie un numero non usato

#define TIPO_GP

0x28

Passo 2. Caricamento gate con tale tipo

carica_gate	TIPO_GP	a_getprec	LIV_UTENTE
-------------	---------	-----------	------------

Passo 3. Scrivere parte amembler primitiva

```
a_getprec:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_getprec
    call carica_stato
    iretq
    .cfi_endproc
```

Passo 4. Scrivere la primitiva in C nel modulo sistema

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

→ dobbiamo restituirla nel contesto

Passo 5. Lo inserisco in sys.h per poter avere chiamata facilmente

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

Punto 6. Dichiariamo la primitiva in utente.s

```
.global getprec
getprec:
    .cfi_startproc
    int $TIPO_GP
    ret
    .cfi_endproc
```

Punto 7. Utile ??

```
void main()
{
    activate_p(mioproc, 10, 10, LIV_UTENTE);
    natl prec = getprec();
    printf("la precedenza del primo processo in coda pronti e': %d\n", prec);
    pause();
    terminate_p();
}
```

↓ Debugger

process list:

```
(gdb) process list
==> Processo 0
    livello      : sistema
    corpo        : sistema:dummy(0)
    rip          : 0x202b9b <dummy(unsigned long)>
==> Processo 3
    livello      : sistema
    corpo        : io:estern_kbd(0)
    rip          : 0x100000005d <estern_kbd(int)>
==> Processo 4
    livello      : sistema
    corpo        : io:esternAta(0)
    rip          : 0x10000000d80 <esternAta(int)>
==> Processo 5
    livello      : utente
    corpo        : utente:start(0)
    rip          : 0xffff8000000001d4 <activate_p+2>
==> Processo 6
    livello      : utente
    corpo        : utente:mioproc(10)
    rip          : 0xffff800000000e8 <mioproc(unsigned long)>
==> Processo 7
    livello      : utente
    corpo        : utente:mioproc(11)
    rip          : 0xffff800000000e8 <mioproc(unsigned long)>
==> Processo 8
    livello      : utente
    corpo        : utente:mioproc(12)
    rip          : 0xffff800000000e8 <mioproc(unsigned long)>
```

Mostra i processi attivi

P PROC_tabelle:

```
$1 = {
100010 -> { id: 0, corpo: "sistema:dummy(0)", prec: 0 0x0, rax: 0 0x0 },
null,
null,
100250 -> { id: 3, corpo: "io:estern_kbd(0)", prec: 1104 0x450, rax: 0 0x0 },
100310 -> { id: 4, corpo: "io:esternAta(0)", prec: 1120 0x460, rax: 0 0x0 },
100190 -> { id: 5, corpo: "utente:start(0)", prec: 4294967295 0xffffffff, rax: 8 0x8 },
1000d0 -> { id: 6, corpo: "utente:mioproc(10)", prec: 10 0xa, rax: 0 0x0 },
1003d0 -> { id: 7, corpo: "utente:mioproc(11)", prec: 55 0x37, rax: 0 0x0 },
100490 -> { id: 8, corpo: "utente:mioproc(12)", prec: 5 0x5, rax: 0 0x0 },
null <repeats 1015 times>
```

process dump:

```

livello      : utente
corpo        : utente:start(0)
-- pila sistema (000000fffffd8 → 453fd8)
rip          : 0xffff80000000204 <getpr
cs           : [SEL_CODICE_UTENTE]
rflags       : [IF SF IOPL=sistema]
rsp          : 0xfffffffffffffd8
ss           : [SEL_DATI_UTENTE]
-- contesto:
rax          : 0xa
rcx          : 0x3
rdx          : 0x1
rbx          : 0x0
rsp          : 0xfffffd8
rbp          : 0xfffffffffffff0
rsi          : 0xe
rdi          : 0xffff800000000e8
r8           : 0xffff800000103010
r9           : 0x0
r10          : 0x0
r11          : 0x0
r12          : 0x0
r13          : 0x0
r14          : 0x0
r15          : 0x0
cr3          : 0x0044f000
-- prossima istruzione:
file: utente/utente.s function: None
129

```

⚠ possibili ottimizzazioni: possiamo non chiamare salva_stato e carica_stato se la primitiva non modifica la variabile esecuzione. Possiamo in questo modo restituire un percorso in %RAX → Non voleva farci sentire ed ha poco senso ed inoltre il chiamante può trovare nei registri scratch dei valori lasciati dal sistema. Inoltre, queste sono utili in caso di errori

↓

Modifichiamo la primitiva in modo che ci restituisca la precedenza di un processo di cui forniamo noi l'identificatore

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

```
natl id = activate_p(mioproc, 13, 10, LIV_UTENTE);
activate_p(mioproc, 14, 1, LIV_UTENTE);
natl prec = getprec(id);
```

```
extern "C" void c_getprec(natl id) → potrebbe essere errato
{
    esecuzione->contesto[I_RAX] = proc_table[id]->precedenza;
}
```



Ottieniamo in tal caso un errore nel sistema perché in proc-table all'inizio c'è null

↓

Vogliamo prevenire questa tipologia di errori, quindi controlliamo prima il percorso

```
extern "C" void c_getprec(natl id)
{
    if (proc_table[id]) {
        natl prec = proc_table[id]->precedenza;
        esecuzione->contesto[I_RAX] = prec;
    } else {
        esecuzione->contesto[I_RAX] = (natl)-1;
    }
}
```

→ L'utente può ancora fare qualunque cosa e quindi può accedere ad un indirizzo qualiasi in memoria purché sia accessibile (quindi anche fuori della proc-table)

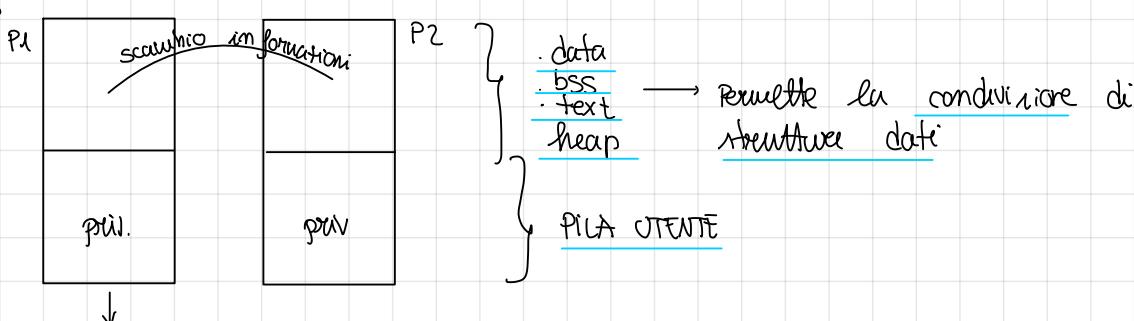
Aggiungiamo

```

if (id >= MAX_PROC) {
    log(LOG_WARN, "id %d non valido (max %d)", id, MAX_PROC);
    c_abort_p();
    return;
}

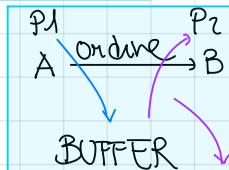
```

- È necessaria la carica stato in quanto execuzione viene modificata
- SENATORI
Dobbiamo prevedere che gli utenti ponano organizzare i programmi in molti processi.
Per questo dobbiamo prevedere una memoria condivisa in cui i processi ponano sovra
scrivere



Questo crea problemi di interfaccia che in questo caso asengono perciò in un modo qualsiasi → non possono dare agli utenti delle primitive per usare `click()` e `sti()` perché non ci fanno dell'utente

Vogliamo quindi che ci sia un principio di mutua esclusione che riguarda le sezioni critiche delle strutture dati condivise. È inoltre possibile che sia richiesto invece una sincronizzazione:



: tipico nel caso in cui un processo produce dati per l'altro

P2 deve leggere dal buffer solo dopo che P1 ci ha scritto e questo non deve ricucchiare il buffer finché P2 non ci ha letto

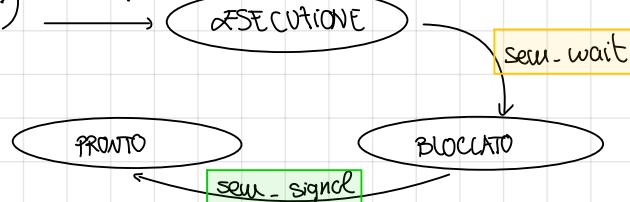
semaphori:



scatola che contiene operazioni → inserire un gettone
gettoni → prelevare un gettone: STOP fino a che qualcuno non ne inserisce uno

Dobbiamo stabilire il numero di rotoli, i gettoni presenti inizialmente e chi può prendere e lasciare i gettoni

mutua esclusione: immaginiamo di avere un luogo in cui puoi andare una persona per volta. Per risolvere il problema immaginiamo che chi deve andare in luogo deve avere un gettone e mettendo in tabù un gettone nello scatola. Ad un momento possono essere più persone → per questo riguarda i processi, se queste "non trovano il gettone", rimane nello stato bloccato: il processo non può andare avanti e non è pronto perché prima della sua esecuzione deve ricevere un evento (processo mette il gettone)



Se ci sono tanti processi bloccati si sceglie quello con precedenza più alta. La soluzione è di tipo cooperativo: tutti i processi devono eseguirsi. È una soluzione di livello che è facile implementare. Comunque quanto gli eventuali errori sono locali e relativi a quelli interrotti dalla interruzione.

Sincronizzazione: In termini di mattei, ne prendiamo una all'inizio vuota ed A interro il gestore della fine del processo e B lo prende quando inizialmente A ha il gestore.

Forniscono all'intente 3 primitive:

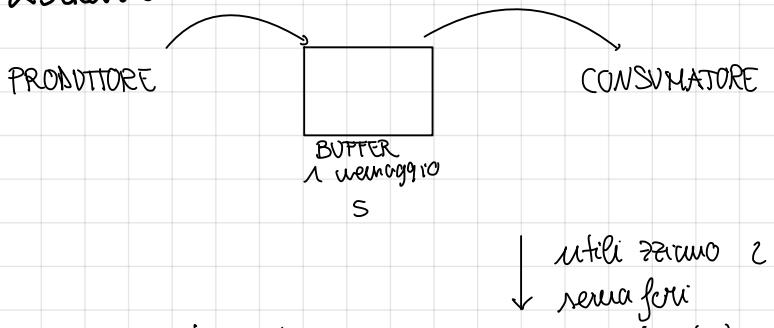
sem - init (nato v) → blocca un nuovo semaforo con v gestioni iniziali e ne restituisce l'identificatore

sem - wait (id) → prende un gestore da ID. Blocca il processo solo se la matraccia è vuota

sem - signal (id) → incrementa un gestore e quindi potrebbe far partire un processo da bloccato a pronto o in esecuzione se ha priorità maggiore cui processi pronti. Il processo attualmente in esecuzione viene messo in pratica (preemption)

Verde: un gestore; Rosso: nuovo gestore

Esercizio



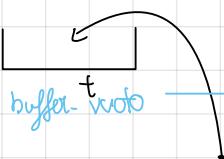
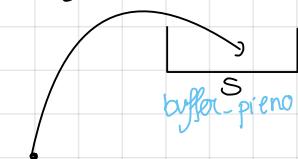
sem - wait (t)

PRODUCCO

sem - signal (s)

dovranno garantire che il consumatore non legga mai due volte lo stesso messaggio e che il produttore non sovrascriva il messaggio prima che venga letto dal consumatore

: faccio una sorta di ls DAV - / RFD



sem - wait (b.v.)
PRODUCCO

sem - signal (b.p.)

sem - wait (b.p.)
CONSUMO

sem - signal (b.v.)

Sincronizzazione:

Pa A semaforo A

↓ ↓ semaforo A
Pb B sem - signal (A - eseguita)

sem - wait (A - eseguito)
B

mutua esclusione:

P1 P2 ... Pn

sem - wait (libero)

A1 A2 An

sem - signal (libero)

realizzazione (struttura dati):

```
int contatore  
des proc *puntatore
```



Le primitive che utilizzano esclusione in intercomunicabili per i semafori non possono creare problemi cronici non

```
#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)  
#define TIPO_GP 0x28 // getprec
```

```
a_sem_wait:  
.cfi_startproc  
.cfi_def_cfa_offset 40  
.cfi_offset rip, -40  
.cfi_offset rsp, -16  
call salva_stato  
call c_sem_wait  
call carica_stato  
iretq  
.cfi_endproc  
  
.extern c_sem_signal
```

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

? struttura dati
Semaforo vero
e proprio

// Usiamo due insiemi separati di semafori, uno per il livello utente e un altro per il livello sistema. I primi MAX_SEM semafori di array_des sono per il livello utente, gli altri MAX_SEM sono per il livello sistema.
des_sem array_des[MAX_SEM * 2];

sem_ini

```
extern "C" void c_sem_ini(int val)  
{  
    natl i = alloca_sem();  
    mom si possono distinguere  
    if (i != 0xFFFFFFFF)  
        array_des[i].counter = val;  
    imposta new i-errore con valore utente  
    esecuzione->contesto[I_RAX] = i;  
}  
    restituisce indice semaforo
```

```
natl alloca_sem()  
{  
    int liv = liv_chiamante();  
    natl i; distingue semafori tra utente e sistema  
    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;
```

sem_wait

```
extern "C" void c_sem_wait(nat1 sem)
{
    // una primitiva non deve mai fidarsi dei parametri
    if (!sem_valido(sem)) { → controllo che il parametro
        flog(LOG_WARN, "semaforo errato: %d", sem);
        c_abort_p();
        return;
    } → puntatore a semaforo specificato
    des_sem *s = &array_des[sem];
    s->counter--; → può diventare negativo
    if (s->counter < 0) { → non c'erano gettoni
        inserimento_lista(s->pointer, esecuzione);
        schedulatore(); → ▲ NO cambio processo
    } il processo viene inserito in una lista perché
    deve contendere
}
```

sem - signal

```
extern "C" void c_sem_signal(nat1 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_des[sem]; → inseriamo un gettone
    s->counter++;
    if (s->counter <= 0) { → c'è qualcuno che aspetta
        des_proc* lavoro = rimozione_lista(s->pointer);
        inspronti(); → se ci sono processi con precedenza maggiore
        inserimento_lista(pronti, lavoro); → si occupano di fare
        schedulatore(); → eventuali preemption
    }
}
```

Non toglie dall'esecuzione un eventuale processo con lo stesso livello di precedenza.

ESEMPIO:

```
void prod(natq a)
{
    for (int i = 0; i < 10; i++) {
        sem_wait(buffer_vuoto);
        buffer++; // nuovo messaggio
        sem_signal(buffer_pieno);
    }
    terminate_p();
}

void cons(natq a)
{
    for (int i = 0; i < 10; i++) {
        sem_wait(buffer_pieno);
        printf("ho ricevuto il messaggio: %d\n", buffer);
        sem_signal(buffer_vuoto);
    }
    pause();
    terminate_p();
}
```

```

void main()
{
    buffer_pieno = sem_ini(0);
    buffer_vuoto = sem_ini(1);
    activate_p(prod, 0, 10, LIV_UTENTE);
    activate_p(cons, 0, 20, LIV_UTENTE);
    terminate_p();
}

```

} funziona anche se cambiamo la priorità

```

ho ricevuto il messaggio: 1
ho ricevuto il messaggio: 2
ho ricevuto il messaggio: 3
ho ricevuto il messaggio: 4
ho ricevuto il messaggio: 5
ho ricevuto il messaggio: 6
ho ricevuto il messaggio: 7
ho ricevuto il messaggio: 8
ho ricevuto il messaggio: 9
ho ricevuto il messaggio: 10
Premere un tasto per continuare - ↵

```

debugger:

```

processi: 2
esecuzione: [6, 10]
pronti: [0, DUMMY]
sem[1]: { -1, [7, 20] }
p_sospesi:

```

→ Code di ciascun reinforce

delay: sospende un processo per un certo tempo

```

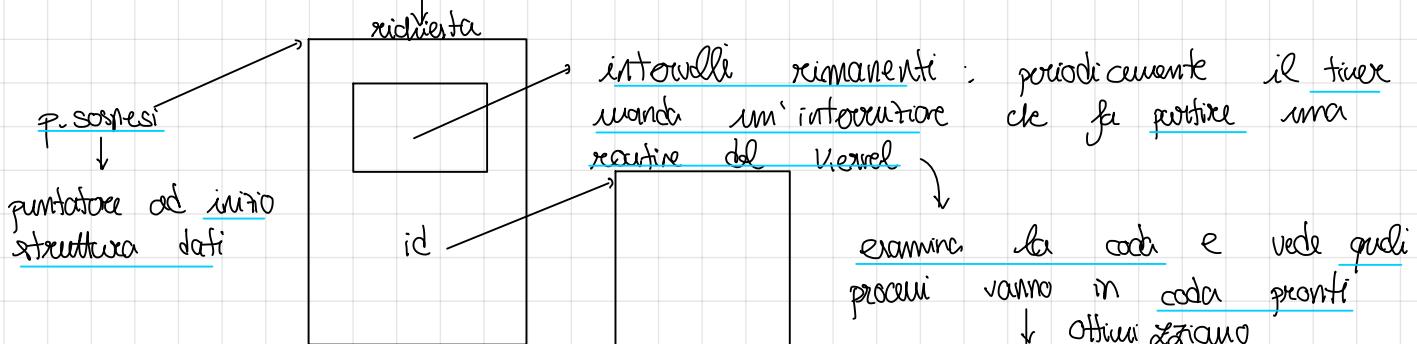
#include <all.h>

void main()
{
    for (int i = 0; i < 10; i++) {
        printf("hello, world\n");
        delay(100);
    }
    pause();
    terminate_p();
}

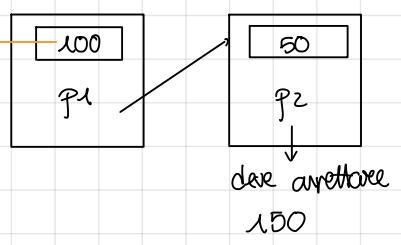
```

100 è definito in intervalli di tempo
definiti attraverso il timer
è vero da attendere: passaggio del tempo

Nel mentre può andare in esecuzione altro
| coda in struttura dati



Memorizza quante inter-
valli im più mancano
rispetto al precedente



Se arriva un processo che richiede di cancellare uno di quelli presenti questo viene messo in testa e viene aggiornato il contatore del processo che è scolato in seconda posizione. Quando un processo chiama delay deve capire dove va meno facendo lo scorciatoio della lista. Se la somma dei contatori è < del numero di intervalli richiesti si continua a correre, altrimenti viene inserito ed aggiornato il contatore del processo successivo. Quando si corre, il contatore del processo successivo viene sempre decrementato notificando ogni volta il contatore di ciascun processo "affievolato".

Il funzionamento è facilitato in quanto deve decrementare solo il primo contatore ed eventualmente inserire in lista pronta estraendolo dalla testa.

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

→ Routine che deve essere eseguita periodicamente (interruzione esterna)

```
.extern c_driver_td
driver_td:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_driver_td (*)
    call apic_send_EOI
    call carica_stato
    iretq
    .cfi_endproc
```

Dove andare in esecuzione il processo con quindi eventualmente maggiore priorità, che deve essere cambiato (preemption).

```
a_delay:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call c_delay
    call carica_stato
    iretq
    .cfi_endproc
```

```
extern "C" void c_delay(natl n)
```

```
    richiesta *p;
    p = new richiesta;
    p->d_attesa = n;
    p->pp = esecuzione;
```

```
struct richiesta {
    natl d_attesa;
    richiesta *p_rich;
    des_proc *pp;
```

```
    inserimento_lista_attesa(p);
    schedulatore();
}
```

→ Algoritmo visto prima

```
void inserimento_lista_attesa(richiesta *p)
{
    richiesta *r, *precedente;
    r = p_sospesi;
    precedente = nullptr;

    while (r != nullptr && p->d_attesa > r->d_attesa) {
        p->d_attesa -= r->d_attesa;
        precedente = r;
        r = r->p_rich;
    }
    p->p_rich = r;
    if (precedente != nullptr)
        precedente->p_rich = p;
    else
        p_sospesi = p;
    if (r != nullptr)
        r->d_attesa -= p->d_attesa;
}
```

(*)

```

extern "C" void c_driver_td(void)
{
    inspronti();
    if (p_sospesi != nullptr) {
        p_sospesi->d_attesa--;
    }

    while (p_sospesi != nullptr && p_sospesi->d_attesa == 0) {
        inserimento_lista(pronti, p_sospesi->pp);
        richiesta *p = p_sospesi;
        p_sospesi = p_sospesi->p_rich;
        inserire_gliel
        con contatore
        D. in cima a
        coda }
        schedulatore();
    }
}

```

decrementa i contatori
ad ogni interruttore

inserire quelli con contatore
D. in cima a coda

punto a nuovo primo
elemento : funzione Standard
su lista

Esempio

```

esecuzione: [6, 10]
pronti: [0, DUMMY]
p_sospesi: {100, [5, MAX_PRIO]}

```

introducendo il timer abbiamo il non determinismo: le interruzioni del timer sono imprevedibili → i programmi devono avere corretti guadagni sia la relativa deklaration

⚠ NOTA: new e delete → funzioni di libreria del C++

```

void* operator new(size_t size)
{
    return alloca(size);
}

```

```

void operator delete(void *p)
{
    dealloca(p);
}

```

funtionano in quanto all'inizializzazione del sistema vi inizializza lo heap (1 MB) → le informazioni sull'allocation della memoria vengono scritte nello heap stesso:



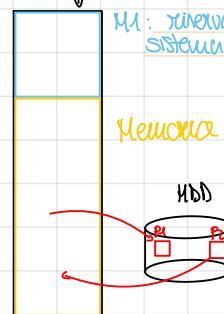
info memoria: spazio consecutivo in memoria disponibile

Le parti libere sono contenute in una lista di cui alloca fa lo scorrimento fino a trovare una sufficientemente grande.

alloca(100): nuovo descrittore e sumatore a zona di memoria allocata (inizio)
dealloca(p): dal puntatore si ottengono le informazioni necessarie al descrittore semplice - muovendo facendo una notifica

La paginazione

Stato di un processo = stato CPU + stato memoria
 il cambio di processo implica il cambio del processo in esecuzione
 della memoria privata con quella del processo entrante



→ prima soluzione rudimentale

↓
costo invincibile

vogliamo

? MU

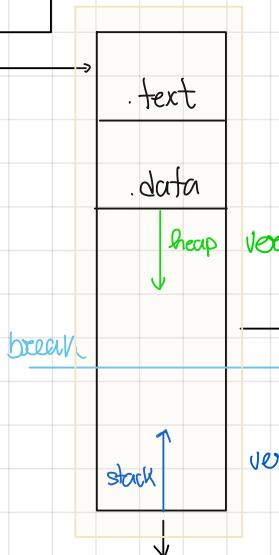
P1

P2

P3

→ Più processi in memoria: evito accesso ad HDD

Memoria di un processo



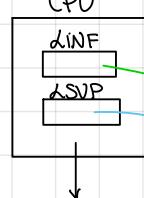
verso indirizzi crescenti

→ **Rischio di collisione:** non c'è una soluzione efficace perché le sezioni, cercando, provano ottenere il **break**
 verso indirizzi decrescenti

Allocata con dimensione massima

⚠ Che problema c'è ad avere più processi in memoria? Un processo può sovrapporre in una parte di memoria appartenente ad un altro processo
 ↓ Necesario più **hardware**: soluzioni

1. **registri aggiuntivi**



→ Viene modificata la CPU perché può controllare processi attualmente in esecuzione e vedere se rispetta i "confini" della sua memoria

Contenuto non modificabile dell'utente dentro descrittore di processo: cambiato al cambio di processo

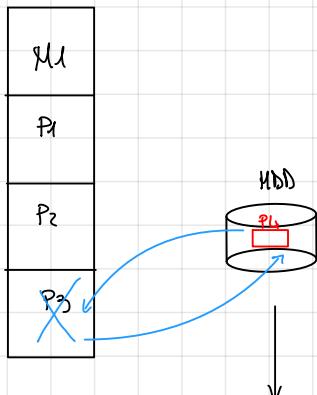
Saranno gli indirizzi a cui vengono **risolte** al momento della loro creazione

1. allocati i processi vengono decisi solo caricatore rilocalizzante: ma le implementazioni di rilocazione del linker per assestarsi

gli indirizzi di memoria del corrente programma

2. Scrive tutto il programma con indirizzi relativi (non dipendenti da posizione)

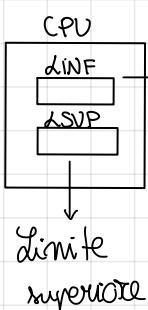
⚠ Problema se abbiamo più processi che memoriano



→ tratta la memoria utente come una cache
copiando in HDD il processo non titolato
↓
Molti meno frequenti di farlo ogni sostituzione

Se ora abbiamo in memoria P4 e ci libera spazio per P3, non possiamo ricaricarlo da un'altra parte perché, ad esempio, gli indirizzi coricati dal processo in pila sono relativi alla posizione precedente del processo in memoria e non possono essere ricomposti. → Dobbiamo quindi ricaricare i processi allo stesso indirizzo da cui sono stati tolti

↓ Modifica



→ interpretato come base

↓

Ogni volta che c'è un ceppo in memoria, l'indirizzo generato dal programma in esecuzione viene sommato alla base prima di accedere alla RAM

↓

P.e.: $a \rightarrow \text{base} + a \rightarrow \text{RAM}$ risolve

Adesso tutti gli indirizzi sono relativi alla base, quindi basta modificare quella: si mette un punto della RAM dove c'è spazio

↓

soluzione trasparente: il programmatore può collegare sempre il suo programma all'indirizzo 0 e non deve renderlo indipendente dalla posizione

↓ perché la base viene sommata automaticamente

⚠ PROGRAMMAZIONE = decidere cosa c'è in memoria inizialmente: lo posso fare indipendentemente dalla base (che può essere qualsiasi). Se RAM vede sempre l'indirizzo risultante dalla somma dell'indirizzo specificato dal programma con la base → il processo ha quindi un proprio spazio numerato e non ci interessa sapere quale è perché i suoi indirizzi variano da 0 al massimo: memoria virtuale indipendente dagli altri processi

La rilocazione viene fatta durante l'esecuzione del programma della CPU.

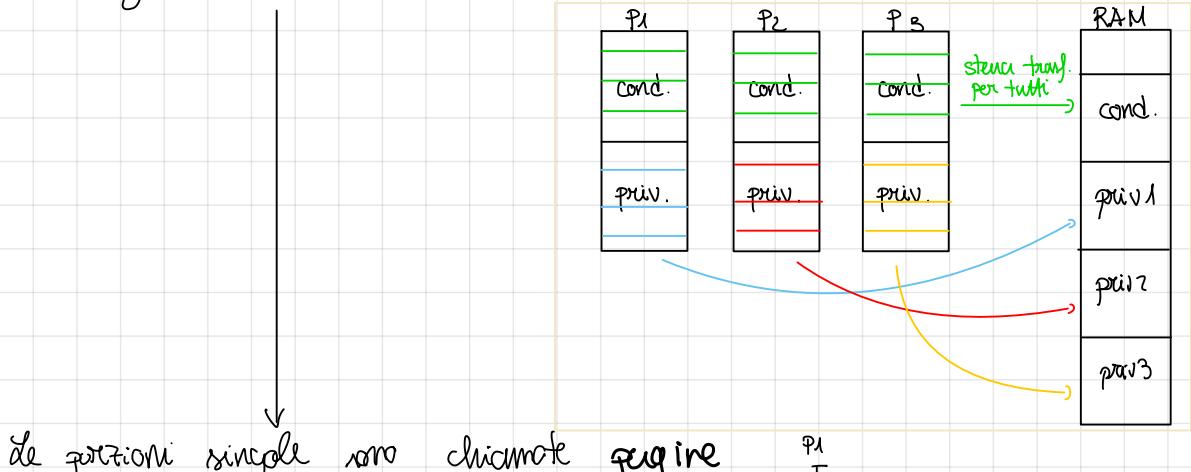
Difetto: la trasformazione è poco fluida. Se dei processi toccano, i processi adiacenti non vengono modificati e che può portare ad avere RAM non sufficiente soluzione costosa → si produce fragmentation (RAM libera spazio a tratti), processi in attesa nonostante lo spazio libero totale in RAM. Ricompattamento attraverso lo spostamento dei processi.

Difetto 2: da solitò permette di includere here i processi ma questo rende difficile farli

comunicare attraverso le sezioni a comune.

↓ Pagine

Permette di comporre la memoria di un processo in pezzi più piccoli e ciascuno di cui metterlo dove voglio sentendo che il processo se ne accorga. È possibile fare questo per ciascuna sezione ovunque ci sia spazio. Risolve il problema della fragmentazione e della condivisione.



Le sezioni singole sono chiamate pagine

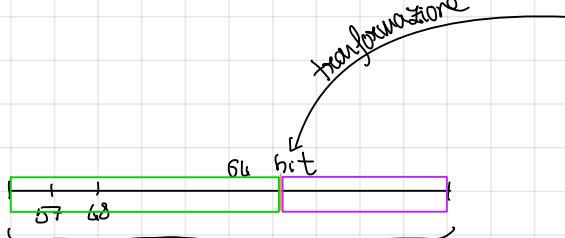
Spazio di indirizzamento del processo diviso in regioni notevoli della stessa grandezza (4 KiB per Intel) → pagine

Spazio di indirizzamento fisico

→ Frame (cornice): 4 KiB

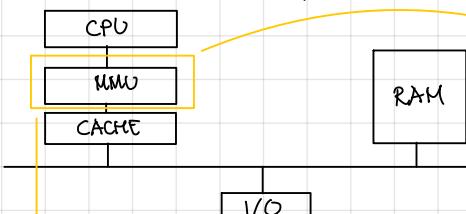


A ciascuna pagina può essere associato un frame (o meno) e a ciascun frame possono essere associate più pagine



L'indirizzo virtuale ancora da trasformare può essere diviso in 2. La parte più significativa è il numero di pagina mentre la meno significativa è l'offset all'interno della pagina, che su 4 KiB è grande il 2 hit. Una pagina può essere mappata su qualsiasi frame sostituendo il numero di pagina con il numero di frame e lanciando l'offset inviato (corrispondente 1:1) con una funzione di trasformazione → linea sua gestibile dal sistema ed applicabile dall'hardware: struttura dati esempio a [n. pagina]: n. frame

↓ nuovo dispositivo: memory management unit



tutti gli indirizzi generati dalla CPU vengono tradotti dalla MMU e quindi tutti gli indirizzi virtuali sono indirizzi fisici. Dopo la MMU ci sono gli indirizzi fisici

Dove everci era tabella a per ogni processo. Questa aiuta ad indire perché il processo può accedere solo agli indirizzi facenti parte del codominio della funzione. Se si entra inoltre permette una più indiretti condivisori dell'hardware applicando virtuali. Viene inoltre risolto il problema della frammentazione potendo disporre di pagine libere. La MMU inoltre li trasci in ogni caso in memoria e quindi può essere utilizzata per altre copie:

nº pagina $\xrightarrow{\text{avocato}}$

traduzione (nº frame)

Q: specifica se traduzione esiste o no → il processo non può accedere ad alcune pagine, pena eccezione: utile per puntatori null per restituire errore e per proteggere lo stack e lo heap attraverso l'inserimento di guard pages che li intervallano

r/w: protegge le pagine della scrittura (.text e R/O). Serve di proteggere il programma e di condividere settori.

U/R: livello necessario per accedere a pagine: proteggere dati sistema perché dare info di sistema devono essere mappate per permettere al processo di eseguire delle funzioni (interazioni). Per ottimizzare consideriamo tutta la memoria mappata

pcd
pwnt

A/R

Il bit PCD permette di dichiarare la cache e quindi far partire l'indirizzo sul bus direttamente. Il bit pwt (page write through) permette la modifica write through per la scrittura in cache. Ha senso dichiarare la cache per l'I/O ed impostare la modalità wt per la memoria video. Il bit A/R non aggiornati della MMU stessa → A (accen): se viene fatta traduzione per quel numero di pagina

D: se l'accesso è fatto in scrittura

→ permettono di redigere statistiche da parte del sistema

ESEMPIO di programma con MMU attiva

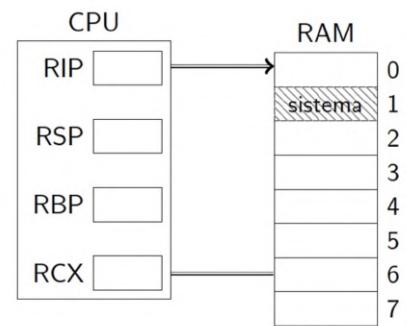
```
char buf[0x2000] = { 2, 6, -1, 200, ...,
    15, 3, -32, 1};
int main()
{
    int sum = 0;
    for (int i = 0; i < 0x2000; i++)
        sum += buf[i];
    return sum;
}
```



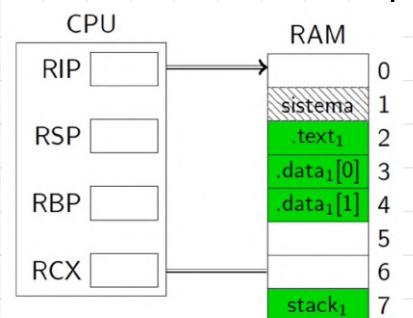
	.text	.global main
2000	main:	pushq %rbp
2001		movq %rsp, %rbp
2004		subq \$8, %rsp
2008		movl \$0, -8(%rbp)
200f		movl \$0, -4(%rbp)
2016	for:	cmpq \$0x2000, -4(%rbp)
2013		jge fine
2020		movslq -4(%rbp), %rcx
2024		movsbl buf(%rcx), %eax
202b		addl %eax, -8(%rbp)
202e		addl \$1, -4(%rbp)
2032		jmp for
2034	fine:	movl -8(%rbp), %eax
2037		popq %rbp
2038		ret
	.data	...
3000	buf:	.byte 2, 6, -1, 200
		...
4ffc		.byte 15, 3, -32, 1

Semplificazioni

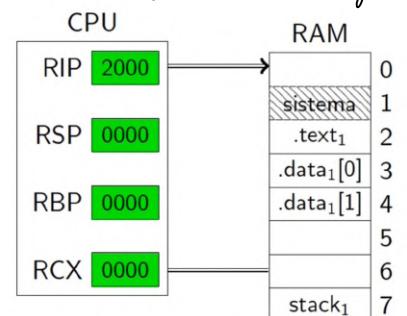
1. Spazio di memoria = 32 KiB quindi indiritti da 0000 a 7fff
2. Spazio diviso in 8 pagine da 4 KiB
3. Pagine 0 e 1 riservate al sistema
4. Codice nella pagina 2, variabile buf nelle pagine 3 e 4 e la pila nella pagina 7



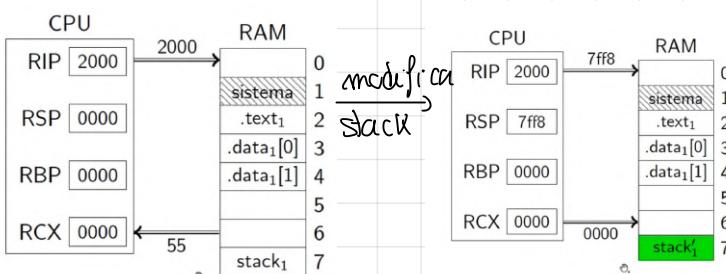
prima che il programma parte solo sistema
dopo caricamento



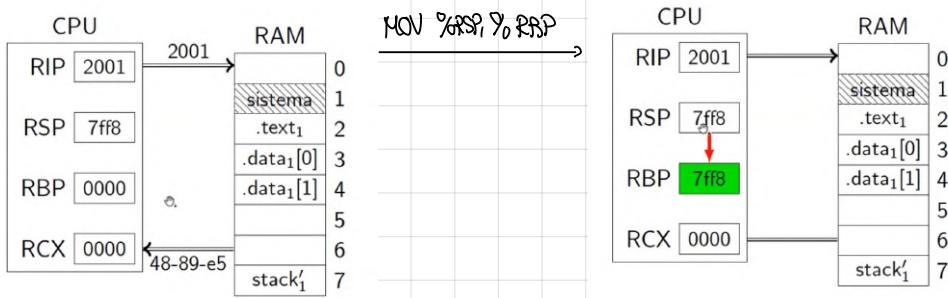
con initializzazione registri



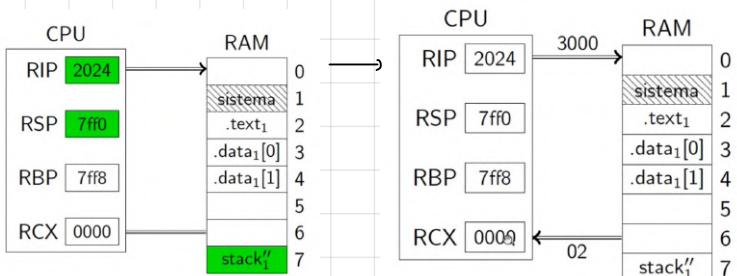
Punto 1



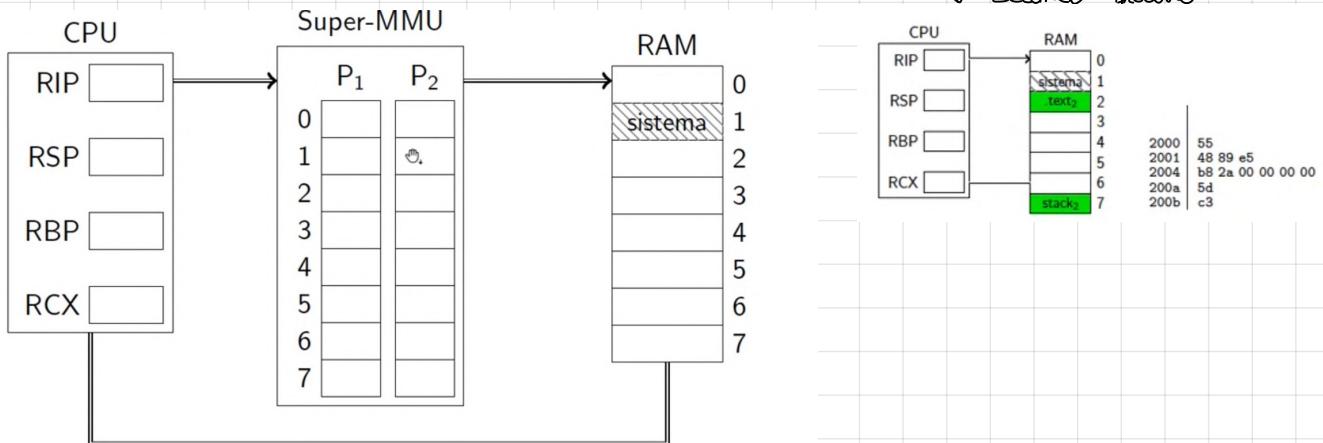
Punto 2



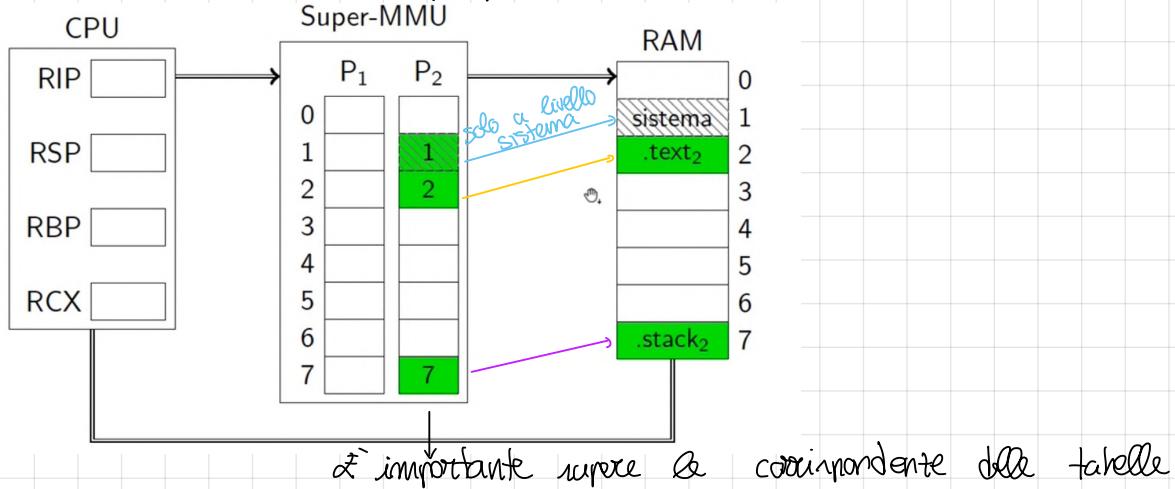
Punto 2c



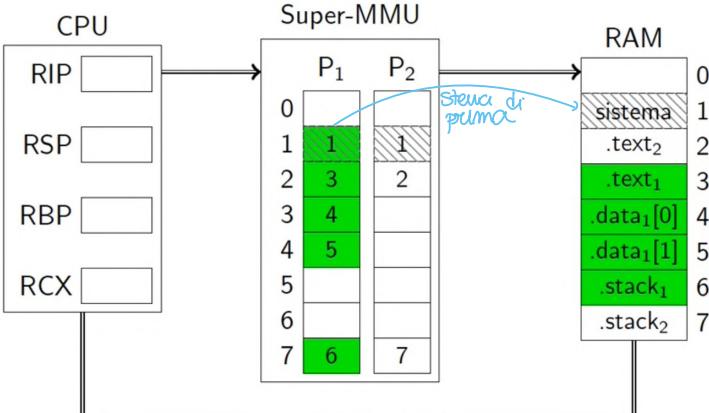
Il processo non sa che ci sono altri processi che condividono, com qui la RAM secondo questo



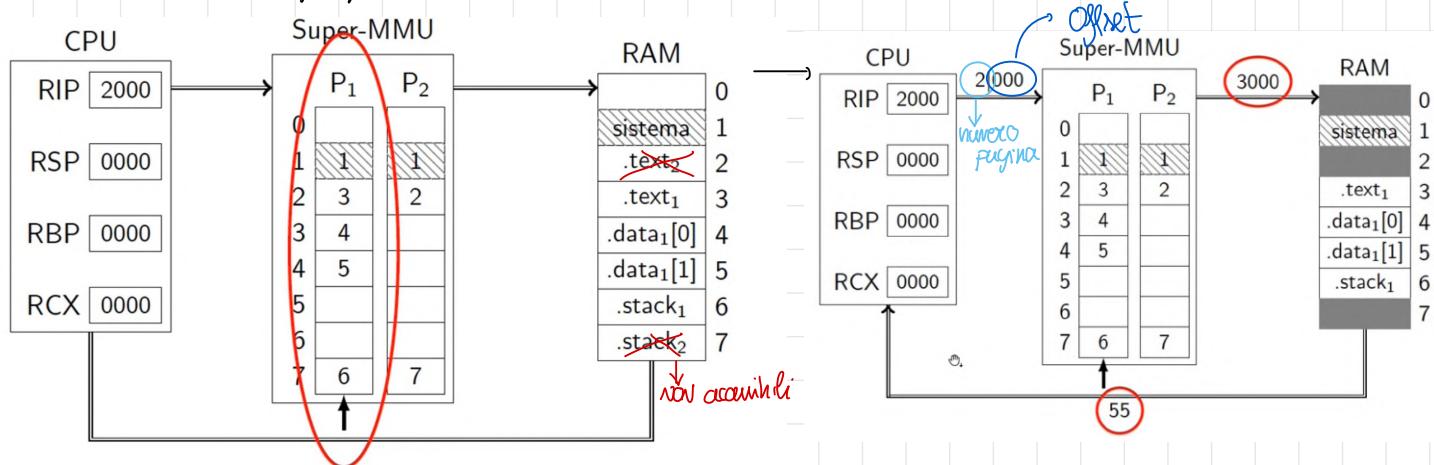
Atto di rovinamento



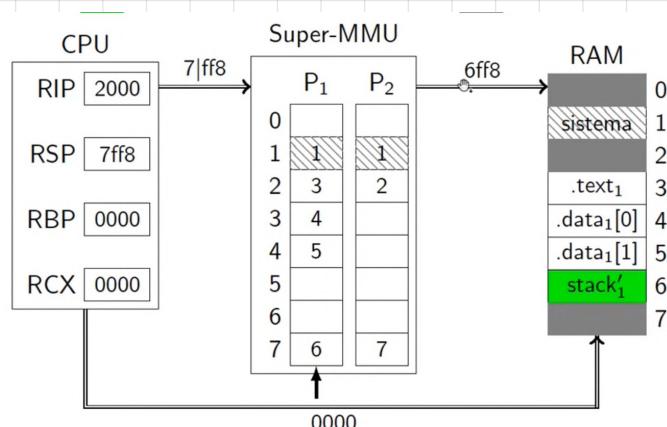
Caricando il secondo processo



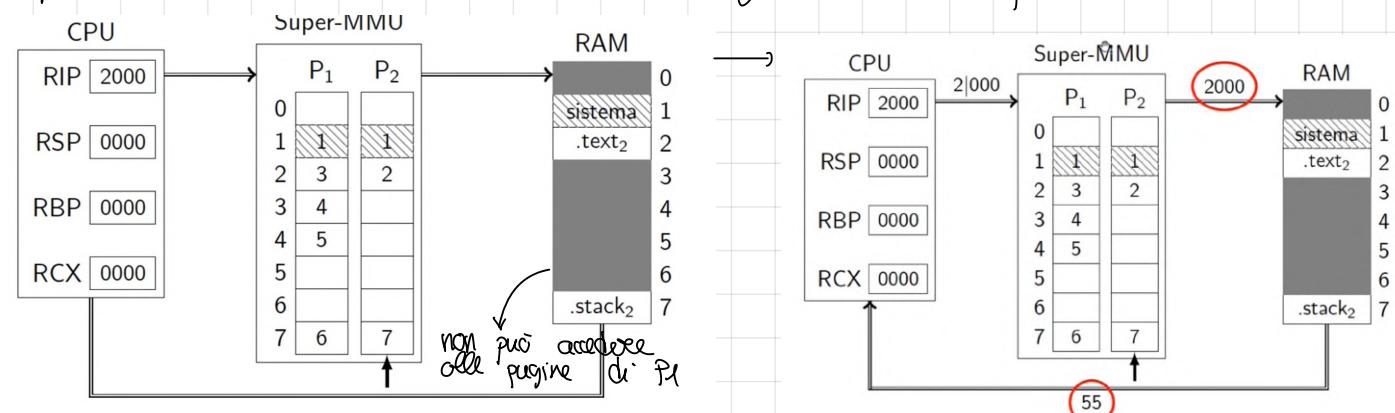
Esecuzione del programma → deve rimanere invariata



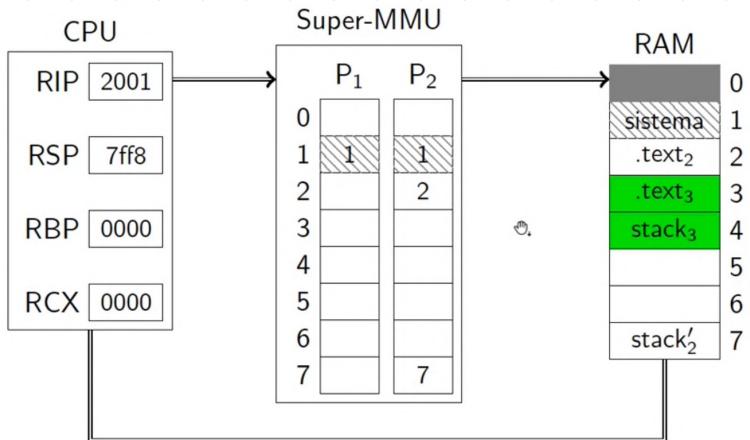
Stesso funzionamento di prima
(anche per passaggi successivi)



Supponiamo che ci sia un'interruzione che fa cambiare processo



Supponiamo che Pe venga fatto



E che, dopo la terminazione di P2 si voglia ricaricare P1: domani fare suoi iniziali sono occupati → non importa: ponono essere rallocati

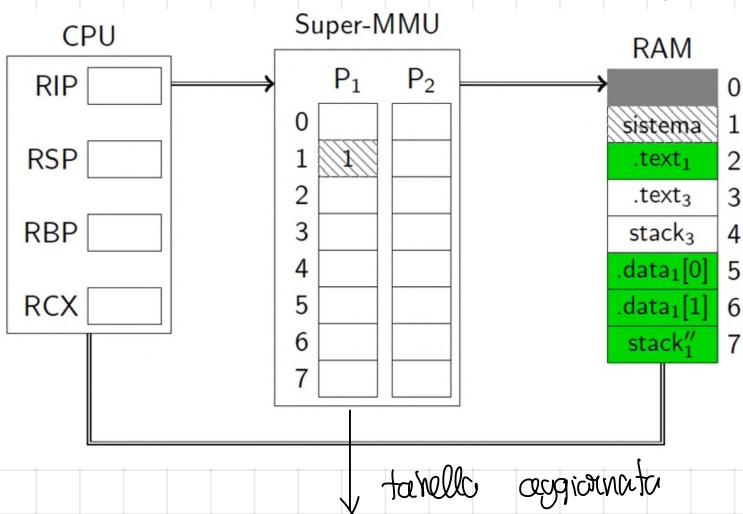
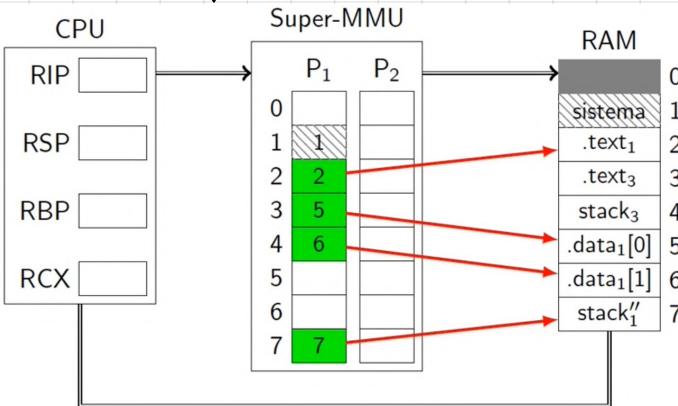
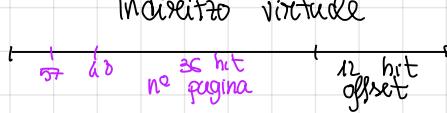


tabelle aggiornata



: Pe non si accorgere di nulla e quindi continua l'esecuzione normale

La super MMU non può essere realizzata in questo modo: Quanto è grande la tabella di corrispondenza?



→ da tabella, nel nostro caso l'array, deve contenere ogni frame col ogni numero di pagine

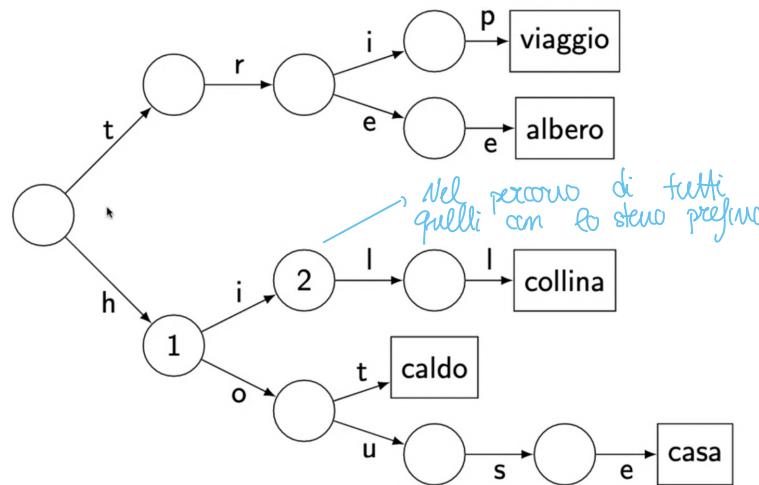
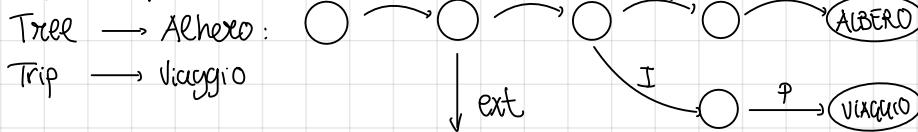
2^{36} righe che devono contenere il numero di frame che può arrivare a 52 hit e gli altri hit → approfondendo, ci servono 3 bufer per regola

$$2^{36} \cdot 2^3 = 2^{39} = 512 \text{ GiB} : \text{non è possibile}$$

Utilizziamo una nuova struttura dati: TRIE → permette di associare chiavi e valori esempio KEY: parole inglese VALUES: traduzione in italiano

Albero sui cui archi ci sono le singole lettere della chiave che guidano il percorso e nei nodi arrivano i valori

↓ Esempio



→ da chiave non è memorizzata in modo ma è lungo il percorso

array di puntatori
in questo caso indirizzati da codice ASCII

✓ de chiavi vuote non occupano spazio

Variazione: sui rami ci sono alcuni hit della chiave

↓
qhit, ahit, qhit, ahit
36 hit
n° pagine

analogo alle lettere di prima: in hue 8

↓
in ogni nodo c'è una tabella con 512 entrate
che puntano ad un'altra tabella. Nell'ultima tabella si trova la traduzione (n° di frame)

↓

↓

tabella di corrispondenza
da 512 GiB

Avendo che anche le altre colonne occupano spazio,
avremo più spazio occupato
rispetto all'array.

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

↓

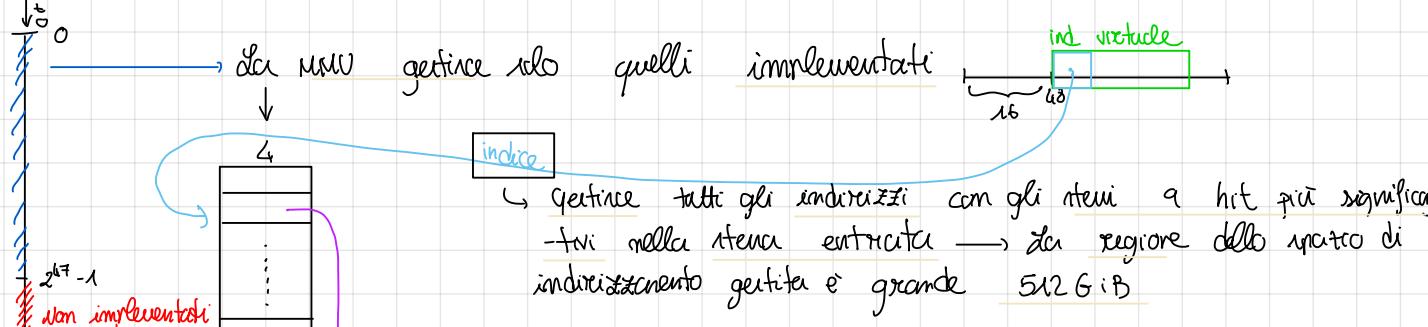
↓

↓

↓

↓

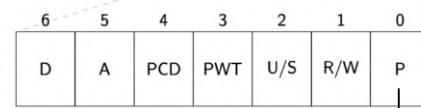
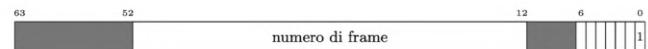
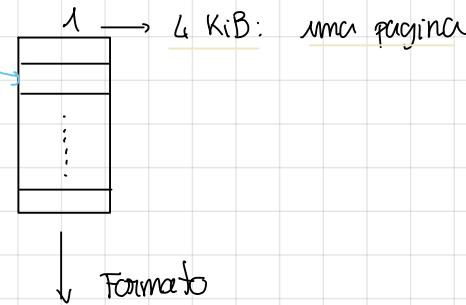
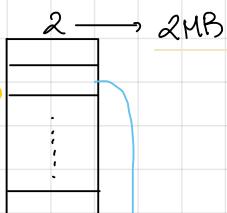
</



La gerarchia delle regioni permette di mem allocare molti utenti in caso di processi piccoli che occupano la stessa sezione di memoria

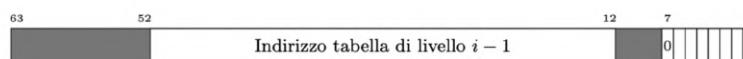
Per processi più grandi possono avere pagine più grandi. Ad esempio, unico pagina da 2 MiB utilizzando offset da 12 byte e facendo riconciliazione a livello. L'organizzazione permette inoltre di non avere tutta la struttura in memoria.

ciascuna pagina si occupa della traduzione di 1 GiB



Potrebbe non essere valida: in tal caso la MMU si diventerà della regione che può essere utilizzata per altri scopi dal programmatore

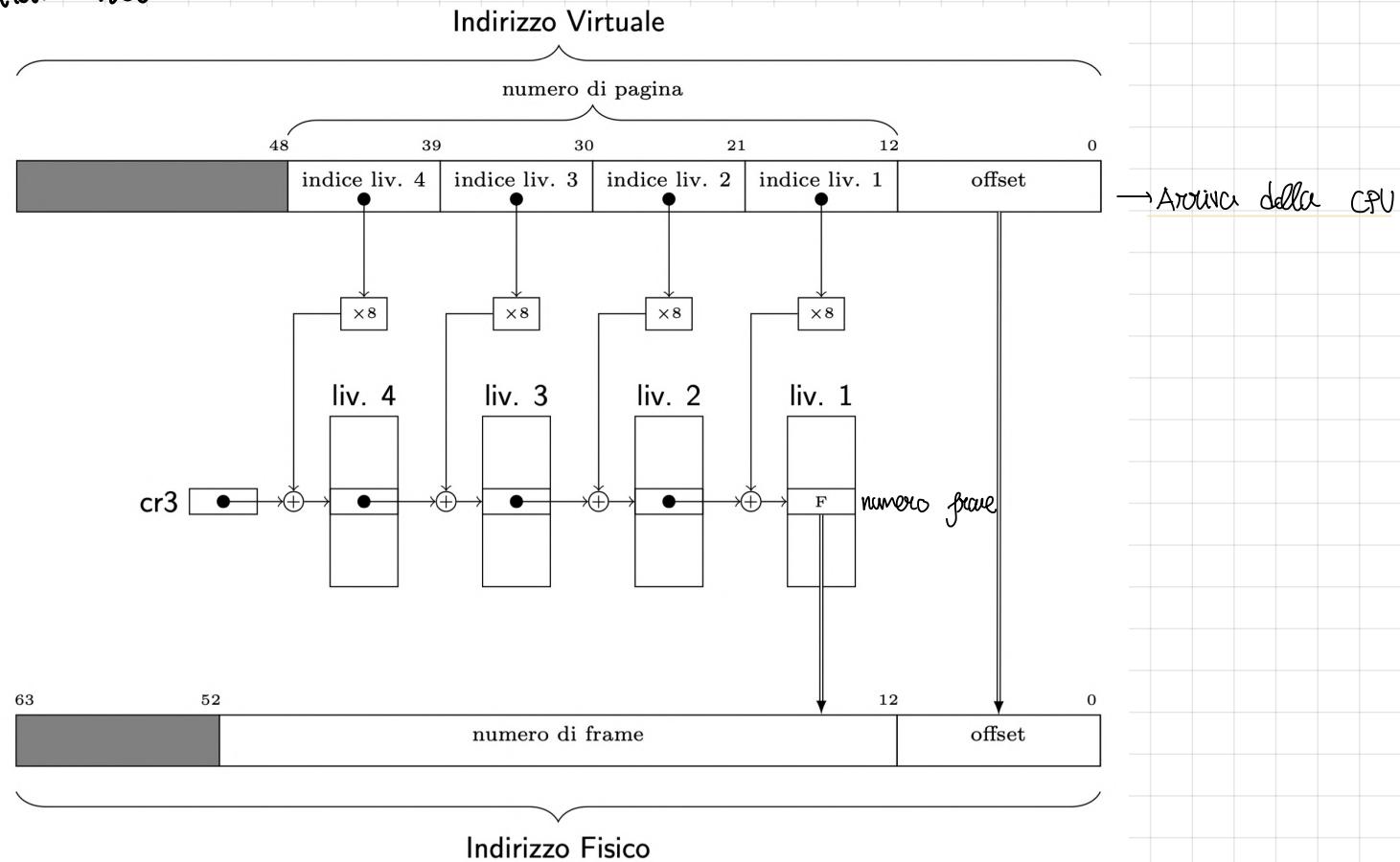
≠ da tab intermedia



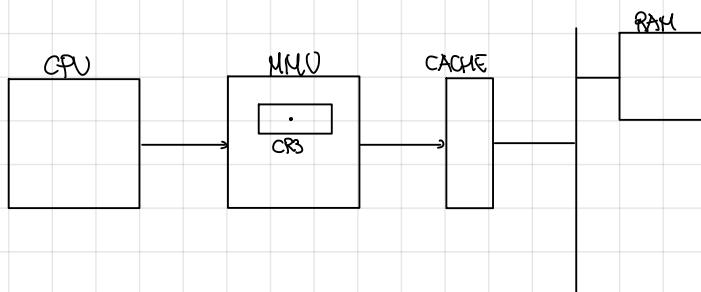
P=0 se manca la traduzione della sotto-regione: interrompe la traduzione

Il programma di scrittura deve avere concavo da tutti i hit
Stessa cosa di R/W

Ricordo



Avviciniamoci alla MMU sede



: contiene le tabelle → comodo pochi create di memoria e accorciati senza istruzioni speciali.
La MMU sa solo dove la tabella di livello 4, per sapere dove sono le altre fa accorci in memoria

La MMU deve sapere l'indirizzo fisico della tabella di livello 4. Anche questa contiene l'indirizzo fisico della tabella di livello 3 e così via

ricordiamo le righe di tali tabelle:



Mancano hit meno significativi: **limiti posizionamento**
in RAM → Avendo 12 zeri meno significativi le tabelle devono essere allineate naturalmente (a 4KiB) cache della

Moltre, gli accorci in memoria efficace perché avvia al numero di pagina direttamente F saltando i puntigli intermedi

Creazione di un altro di traduzione

```
#include <libce.h>

void main()
{
    pause();
}
```

→ modo: sistema intr: abilitati io: permesso [protezione] cr3: 0x00001000

contenuto corrente registro cr3: puntatore alla radice struttura di traduzione ~~per la~~ traduzione identità: numero frame numero pagina

↓ VM maps: riassunto altro di traduzione

da a [con comincia]
(gdb) vm maps
0000000000000000 S-000-000-000-000: U W PWT PS A D
0000000000200000 S-000-000-001-000: U W PS A D
0000000000400000 S-000-000-002-000: U W PS
0000000040000000 S-000-001-000-000: U W PWT PCD PS
00000000fec00000 S-000-003-766-000: U W PWT PCD PS A D
00000000fec00000 S-000-003-767-000: U W PWT PCD PS
0000000010000000 S-000-004-000-000: unmapped

2^{32} : Traduzioni per 4GB

+
(gdb) vm path 0 → percorso traduzione
cr3: 0x00001000: puntatore indirizzo
000: [WU--A--] 0x00002000: livello 3
000: WU--A-- → 0x00003000:
000: WUw-ADs → 0x00000000

implementazione traduzione identità per 8MB

.data
.align 4096

→ Allineamento naturale a 4 byte

↓ 8 MB

da 0000 0000 offset a 00080000

Scomposti in gruppi di 3 cifre: utilizziamo la base 8, quindi il numero di pagina sono 12 cifre ottime mentre l'offset è dato da 4 cifre ottime

in base 8: 000 000 000 000 000 : da

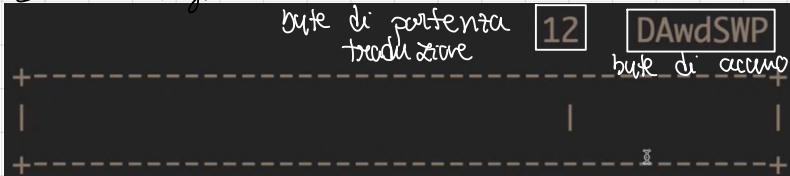
000 000 004 000 0000 : a → primo non tradotto
↓ tabella di livello 4

Avendo tutti 0, passano tutti da una sola entrata, il che implica la presenza di una sola tabella di livello 3. Anche in questa tabella tutti gli indirizzi saranno della prima entrata e quindi ci saranno un'unica tabella di livello 2. Nel caso di questa tabella avremo bisogno di 4 entrate (000, 001, 002, 003).

tab_4
0 -> tab_3
0 -> tab_2
0 -> tab_1_0
1 -> tab_1_1
2 -> tab_1_2
3 -> tab_1_3

.data
.align 4096
tab_4:
 .space 4096, 0
tab_3:
 .space 4096, 0
tab_2:
 .space 4096, 0
tab_1_0:
 .space 4096, 0
tab_1_1:
 .space 4096, 0
tab_1_2:
 .space 4096, 0
tab_1_3:
 .space 4096, 0

iniziali di traduzione delle tavelle
Sintesi reghe



tah 4

init_vm:

```
    movq $tab_3, tab_4 → posso sovraccoprire $tab_3 perché bootloader fa traduzione 1:1  
    movb $0b0000111, Swp → setto hit
```

tah 3

```
    movq $tab_2, tab_3  
    movb $0b0000111, tab_3
```

tah 2

```
    movq $tab_1_0, tab_2  
    movb $0b0000111, tab_2 → Grandezza rega  
    movq $tab_1_1, tab_2 + 8 → tah2  
    movb $0b0000111, tab_2 + 8
```

```
    movq $tab_1_2, tab_2 + 8 * 2  
    movb $0b0000111, tab_2 + 8 * 2  
    movq $tab_1_3, tab_2 + 8 * 3  
    movb $0b0000111, tab_2 + 8 * 3
```

tah 1 : contiene le traduzioni (riconducono, traduzione identità)

```
    mov $0, tab_1_0 → Allora tavella si avvia con indice 0  
    movb $0b0000111, tab_1_0
```

```
    mov $4096, tab_1_0 + 8 : seconda rega  
    Salvo 1 seguito da 12 zeri
```

```
    mov $(2 * 4096), tab_1_0 + 8 * 2 : terza rega
```

2 seguito da 12 zeri

per ricorrere tutte inizie

```
    movq $tab_1_0, %rdi  
    xorq %rax, %rax  
.Loop:  
    movq %rax, (%rdi)  
    movb $0b0000111, (%rdi)  
    addq $0x1000, %rax  
    addq $8, %rdi  
    cmp $last, %rdi  
    jb .Loop  
  
    ret
```

→ faccio le cose di prima per tutte le tavelle

```
#include <libce.h>  
  
extern "C" void init_vm();  
void main()  
{  
    init_vm();  
    pause();  
}
```

Al hexo generato
(gdb) vm tree &tab_4

0x00204000

000: WU----- → 0x00205000

000: WU----- → 0x00206000

000: WU----- → 0x00207000

000: WU----- → 0x00000000

001: WU----- → 0x00001000

002: WU----- → 0x00002000

003: WU----- → 0x00003000

004: WU----- → 0x00004000

005: WU----- → 0x00005000

006: WU----- → 0x00006000

007: WU----- → 0x00007000

010: WU----- → 0x00008000

011: WU----- → 0x00009000

012: WU----- → 0x0000a000

013: WU----- → 0x0000b000

014: WU----- → 0x0000c000

Per fare sì che questa traduzione venga usata scriviamo l'indirizzo di tab4
in cr3

```
// carica il registro cr3
// parametri: indirizzo fisico del nuovo direttorio
.global loadCR3
loadCR3:
.cfi_startproc
    movq %rdi, %cr3
    retq
.cfi_endproc
```

funzionamento

```
#include <libce.h>

extern "C" void init_vm();
extern natq* tab_4;
natq var = 3;
void main()
{
    init_vm();
    var = 5;
    pause();
}
```

(gdb) p &var
\$1 = (natq *) 0x205000 <var>
(gdb) vm path &var
cr3: 0x00206000
000: WU--A-- → 0x00207000
000: WU--A-- → 0x00208000
001: WU--A-- → 0x0020a000
005: WU----- → 0x00205000

scriviamo su toyhere il diretto di scrivere alla pagina contenente var

205	2	1
000/000/001	000/000/101	

movb \$0, tab_1_1 + 8 * 0b101

→ proviamo togliendo tutti i direct:

(gdb) vm path &var
cr3: 0x00206000
000: WU--A-- → 0x00207000
000: WU--A-- → 0x00208000
001: WU--A-- → 0x0020a000
005: RS---- b: 517 → manca traduzione

ottengo eccezione 14 (page fault)

exc_int_tipo_pf:

provo a vedere un click (oltre indirizzo virtuale tradotto nello stesso indirizzo di var)

4	3	2	1
000/000/000	000/000/000	000/000/011	777

: Lo vogliamo far corrispondere a var (0x205)

movq \$0x205000, tab_1_3 + 8 * 0777

+ flag

(gdb) vm path &var

cr3: 0x00206000

000: WU--A-- → 0x00207000

000: WU--A-- → 0x00208000

001: WU--A-- → 0x0020a000

005: WU----- → 0x00205000

(gdb) vm path 0x7ff000 i

cr3: 0x00206000

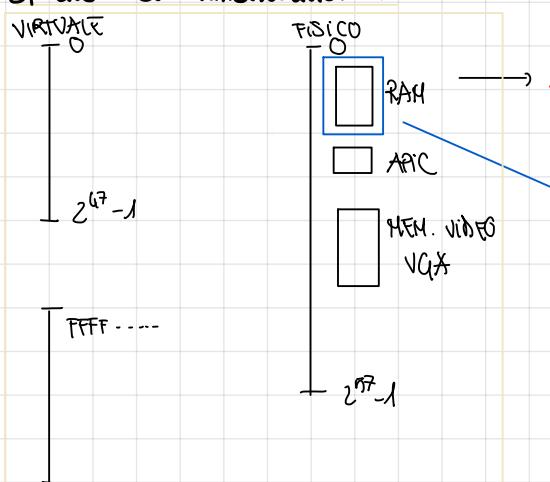
000: WU--A-- → 0x00207000

000: WU--A-- → 0x00208000

003: WU----- → 0x0020c000

777: WU----- → 0x00205000

spazio di indirizzamento



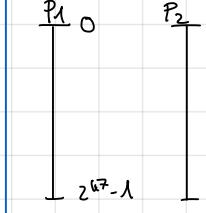
⚠ La CPU utilizza solo indirizzi virtuali

Nelle tabelle di traduzione generate dal sistema

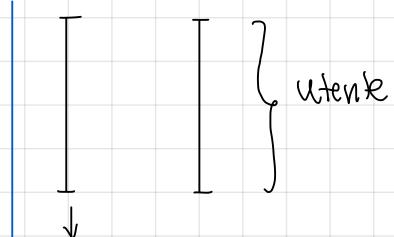
Ha bisogno di cercare, modificare e distruggere



Per vedere la lista dei occulti - p deve creare tutto l'elenco. Le tabelle vengono allocate in N2 perché i frame sono della dimensione giusta ed allineati nel modo giusto. Ricordiamo che tutti gli indirizzi non vengono tradotti dalla MMU e questo comporta che si può accedere solo agli indirizzi fusi appartenenti al codominio della funzione di traduttore → i frame devono far parte di questo codominio. Fatto che lo siano, il su deve avere l'indirizzo virtuale delle tabelle. Per puntare alle tabelle di livello inferiore infatti in ciascuna tabella deve essere presente il suo indirizzo fisico. Le tabelle sono alla MMU. → **RISERVAZIONE**: di default viene utilizzata la traduzione identità posta nella memoria virtuale di tutti i processi.



sistema → qua mettiamo la traduzione identità di default. Gli indirizzi tradotti sono quelli necessari ad indirettamente lo spazio di indirizzamento fisico denotato



Risolve il problema dei puntatori nelle tabelle e dell'accordo delle tabelle di livello 4. Il sistema non fa quindi distinguere tra indirizzi fusi e virtuali per le sue strutture.

In questa memoria virtuale dove essere presente anche la LDT perché gli indirizzi delle interruzioni sono infine la pila utente ed il fisico per risolvere tutti i problemi. **semplicifica** attivazione paginazione:

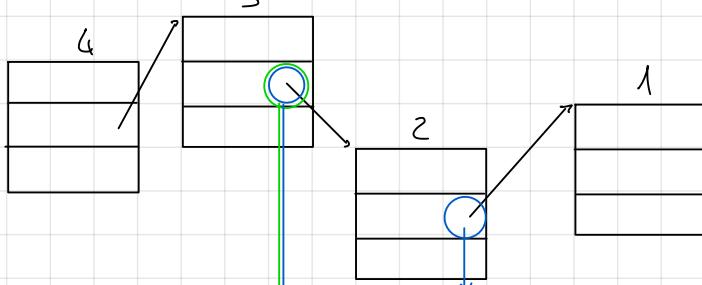


bit 1 = paginazione attivata
il sistema parte con la MMU disattivata e questa viene attivata successivamente
↓ momento attivazione

MOV \$1, %CRO

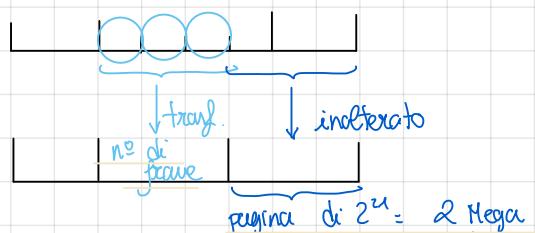
%RIP → contiene indirizzo fisico: al momento dell'attivazione viene interpretato come virtuale → il programma è banale utilizzando la traduzione identità

utilizzo di pagine più grandi



PS: page size → dice alla MMU che la traduttore è già lì e non c'è la tabella inferiore

Se pagine sono più grandi:



Se PS=1 qui la pagina è grande 1GB → la regione di 2Mega viene tradotta tutta nello stesso modo

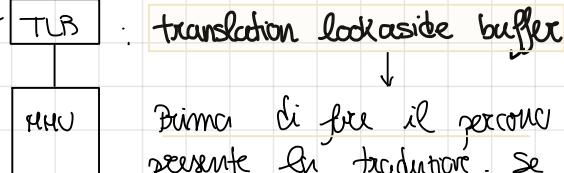
Poi avviene per tutte le pagine. Permette di risparmiare memoria e risparmiare tempo di ricerca. Lo svantaggio è la fragmentazione interna che si crea e quindi si fa solo dove possibile.

È utile in particolar modo per la traduzione identità in quanto permette di correttamente alla tabella di livello 3 con pagine di 1GB ciascuna.

⚠ PROBLEMA: lentezza

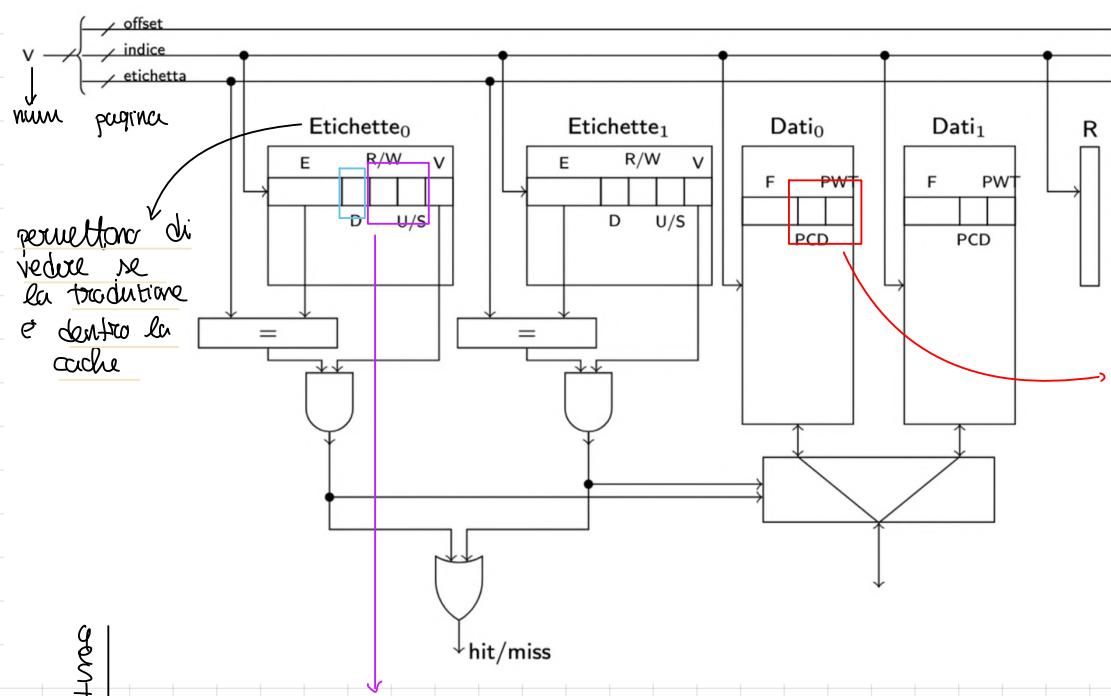
Aprendo le tabelle, in genere per tradurre l'indirizzo si devono fare 6 accesi in memoria RAM: 500 cicli di clock. Inoltre, la RAM deve aggiornare i BT e quindi leggerli e scriverli → 6 accessi in memoria

↓ risolutore



: translation lookaside buffer

: contiene traduzione num. pagina → num. fizico
rimuovi di fare il percorso
verifica se è già
presente la traduzione. Se non è presente fa
il percorso e porta la traduzione nel
TLB



→ Funziona allo stesso modo della cache della CPU

Vogliono anche gli stessi principi di località

informazioni gestione cache

su dati perché prima l'accesso deve essere garantito da etichette

Può vietare accesi utilizzando TLB → non sovraccarichi tutti e 4 i hit del percorso ma fa un AND

BIT A: se la traduzione è già in TLB non lo aggiorna in quanto era già ad A per avere portato in TLB. Il hit regola che c'è stato dunque un accesso alla pagina e quindi per garantire il corretto funzionamento questi hit vengono attesi per il corretto pericolo corrente del sistema. Non aggiornando il hit delle traduzioni nel TLB il servizio fornito da A non funziona più. Si fatto che il sistema attesi il hit A invia da il funzionamento della cache in quanto le tabelle vengono combinate da qualche che non è la MMU. La cache va in uno stato inconsistente e quindi il software invalida il TLB totalmente al imposta, che viene quindi nuotato e ricreato.

↓ 2 modi per invalidare
modo 1: soffocare in %CR3: invalida tutto TLB poiché comincio

d'errore di traduzione (anche se ci sono lo stesso concetto)

modo 2: INV LGY indirizzo: invalida la traduzione dell'indirizzo se questa è presente in TLR

BIT D: Va gestito correttamente perché è utilizzato nello swap-out dei processi. Il TLR ricorda il suo stato alla lettura nell'elenco. Se l'accesso è in lettura il bit non viene cambiato. Se l'accesso è in scrittura ed il TLR è 0 allora viene restituita una in modo che D venga aggiornato e la traduzione riconosciuta in TLR mentre se è 1 viene restituita hit e tutto si evolge correttamente. Nel primo caso non si aggiornerà solo il bit D in TLR perché non esistono per accedere al TLR. Anche se l'indirizzatore per accedere non si potrebbe fare in quanto l'aggiornamento di D in TLR comporterebbe l'accesso successivo all'elenco che può essere fatto solo dalla MMU.

Cosa accade con traduzioni dei più di 4K? La traduzione può essere inserita spartendola in 512 parti ed occupando quindi 512 entrate. Oggi c'è un TLR diverso per ogni dimensione di pagina. La MMU deve per questo guardare paralleamente tutti i TLR e nel caso di esito deve correggere la pagina in quello giusto → **VANTAGGIO** avere pagine grandi funzioni utili

$L_{MAX_LIV} = 4 \rightarrow 4$ livelli di tabella

BIT_SEGNO: serve per normalizzare gli indirizzi $\rightarrow 16$ MSB = bit 47 (0 o 1)

MASCHERA: $n \downarrow 1 \rightarrow$ seguito da 47 1
↓ Moudulo 2^16 → seguito da 47 1

static inline vaddr norm(vaddr a)

```
{  
    return (a & BIT_SEGNO) ? (a | ~MASCHERA_MODULO) : (a & MASCHERA_MODULO);  
}
```

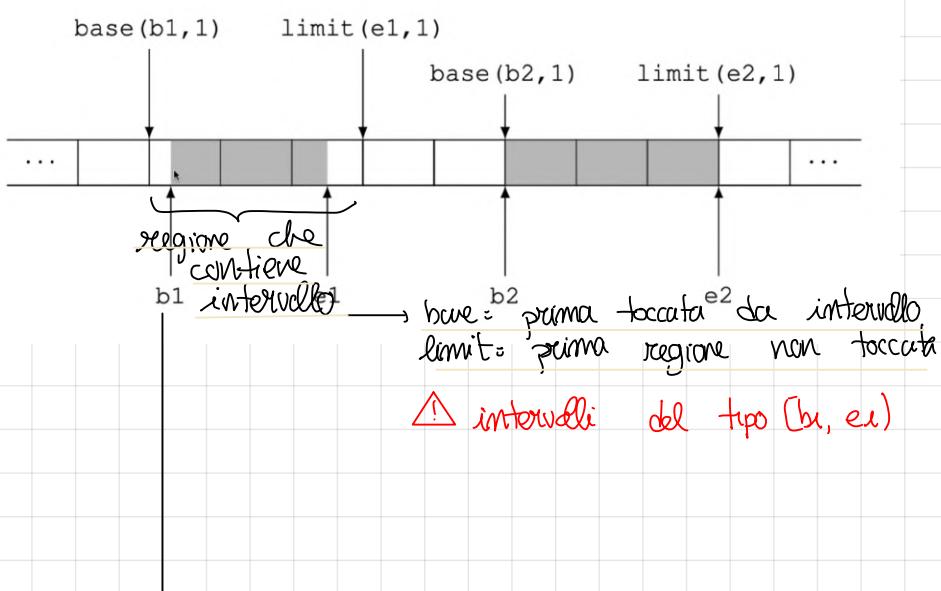
Mette a 1 i 16 MSB anteriori parte significativa

dimensione regione dato livello

static inline constexpr natq dim_region(int liv)

```
{  
    se fanno una costante la f() restituisce sempre lo stesso valore; permette ottimizzazioni  
    natq v = ULL << (liv * 9 + 12); → parte del Ø: pagina  
    return v; unsigned long long per non usare ou  
}
```

base / limite



base

```
static inline vaddr base(vaddr v, int liv)
{
    natq mask = dim_region(liv) - 1; → n 1 nella parte meno significativa (grand. livello)
    return v & ~mask; → nella parte meno sign.
```

limite

```
static inline vaddr limit(vaddr e, int liv)
```

```
{
    natq dr = dim_region(liv);
    natq mask = dr - 1;
    return (e + dr - 1) & ~mask;
```

se sono anche di rhute dentro vado alla successiva mentre resto dentro se sono dell'inizio

dov vedrò prima regole a destra

```
typedef natq tab_entry;
```

→ entro tabelle:

const natq BIT_P	= 1U << 0; // il bit di presenza
const natq BIT_RW	= 1U << 1; // il bit di lettura/scrittura
const natq BIT_ÜS	= 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"

set /effrazione ind. finico

resetto: And con costante negata

```
const natq ACCB_MASK = 0x00000000000000FF; // maschera per il byte di accesso
```

```
const natq ADDR_MASK = 0x7FFFFFFFFF000; // maschera per l'indirizzo
```

```
static inline paddr extr_IND_FISICO(tab_entry descrittore)
{ // (
    return descrittore & ADDR_MASK; // )
}
static inline void set_IND_FISICO(tab_entry& descrittore, paddr ind_fisico) //
{ // (
    descrittore &= ~ADDR_MASK;
    descrittore |= ind_fisico & ADDR_MASK; // )
}
```

estrazione indice a partire da livello ed indirizzo

```
static inline int i_tab(vaddr ind_virt, int liv)
{
```

```
    int shift = 12 + (liv - 1) * 9;
```

natq mask = 0x1ffULL << shift; → 1 solo dove c'è indice che mi serve
return (ind_virt & mask) >> shift; → Moreno solo in parte meno sign

estrazione indirizzo finico

```
static inline int i_tab(vaddr ind_virt, int liv)
{
```

int shift = 12 + (liv - 1) * 9; → bit che mi serve (avrei potuto usare anche dim_region)
natq mask = 0x1ffULL << shift; → 1 solo dove serve
return (ind_virt & mask) >> shift;

→ effetto netto ciò che non mi serve è

riferimento ad entrota

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

differenza tra vedere e vedere nel debugger

v = 000024a49b724b6d S-111-222-333-444 → otto e percosi im elero
p = 0x00001000 → solo esadecimale

Vogliamo mappare v in p → reutilizziamo la tabella del bootstrap ed implementiamo (leggiamo ora con readCR3());

definiamo le tabelle e l'elenco

```
extern tab3, tab2, tab1
.data
.align 4096
tab3:
    .space 4096, 0
tab2:
    .space 4096, 0
tab1:
    .space 4096, 0
```

```
int main()
{
    vaddr v = 01112223334445555;
    paddr p = 4096;

    paddr tab4 = readCR3();
    tab_entry& e4 = get_entry(tab4, i_tab(v, 4));
```

prendo il riferimento alla tabella di livello 4 (dell'entrota) dell'indice specificato.

set_IND_FISICO(e4, (paddr)&tab3); → collegiamo l'entrota della tabella 4 alla tabella 3
e4 |= BIT_P | BIT_RW;

set_P e RW

```
tab_entry& e3 = get_entry((paddr)&tab3, i_tab(v, 3));
set_IND_FISICO(e3, (paddr)&tab2);
e3 |= BIT_P | BIT_RW;
tab_entry& e2 = get_entry((paddr)&tab2, i_tab(v, 2));
set_IND_FISICO(e2, (paddr)&tab1);
e2 |= BIT_P | BIT_RW;
tab_entry& e1 = get_entry((paddr)&tab1, i_tab(v, 1));
set_IND_FISICO(e1, p);
e1 |= BIT_P | BIT_RW;
```

gestione del TLB
invalida TLB

```
// invalida tutto il TLB
.global invalida_TLB
invalida_TLB:
    .cfi_startproc
    // è sufficiente riscrivere in cr3 il suo contenuto corrente
    movq %cr3, %rax
    movq %rax, %cr3
    retq
    .cfi_endproc
```

invalida entrata TLB

```
// 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
```

iteratore: potrete di gestire altro di traduzione come correggere (visita oltre)
↓ utilizzando start fine condizione: false a fine

for (tab_iter it(tab4, v); it; it.next()) → visita ANTICIPATA AL DI FUORI
avanzamento

for (tab_iter it(tab4, v, v + 4*4096); it; it.next())

Le seguenti possono avere a cavallo di tabella di livelli diversi: ci pensa l'iteratore a fare una visita anticipata → si ferma su un elemento prima di inizio e quindi possono muovere l'iteratore per cercare l'elenco

```
for (tab_iter it(tab4, v); it; it.next()) {
    tab_entry& e = it.get_e(); → rig. entrata
    int l = it.get_l(); → livello oltre
    printf("livello %d: -> %x\n", l,
        extr_IND_FISICO(e));
```

}

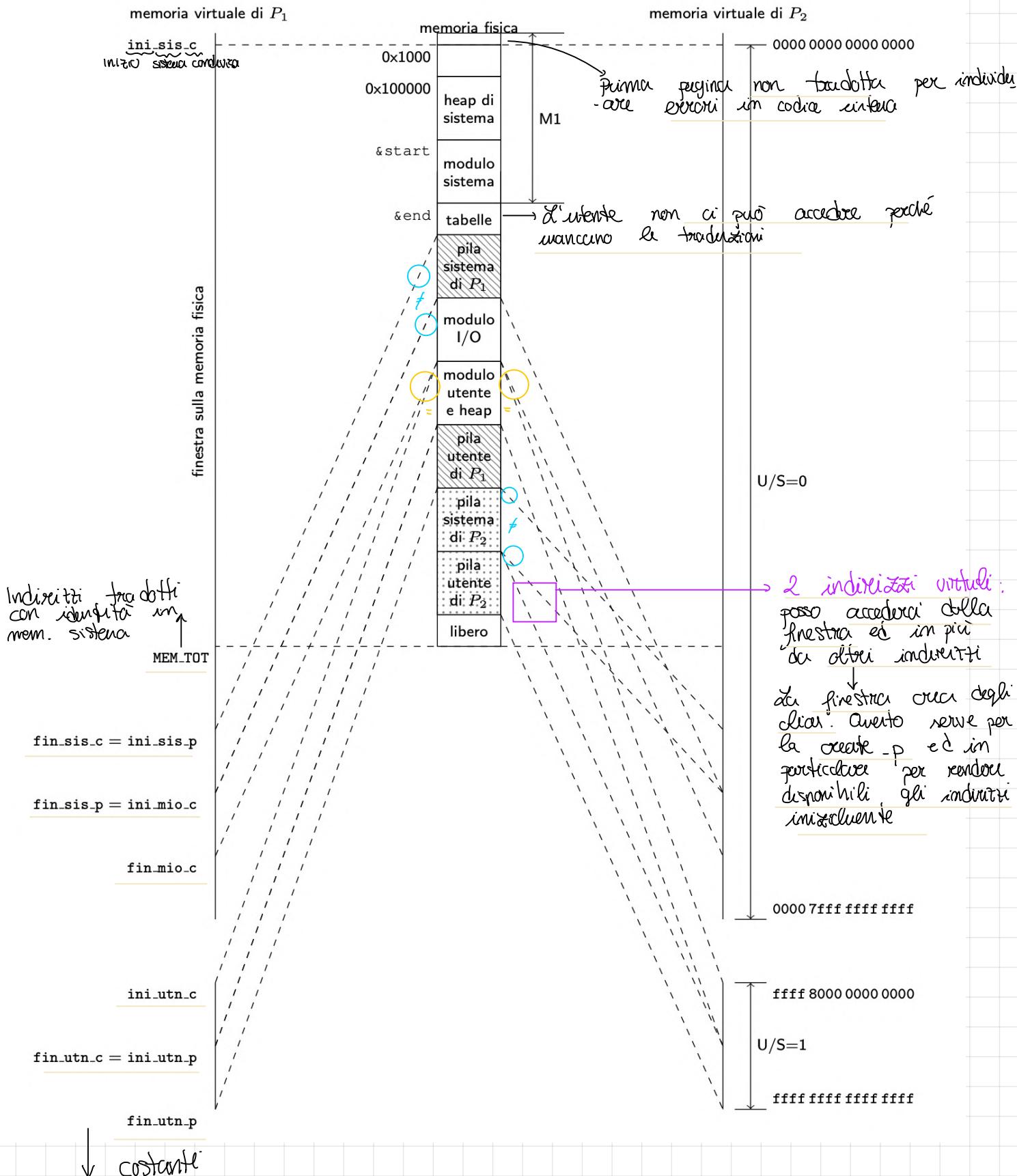
↓ visita anticipata in profondità quindi scende fino in fondo prima di cambiare



memoria virtuale processi

V(S=0) {cond (start, identità)} le sezioni text e data sono condivise quindi la loro traduzione, anche se proveniente da processi diversi, porta sempre uguali indirizzi di memoria fisica.
V(S=1) {cond} La zona utente "corre" invece privata e quindi la traduzione degli indirizzi non è diversa a seconda del processo

V(S=1) } utente / condivisa
(utente) } utente / privata



```

#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)

```

→ Sistema cerca di mappare meno memoria possibile

```

// C 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
#define N_SIS_P 1
#define N_MIO_C 1
#define N_UTN_C 128
#define N_UTN_P 128

```

↓
Sistema caricato a partire dai LMR, utente da indiritti che iniziano con FFFF
gestione dei frame

```

struct des_frame {
    union { → Le informazioni non sono mai insieme
        natw nvalid; → abbiamo scelto per
        // numero di entrate valide (se il frame contiene una tabella)
        natl prossimo_libero; → deallocare frame
    };
};

```

```

// numero totale di frame (M1 + M2)
natq const N_FRAME = MEM_TOT / DIM_PAGINA;
// numero totale di frame in M1 e in M2
natq N_M1;
natq N_M2;

// array dei descrittori di frame
des_frame vdf[N_FRAME];
// testa della lista dei frame liberi
natq primo_frame_libero;
// numero di frame nella lista dei frame liberi
natq num_frame_liberi;

```

inizializzazione

```

void init_frame()
{
    // primo frame di M2 → multiplo di pagina creato da compilatore
    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;
}

```

⚠ Traduzioni di dimensioni multiple di 512 GB: occupano righe intere di tavelle di livello 4

Allocazione

```
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; —> Restituisce indirizzo fisico frame allocato
}
```

Dallocazione

```
void rilascia_frame(paddr f)
{
    natq j = f / DIM_PAGINA; —> indice corretto
    if (j < N_M1) {
        fpanic("tentativo di rilasciare il frame %p di M1", f);
    } —> errore di interfaccia
    vdf[j].prossimo_libero = primo_frame_libero;
    primo_frame_libero = j;
    num_frame_liberi++;
}
```

Gestione indirizzi: Jrutucl

prima entrata liv. 0 due regioni livello 3 (dim pagina regione con un livello in meno)

```
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;
```

Allocazione

```
paddr alloca_tab()
{
    paddr f = alloca_frame();
    if (f) {
        memset(reinterpret_cast<void*>(f), 0, DIM_PAGINA);
    }
    vdf[f / DIM_PAGINA].nvalide = 0; —> stato corrispondente
    return f;
}
```

Dallocazione

```
void rilascia_tab(paddr f)
{
    if (int n = get_ref(f)) { —> le entrate valide devono essere 0
        fpanic("tentativo di deallocare la tabella %x con %d entrate valide", f, n);
    }
    rilascia_frame(f);
}
```

set delle entrate

```
void set_entry(paddr tab, natl j, tab_entry se)
{
    tab_entry& de = get_entry(tab, j);
    if ((se & BIT_P) && !(de & BIT_P)) { → Nuova entrata ha hit P=1
        inc_ref(tab);
    } else if (!(se & BIT_P) && (de & BIT_P)) { → nuova entrata ha hit P=0
        dec_ref(tab);
    }
    de = se;
}
```

Map

```
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T& getpaddr, int ps_lvl)
```

↓ Reduce Trie ↳ [)] → template
flag funzione che dà un indirizzo virtuale,
 restituendo l'indirizzo fisico che vogliamo
Guarda tutte le tabella necessarie per avere le traduzioni (begin, end) e quando
corrisponde a quelle di livello i ci mette l'indirizzo dato della funzione
parata
↓ Algoritmo

```
for /* tutto il sotto-albero visitato in pre-ordine */ { → una iterazione
    tab_entry& e = /* riferimento all'entrata corrente */
    vaddr v = /* indirizzo virtuale corrente */
    if /* livello non foglia */ {
        if /* tabella assente */ {
            new_f = alloca_tab();
        }
    } else {
        new_f = getpaddr(v);
    }
    if (new_f) {
        /* inizializza 'e' in modo che punti a 'new_f' */
    }
}
```

+ controlli: [begin, end] non deve contenere traduzioni precedenti

[begin, end] non deve sovrapporre il buco → intervallo valido

+ Ritorno: primo indirizzo per cui non c'è traduzione (end). Se il progetto restituito
è diverso si è verificato un errore → si distrugge la traduzione

```
void unmmap(paddr tab, vaddr begin, vaddr end, T& putpaddr)
```

Distrugge tab con multilevel=0

```
// otteniamo il corrispondente indirizzo fisico
// chiedendolo all'oggetto getpaddr
```

```
new_f = getpaddr(v); → possiamo passare funzione o oggetto che si comporta come funzione
```

Si puca l'indirizzo fisico e ci dice cosa fare

Vantaggio

L'oggetto ha uno stato proprio che sopravvive all'invocazione
dei suoi metodi e non è condiviso con altri oggetti.
Questo è comodo nel caso in cui valori passate
in array di traduzione.

Applicazione: creazione finestra di memoria

```
paddr root_tab = alloca_tab();
if (!root_tab)
    goto error;
```

↓ Semplificando

```
paddr identity(vaddr v)
```

```
{
    return v; // traduzione identità
}
```

```
map(root_tab, 0, MEM_TOT, flag, identity);
```

Realtà

non mappiamo la

non controllato di default ma
nucleo perché ha flag setto

controllato dal

```
if (map(root_tab, DIM_PAGINA, first_reg, BIT_RW, identity_map) != first_reg)
    return false; —> non riuscito a mappare
```

```
tab_iter it(root_tab, 0xb8000, DIM_PAGINA); —> PWT = 1 per mem. video
while (it.down()) —> iteratore corre solo nel percorso di quell'indirizzo e si ferma
;
tab_entry& e = it.get_e(); —> iteratore fermo sulla foglia
e |= BIT_PWT;
```

```
if (MEM_TOT > first_reg) {
    if (map(root_tab, first_reg, MEM_TOT, BIT_RW, identity_map, 2) != MEM_TOT)
        return false;
}
```

—> pagine da LMB
(livello 2)

```
// 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 beg_pci = 4*GiB - 20*MiB,
end_pci = 4*GiB;
if (map(root_tab, beg_pci, end_pci, BIT_RW|BIT_PCD|BIT_PWT, identity_map, 2) != end_pci)
    return false;
```

disabilita cache

Creazione Spazio condiviso —> dove creare corrispondente con sezioni data e text

```
if (map(root_tab,
        virt_beg,
        virt_end,
        flags,
        copy_segment{mod_beg, mod_end, virt_beg}) != virt_end)
    return 0;
```

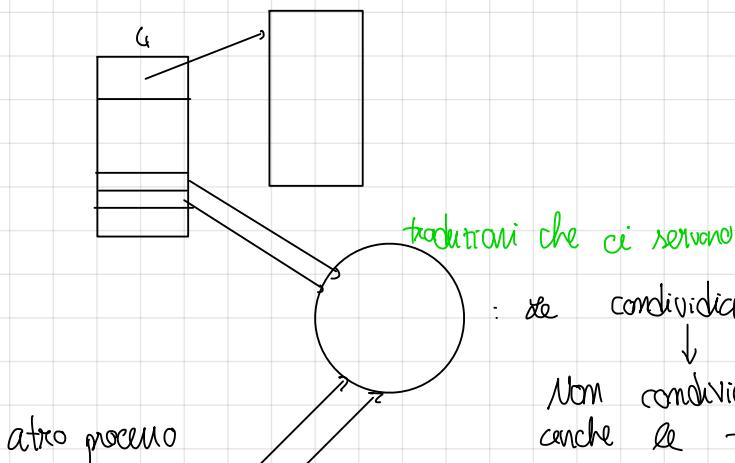
nella creazione dei processi

```
// creazione della tabella radice del processo
p->cr3 = alloca_tab(); : tabella di livello 4
if (p->cr3 == 0)
    goto err_rel_id;
init_root_tab(p->cr3);
```

} SEMPLIFICAZIONE: punti condivisi tra tutti i processi non cambiano

{

procedura esistente (quello che crea il processo nuovo)



: se condividiamo: puntano agli stessi traduttori

↓
Non condividono la tabella di livello 0 perché contiene anche le traduzioni per le parti private

```
→ void init_root_tab(paddr dest)
{
    proc che ha chiamato activate_p
    paddr pdir = readCR3();

    // ci limitiamo a copiare dalla tabella radice corrente i puntatori
    // alle tavole di livello inferiore per tutte le parti condivise
    // (utente, I/O). Quindi tutti i sottoalberi di traduzione
    // delle parti condivise saranno anch'essi condivisi. Questo, oltre a
    // semplificare l'inizializzazione di un processo, ci permette di
    // risparmiare un po' di memoria.
    copy_des(pdir, dest, I_SIS_C, N_SIS_C);
    copy_des(pdir, dest, I_MIO_C, N_MIO_C); ↳ copia del direzio sopra
    copy_des(pdir, dest, I_UTN_C, N_UTN_C);
}
```

Creazione pila *↳ Radice* *↳ Bottom*

```
if (!crea_pila(p->cr3, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
    goto err_rel_tab;
```

↓ che indirizzo possiamo?

```
paddr miagetpaddr(vaddr v)
{
    return alloca_frame();
}
```

: Mettiamo la pila sottocima in un nuovo frame

```
bool crea_pila(paddr root_tab, vaddr bottom, natq size, natl liv)
{
    vaddr v = map(root_tab,
                  bottom - size, begin
                  bottom, end
                  BIT_RW | (liv == LIV_UTENTE ? BIT_US : 0),
                  0(vaddr) { return alloca_frame(); });

    if (v != bottom) { → map ha fallito
                      unmap(root_tab, bottom - size, v,
                            0(paddr p, int) { rilascia_frame(p); });
                      return false; ↳ livello
    }
    return true;
}
```

inizializziamo

```

if (liv == LIV_UTENTE) {
    // inizializziamo la pila sistema.
    pl[-5] = reinterpret_cast<natq>(f); // RIP (codice utente)
    pl[-4] = SEL_CODICE_UTENTE; // CS (codice utente)
    pl[-3] = BIT_IF; // RFLAGS
    pl[-2] = fin_utn_p - sizeof(natq); // RSP
    pl[-1] = SEL_DATI_UTENTE; // SS (pila utente)
}

```

Per accedere alla pila del processo appena creato (e non nel creatore) utilizziamo l'indirizzo fisico otto della finestra di memoria

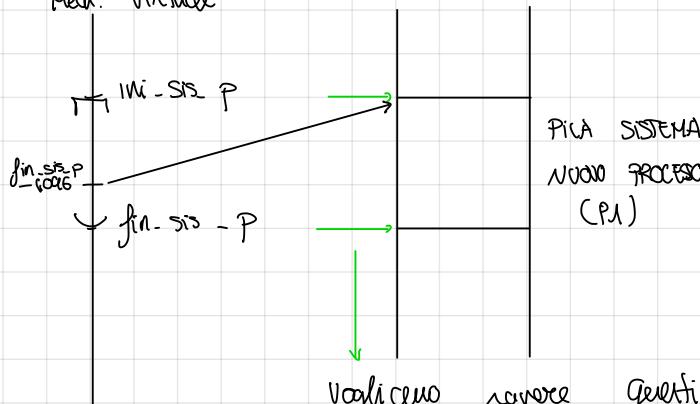
```

pila_sistema = trasforma(p->cr3, fin_sis_p - DIM_PAGINA) + DIM_PAGINA;
// convertiamo a puntatore a natq, per accedervi più comodamente
pl = reinterpret_cast<natq*>(pila_sistema); // ultima pila quadrupla

```

→ fa la traduzione
"a mano"

Mem. virtuale



Primo byte della RIM fuori dalla pagina

Vogliamo sapere quali indiritti finci: dato che la pila è grande non è sufficiente una pagina. Ci interessa tradurre fin-sis-p- 60% fino a fin-sis-p (escluso)

```

if (!crea_pila(p->cr3, fin_utn_p, DIM_USR_STACK, LIV_UTENTE)) {
    flog(LOG_WARN, "creazione pila utente fallita");
    goto err_del_sstack;
}

```

```

paddr trasforma(paddr root_tab, vaddr v)
{
    // usiamo un tab_iter sul solo indirizzo 'v', fermandoci
    // sul descrittore foglia lungo il percorso di traduzione
    tab_iter it(root_tab, v);
    while (it.down())
        ;

    // si noti che il percorso potrebbe essere incompleto.
    // Ce ne accorgiamo perché 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(); // nuovo frame
    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; // n hit col 1(offset)
    return (e & ~mask) | (v & mask);
}

```

isiliamo nr. atteniamo nr. di pagina
di frame a teniamo offset

Distruttore processo

```

void distruggi_processo(des_proc* p)
{
    paddr root_tab = p->cr3;
    if (p->livello == LIV_UTENTE)
        distruggi_pila(root_tab, fin_utn_p, DIM_USR_STACK);
    ultimo_terminato = root_tab;
    if (p != esecuzione_precedente) {
        distruggi_pila_precedente(); // Rilascia pila
        sistema quando non ha uscito
    }
    rilascia_proc_id(p->id);
    delete p;
}

void distruggi_pila(paddr root_tab, vaddr bottom, natq size)
{
    unmap(
        root_tab,
        bottom - size,
        bottom,
        [](paddr p, int) { rilascia_frame(p); });
}

```

Cerca stato

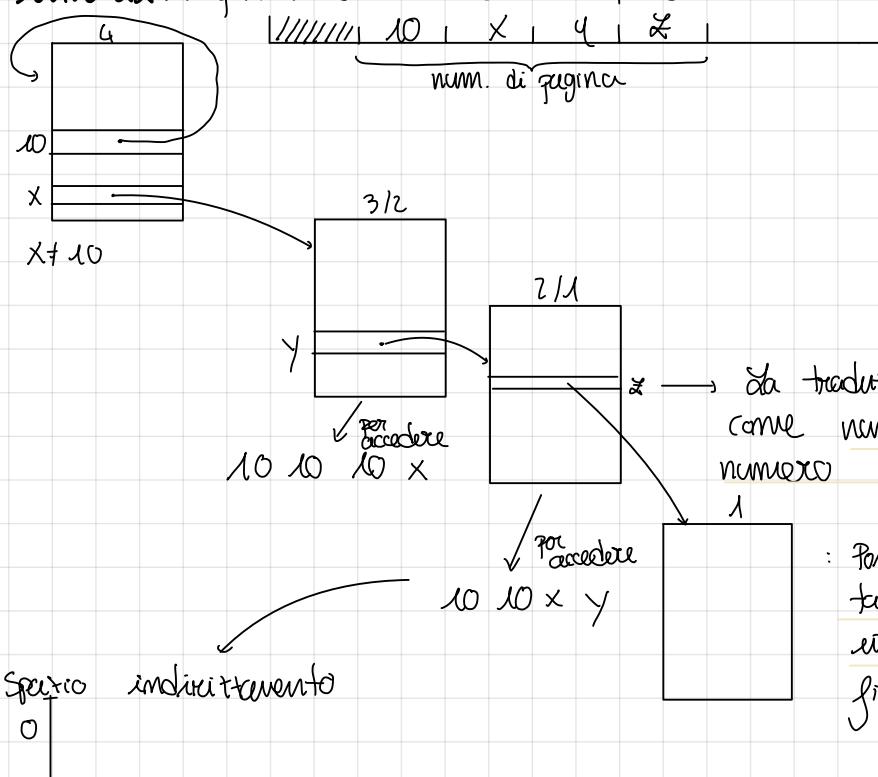
```

// nuovo valore per cr3
movq CR3(%rbx), %r10 → cerca traduzione
movq %cr3, %rax
cmpq %rax, %r10 → Accessibile memoria nuovo processo
je 1f
movq %r10, %rax → evitiamo di invalidare il TLB → può capitare che il TRIE
movq %rax, %cr3 → se cr3 non cambia non venga cambiato e
                     che venga ri eseguito lo stesso processo

```

⚠ La unmap deve fare una visita posticipata, ovvero deve arrivare fino in fondo prima di eliminare → funzione ut.post()

Utilizzo: puntatore alla stessa tabella



Da traduzione finale: l'indirizzo viene mappato come numero di frame per sostituire il numero di pagina

: Possiamo leggere / scrivere in questa tabella tramite l'offset: utile per interno → metodo alternativo alla finestra

→ tutte tabelle di livello 1 visibili

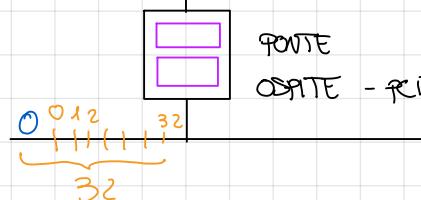
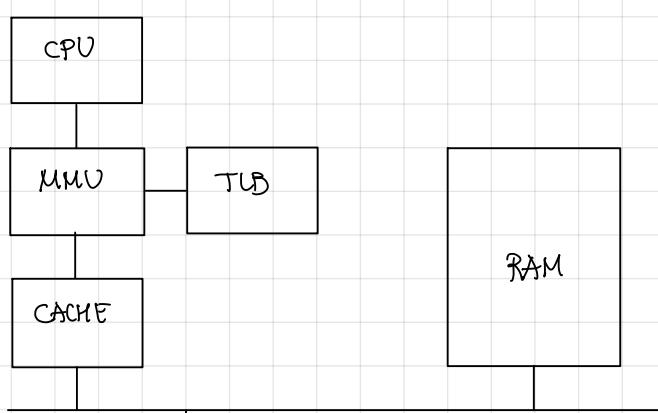
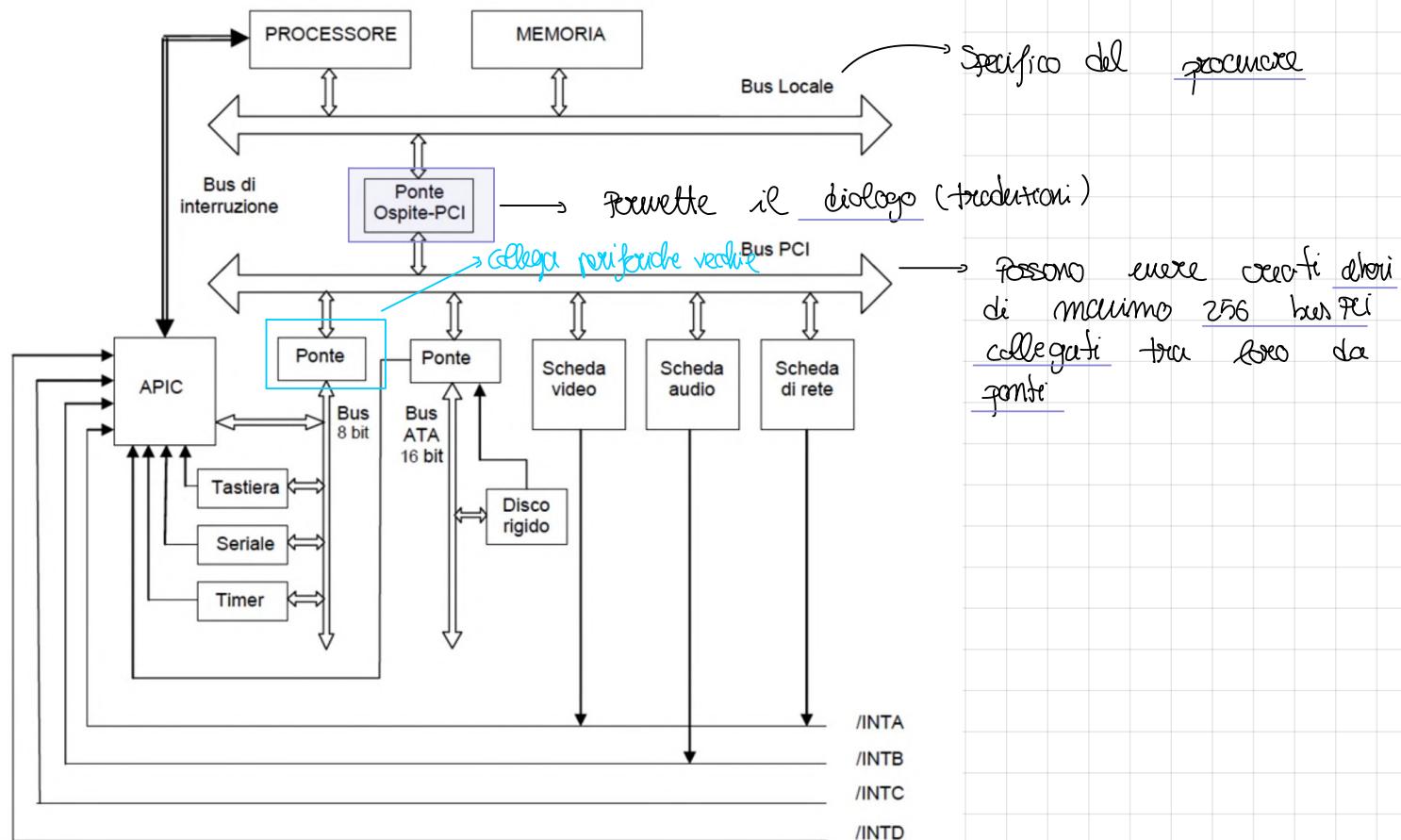
rimuova il problema di avere la finestra di memoria che "ruha" spazio

Ingresso / uscita

o Bus PCI

Standard: sostituisce il PC-AT perché aveva vari problemi → il PC IBM era expandibile con altri dispositivi. Altre aziende producevano schede per questo. È necessario che queste siano compatibili tra di loro e c'è bisogno di un software (driver) che ci permette di muoverle e dare sapere se è collegata o no. Per farlo il driver deve ricevere all'indirizzo fornito dal produttore e vedere se legge poi il risultato atteso. Ci sono problemi (c'erano) se più schede usano gli stessi indirizzi. Il bus PCI nasce per risolvere questi problemi.

Struttura

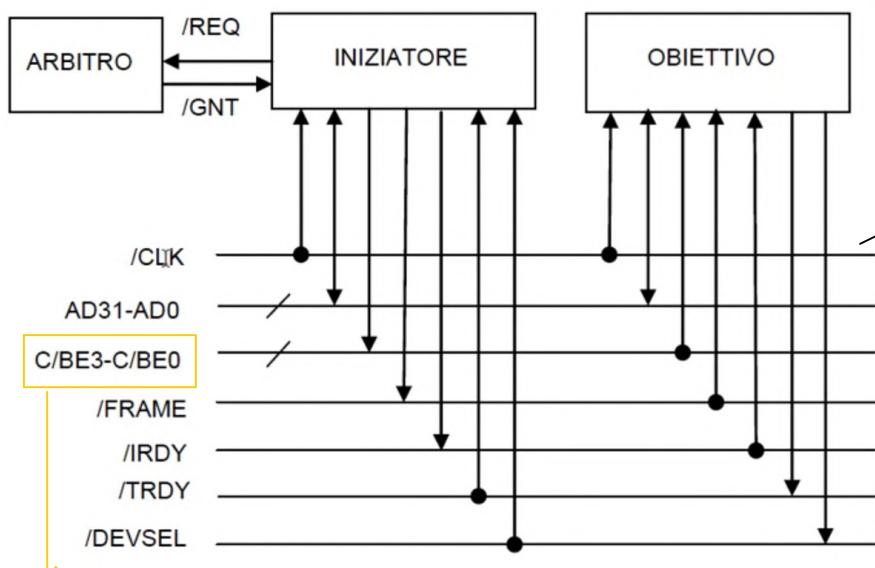


BUS PCI → Risolve problemi di prelievo: definisce 3 nuovi spazi di indirizzamento

I/O
MEMORIA
CONFIGURAZIONE → i dispositivi acquisiscono un indirizzo automaticamente

Identificazione geografica

In ogni BUS (max 256) ci sono vari Spazi ove inviare schede (maxim 32). Ogni scheda ha un identificatore dato dell'indirizzi (numero). Le schede multifunzione possono avere al massimo 3 funzioni, numerate da 0 a 7 del costruttore. Alla fine, ogni scheda è identificata univocamente. Non sono però dimensioni hardware che permettono alla CPU di accedere allo spazio di configurazione e quindi ci si deve accedere indirettamente in fase di avvio tramite i registeri del ponte nello spazio di I/O della CPU → molto costoso: le operazioni devono coinvolgere dimensioni hardware. com un registro. Una volta configurato il commutatore, alla scheda si accede con le normali istruzioni IN /OUT → vantaggio: configurati da SW sistema che si cura di scegliere indirizzi diversi d'acesso. Inoltre, il BIOS stabilisce anche cosa fare se un dispositivo non risponde: sorge una configurazione per indicare riconosciuti del suo come orario collegamenti.



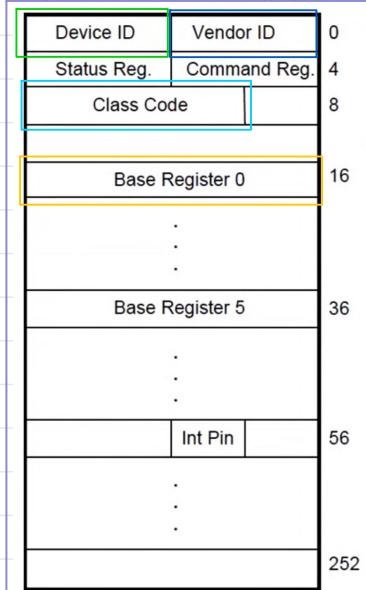
- Linee campionate secondo il clock: 66 MHz (prima 33)
- Linee di indirizzo/dati condivise.
 - ↓ Tutte transazioni BIOS
- 1. indirizzi esaurito
- 2. scambio dati: possono essere più di una

Linee di controllo e byte-enable: nella fase di indirizzamento fungono da linee di controllo e quindi codificano il tipo di operazione (R/W) e lo spazio di indirizzamento. Nelle fasi successive servono come byte enable, ovvero specificano quali byte della parola trasferire. I dati sono trasportati con un parallelismo di 4 e quindi ADL e TDO sono sempre 0. Per stabilire quali byte trasferire tra i 4 indirizzati sempre viene utilizzato il byte enable.

Le operazioni sono dette transazioni, ciascuna con un proprio iniziatore o obiettivo. Qualunque dispositivo può essere iniziatore o obiettivo (un solo per volta). Tipicamente l'iniziatore è il ponte. L'iniziatore pilota la fase di indirizzamento e poi fase quanto è necessario che gli iniziatori siano collegati con un arbitro che esamina le richieste e prima o poi dai l'acknowledge all'iniziatore. Questo poi deve attendere che le altre transazioni terminino e poi può pilotare il bus. Deve attendere in quanto l'architettura può avere fatto mentre è in esecuzione un'altra transazione. Per pilotare il bus deve: attivare /FRAME, pilotare l'indirizzo di controllo ed ottenere che i dispositivi attivino la linea /DEVSEL. Se questo non viene ottenuto entro un certo numero di clock questo implica che il dispositivo non è presente. Fatto questo si puota al trasferimento dati utilizzando /IRDY e /TRDY come handshake. Finché /FRAME viene tenuto a 0 si continuano a trasferire dati in quanto viene messo ad 1 all'ultima fase. Quando anche /IRDY e /TRDY sono disattivati si può dare il via ad un nuovo trasferimento. I byte

enable come sempre pilotati dall'initiatore mentre le linee dati sono pilotate dal destinatario o dell'obiettivo a seconda che l'operazione sia di lettura o scrittura.

Spazio di configurazione



PC
S
G
A
D
I
V
E

Ciascuna funzione può acquisire un indirizzo nello spazio di configurazione automaticamente. Ciascuna di esse deve avere al suo interno 256 byte di configurazione. Questi vengono chiamati registri ed alcuni sono obbligatori:

- Venditore
- Scelto architetturale del venditore
- Ci permette di sapere cos'è il dispositivo: è possibile solo anche se non si ha un driver specifico

Class Code (byte più significativo)	Significato PCI
0x00	Pre revision 2.0
0x01	Mass storage controller
0x02	Network controller
0x03	Display controller
0x04	Multimedia device
0x05	Memory controller
0x06	Bridge device
0x07	Simple communication controller
0x08	Base system peripheral
0x09	Input device
0xA	Docking station
0xB	Processor
0xC	Serial bus controller
0xFF	Other

→ Permettono di programmare i componenti: sono al massimo.

Allora cosa fa la macchina per il venditore (quanti registri e ne devono stare in I/O o memoria). Il software decide invece dove disporli e dove comunicarli alla scheda. Il hardware gestisce un blocco di register e codifica quanto è grande il blocco. Il hardware viene letto dal HW ed assegna l'indirizzo di partenza del blocco sovrapponendo nel registro.



Il primo bit b non scrivibile codifica la dimensione del blocco: grande 2^b . Per scoprire quanto è b il HW riceve tutti 1 e vede fin dove è rimasto a zero.

Dove poi decide dove allocare il blocco: vengono scelte regioni naturali di 2^b . Per comunicare al dispositivo qual è il suo blocco vengono scritte le coordinate dell'indirizzo: $32-b$ | b | offset. Ciò che va a finire in R/W. Per il resto ci pensa il driver.

Queste operazioni vengono fatte dall'alto: il ponte ha 2 registri nello spazio di I/O (0x0CF0 e 0x0CF1) che permettono di accedere allo spazio di configurazione indirettamente: nel primo (CAP) si riceve l'indirizzo a cui accedere, dopodiché il secondo (COP) dà una finestra all'interno della memoria di configurazione di quella particolare funzione. Per ricevere in CAP:



Qui tutte quelle vengono presi dai hub enable dell'operatore fatto su COP.

Se nessuno risponde vengono scritti tutti i dispositivi installati e può enumerare i dispositivi collegati e può configurarli per scegliere indiritti non sovrapposti. Per ricevere quelli collegati, il software pone a leggere qualche dello spazio di configurazione di ogni possibile dispositivo e funzione ed in particolare legge il vendor ID. Se legge tutti 1 (inexistenti) allora conclude che non c'è niente, altrimenti può leggere anche le altre informazioni e configurerle ai loro register.

UTILIZZO LIBCE

```
natb pci_read_conf8(natb bus, natb dev, natb fun, natb off)
{
    natl confaddr = make_CAP(bus, dev, fun, off);
    outputl(confaddr, CAP); // scrive configurazione
    return inputb(CDP + (off & 0x03));
}
```

utilizza tutte anche

"componi" CAP come detto precedentemente

```
natl make_CAP(natb bus, natb dev, natb fun, natb off)
{
    return 0x80000000 | (bus << 16) | (dev << 11) | (fun << 8) | (off &
0xFC); // bit meno significativi = 0
```

```
void pci_write_conf8(natb bus, natb dev, natb fun, natb off, natb data)
{
    natl confaddr = make_CAP(bus, dev, fun, off);
    outputl(confaddr, CAP);
    outputb(data, CDP + (off & 0x03));
}
```

Scrivere config

Lettura elenco dispositivi attivi

```
#include <libce.h>
int main()
{
    natb bus = 0, c;
    for (natb dev = 0; dev < 32; dev++) {
        for (natb fun = 0; fun < 8; fun++) {
            natw vendorID, deviceID;

            vendorID = pci_read_conf8(bus, dev, fun, 0);
            if (vendorID == 0xFFFF) continue;
            deviceID = pci_read_conf8(bus, dev, fun, 2);
            printf("%2x.%2x.%2x: %4x.%4x\n",
                   bus, dev, fun,
                   vendorID, deviceID);
        }
    }
    pause();
    return 0;
}
```



Com CTRL+ALT e info PCI

```
QEMU
wavcapture path audiodev [frequency [bits [channels]]] -- capture audio to a wav
e file (default frequency=44100 bits=16 channels=2)
x /fmt addr -- virtual memory dump starting at 'addr'
x_colo_lost_heartbeat -- Tell COLO that heartbeat is lost,
a failover or takeover is needed.
xp /fmt addr -- physical memory dump starting at 'addr'
(qemu) info pci
Bus 0, device 0, function 0:
  Host bridge: PCI device 8086:1237
    PCI subsystem 1af4:1100
    id ""
Bus 0, device 1, function 0: testiera
  ISA bridge: PCI device 8086:7000
    PCI subsystem 1af4:1100
    id ""
Bus 0, device 1, function 1: IDE
  IDE controller: PCI device 8086:7010
    PCI subsystem 1af4:1100
    BAR4: I/O at 0xc000 [0xc00f].
    id ""
Bus 0, device 1, function 3:
  Bridge: PCI device 8086:7113
    PCI subsystem 1af4:1100
    IRQ 9, pin A
```

```
QEMU
PCI subsystem 1af4:1100
id ""
Bus 0, device 1, function 0:
  ISA bridge: PCI device 8086:7000
    PCI subsystem 1af4:1100
    id ""
Bus 0, device 1, function 1:
  IDE controller: PCI device 8086:7010
    PCI subsystem 1af4:1100
    BAR4: I/O at 0xc000 [0xc00f].
    id ""
Bus 0, device 1, function 3:
  Bridge: PCI device 8086:7113
    PCI subsystem 1af4:1100
    IRQ 9, pin A
    id ""
Bus 0, device 2, function 0: scheda video
  VGA controller: PCI device 1234:1111
    PCI subsystem 1af4:1100
    BAR0: 32 bit prefetchable memory at 0xfd000000 [0xfffffff]. : 16 MB memoria video mod. grafica
    BAR2: 32 bit memory at 0xebf0000 [0xebf0ffff].
    BAR6: 32 bit memory at 0xfffffffffffffff [0x0000ffff].
    id ""
(qemu)
```

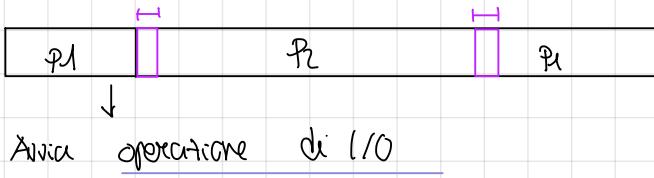
Come facciamo a spostare le operazioni nello spazio di indirizzamento?
Ci sono il ponte che due camere, in base agli indirizzi, quelli non le operazioni dirette dei dispositivi e quelli no.
↓ 2 modi

Attivo: ricompare finestra PCI → quando un indirizzo è in tale intervallo capisce che l'operazione è destinata al BUS PCI (tipico della memoria: ultimi 20MB)

Passivo: se il ponte vede che nonno risponde per un certo tempo provoca a trasferire la transazione al BUS PCI (utilizzato per I/O)
interrupt

Le possibili linee per richiesta interruzione: A, B, C, D e deve dire se una funzione è collegata ad una di queste 4 linee (se eventualmente a qualche). Questo è scritto in INTIN. Non viene detto come sono collegate a linee per rendere tutto indipendente dal processore. In QEMU si utilizzano le linee dell'APIC dopo la 16. Inoltre, il ruo di configurare come a che piedino dell'APIC è collegata la funzione

o 1/10

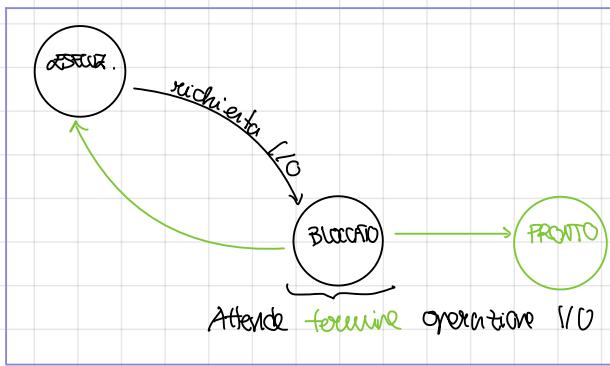


→ i processi non hanno intenzione a collaborare ed in particolare a cedere la CPU l'uno all'altro
↓

d'I/O può essere eseguita solo a livello sistema con IOPL o traduzione ed utilizzano le interruzioni

Le parti viste sono parti di codice di un'operazione di sincronizzazione.

P1:



Possiamo avere i segnali: vogliamo che P1 si blocca in un segnale di sincronizzazione finché l'operazione di I/O non termina

Un altro problema risolto

Possiamo sindicare anche il problema di due processi vogliano usare la stessa periferica

a.in.↓ organizzazione operazione

Seu - wait (mutex)

: → avvia operazione

Seu - signal (mutex) : blocca processo e ne schedula un altro

tutte chiamate di esterna chiamate da altre primitive di intesa.

⚠ a.in:

call solva_stato

call c-in

call carica_stato

IRETQ

⚠ anche tutte le altre chiamate fanno solva_stato e carica_stato

da solva_stato di a e di seu-wait ritrovano sullo stesso dispositivo di processo e quindi la seconda non muore

terzo concetto: a non deve fare solva/carica stato perché due eseguire processi ed invocare le operazioni di a possono essere interrotte e ripetute. Sono quindi simili ad un processo utente e non una primitiva atomica. Non è un problema perché stiamo utilizzando i segnali

Parte codice sintesi che interrompe P2

a - driver:

call solva_stato (P2)

:

// seu signal (sync) →

call avvia_stato

IRETQ

Non possono marca e quindi dato che vanno a livello sistema possono fare la stessa cosa "a suono"

Primitive fornite per dialogare con I/O:

read (int id, identificatore numero, periferica dato da sistema)

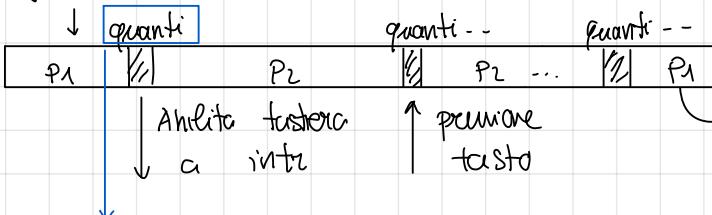
char *buf, contiene byte letto, (ad esempio da tastiera)

, natl quanti) quanti byte legge

write (int id, const char *buf, natl quanti)

↓

Non è detto che una periferica possa svolgere da sola tutti l'operazione: il driver fornisce la system call con ciò che la periferica ha fatto non tutti gli interrupt segnalano la fine dell'operazione (ad esempio, dovremo leggere più caratteri da tastiera e si riceverà qualche byte ormai scritti). La system call si ricorda quanti byte erano stati richiesti dall'utente e decrementa ogni volta



Come fanno quanti? Utilizziamo lo spazio di sistema condiviso. Abbiamo in memoria di deviitori di periferiche con queste informazioni e gli indici dei remotori associati a tali periferiche. Il driver deve sapere la periferica a cui era associato in modo da accedere alla struttura dati corretti dove trovare tutte le sue informazioni e le informazioni nelle operazioni di I/O in corso. Per sapere a che periferica è associato il driver è possibile guardare il tipo di interruzione. Si deve inoltre sapere quale è il tasto e nel quale ordine della prima. Se memoria virtuale attiva dove è quello di P2 e quindi il driver non può ricevere l'indirizzo buf specificato in read perché in questo modo si riferirebbe a P2

Imponiamo limitazioni sul buffer passato a read: deve essere nella memoria condivisa → deve essere dichiarato a livello globale.

nella macchina QEMU

extern "C" void ceread_n(nat1 id, char *buf, natl quanti);

possono avere al massimo 4

ceread_n:
int \$IO TIPO_CEREAD
ret

A livello sistema

```
// i tipi da 0x50 a 0xFE verranno usati per gli handler  
// (si veda load_handler() più avanti)  
carica_gate 0xa0      a_driver_ce_0  LIV_SISTEMA  
carica_gate 0xa1      a_driver_ce_1  LIV_SISTEMA  
carica_gate 0xa2      a_driver_ce_2  LIV_SISTEMA  
carica_gate 0xa3      a_driver_ce_3  LIV_SISTEMA
```

: 4 driver per differenziare le 4 periferiche possibili

```

a_driver_ce_0:
    .cfi_startproc
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    mov $0, %rdi → Comunica che è stata fin puferucca O a fare la
    call c_driver_ce
    call carica_stato
    iretq
    .cfi_endproc

```

Requisito:

```

a_ceread_n:
    .cfi_startproc
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call c_ceread_n
    iretq
    .cfi_endproc

```

Comunica che è stata fin puferucca O a fare la richiesta

⚠ Non ci sono salva / carica stato: il processo provoca dopo aver immolato il livello di privilegio facendo altre cose

Descrizione

```

static const int MAX_CE = 4;
struct des_ce {
    ioaddr iCTL, iSTS, iRBR;
    char *buf;
    natl quanti;
    natl sync;
    natl mutex;
};
des_ce array_ce[MAX_CE]; } quante ce abitano fronte
natl next_ce;

```

Periferica PCI: I registri possono avere comunque nello spazio di I/O → L'indirizzo viene scambiato all'invio e poi non cambiano

Initializzazione

```

bool ce_init()
{
    for (natb bus = 0, dev = 0, fun = 0;
        pci_find_dev(bus, dev, fun, 0xedce, 0x1234); → RIFERIMENTO: bus, dev, fun
        pci_next(bus, dev, fun)) verifica ID device
    {
        if (next_ce >= MAX_CE) {
            flag(LOG_WARN, "troppi dispositivi ce");
            break;
        }
        des_ce *ce = &array_ce[next_ce]; → percorsoo all'utilizzo
        flag(LOG_INFO, "ce%d %2x:%1x:%1x base=%4x IRQ=%d", next_ce, bus, dev, fun, ce->iSTS, irq);
        next_ce++;
    }
    return true;
}

```

```

natl base = pci_read_confL(bus, dev, fun, 0x10); → Da specifica era il punto base
base &= ~1; → LSB ad uso spazio di I/O: il resto
ce->iCTL = base; → base = base è l'indirizzo
ce->iSTS = base + 4; → Registro stato
ce->iRBR = base + 8; → RBR
ce->sync = sem_ini(0); → semafore utilizzati per
ce->mutex = sem_ini(1); → gestire la puferucca
natb irq = pci_read_confB(bus, dev, fun, 0x3c); →
apic_set_VECT(irq, 0xa0 + next_ce); → Tipo
apic_set_MIRQ(irq, false);
apic_set_TRGM(irq, false); // riconoscimento sul fronte

```

Da specifica era il punto base Address Register (offset 16)

Associazione a richieste di interruzione

0x3C è il punto dove il BIOS riceve a che piedino dell'APIC è collegata la puferucca

permesso:

dobbiamo controllare che va nello spazio utente, dove avere il spazio utente + traduttore mappato e incendiare

dove avere il spazio utente + traduttore

```
extern "C" void c_ceread_n(nat1 id, char *buf, nat1 quanti)
{
    if (id >= MAX_CE) { controllo identificatore
        flag(LOG_WARN, "id non valido");
        abort_p(); deve file relva_state e
    }

    if (id >= next_ce) { L'utente non sa quale periferica
        flag(LOG_WARN, "periferica non presente");
        return;
    }

    des_ce *ce = &array_des[id]; ID valido
    sem_wait(ce->mutex); Schema di prima
    ce->buf = buf;
    ce->quanti = quanti;
    outputb(1, ce->iCTL);
    sem_signal(ce->sync);
    sem_signal(ce->mutex);
}
```

Avvio operazioni

Modulo interruzione ma
non attivata: ci sono INTR

```
extern "C" bool c_access(vaddr begin, nat4 dim, bool writeable, bool shared = true)
{
    if (!tab_iter::valid_interval(begin, dim)) controllo se sta in
        return false; spazio utente cond: int.
    spazio controllato
    if (shared && (!lin_utn_c(begin) || (dim > 0 && !lin_utn_c(begin + dim - 1))) 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->r3, 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;
}

if (!c_access(vaddr, buf, quanti, true, true))
    flag(LOG_WARN, "indirizzo non valido");
    abort_p();
```

modifica ceread_n

tutti gli indirizzi puntati dall'esterno
devono essere controllati altrimenti
⚠️ cavallo di Troia

Driver:

```
extern "C" void c_driver_ce(int id)
{
    des_ce *ce = &array_des[id]; ID proviene dal driver davanti alla
    ce->quanti--;
    if (ce->quanti == 0) {
        outputb(0, ce->iCTL); disattiva interruzioni
        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
        }
    }

    char b = inputb(ce->iRBR); prendiamo byte
    *ce->buf = b; mettiamo un carattere dietro l'altro
    ce->buf++;
}
```

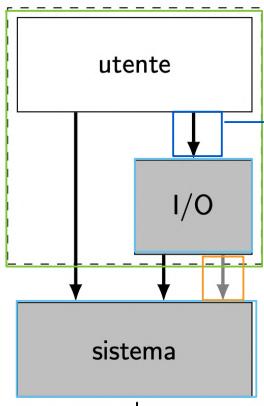
do mettiamo qua perché non vogliano che avvenga più reclame di interruzione
nel caso di più byte pronte

Le interruzioni in questo caso non possono essere cancellate

Nuovo modulo che elimina le incompatibilità: permette di avere INTR e
invocare chiamate sistema → permette di trovare più facilmente i BUG di I/O
(più frequenti). Possiamo eseguire il driver a livello utente? Nota che lo trasformano
in un processo, dobbiamo fare in modo che IOP per questo tipo di processi sia
utente. Questo però permette di fare C1 e STI e non ci fanno perché stato di
fatto dando questo privilegio all'utente. → Il modulo verrà eseguito a livello interno
con interruzioni abilitate

Risolve errori involontari: nuovi strumenti sistema non disponibili

⚠️ ATT: non possono
avere relva_state perché
verrebbe usato il driver
di P2 da parte del
driver. Non sono usate
memoria la sem-signal
in C perché il controllo
è al livello del software
fornito in quanto verrebbe
visto quello di P2 (utente)



INT abilitate

Per primitive non attecchite: INT abilitate: gest di tipo TRAP → No soluz. stato e cerca - stato

Livello sistema

primitive utilibili solo dai moduli I/O: DPL a livello sistema
Moduli offrono system call a moduli più in alto

Il modello di I/O parte ed utilizza delle primitive rinviate per installare system call nella INT di tipo TRAP (analoghe a funzioni già viste in precedenza). Le interruzioni vengono intercettate dall'hardware nel modello sistema che mette in esecuzione il processo che permette di eseguire tali funzioni. Il processo è analogo al driver ma non ha accesso alle strutture dati.

```
extern "C" void c_ceread_n(natl id, char *buf, natl quanti)
{
    if (id >= next_ce) {
        flog(LOG_WARN, "ce non riconosciuto: %d", id);
        abort_p();
    }

    if (!access(buf, quanti, true)) {
        flog(LOG_WARN, "buf non valido\n");
        abort_p();
    }

    des_ce *ce = &array_ce[id];
    sem_wait(ce->mutex);
    ce->buf = buf;
    ce->quanti = quanti;
    outputb(1, ce->iCTL);
    sem_wait(ce->sync);
    sem_signal(ce->mutex);
}
```

access:
.cfi_startproc
int \$TIPO_ACC
ret
.cfi_endproc

```
handler_0:
.cfi_startproc
.cfi_def_cfa_offset 40
.cfi_offset rip, -40
.cfi_offset rsp, -16
call salva_stato
call inspronti
    Mette processo in testa pronti perché era in esecuzione

    movq $0, %rcx
    movq a_p(%rcx, 8), %rax
    movq %rax, esecuzione
    Forse è in esecuzione: pronti I/O > utente
    ↳ array costituito prima con process corrispondente al device

    call carica_stato
    iretq
.cfi_endproc
```

Mette processo in testa pronti perché era in esecuzione

Forse è in esecuzione: pronti I/O > utente
↳ array costituito prima con process corrispondente al device

Processo deve avere stessa funzione del driver: non deve mai interrompersi per rispondere a richieste di INT

SPECIALE: sem_signal notifica trasferito tutti dati. Se viene prima mancarebbe ancora l'ultimo dato

PRIMITIVA: attende interruzione

```
extern "C" void estern_ce(int id)
{
    des_ce *ce = &array_ce[id];

    for (;;) {
        ce->quanti--;
        if (ce->quanti == 0) {
            outputb(0, ce->iCTL);
        }
        char b = inputb(ce->iRBR);
        *ce->buf = b;
        ce->buf++;
        if (ce->quanti == 0) {
            sem_signal(ce->sync);
        }
        wfi();
    }
}
```

```

a_wfi:
    .cfi_startproc
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call apic_send_EOI
    call schedulatore
    call carica_stato
    iretq
    .cfi_endproc

```

: Procedo esterno si blocca e ne viene eseguito un altro.
 Non si salva quello che era in esecuzione perché viene preto dall'array.

modulo I/O unione video/tastiera

```

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* vettore, natl primo, natb quanti);
extern "C" void dmareadhd_n(void* vetti, natl primo, natb quanti);
extern "C" void dmawritehd_n(const void* vettore, natl primo, natb quanti);

extern "C" natq getiomeminfo();

```

primitive fornite dal sistema ad I/O

```

extern "C" natl activate_pe(void f(int), int a, natl prio, natl liv, natb irq);
extern "C" void wfi(); crea processo esterno piuttosto apic a cui è associato
extern "C" void abort_p(); (sostituisce driver)
extern "C" void io_panic();
extern "C" paddr trasforma(void* ff);
extern "C" bool access(const void* start, natq dim, bool writeable, bool shared = true);
extern "C" void fill_gate(natl tipo, vaddr f);

```

Torte Auswahller: formazioni del modello utente + specifiche + primitive fornite dal sistema
 non comuni e senza salva stato e carica stato

fill_io_gates: crea gate della IDT per primitive fornite da I/O : primitive TRAP

```

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

```

activate_pe

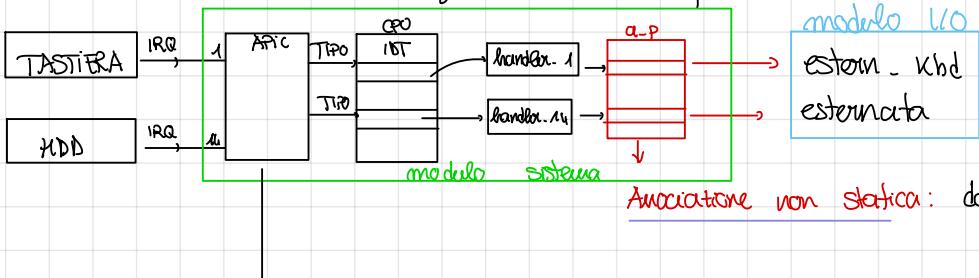
```

p = crea_processo(f, a, prio, liv);
if (p == 0)
    goto err_out;

```

: Stessa di activate_pe ma solo liv. sistema

↓ creiamo il collegamento tra processo e interrupt



Codifica la precedenza che compare anche nel processo: calcoliamo il tipo in base ad enca → assumiamo che tutti i processi esterni alla priorità totale zero data da MIN_PRIO_IO + tipo.

Assegno ad APIC. Nella IDT viene poi inserito l'indirizzo dell' handler corrispondente.

```
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;
}
```

New e delete: mano la multa escludere in quanto le INTR sono abilitate

```
void* operator new(size_t s)
{
    void *p;

    sem_wait(ioheap_mutex);
    p = alloca(s);
    sem_signal(ioheap_mutex);

    return p;
}
```

```
void operator delete(void* p)
{
    sem_wait(ioheap_mutex);
    dealloca(p);
    sem_signal(ioheap_mutex);
}
```

Gestione della console

```
write_console
extern "C" void c_writeconsole(const char* buff, natq quanti) //
{
    des_console *p_des = &console; // non importa spazio condiviso: stessa procedura di writeconsole
    if (!access(buff, quanti, false, false)) { // controllo se posso leggere i parametri
        flog(LOG_WARN, "writeconsole: parametri non validi: %p, %d:", b
uff, quanti);
        abort_p();
    }
    sem_wait(p_des->mutex);
#endif AUTOCONN
    for (natq i = 0; i < quanti; i++)
        char_write(buff[i]);
```

```

#else /* AUTOCORR */
    if (buff[quanti - 2] == '\n')
        quanti -= 2;
    flog(LOG_USR, "%.*s", quanti, buff);
#endif /* AUTOCORR */
    sem_signal(p_des->mutex);
}

```

readconsole

```

extern "C" natq c_readconsole(char* buff, natq quanti) //
{
    des_console *d = &console;
    natq rv;

    if (!access(buff, quanti, true)) {
        flog(LOG_WARN, "readconsole: parametri non validi: %p, %d:", bu
ff, quanti);
        abort_p();
    }

#ifdef AUTOCORR
    return 0;
#endif

    if (!quanti)
        return 0;

    sem_wait(d->mutex);
    startkbd_in(d, buff, quanti);
    sem_wait(d->sincr);
    rv = d->dim - d->cont; → Numero caratteri letti
    sem_signal(d->mutex);
    return rv;
}

```

extern - kbd

```

for(;;) {
    disable_intr_kbd(); → Pausa di leggere RDR: Non voglio nuovo carattere se
    a = char_read_int();

default: → carattere standard
    *d->punt = a;
    d->punt++;
    d->cont--;
    char_write(a);
    if (d->cont == 0) {
        fine = true;
    }
    break;
if (fine)
    sem_signal(d->sincr);
else
    enable_intr_kbd();
wfi();

case '\n': a capo
    fine = true;
    *d->punt = '\0';
    str_write("\r\n");
    break;

case '\b': cancellazione carattere
    if (d->cont < d->dim) {
        d->punt--;
        d->cont++;
        str_write("\b \b");
    }
    break;
}

```

buffer è pieno o carattere è INVIO

```

handler_1:
    .cfi_startproc
    .cfi_def_cfa_offset 40
    .cfi_offset rip, -40
    .cfi_offset rsp, -16
    call salva_stato
    call inspronti

    movq $1, %rcx
    movq a_pc(%rcx, 8), %rax
    movq %rax, esecuzione

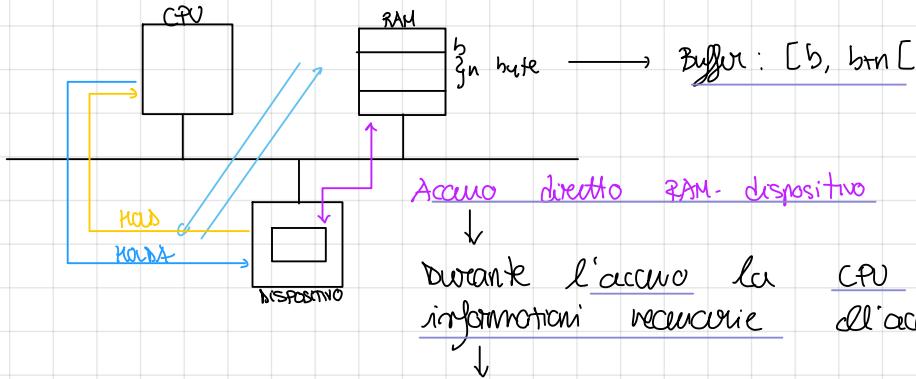
    call carica_stato
    iretq
    .cfi_endproc

```

Penso dire da che punto rispetterà: da prima volta fatta dell'inizio, la seconda da dopo cfi. Questo accade perché se è partito l'handler, necessariamente l'APIC ha mandato la richiesta di INTR. Se questo ha mandato l'interrupt, non c'era nessuna richiesta di INTR con la stessa priorità pendente. Ma richiesta di INTR è in gestione fin quando il processo esterno è ancora in esecuzione.

⚠ Mutua esclusione: L'acceso alla comune viene fatto senza semafori in varie parti. Questo si può fare poiché le interruzioni sono di norma discibilitate e vengono abilitate solo dopo l'inizializzazione di comune. In questo momento in poi possono essere eseguiti i processi esterni.

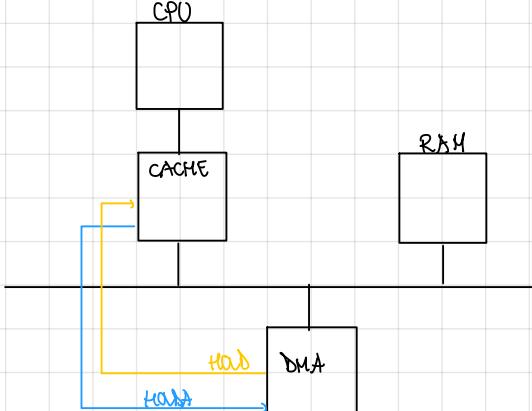
o DMA



Durante l'acceso la CPU è libera in quanto le informazioni necessarie all'acceso sono comunicate prima.

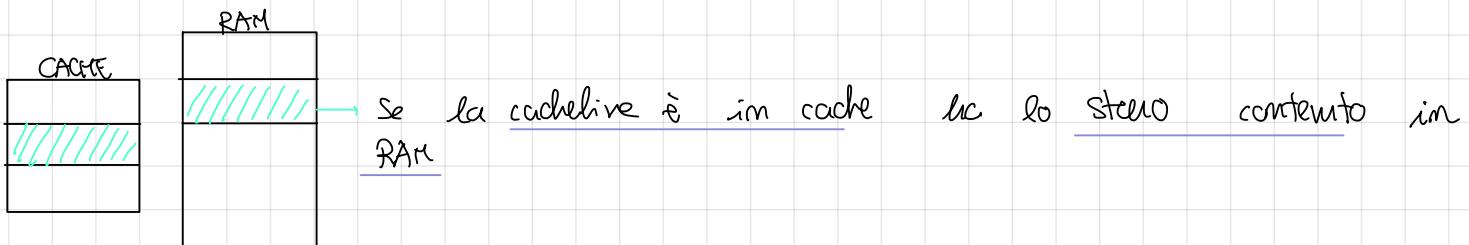
Avendo il bus unico, il dispositivo vi accede con le stesse modalità già usate dalla CPU e deve avere registri per contenere delle informazioni: b, un commutatore per incrementarlo e un commutatore per sapere quando uscire. Non è possibile avere due dispositivi che pilotano lo stesso bus e quindi introduciamo i fili HAD e HDA con i quali viene fatto un bus. Utilizzando HAD richiede l'accesso al bus della CPU. Per concedere il controllo deve mettere le uscite in flit e ottenere HDA. Alla fine, il dispositivo dicondra HAD e mette le linee in flit. La CPU dà precedenza al dispositivo e quindi si fa "ritorno" cicli anche in merito ad interruzioni. La CPU può continuare a fare delle operazioni nel mentre. **Vantaggio** nel caso in cui l'operazione di ingresso debba essere eseguita il prima possibile (scheda di rete).

Agiungiamo la cache

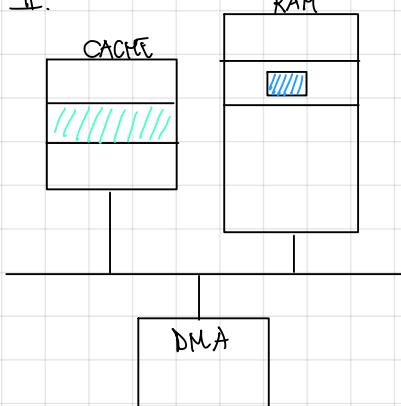


→ Cache è l'oggetto fondamentale su cui si basa la Cache: la RAM può comunicare anche se la CPU non ci rivela niente
 ↓ Caso diverso a seconda di:
 a. Cache write through
 b. Cache write back
 I. Trasferimenti completati di cache-line: il dispositivo fa operazioni di r/w di una sola cache-line intera
 II. Trasferimenti parziali di cache-line

Caso a.



II.

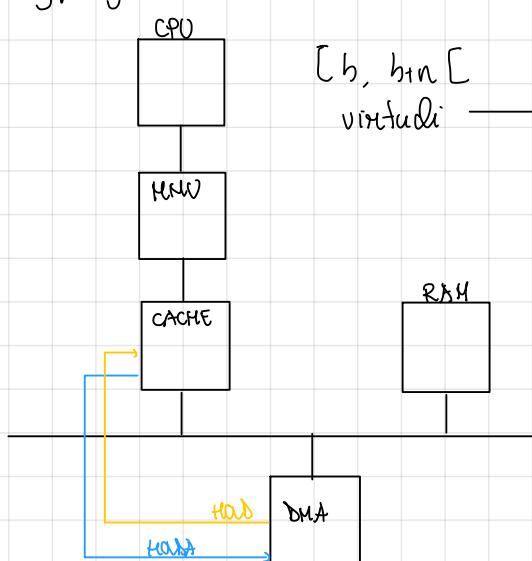


perché facciamo trasf. portati?



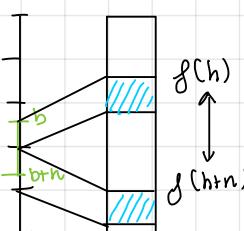
+ modo in cui dati vengono al dispositivo (in genere non riempiono cache line)

Aggiungiamo la MMU



DMA conosce solo indirizzi fisici
↓ Conseguente

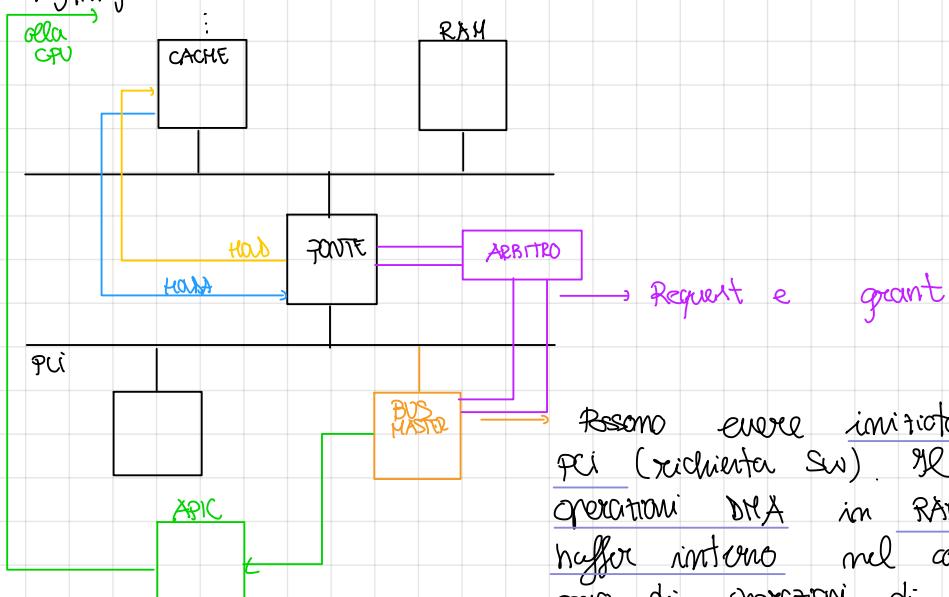
- Quando si comunicano dati a dispositivo per avviare operazioni gli va comunicato $f(h)$ [traduzione]
- Vogliamo comunicare sempre tutto in



2 trasformazioni: da immesso al fine di evitare frange

Alcuni dispositivi possono ricevere una lista di trasformazioni da fare. Anche se ciò non è supportato, comunque il problema dove avere reinolti via SW
3. La traduzione degli indirizzi virtuali non deve cambiare mentre è in corso il trasferimento: possibile in caso di swap-in e swap-out dei processi.

Aggiungiamo il bus PCI



Riscono essere iniziatori di trasferimenti sul bus PCI (richiesta SW). Il ponte è obiettivo e fa le operazioni DMA in RAM mettendo prima i dati in un buffer interno nel caso di operazioni di scrittura. Nel caso di operazioni di lettura, il dispositivo PCI fa operazioni di lettura nello spazio PCI ed il ponte, obiettivo,

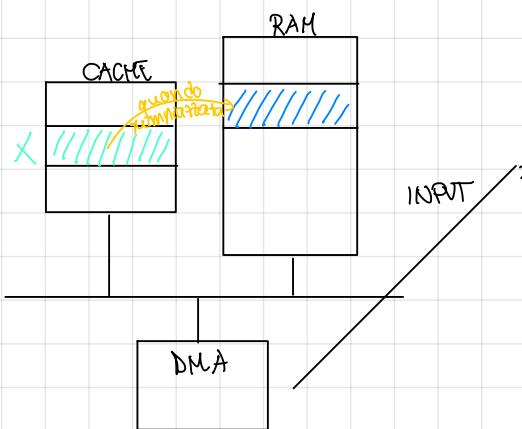
legge i dati della RAM e riporta al dispositivo con tali dati

Come interagiscono con la interruzione? Nel dispositivo, il termine dell'operazione, insieme una richiesta di interruzione all'APIC che a sua volta fa inviare alla CPU. Ecco però comunicare solo che i dati sono arrivati al ponte e non alla RAM → è possibile che l'INTR venga accettata prima che i dati arrivino in RAM

↓ Soluzioni

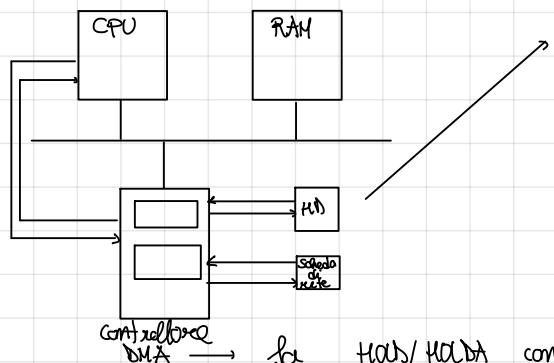
Sol: proprietà ponte: gestisce in ordine FIFO il trasferimento dati dal bus per. Quando riceve l'INTR il bus può leggere un registro del dispositivo cercando una transazione che si accoda alle altre. Quando questa operazione termina notifica la fine del trasferimento dei dati precedenti. Non è in costo in più poiché il driver avrà già dato leggere un registro del dispositivo a seguito dell'INTR in quanto il dispositivo si aspetta una risposta.

Inv: handshake tra apic e ponte → prima di inviare richiesta chiede OK al ponte che glielo concede solo dopo aver trasferito i dati che aveva fino a quel momento. → Rollenta le interruzioni anche se non serve.



È possibile che il controllore cache faccia il write-back prima di ricevere un X davanti al DMA? Si perché può causare rimpiazzamento. Poi accade che arriva a freccia prima di wh. Operare nel secondo caso viene tutto sovrascritto e quindi va bene

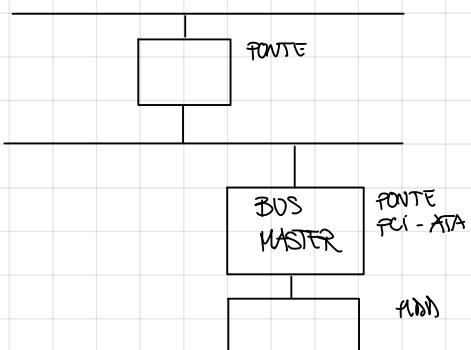
Situazione pre-fci



Venne comunicato al controllore dove trasferire e poi c'è un handshake, ad esempio con l'hard disk, che comunica quando è pronto a trasferire. A questo punto il controllore si imposta sul bus ponendo indirizzo e controllo e l'hard disk invia i dati

Per compatibilità anche oggi l'hard disk pensa di parlare con il controllore DMA che però in realtà

è il ponte PCI-ATA

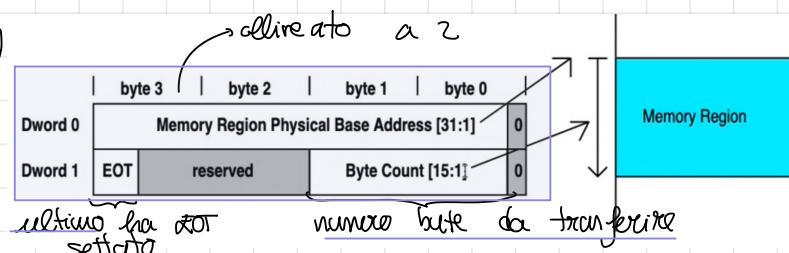


de informazioni su dove leggere e quanti byte trasferire vanno dette nel fonte PCI-ATA ed il resto delle informazioni (nr settori e quali settori) vanno dette dall'ATA. Per gestire le interruzioni sono e fonte PCI-ATA.

Come programmare il bus-mスター

Il bus master può eseguire più operazioni una dietro l'altra e invia un'interruzione alla fine di tutte

↓
Sono dei deviatori in memoria (array) e poi si deve comunicare l'indirizzo dell'array



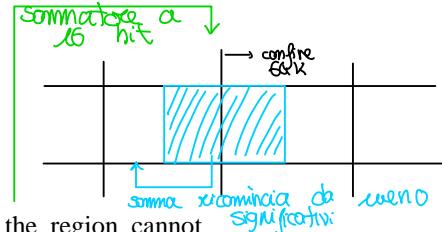
Registri del dispositivo:

The bus master IDE function uses 16 bytes of IO space. All bus master IDE IO space registers can be accessed as byte, word, or Dword quantities. The description of the 16 bytes of IO registers follows:

Offset from Base Address	Register	Register Access
00h	Bus Master IDE Command register Primary	R/W
01h	Device Specific	
02h	Bus Master IDE Status register Primary	RWC
03h	Device Specific	
04h-07h	Bus Master IDE PRD Table Address Primary	R/W
08h	Bus Master IDE Command register Secondary	R/W
09h	Device Specific	
0Ah	Bus Master IDE Status register Secondary	RWC
0Bh	Device Specific	
0Ch-0Fh	Bus Master IDE PRD Table Address Secondary	R/W

Vedi sotto per la descrizione dei registri, operazioni ed ordini e specifiche PCI

Registri Primary e Secondary per funzione parallela ATA



Note

The memory region specified by the descriptor is *further* restricted such that the region cannot straddle a 64K boundary. This means that the byte count can be limited to 64K, and the incrementer for the current address register need only extend from bit [1] to bit [15]. Also, the total sum of the descriptor byte counts must be equal to, or greater than the size of the disk transfer request. If greater than, then the driver must terminate the Bus Master transaction (by resetting bit zero of the command register to zero) when the drive issues an interrupt to signal transfer completion.

2.0. Bus Master IDE Register Description

The bus master IDE function uses 16 bytes of IO space. All bus master IDE IO space registers can be accessed as byte, word, or Dword quantities. The description of the 16 bytes of IO registers follows:

Offset from Base Address	Register	Register Access
00h	Bus Master IDE Command register Primary	R/W
01h	Device Specific	
02h	Bus Master IDE Status register Primary	RWC
03h	Device Specific	
04h-07h	Bus Master IDE PRD Table Address Primary	R/W
08h	Bus Master IDE Command register Secondary	R/W
09h	Device Specific	
0Ah	Bus Master IDE Status register Secondary	RWC
0Bh	Device Specific	
0Ch-0Fh	Bus Master IDE PRD Table Address Secondary	R/W

2.1. Bus Master IDE Command Register

Register Name: Bus Master IDE Command Register
 Address Offset: Primary Channel: Base + 00h
 Secondary Channel: Base + 08h
 Default Value: 00h
 Attribute: Read / Write
 Size: 8 bits

Bit	Description
7:4	Reserved. Must return 0 on reads.
3	Read or Write Control: This bit sets the direction of the bus master transfer: when set to zero, PCI bus master reads are performed. When set to one, PCI bus master writes are performed. This bit must NOT be changed when the bus master function is active.
2:1	Reserved. Must return 0 on reads.
0	Start/Stop Bus Master: Writing a '1' to this bit enables bus master operation of the controller. Bus master operation begins when this bit is detected changing from a zero to a one. The controller will transfer data between the IDE device and memory only when this bit is set. Master operation can be halted by writing a '0' to this bit. All state information is lost when a '0' is written; Master mode operation cannot be stopped and then resumed. If this bit is reset while bus master operation is still active (i.e., the Bus Master IDE Active bit of the Bus Master IDE Status register for that IDE channel is set) and the drive has not yet finished its data transfer (The Interrupt bit in the Bus Master IDE Status register for that IDE channel is not set), the bus master command is said to be aborted and data transferred from the drive may be discarded before being written to system memory. This bit is intended to be reset after the data transfer is completed, as indicated by either the Bus Master IDE Active bit or the Interrupt bit of the Bus Master IDE Status register for that IDE channel being set, or both.

2.2. Bus Master IDE Status Register

Register Name:	Bus Master IDE Status Register
Address Offset:	Primary Channel: Base + 02h Secondary Channel: Base + 0Ah
Default Value:	00h
Attribute:	Read/Write Clear
Size:	8 bits

Bus Master IDE Status Register

Bit	Description
7	Simplex only: This read-only bit indicates whether or not both bus master channels (primary and secondary) can be operated at the same time. If the bit is a '0', then the channels operate independently and can be used at the same time. If the bit is a '1', then only one channel may be used at a time.
6	Drive 1 DMA Capable: This read/write bit is set by device dependent code (BIOS or device driver) to indicate that drive 1 for this channel is capable of DMA transfers, and that the controller has been initialized for optimum performance.
5	Drive 0 DMA Capable: This read/write bit is set by device dependent code (BIOS or device driver) to indicate that drive 0 for this channel is capable of DMA transfers, and that the controller has been initialized for optimum performance.
4:3	Reserved. Must return 0 on reads.
2	Interrupt: This bit is set by the rising edge of the IDE interrupt line. This bit is cleared when a '1' is written to it by software. Software can use this bit to determine if an IDE device has asserted its interrupt line. When this bit is read as a one, all data transferred from the drive is visible in system memory.
1	Error: This bit is set when the controller encounters an error in transferring data to/from memory. The exact error condition is bus specific and can be determined in a bus specific manner. This bit is cleared when a '1' is written to it by software.
0	Bus Master IDE Active: This bit is set when the Start bit is written to the Command register. This bit is cleared when the last transfer for a region is performed, where EOT for that region is set in the region descriptor. It is also cleared when the Start bit is cleared in the Command register. When this bit is read as a zero, all data transferred from the drive during the previous bus master command is visible in system memory, unless the bus master command was aborted.

2.3. Descriptor Table Pointer Register

Register Name:	Descriptor Table Pointer Register
Address Offset:	Primary Channel: Base + 04h Secondary Channel: Base + 0Ch
Default Value:	00000000h
Attribute:	Read / Write
Size:	32 bits

Descriptor Table Pointer Register

Bit	Description
31:2	Base address of Descriptor table. Corresponds to A[31:2]
1:0	reserved

The Descriptor Table must be Dword aligned. The Descriptor Table must not cross a 64K boundary in memory.

3.0. Operation

3.1. Standard Programming Sequence

To initiate a bus master transfer between memory and an IDE DMA slave device, the following steps are required:

- 1) Software prepares a PRD Table in system memory. Each PRD is 8 bytes long and consists of an address pointer to the starting address and the transfer count of the memory buffer to be transferred. In any given PRD Table, two consecutive PRDs are offset by 8-bytes and are aligned on a 4-byte boundary.
- 2) Software provides the starting address of the PRD Table by loading the PRD Table Pointer Register . The direction of the data transfer is specified by setting the Read/Write Control bit. Clear the Interrupt bit and Error bit in the Status register.
- 3) Software issues the appropriate DMA transfer command to the disk device.
- 4) Engage the bus master function by writing a '1' to the Start bit in the Bus Master IDE Command Register for the appropriate channel.
- 5) The controller transfers data to/from memory responding to DMA requests from the IDE device.
- 6) At the end of the transfer the IDE device signals an interrupt.
- 7) In response to the interrupt, software resets the Start/Stop bit in the command register. It then reads the controller status and then the drive status to determine if the transfer completed successfully.

3.1. Data Synchronization

When reading data from an IDE device, that data may be buffered by the IDE controller before using a master operation to move the data to memory. The IDE device driver in conjunction with the IDE controller is responsible for guaranteeing that any buffered data is moved into memory before the data is used.

The IDE device driver is required to do a read of the controller Status register after receiving the IDE interrupt. If the Status register returns with the Interrupt bit set then the driver knows that the IDE device generated the interrupt (important for shared interrupts) and that any buffered data has been flushed to memory. If the Interrupt bit is not set then the IDE device did not generate the interrupt and the state of the data buffers is unknown.

When the IDE controller detects a rising edge on the IDE device interrupt line (INTRQ) it is required to:

- Flush all buffered data
- Set the Interrupt bit in the controller Status register
- Guarantee that a read to the controller Status register does not complete until all buffered data has been written to memory.

Another way to view this requirement is that the first read to the controller Status register in response to the IDE device interrupt must return with the Interrupt bit set and with the guarantee that all buffered data has been written to memory.

3.1. Status Bit Interpretation

The table below gives a description of how to interpret the Interrupt and Active bits in the Controller status register after a DMA transfer has been started.

Interrupt	Active	Description:
0	1	DMA transfer is in progress. No interrupt has been generated by the IDE device.
1	0	The IDE device generated an interrupt. The controller exhausted the Physical Region Descriptors. This is the normal completion case where the size of the physical memory regions was equal to the IDE device transfer size.
1	1	The IDE device generated an interrupt. The controller has not reached the end of the physical memory regions. This is a valid completion case where the size of the physical memory regions was larger than the IDE device transfer size.
0	0	This bit combination signals an error condition. If the Error bit in the status register is set, then the controller has some problem transferring data to/from memory. Specifics of the error have to be determined using bus-specific information. If the Error bit is not set, then the PRD's specified a smaller size than the IDE transfer size.

4.0. Error Conditions

IDE devices are sector based mass storage devices. The drivers handle errors on a sector by sector basis; either a sector is transferred successfully or it is not.

If the IDE DMA slave device never completes the transfer due to a hardware or software error, the Bus Master IDE command will eventually be stopped (by setting Command Start bit to zero) when the driver times out the disk transaction. Information in the IDE device registers will help isolate the cause of the problem.

If the controller encounters an error while doing the bus master transfers it will stop the transfer (ie. reset the Active bit in the Command register) and set the ERROR bit in the Status register. The controller does not generate an interrupt when this happens. The device driver can use device specific information (e.g.; PCI Configuration Space Status register) to determine what caused the error.

Whenever a requested transfer does not complete properly, information in the IDE device registers (Sector Count) can be used to determine how much of the transfer was completed and to construct a new PRD table to complete the requested operation. In most cases the existing PRD table can be used to complete the operation.

5.0. PCI Specifics

Bus master IDE controllers built to attach to a PCI bus must have the following characteristics:

- 1) The Class Code in PCI configuration space indicates IDE device (top two bytes have the value 0x0101) and bit 7 of the Programming Interface register (offset 0x09) in PCI configuration space must be set to 1 to indicate that the device supports the Master IDE capability.
- 2) The control registers for the controller are allocated via the devices Base Address register at offset 0x20 in PCI configuration space.
- 3) In the controller Status register the Error bit will be set and the Active bit reset if any of the following conditions occur on the PCI bus while the controller is doing a master operation on the bus. The exact cause can be determined by examining the Configuration Space Status register.

Error Condition	Configuration Space Status bits
Target Abort	Anytime bit 12 of the Config Space Status register is set.
Master Abort	Anytime bit 13 of the Config Space Status register is set.
Data Parity Error Detected	Anytime bit 8 of the Config Space Status register is set.

ESERCITO: trasferimento in ingresso

```
volatile bool done = false;
extern char vv[]; buffer
const natl BUFSIZE = 65536;
extern natl prd[]; → prd:
→ dev_out[dev]
extern "C" void a_bmide();
extern "C" void c_bmide()
```

```
int main()
{
    natb nn = BUFSIZE / 512; → numero settori
    natb lba = 0; → logical block address fattoria
    natb bus = 0, dev = 0, fun = 0; numero di bus, device, funzione ha master
    clear_screen(0x0f);
→ assegna bus, dev, fun
    if (!bm_find(bus, dev, fun))
        printf("bm non trovato!\n");
    printf("PCI-ATA at %2x:%2x:%2x\n", bus, dev, fun);
    bm_init(bus, dev, fun); → initializzazione

    apic_set_VECT(14, HD_VECT);
    gate_init(HD_VECT, a_bmide); } per interruzione
    apic_set_MIRQ(14, false); HDD (1u)

    for (int i = 0; i < BUFSIZE; i++)
        vv[i] = '-';

    printf("primi 80 caratteri di vv:\n");
    for (int i = 0; i < 80; i++)
        char_write(vv[i]);
    printf("ultimi 80 caratteri di vv:\n");
    for (int i = BUFSIZE - 80; i < BUFSIZE; i++)
        char_write(vv[i]);
```

```
extern "C" void a_bmide();
extern "C" void c_bmide()
{
    → flag globale
    done = true;
    hdd bm_ack(); bus write
    hdd hd_ack_intr();
    apic apic_send_EOI();
}
```

```
prd[0] = reinterpret_cast<natq>(vv);
prd[1] = 0x80000000 | ((nn * 512) & 0xFFFF);
bm_prepare(reinterpret_cast<natq>(prd), false);
```

prevediamo prd (vv indirizzo fisico)

prendiamo nr byte e scegliamo solo i primi 16 settando bit più significativo

→ fa le cose richieste dalla documentazione

```
void bm_prepare(natq prd, bool write)
{
    outputl(prd, iBMDTPR);
    natb work = inputb(iBMCMD);
    if (write) {
        work &= ~0x8;
    } else {
        work |= 0x8;
    }
    outputb(work, iBMCMD);
    work = inputb(iBMSTR);
    work |= 0x6;
    outputb(work, iBMSTR);
}
```

```

hd_enable_intr();
hd_start_cmd(lba, nn, READ_DMA);
bm_start();

```

↓ Punto 4 documentazione

```

void bm_start()
{
    natb work = inputb(iBMCM);
    work |= 1;
    outputb(work, iBMCM);
}

```

```

printf("aspetto l'interrupt...\n");
while (!done)
;

printf("primi 80 caratteri di vv:\n");
for (int i = 0; i < 80; i++)
    char_write(vv[i]);

printf("ultimi 80 caratteri di vv:\n");
for (int i = BUFSIZE - 80; i < BUFSIZE; i++)
    char_write(vv[i]);

```

Per fare il trasferimento in più passaggi detto prima
2 casi di regione non allineata:

```

// primo_tratto
// |-----|
// +-----+-----+
// |       XXXXXXXX|XXXX      |
// +-----+-----+
// |<---vv-->|
//
// +-----+-----+
// |       XXXX---|      |
// +-----+-----+
// |vv| 

```

Caso 1

```

prd[0] = reinterpret_cast<natq>(vv);
prd[1] = 0x00000000 | (primo_tratto & 0xFFFF);
prd[2] = tmp2 + 65536;
prd[3] = 0x80000000 | (BUFSIZE - primo_tratto);

```

```

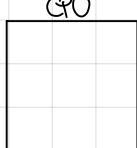
natq tmp1 = reinterpret_cast<natq>(vv);
natq tmp2 = tmp1 & ~((1U<<16)-1);

```

inizio regione che contiene frag 2

La CPU moderna

• VEWICIZZAZIONE delle OPERAZIONI



Legge Dennard Scaling: aumentando la frequenza aumenta la potenza (il qualsiasi) e riducendo la dimensione dei transistor l'area diminuisce (risparmio di qualsiasi). Oggi non è più possibile aumentare la frequenza perché non è possibile ridurre la potenza → si fanno fare più cose per clock: esecuzione in pipelining o speculativa.

pipelining

↓ fasi esecuzione istruzione



I circuiti che fanno prelievo e decodifica non fanno niente qua: li uso per la prossima istruzione

Penso dividere ulteriormente questa cosa allargabile a tutte le singole fasi: oso una catena di istruzioni → è più veloce non col giro della singola istruzione ma globalmente in quanto i tempi delle istruzioni si sovrappongono e migliora anche il numero di istruzioni completate al secondo

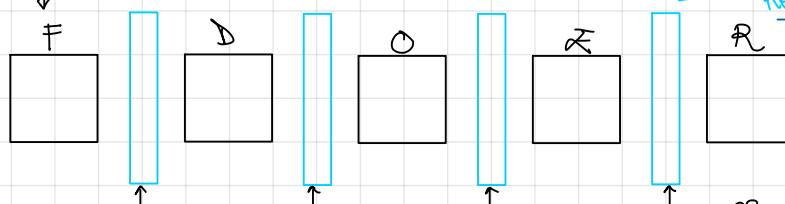
↓ Come fare

È possibile prelevare l'istruttore $i+1$ mentre si decodifica la i ? No perché le istruzioni hanno dimensione diversa e possono superare solo dopo la decodifica. Inoltre, il meccanismo funziona solo se le istruzioni sono recyclate, ovvero fanno cose soltanto in determinati punti della loro esecuzione. Questo non accade nei processori Intel in quanto le istruzioni non sono segnate per l'esecuzione in pipeline → corali tabelle RISC contrapposte ad Intel (SISD). Nelle RISC si fanno solo ADD e STORE in memoria e poi si lavora sui registri: OP DST, SRC1, SRC2. I dati inoltre

sono del tipo OP REG, offset e non vi hanno sui flag. Le istruzioni inoltre sono sempre lunghe 6 byte e strutturate così segue



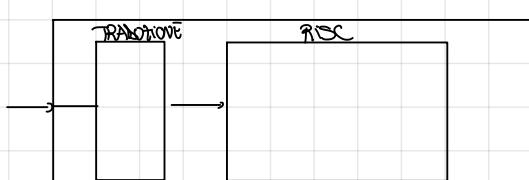
I hit che nominano i registri sono inoltre tenuti nella stessa posizione. Queste cose risolvono il problema di far regolarità e di fare il prelievo di istruzioni consecutive nella semplicità aiuta inoltre la pipeline



Registri di pipeline: memorano dati intermedi

Clock: tempo successivo tra registro e successivo → Affinché il pipelining sia efficace le istruzioni devono fare poco

Nell'Intel



: da traduzione utilizzata registri di appoggio
↓ Esempio:
ADD %RAX, 1000 (%RCX, %RBX, 6)
diventa

SHL R3, %RCX, \$4
 ADD R2, %RDX, R3
 LD R1, 1000(R2)
 ADD %RAX, %RAX, R1

} Istruzioni elementari di tipo RISC

Naturalmente conviene se ogni istruzione intel corrisponde ad una sola istruzione RISC

Ha come lavora il traduttore? Fa una prefetch in cui pedisca 16 byte della memoria a regioni naturali e nei quanti 16 byte decodifica le istruzioni ma alla volta fraendole e scendendo alla pipeline. Inoltre ha anche una cache delle traduzioni ed il compilatore la utilizza per ottimizzazione. Nel fare una pipeline bisogna stare attenti alle alee: arrivando troppo l'esecuzione delle istruzioni possono crearsi inconveniente rispetto a come il processore le avrebbe eseguite una dietro l'altra oppure situazioni in cui non ci sono possibili prelevare una nuova istruzione ad ogni clock. esistono 3 tipologie di alee

- **strutturali**: 2 istruzioni nella pipeline stessa risrono
- **dati**: un'istruzione ha bisogno di dati di un'altra istruzione non ancora disponibili
- **controllo**: per sapere quel è l'istruzione successiva ad una di solito si deve avere accenni all'esecuzione

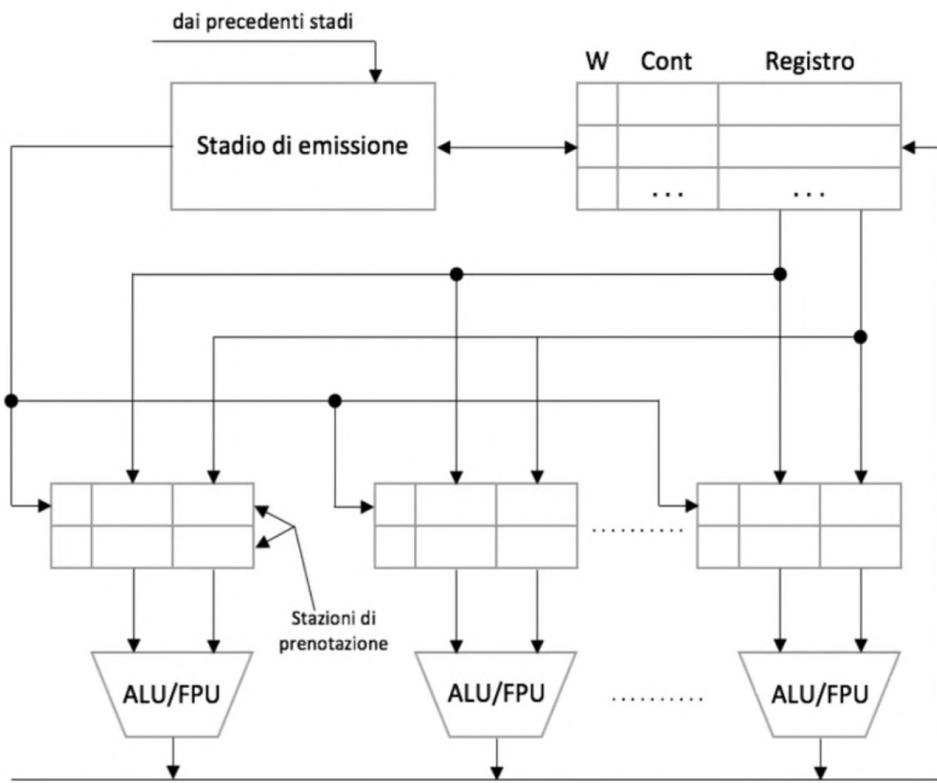
Risoluzione: La correttezza del programma è assicurata mettendo in stallo qualche unità della pipeline cioè fermando un'istruzione e le precedenti e facendo condurre avanti le altre. Questo rende le alee strutturali, esse di controllo. Ciascun stallo ci fa perdere un clock ed è massimo temporaneo alle situazioni seguenti. Se le due istruzioni possono essere eseguite anche tramite circuiti di bypass che permettono ad esempio di riportare indietro dati prodotti utili per istruzioni successive. Per le alee di controllo si fa branch prediction, cercando quindi di indovinare il risultato → del 16 al 25% delle istruzioni in un programma sono salti. Si utilizza una predizione statica: se un salto è di indietro probabilmente è in ciclo probabilmente eseguito molte volte. Se il salto è in avanti probabilmente è un errore. Quello che è importante è di cominciare a predire statica le istruzioni vengono sempre predette allo stesso modo. Per fare meglio si fa una predizione dinamica utilizzando una sorta di cache indirizzata facendo hash dell'indirizzo dell'istruzione. Per ogni istruzione di salto ci si ricorda una serie di hit che sono il recordo del comportamento dell'istruzione in passato. Si utilizzano più hit per memorizzare più volte precedenti e sbagliare di meno le predizioni. A differenza della cache dei dati non utilizziamo i tag poiché la previsione può essere sbagliata in ogni caso, sia nel caso in cui l'informazione sia relativa ad un'altra istruzione, sia nel caso in cui la predizione sia sbagliata. Quando la predizione è sbagliata nella pipeline ci sono istruzioni prelevate a causa della predizione sbagliata. Queste vanno cancellate e prelevate quelle giuste. L'annullamento consiste nel cancellare e basta in quanto non hanno ancora salvato il risultato in modo tale che al momento del salvataggio del risultato questo venga scartato. Il problema della pipeline sono gli stalli in quanto coinvolgono tutte le istruzioni e non solo quelle che li provocano. Gli stalli sono spesso provocati dalle istruzioni che interagiscono con la memoria che non trovano i dati interessanti in cache. Questo provoca stalli di centinaia di clock da cache

di livello 1 è divisa in istruzioni e dati. Altrimenti ci sarebbero sempre delle strutture perché le istruzioni di load si troverebbero sempre in una struttura comune alle altre perché devono leggere dati che sarebbero sempre nella stessa cache. → come velocizziamo? → non è necessario mandare tutto in stadio?

```
for (int i = 0; i < 10000; i++)
    v1[i] = v2[i] + v3[i];
```

quindi istruzioni non lo fanno in maniera automatica → è compito del processore che utilizza la branch prediction per ipotizzare i risultati dei cicli precedenti non eseguiti. La tecnica viene usata inoltre per eseguire alcune istruzioni prima di scoprire se ce ne sono necessarie → esecuzione speculativa (ad esempio ipotizzando il risultato di una condizione in un solo)

l'ordine: penichiamo il primo momento utile in cui possiamo eseguire un'istruzione ed eseguendola sentire considerare l'ordine in cui si trova nel programma. Organizziamo quindi la pipeline in un modo diverso



Per sapere quando gli operandi sono pronti un flag W che mi dice se c'è un'istruzione → associato ad ogni registro che vuole scrivere nel registro. Se è 0 indica che gli operandi sono pronti. Anche se arrivano più AW ed altrimenti garantito di leggere gli stessi dati del caso precedente, non immortala più l'ordine.

↓ vanno gestite le

Dipendenze tra le istruzioni
(se gestite male generano delle)

- 1. DATI: istruzioni i e j con i che viene prima di j nel flusso di esecuzione sequenziale. C'è dipendenza sui dati e i prodotti un risultato che serve a j (ADD %RAX, %RBX e SUB %RCX, %RDX)
- 2. NOMI:
 - a. ANTIDIPENDENZE → j riceve in un generico registro Rx e i legge Rx
 - b. SIPENDENZE DI USCITA → j scrive su Rx e i scrive anche su Rx

→ Alliniamo tutte AW con una stazione di prenotazione circondata davanti: memorizzano le istruzioni elementari con gli operandi. Sostituiamo prelievo operandi, esecuzione e scrittura risultato. Aggiungiamo però lo stadio di emissione che riceve le istruzioni da prelievo e decodifica: capisce quando un'istruzione può essere inserita in una stazione di prenotazione perché da questo momento in poi verrà prima o poi eseguita quando gli operandi e la sua AW sono pronti.

	R	W
R	-	.
W	.	.

- Dipendenze sui dati
- Anti-dipendenza
- Dipendenza di controllo

Se le dipendenze sui nomi possono creare loop: ADD R1, R2, R3 } anti-dip.
SUB R2, R4, R5 }

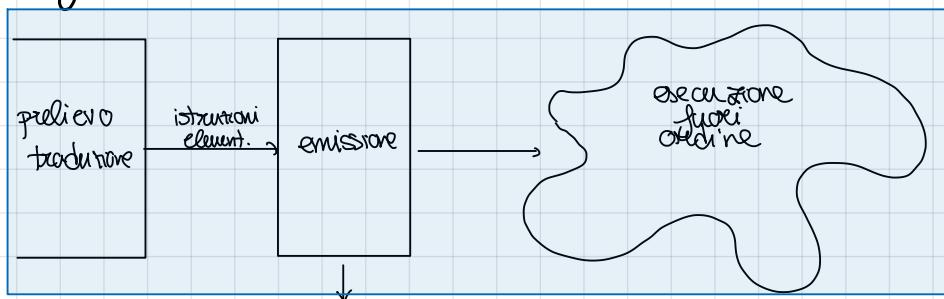
Se eseguo prima la SUB e poi la ADD ottengo un risultato errato

3. CONTROLLO:

Je dopo
altra istruzione → Eseguite in base alla decisione
della jump. dipendenze da cosa
succede in tutto il programma
(dipendenze per controllo)

Il processore non può sapere perché dovrebbe analizzare tutto il programma → le considera dipendenze per controllo da dopo la jump

organizzazione interna



riconosce dip. e anche loop. Se non sono possibili
altri modi si utilizzano gli stalli
↓ riconoscimento di dipendenze

SCOREBOARD

R1			
R2			
R3			
:			
Rn			

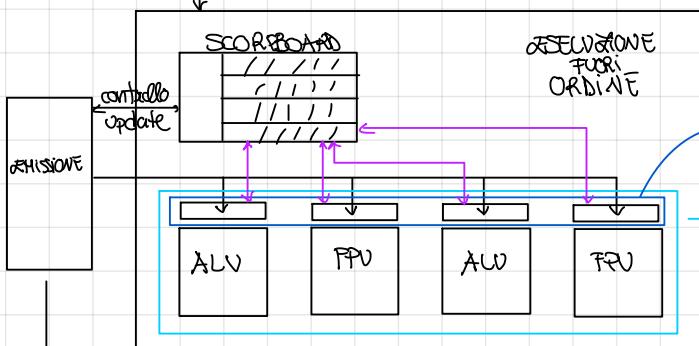
→ Gestita dello stato di emissione.

w cont
contento

↓ quant istruzioni vogliono leggere dal registro

l se nell'esecuzione c'è un'istruzione che vuol scrivere sul registro
↓ Come vengono riconosciute (ridere →)

- Dipendenze sui dati: J: op R1, R2, R3 → dipendenze se W di R1, R3 = 1
- Dipendenze sui nomi: Si guarda il registro di uscita e si vede se w / cont ≠ 0
- Dipendenze sul controllo: Se c'è istruzione di controllo nello stato di emissione



stazioni di prenotazione: se le code si riempiono
si ha un'overflow strutturale
più kW / FPV: da già risolte

Metodo più semplice risolvere: Stallo → programma si ferma ogni volta che c'è dipendenza. Questo accade raramente e quindi ci occupiamo di ridurre di nuovo il quindì interruzioni anche se dipendono dai dati da altre interruzioni emesse non ancora terminate. Però collegiamo le stazioni di prenotazione alla scoreboard in quanto aggiornano il risultato nel registro ed aggiornano anche gli altri campi d'interruzione appena emessa va nella stazione di prenotazione e viene eseguita solo quando i sorgenti sono pronti (monitmando costantemente la scoreboard). È possibile inoltre ricevere quale stazione produceva il risultato in modo da poter lavorare i registri. Per quanto riguarda le dipendenze sui nomi, queste possono essere eliminate sentire offrire la semanticà del programma → Tomasulo → Ho la scoreboard e l'interruzione com i sorgenti i sorgenti già pronti li leggo subito e li metto nella stazione di prenotazione. Quelli non pronti ricordo la stazione di prenotazione che produceva il risultato → Risolve tutte le dipendenze sui nomi perché prende subito il valore già pronto o no da dove riceverci il valore. È possibile, in alternativa, utilizzare la tecnica di renomina dei registri → di distinguere im logici, ma di istruzioni, e registri finiti, ma della scoreboard. Da corrispondenza fra questi è codificata la corrispondenza viene cambiata ogni volta che un'istruzione vuole ricevere un qual registro e gli operandi sorgenti vengono introdotti con tel corrispondente. I registri finiti possono avere molti senti codificare istruzioni ed il loro aumento non richiede la modifica del programma. Utilizzano infine le dipendenze sul controllo. Finché non termina, tutte le altre dipendono da questa. Utilizzano sempre il branch predictor per produrre il risultato. Se viene predetto che non si salterà, le istruzioni successive al sotto vengono senti in esecuzione ed eseguite appena possibile producendo dei risultati. Se la predizione è sbagliata vanno buttati via i risultati (esecuzione speculativa) ed è quindi importante che il branch predictor sbagli il meno possibile. D' algoritmo di Tomasulo può avere etero introducendo un register buffer (ROB) → coda: i sorgenti sono già pronti o referiti tramite la stazione che produceva il risultato, la destinazione va a finire nel ROB dove le istruzioni inseriscono il risultato in ordine di esecuzione → Memorizza i dati finiti non so se l'istruzione doveva essere eseguita o no → Si ha anche un flag che mi dice se l'istruzione è terminata e le istruzioni vengono ritirate allo della testa: stadio di ritiro → Monitora il ROB e vede quando è terminata l'interruzione in testa: in questo caso rende attuali le modifiche → Se l'istruzione in cima è operativa si notifica il risultato, se è un sotto si salta invece se avevamo indovinato o no. Se aveva indovinato il ROB sotto è valida, altrimenti si svuota tutto il ROB. Con la renomina dei registri bisogna ricordarsi i registri fisici per ogni registro logico: quello in corso di calcolo (speculativo) e quello che contiene l'ultimo valore che andava vicinamente calcolato (non speculativo), si ha una coda e se l'istruzione operativa arriva alla cima si salva il risultato trascrivendo i dati del registro speculativo a quello non speculativo.

Le istruzioni di LOAD e STORE hanno gli stessi problemi di dipendenze ma sono più complicate dai registri: LOAD dst, base [reg]
STORE src, base [reg]

Più complicato perché l'indirizzo va sempre calcolato
 ↓
 Esecuzione in 2 fasi: calcolo indirizzo + esecuzione op.

Dipendente visto dopo prete e cuore
in LOAD/STORE buffer

Inoltre, in caso di speculazione, devo tenere tutto in LOAD/STORE buffer finché non sono sicuro, altrimenti occorrei troppe speculazioni che avvicede, infine, in caso di recalculazioni su interruption? Si salva tutto nel ROB mentre però potrebbe eseguirsi e si eseguono solo se in cima al ROB.