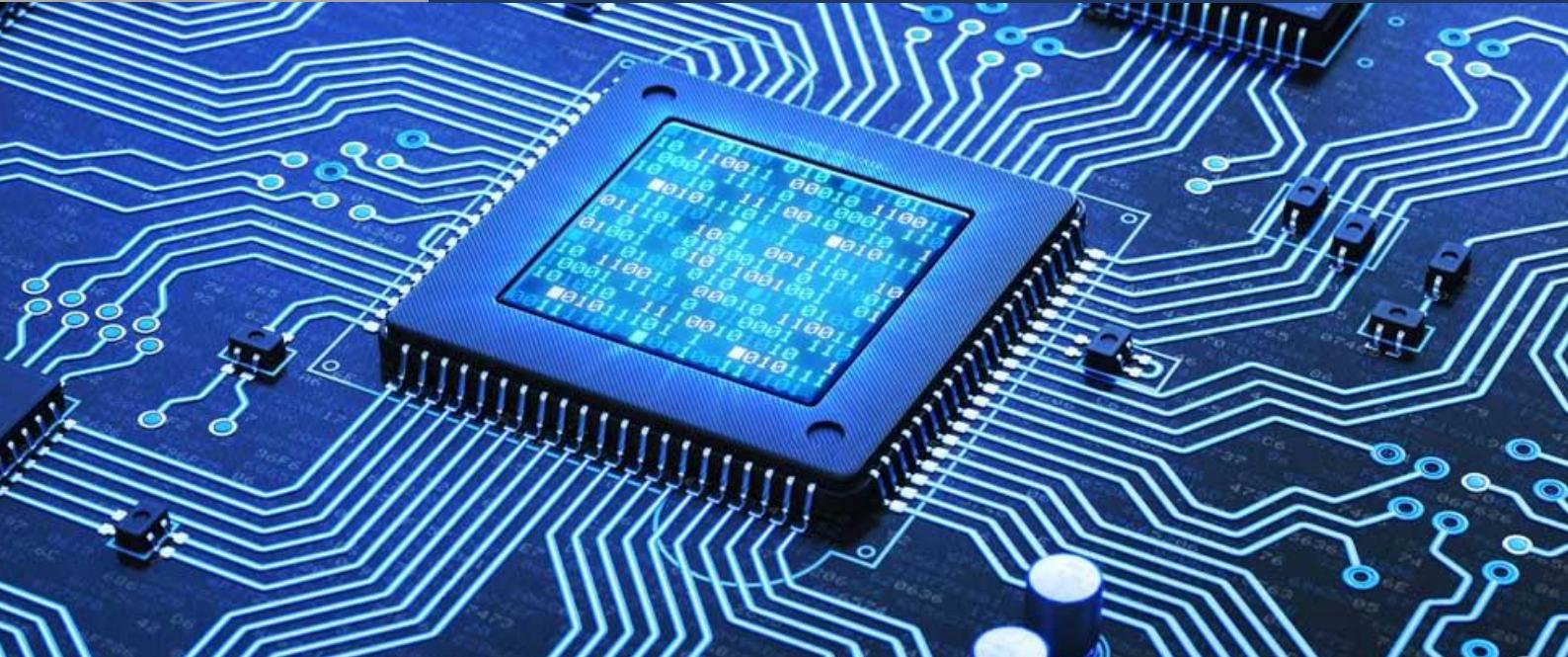


Sansone Giacomo

Revisione a cura di Marco Lampis

CALCOLATORI ELETTRONICI



I presenti appunti sono stati presi durante le lezioni del professor Lettieri per il corso di Calcolatori Elettronici al C.d.L. di Ingegneria Informatica all'Università di Pisa, nell'A.A. 2020-2021.

Gli appunti non sono stati rivisti dal docente, e non costituiscono materiale ufficiale per il corso.

Alcune delle immagini utilizzate sono tratte dalle dispense ufficiali, disponibili all'indirizzo <https://calcolatori.iet.unipi.it>.

Nella seconda parte degli appunti, a partire dal capitolo sull'accesso a basso livello, si fa riferimento alla versione 2.5 della libce, 7.0 degli esempi di IO e 6.5 del Nucleo.

In bocca al lupo!

GS

SUL MANCHESTER BABY	5
SULL'ARCHITETTURA DEL CALCOLATORE	7
SUGLI INDIRIZZI E LO SPAZIO DI MEMORIA	14
SULLA PROGRAMMAZIONE ASSEMBLER E C++	19
SULLA MEMORIA CACHE	56
PER UN PRIMO ACCESSO A BASSO LIVELLO	61
SULLE PERIFERICHE	63
SULLE INTERRUZIONI	73
SUL BUS PCI	82
SUL DMA	85
SULLE ECCEZIONI	89
SULLA PROTEZIONE	93
SUI PROCESSI	98
SULL'IMPLEMENTAZIONE SOFTWARE DEI PROCESSI	105
SUI SEMAFORI	108
ALCUNE PRIMITIVE D'ESAME	111
LA PRIMITIVA DELAY	115
SULLA PAGINAZIONE	117
ANCORA SULLA PAGINAZIONE	126
SULLA PAGINAZIONE NEL NUCLEO	134
L'I/O NEL NUCLEO	140

Sul Manchester Baby

Il corso completa il lato hardware introducendo tre aspetti fondamentali: le interruzioni, la protezione e la memoria virtuale. Questi tre aspetti insieme permettono di realizzare la multiprogrammazione, ovvero la capacità per un processore di eseguire più programmi contemporaneamente. Ciò non è la conseguenza di avere più processori all'interno di un computer, ma della capacità di alternare esecuzioni di programmi diversi. Spesso risulta complicato capire *chi fa cosa* all'interno di un calcolatore. Infatti, nei computer che usiamo, c'è un'ampia dose di software che ci separa dal processore e dalle componenti più elementari del calcolatore. Per chiarirci le idee, torniamo al 1948 con il Manchester Baby o SSEM, un calcolatore sperimentale nato per testare alcuni aspetti architetturali.

Nell'SSEM, troviamo un'impostazione della memoria per come si è sviluppata negli anni. Ci sono diverse celle, ciascuna con un indirizzo tramite la quale può essere acceduta (sia in lettura che in scrittura) senza bisogno di scorrere tutte quelle precedenti. La memoria non contiene solo gli operandi delle elaborazioni, ma anche le istruzioni, e quindi i programmi stessi. Questa assenza di distinzione si è sviluppata negli anni, portando con sé numerosi vantaggi e svantaggi. L'idea che vi stava alla base, introdotta da von Neumann ma alla lunga superata, è quella di modificare le istruzioni con il programma stesso, in modo da dover scrivere meno codice (in relazione anche alla piccola dimensione della memoria). L'istruzione dei programmi avveniva in modo sequenziale, ma già si aveva la possibilità di fare salti in avanti nelle istruzioni (condizioni) e salti indietro (cicli). Si voleva soprattutto realizzare algoritmi matematici, cosa che necessitava di queste funzionalità.

La memoria di SSEM è implementata tramite un tubo catodico, in modo che potesse essere anche visualizzata bit per bit attraverso uno schermo. Comprendeva 32 celle, ciascuna da 32 bit. Nonostante la memoria sia osservabile, da nessuna parte si vedono gli indirizzi: è compito del calcolatore effettuare questo accesso. Con dei pulsanti era possibile modificare la memoria, per poter scrivere il programma e inserire i dati. Logicamente, non essendoci un assemblatore che si occupava della traduzione da codice mnemonico a codice macchina, era necessario scrivere direttamente il codice macchina delle istruzioni. Il processore comprendeva 3 registri da 32 bit: *A*, accumulatore, *C_I*, per puntare l'istruzione corrente e *P_I*, per memorizzare l'istruzione attuale. Vi erano solo alcune istruzioni:

- JMP e JRC, rispettivamente per modificare il valore di *C_I* con un nuovo valore e per sommarvi il contenuto di una locazione di memoria.
- LDN, per caricare in *A* l'opposto del contenuto di una locazione di memoria (in complemento alla radice)
- STO, per caricare in una locazione il contenuto di *A*.
- SUB, per sottrarre ad *A* il contenuto di una locazione di memoria. Nella realizzazione del calcolatore, si riteneva che la sottrazione potesse essere più versatile della somma, dato che questa si otteneva dalla prima tramite un opposto.
- CMP, per incrementare di 1 *C_I* nel caso in cui *A* sia negativo. Questo metodo era molto diffuso nei processori che non appartenevano alla famiglia Intel, anche se alla lunga si è preferito il confronto di operandi per condizionare un salto.
- STP, per bloccare l'esecuzione del programma.

Programmare SSEM significa impostare la memoria a dei valori iniziali, tenendo presente che la prima istruzione eseguita è quella alla prima locazione di memoria. Da quel momento in avanti, il programma si evolve in modo deterministico, con il processore che preleva un'istruzione, incrementa C_I e la esegue (eventualmente modificano C_I o fermandosi se si trattava di un'istruzione di *STOP*). L'unica cosa che conta è quindi il modo con cui è caricata la memoria prima dell'esecuzione. Un programma in codice mnemonico per calcolare l'opposto di un numero contenuto in $mem[addr]$ potrebbe essere

```
LDN addr // carico in A l'opposto di mem[addr]
CMP      // se A è negativo, allora mem[addr] è positivo e non serve modificarlo
STO addr // viceversa, mem[addr] era negativo, ma in A adesso c'è il suo valore assoluto
STP
```

Nella scrittura di questo pseudocodice, non abbiamo tenuto in considerazione l'indirizzo nel quale carichiamo *addr*. Questo andrà risolto nel momento in cui andiamo a scrivere le istruzioni, che dovranno contenere il valore di *addr* al loro interno secondo la sintassi delle istruzioni di SSEM. Tale complicazione nei calcolatori moderni è risolta dal collegatore, che assegna a dati e istruzioni degli indirizzi in modo che non ci siano conflitti.

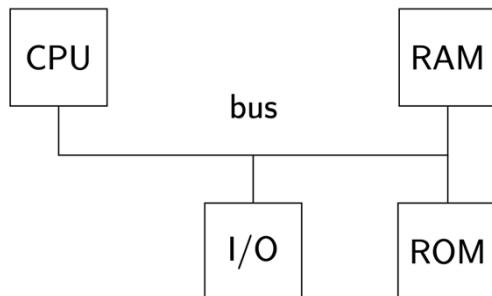
Osserviamo che, in fase di programmazione di questo calcolatore, noi abbiamo libero accesso alla memoria, potendo sfruttare ogni locazione a piacimento. Questo non è vero nei calcolatori moderi, dove uno strato di software si intramezza tra noi e la memoria. Allo stesso tempo, data una locazione non è scontato sapere se si tratta di un'istruzione o di un operando di un certo tipo. In un calcolatore come quello esposto, il controllo ce l'ha il programma/software, non la memoria o il processore. Se è vero che, ad un livello più basso, è il processore ad eseguire le istruzioni, allo stesso tempo esso non conosce l'esecuzione del programma, non sa cosa sta facendo, ma esegue passivamente quanto fornito dalla memoria (anch'essa passiva, in quanto continuamente modificata dalle istruzioni). Quando ci chiediamo chi faccia cosa, spesso la risposta più scontata è "il processore", ma conviene fermarsi a pensare due volte, in quanto, aumentando di livello, il software svolge un ruolo sempre più importante. Per quanto riguarda il processore, non è sempre lo stesso software a controllarlo. Se pensiamo ad un programma in C++ che sfrutta una libreria, alla chiamata di una sua funzione il controllo è ceduto alla libreria. In ogni istante, un solo software ha in mano il processore, ed è lui che cede il controllo ad un altro (si parla di *flusso di controllo*).

Nella macchina sopra analizzata, l'I/O non è interfacciato da alcun software: la memoria è visibile in ogni istante, così come può essere modificata tramite gli appositi pulsanti. Nelle macchine moderne, la memoria non si può visualizzare direttamente, ma è sempre necessario un software che mi permetta di visualizzare qualcosa o realizzare un input.

Sull'architettura del calcolatore

Nel 1981, l'IBM entra nel mercato dei calcolatori con il primo *personal computer*. Alla base vi è il processore 8088 dell'Intel, derivato dal 8086, dalla cui evoluzione nasce il processore che andremo a studiare noi. L'8086 è un processore a 16 bit con interruzioni ma senza protezione e memoria virtuale. Le sue versioni successive introducono queste due funzionalità, con il passaggio ai 32 bit. Successivamente, si è sentita la necessità di spostarsi ad un'architettura a 64 bit. L'Intel produce, senza successo, *Itanium*, surclassato da AMD64, il processore che studiamo e che è alla base di tutti quelli oggi in commercio. La sua fama è allo stesso tempo anche la sua debolezza, essendosi portato dietro tutta una serie di problemi strutturali, risolti lato software, la cui presenza serve per garantire la retrocompatibilità.

Nel SSEM, la CPU è direttamente collegata alla memoria, la quale è a sua volta collegata direttamente con lo spazio di I/O, che non ha quindi bisogno di interfacciamento: l'output si ottiene visualizzando la memoria sul tubo catodico, ed ogni locazione di 32 bit può essere modificata a mano tramite l'apposita pulsantiera. Per quanto questa macchina possa apparire primitiva, la CPU moderna ha gli stessi compiti di allora: prelevare un'istruzione da un indirizzo memorizzato in un registro, modificare il valore di tale registro ed eseguire l'istruzione. Delle sostanziali modifiche si hanno per quanto riguarda la memoria: nonostante la rappresentazione matriciale ancora regga, si può accedere alla memoria anche per multipli del byte (*word*, *double word* e *quad word*).



Lo spazio di I/O, nell'architettura moderna, deve essere intramezzato da del software, che permetta l'ingresso e l'uscita. Scrivendo con la tastiera, i caratteri vengono immagazzinati in un registro, ed è il processore che si deve occupare di leggerli per portarli, ad esempio, in uscita a video. Tale spazio, inoltre, si adegua alla memoria per quanto riguarda la presenza di indirizzi tramite i quali la CPU può stabilire univocamente dove accedere. Ciononostante, mentre la RAM è prettamente passiva, l'I/O spesso esegue delle azioni collaterali a seguito di operazioni di lettura o scrittura. L'indirizzo è proprio del bus. Nel momento in cui la CPU vuole fare un'operazione, scrive sul bus un indirizzo: tutte le entità ad esso collegate verificano se, tra gli indirizzi ad esse associati, vi sia quello richiesto e, in caso affermativo, proseguono con la relazione identificando le locazioni interessate. Per fare questo, la parte più significativa dell'indirizzo indica l'entità scelta, la parte meno significativa la specifica locazione. Lo spazio di memoria comprende tutto quello che è indirizzabile. Ad alcuni indirizzi sarà associata la memoria RAM, ad altre le periferiche dell'I/O o la ROM, altri ancora rimangono scollegati. In alcune architetture anche i registri della CPU sono indirizzabili tramite un indirizzo; nel nostro caso, essi hanno dei nomi. La ROM è un elemento fondamentale per il calcolatore, in quanto

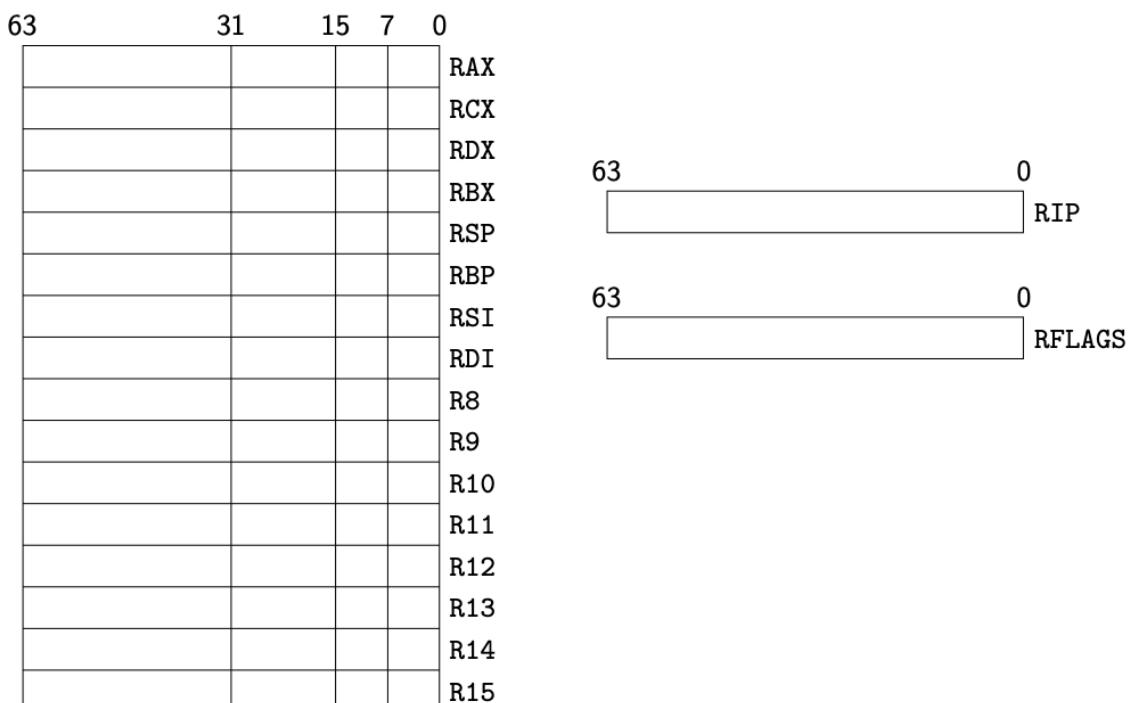
permette di inizializzare la memoria eseguendo un programma di *bootstrap*. Infatti, mentre in SSEM noi abbiamo un controllo diretto sulla memoria, nei calcolatori moderni dobbiamo prima eseguire un programma, mentre all'inizializzazione la memoria contiene dati casuali. La ROM è quindi fondamentale per tutti quei calcolatori non triviali che non vogliono fare sempre la stessa cosa: ci consente di caricare dei programmi.

Il software è tutto ciò che, contenuto in memoria, viene eseguito dal processore. Sebbene molte cose realizzabili tramite software siano fattibili anche lato hardware e viceversa, affinché il primo sia eseguito è necessario che il flusso di controllo vi ci passi. Se so che in un certo istante si realizza l'azione X , ma nello stesso istante il processore sta facendo altro, allora presumibilmente X è realizzata via hardware. La distinzione spesso è legata al singolo calcolatore, nel quale si fa una scelta piuttosto che un'altra: per questo, è bene ricordarsi cosa avviene nel nostro caso di studio, osservando che, se X si realizza via software, allora dovrei essere capace di scrivere il codice associato.

Il formato standard delle istruzioni in linguaggio mnemonico che andremo ad eseguire, nel formato AT&T, è

opcode source, destination

Gli operandi possono trovarsi direttamente nell'istruzione (immediati, per facilitare il parsing sono preceduti dal simbolo \$), possono essere registri (simbolo %) o operandi nello spazio di memoria (che si tratti di I/O o memoria, comunque viene generato un indirizzo). In quest'ultimo caso, solo le istruzioni `in` e `out` possono lavorare con le periferiche nello spazio di IO. I registri disponibili sono i seguenti:



Rispetto alle architetture a 32 bit, sono stati tutti estesi a 64 bit, compreso il registro dei flag, nonostante non ne siano stati introdotti di nuovi. È stata risolta l'irregolarità dei processori Intel per cui solo alcuni registri erano accessibili agli 8 bit meno significativi. Ciononostante, solo %rax, %rbx, %rcx e %rdx permettono di accedere agli 8 bit più significativi dei 16 meno significativi. Come sempre, alcune istruzioni necessitano di specifici registri – è il caso della moltiplicazione – mentre altre sono state introdotte per una maggiore flessibilità, come quelle che manipolano lo stack pointer %rsp.

64b	32b	16b	8b	8b
RAX	EAX	AX	AL	AH
RCX	ECX	CX	CL	CH
RDX	EDX	DX	DL	DH
RBX	EBX	BX	BL	BH
RSP	ESP	SP	SPL	
RBP	EBP	BI	BPL	
RSI	ESI	SI	SIL	
RDI	EDI	DI	DIL	
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	
R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	

L'accesso in memoria si esegue secondo la notazione dell'Intel con

$$\text{offset}(\text{base}, \text{index}, \text{scala}) \Leftrightarrow \text{offset} + \text{base} + \text{index} * \text{scala}$$

base e index sono due registri a 64 bit, mentre scala può assumere 1, 2, 4 o 8. Rispetto alla sintassi, si possono avere alcune variazioni, come: offset (indirizzamento diretto) e (base) (indirizzamento con registro puntatore). Quest'ultima notazione rende inutile l'idea di von Neumann per cui un programma potesse modificare sé stesso. Infatti, non è più necessario modificare l'offset nell'istruzione, basta il contenuto di un registro dal quale l'istruzione preleva un dato. Questo tipo di indirizzamento, associato ad una memoria ad accesso casuale (non c'è bisogno di scorrere tutte le locazioni precedenti), favorisce l'implementazione di specifiche strutture dati, quali gli *array*, le *struct* e i *puntatori*. Nel primo caso, le locazioni degli elementi sono contigue, non ci sono spazi vuoti e tutte assumono la stessa dimensione. Per quanto riguarda le *struct*, la conoscenza della definizione mi consente di determinare, noto il punto di partenza dell'oggetto, a quale cella si troverà un dato campo: lo scostamento infatti è sempre costante.

Un aspetto importante è che l'offset, così come un operando immediato all'interno delle istruzioni, può stare solo su 32 bit, esteso con segno su 64 bit (la scelta, di carattere tecnico, serve per non avere istruzioni di dimensioni inutilmente grandi). `mov offset, %rax` può quindi accedere solo ai primi e agli ultimi 2GiB di memoria, come conseguenza dell'estensione con segno, cosa che risulta una grande limitazione. Per venir meno a questa esigenza, ci sono alcune vie. La prima è l'uso dell'istruzione `movabs`, che accetta sia

operandi immediati che offset a 64 bit. La seconda è la notazione offset(%rip), che mi dà la possibilità di accedere ad un'area di memoria distante al più 2GiB dall'istruzione puntata. Questo è vantaggioso in quanto difficilmente i programmi risultano più grandi di 4GiB: posso quindi inserire in memoria i programmi, inserire i dati e fare riferimento a questi, in ogni parte di codice, tramite un indirizzo relativo (viene salvata la distanza tra l'indirizzo e %rip, e l'indirizzo è calcolato a tempo di esecuzione con queste informazioni). Qualcosa di simile si ha anche nei processori Intel, quando si esegue un'istruzione che altera il flusso del programma (salto o chiamata di funzione). Nonostante si abbia, un'architettura a 64bit, per gli indirizzi ne vengono utilizzati in numero minore (in una prima fase 48, ma alcuni modelli arrivano fino a 57). Usarli tutti comporterebbe una complicazione nelle strutture dati utilizzate, e le società si riservano, secondo necessità, di utilizzarne altri in futuro. Un indirizzo si definisce in *forma canonica* se i suoi 16 bit più significativi assumono lo stesso valore del quarantasettesimo. Gli indirizzi che non sono in questa forma non vengono accettati dal processore. Questa estensione fa sì che ci sia un buco negli indirizzi utilizzabili: si va infatti da 0 a $2^{48}-1$, poi da $2^{64}-2^{48}$ a $2^{64}-1$. Se non si rispettasse una convenzione del genere, troveremmo delle porzioni di memoria libera non contigue, cosa che porta (e ha portato, visto che solo AMD ha introdotto questo vincolo) un alto costo di gestione.

Per realizzare un programma, si scrive il codice con un qualsiasi editor di testo. Questo va in pasto all'assemblatore, che produce un file oggetto. Tale file va in input al collegatore, che realizza un file eseguibile. Sarà il caricatore che, secondo quanto stabilito dalle fasi precedenti, lo carica in memoria e lo esegue. Fino al momento dell'esecuzione, si ha solo un'elaborazione di byte sull'hard disk. L'assemblatore si occupa della traduzione da linguaggio mnemonico a linguaggio macchina. Le singole porzioni del programma vengono assemblate autonomamente, senza che si abbia una visione di gruppo. Ciò fa sì che, in questa fase, non si possano assegnare gli indirizzi delle variabili o dire dove porre in memoria il programma, poiché si potrebbe andare in conflitto con altri blocchi del programma. A fare ciò pensa il collegatore che, avendo in ingresso tutti i file oggetto che compongono l'eseguibile, ha una visione globale.

```

1  .data
2      num1:
3          .quad 0x1111111111111111
4      num2:
5          .quad 0x2222222222222222
6      risultato:
7          .quad -1
8
9  .global _start
10 .text
11  _start:
12      movabs num1, %rax
13      mov %rax, %rbx
14      movabs num2, %rax
15      add %rbx, %rax
16      movabs %rax, risultato
17
18      #fine programma
19      mov $0, %rbx
20      mov $1, %rax
21      int $0x80

```

L'assemblatore ci consente di definire alcune sezioni del codice sorgente. Nel campo .data vi sono le variabili che saranno caricate in memoria, mentre nel campo .text rientrano le istruzioni. La differenza sostanziale è che la prima è accessibile sia in lettura che in scrittura, la seconda no. L'uso dell'istruzione

`movabs` risulta necessaria dal momento in cui, non sapendo dove saranno collocate le variabili, potrei ottenere indirizzi che non stanno su 32 bit. Questa istruzione accetta come destinatario solo `%rax`. Alla riga 13, `num1` è un operando in memoria, non avendo alcun simbolo a precederlo. Un'etichetta serve a dare un nome all'indirizzo del primo byte che la segue. Con queste, ci riferiamo agli indirizzi simbolicamente, delegando ad altri il compito di decidere dove caricare le sezioni. L'indirizzamento è diretto, avendo solamente il *displacement*. L'assemblatore riconoscerà i nomi usati per designare le prime celle dei 64 bit delle variabili in memoria, e, all'interno delle istruzioni, lascerà uno spazio che sarà il collegatore a colmare. Le istruzioni che iniziano con il punto sono dette *direttive*, e si usano per comunicare con l'assemblatore. Per esempio, `.quad` riserva 64 bit in memoria a partire da quella locazione, mentre `.global` dice che l'etichetta `_start` sarà globale, e quindi visibile al collegatore. L'ultima porzione del codice è necessaria per far sì che la terminazione avvenga correttamente. Per ora la prendiamo per buono. Per assemblare, usiamo

```
as -o nomeFileOggetto.o nomeCodiceSorgente.s
```

Con `objdump` osserviamo il contenuto del file oggetto appena assemblato. In particolare, con l'opzione `-d` disassembliamo il file, che è l'operazione opposta dell'assemblamento, passando dal codice oggetto al codice mnemonico associato. Facendolo, vediamo che ogni istruzione è associata ad un codice binario che inizia sempre con l'opcode dell'operazione associata. Dopo sono riservati 64 bit per l'indirizzo che il collegatore vi inserirà. Infatti, nel file oggetto vi sono tutta una serie di informazioni che renderanno quest'operazione possibile.

```
studenti@debian-ce-2:~/Desktop/assembler$ objdump -d sum.o

sum.o:      file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <_start>:
 0: 48 a1 00 00 00 00 00  movabs 0x0,%rax
 7: 00 00 00
 a: 48 89 c3             mov    %rax,%rbx
 d: 48 a1 00 00 00 00 00  movabs 0x0,%rax
14: 00 00 00
17: 48 01 d8             add    %rbx,%rax
1a: 48 a3 00 00 00 00 00  movabs %rax,0x0
21: 00 00 00
24: 48 c7 c3 00 00 00 00  mov    $0x0,%rbx
2b: 48 c7 c0 01 00 00 00  mov    $0x1,%rax
32: cd 80                 int   $0x80
```

```
ld -o nomeFileEseguibile nomeFileOggetto.o
```

Questo comando ci permette di fare il collegamento, grazie al quale otteniamo un file eseguibile con `./nomeFileEseguibile`. Richiedendo, con `objdump`, il disassemblato, vediamo che il collegatore ha sostituito, al posto degli zeri, gli indirizzi scelti per le variabili. Tali indirizzi si possono vedere con

```
nm -n nomeFileEseguibile
```

Usando questo comando con il file oggetto, vediamo che una variabile è posta ad un ipotetico `0x0`, mentre tutte le altre sono indicate in termini di *offset* da tale indirizzo, sulla base della loro dimensione.

Non avendo alcun output, è necessario utilizzare il debugger *gdb* per osservare istruzione per istruzione l'andamento del programma.

```

gdb nomeFileEseguibile
source ~/gef/gef.py
set disassembly-flavor att
gef config context.layout "regs memory code"
start

```

Il secondo comando ci dà un'interfaccia comoda con cui vedere il contenuto di memoria e registri, il terzo imposta la sintassi AT&T di sorgente/destinatario, il quarto ci permette di visualizzare solo gli elementi che vogliamo (scegliendo da una lista che troviamo con `gef config context.layout`), l'ultimo ci consente di iniziare l'esecuzione, con un breakpoint prima dell'inizio del programma. Per poter vedere il contenuto della memoria che ci interessa (visto che, nel campo `data`, le variabili sollo allocate consecutivamente), usiamo `memory watch &num1 3 qword`. Con si eseguiamo un'istruzione e ci fermiamo: nel farlo, il debugger, assistito dal nucleo, cede il controllo del flusso al programma per riprenderselo subito dopo. In alto vediamo lo stato dei registri, evidenziati in rosso quelli che hanno cambiato valore a seguito dell'ultima operazione. Ci sono poi i flag e lo stato della memoria che avevamo evidenziato.

Per capire come si comporta l'assemblatore, proviamo a scrivere cose particolari. Se scrivo `movabs $num1, %rax`, il collegatore sostituirà a `num1` l'indirizzo associato, e dentro `%rax` ci finirà tale valore. Se scriviamo un'istruzione nel campo `.data`, essa viene tradotta in binario secondo la sintassi del processore, e, in fase di esecuzione, sarà considerato come un dato numerico assegnato ad un certo indirizzo. Se inseriamo `.quad var 0x0201020102010201` nel campo `.text`, essa viene tradotta come tale ma sarà considerata come un'istruzione, portando ad istruzioni casuali oppure a stringhe di byte che il processore non riconosce. Questo per dire che, sebbene l'assemblatore si occupi di sintassi, non fa altro che tradurre delle codifiche ascii in dei byte, e non si preoccupa del senso che questo può avere o meno. Si possono al più avere dei problemi durante l'esecuzione.

`.quad` richiede necessariamente un valore con cui inizializzare gli otto byte, ma, fornendo più valori, si possono inizializzare più variabili. Senza un'inizializzazione, la direttiva non riserva alcuno spazio. Ipotizziamo di scrivere

```

movabs avanti, %rax
[...]
avanti: mov $0, %rax

```

L'assemblatore interpreta `avanti` come un'etichetta qualsiasi, e va inserire dentro `%rax` il valore contenuto nella qword il cui primo indirizzo è quello di `avanti`. Nello specifico, vi andranno i primi 8 byte dell'istruzione indicata. E se scrivessi qualcosa che non va bene in termini di dimensioni, come `mov num1, %rax?` Sappiamo che questa operazione è lecita solo se `num1` ha un indirizzo che sta su 32 bit. Ma questo non è affatto scontato, dal momento in cui è il collegatore ad allocare, dove ritiene più opportuno, la variabile `num1`. Se ciò accade, è il collegatore a segnalare un errore. Infatti, esso avrà tra le mani un indirizzo che necessita di più di 4 byte (eventualmente con un po' di zeri in testa che possono essere ignorati), ma nell'istruzione ci sarà spazio per solamente 4 byte.

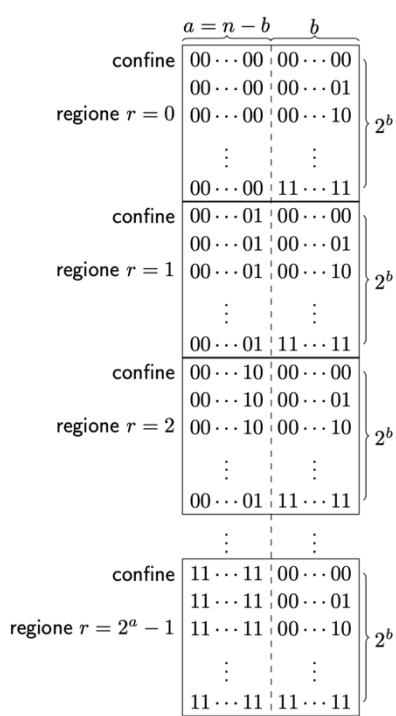
Il fatto che bisogna tener presente della lunghezza dell'operando è una limitazione, visto che il compilatore, che passa dal linguaggio ad alto livello ad istruzioni Assembler, non avrà a disposizione buona parte delle istruzioni. Per risolvere questo, si può imporre un vincolo alla dimensione del programma (noto a priori che quindi devo rispettare). Il collegatore farà in modo che l'intero mio programma si ritrovi in una porzione di memoria per cui è lecito adoperare indirizzi di 32 bit. Un'alternativa è quella di usare lo scostamento da `%rip`. Scrivendo `mov num1(%rip), %rax`, l'assemblatore lascia al collegatore il compito di calcolare quale sia la differenza tra l'indirizzo `num1` e quello contenuto in `%rip` durante l'esecuzione dell'istruzione (quindi, l'indirizzo dell'istruzione successiva). `num1` è rintracciabile in termini di differenza dal valore di `%rip`. Anche in questo caso, il valore di offset è memorizzato con segno dentro l'istruzione su 32 bit, quindi deve distare al più 2GiB da `%rip`.

Nonostante io dichiari delle variabili, come avveniva in SSEM posso indirizzare una qualsiasi delle 2^{64} locazioni. Il collegatore è solo una facilitazione che abbiamo a disposizione per non doverci occupare degli indirizzi. Piuttosto, è possibile (e anzi, probabile) che scrivendo indirizzi numerici si voglia accedere ad una regione di memoria che non era assegnata al nostro programma, e su cui quindi non abbiamo privilegi. Tale situazione è gestita dal nucleo, che può provocare un *segmentation fault*.

Sugli indirizzi e lo spazio di memoria

Gli indirizzi si perdono nella programmazione ad alto livello ma, nel basso livello, sono fondamentali. Utilizzandoli, è bene far riferimento alla notazione esadecimale, grazie alla quale è possibile rappresentare un byte con due cifre: un numero in questa rappresentazione si traduce in binario convertendo cifra per cifra la rappresentazione. A volte, può essere utile usare la base otto, che permette di rappresentare i numeri a gruppi di tre e richiede solamente otto codifiche binarie da utilizzare. Un numero in base 8 si scrive con uno zero in testa. 4K, che corrisponde a 2^{12} , si rappresenta in esadecimale come `0x1000`. Il numero $2^n - 1$ è un numero di n bit, che valgono tutti 1.

Lo spazio di memoria ha indirizzi da 0 a $2^{64} - 1$. La memoria si deve intendere come circolare: l'indirizzo successivo all'ultimo è nuovamente 0. Nei linguaggi ad alto livello come C, si rappresentano come degli unsigned long: per standard, quando si ha un overflow con questo tipo si esegue l'operazione modulo 2^{64} . Sia x un indirizzo, y un altro. Per offset tra i due indirizzi si intende *il numero di byte che devo saltare per passare da x a y* . Banalmente, l'offset vale $y - x$. Questa definizione continua ad avere senso per offset negativi, quando cioè y ha un indirizzo minore di x . Non si hanno neanche problemi per quanto riguarda il *wrap around*.



È fondamentale utilizzare intervalli di indirizzi. Si possono definire dati i due estremi e, tra le quattro notazioni che coinvolgono o meno gli estremi nell'intervallo, si usa $[x, y]$. Il vantaggio di questa scelta è che, dati i due intervalli $[x, y)$ e $[y, z)$, la loro unione è $[x, z)$. Inoltre, la grandezza dell'intervallo si ottiene calcolando $y - x$, supponendo $y > x$. Infatti, se non si avevano problemi per gli offset, *wrap around* è sconveniente negli intervalli, e conviene usare sempre $y > x$. $[x, x)$ è un insieme vuoto, e l'ultimo elemento di un intervallo è sempre $y - 1$. Il primo elemento dell'intervallo è la base, nonché il suo indirizzo.

Per molti casi d'uso, suddividiamo lo spazio di memoria in parti uguali di grandezza pari ad una potenza di 2, 2^b . I punti in cui si passa da una parte, che è detta *regione naturale*, ad un'altra sono i confini. Un confine si riconosce essendo un multiplo della dimensione degli intervalli, e quindi avendo b zeri in coda. Gli indirizzi che appartengono ad una regione hanno gli $n - b$ bit più significativi tutti uguali. All'interno di una regione, i b bit meno significativi assumono tutti i valori compresi tra 0 e $2^b - 1$, e vengono detti *offset*. Ma

allora, fissato b , una regione è identificata grazie ai suoi bit più significativi. Lato hardware, dato un indirizzo è immediato ricavare regione e offset. Lato software, supponiamo di avere un indirizzo dentro un unsigned long identificato con x . Il numero di regione è dato da $x \gg b$, ossia da uno shift a destra di b posizioni. L'offset si ricava con una maschera con i primi b bit ad 1, gli altri a 0. Questa maschera si ottiene con $(1 \ll b) - 1$. Dato l'intervallo $[x, y]$, l'ultima regione naturale toccata dall'intervallo, supposto $y > x$, è la regione di $y - 1$.

Un oggetto è una sequenza di un certo numero di byte. Il suo indirizzo è quello del primo byte da esso occupato. Allocando un oggetto, ci si pone il problema del modo in cui esso è allineato, cioè se il suo indirizzo è il confine di una qualche regione naturale, e quindi è un multiplo di 2^b . L'oggetto x è allineato a 2^b se il suo indirizzo è multiplo di tale valore; si dice che “ x è *allineato con y* ”, con y di dimensione multipla di 2^b , se x è allineato con 2^b . “ x è *allineato naturalmente*” se ha come dimensione una potenza di 2, ed è allineato con la stessa potenza. Un oggetto allineato naturalmente, occupa interamente una regione naturale della

sua stessa dimensione. Osserviamo che l'allineamento è una proprietà dell'indirizzo assegnato, e non dell'oggetto stesso.

Una differenza sostanziale rispetto alla memoria per come era implementata in SSEM è che, nei calcolatori moderni, è possibile accedere sia ad un byte sia a suoi multipli. Ragionevolmente, questo deve avvenire in modo efficiente, senza che il tempo si quadruplichi per leggere un *long* piuttosto che un *byte*. Si pongono allora due questioni.

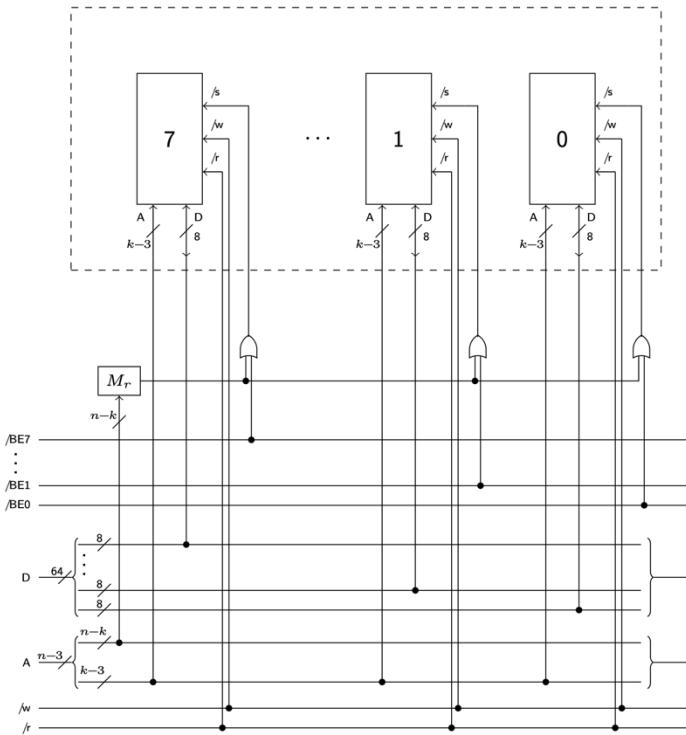
Il primo problema è quello dell'*Endianess*: dovendo accedere ad una parola di più byte, in che ordine sono disposti? Ha indirizzo più piccolo il byte più significativo o il meno significativo? Nel caso in cui inserisca per primo il byte meno significativo, si parla di *little endian*; nel caso opposto, di *big endian*. Le due modalità non hanno sostanziali differenze, ma, durante lo sviluppo delle architetture, ognuno ha preso una scelta differente. Questa situazione ha comportato dei problemi nel momento in cui, ad esempio, due calcolatori si dovevano scambiare dati tramite la rete; un'altra situazione di difficoltà si ha avuta quando sono nati i primi sistemi operativi per più architetture (primo fra tutti Unix). Nel processore Intel di cui ci occupiamo, lavoriamo in *little endian* (ma, ad esempio, i dati in rete sono scambiati in *big endian*). Per questo motivo, quando troviamo un offset dentro un'istruzione, esso viene visualizzato a partire dal byte meno significativo: l'indirizzo 0x3280a1e4 viene memorizzato come e4 a1 80 32, associando indirizzo più piccolo a parte meno significativa.

L'altro problema è quello del parallelismo. Sarebbe conveniente che il tempo di accesso non sia proporzionale al numero di byte coinvolti. Quindi, le istruzioni `mov %al, 0x1000` e `mov %rax, 0x1000` non dovrebbero ricorrere a metodi differenti per l'accesso. Ma, affinché ciò accada, un accesso in memoria non richiede solamente l'indirizzo, ma anche il numero di byte coinvolti, nella forma *[indirizzo, numero di byte coinvolti]*. Si prende lo spazio di memoria, e si suddivide in regioni naturali di dimensione 8 byte (ottenendo 2^{61} regioni). Il processore prende l'indirizzo e ne ricava il numero di regione, dato dai 61 bit più significativi. Successivamente, utilizza l'offset e i byte richiesti dall'opcode tramite 8 linee separate attive basse, dette *byte enable* (`/b0, /b1, ..., /b7`): queste linee specificano quali byte andare a prendere dentro la regione naturale avente quel numero. Questa operazione viene quindi svolta dal processore: sul bus sono impostati solamente il numero della regione naturale e gli 8 *byte enable*. Consideriamo `mov %al, 0x3f7`. L'indirizzo, in binario è `0011|1111|0111`: la regione naturale alla quale appartiene è quella con numero `0011|1111|0`. Si vuole accedere al byte numero 7 della regione, quindi tutti i fili di *enable* saranno ad 1 eccetto che `/b7`. Prendiamo invece `mov %ax, 0x3f4`. La regione è quella con il numero `0011|1111|0`. La parte meno significativa del dato è all'indirizzo più piccolo, quindi con offset 4, l'altro a quello successivo, con offset 5. Si dovranno quindi impostare a 0 i fili `/b4` e `/b5`. Ma cosa succede se usciamo dalla regione naturale? Consideriamo `mov %ax, 0x3f7`. Il primo dei due byte interessati sta nella regione naturale con numero `0011|1111|0`, ma per il secondo byte si deve andare nella regione successiva. Ciò significa che, secondo lo standard utilizzato dal bus, non è possibile fare l'accesso con una sola operazione di scrittura. Tale doppia lettura è svolta automaticamente dai processori Intel, ma si nota l'importanza dell'allineamento: se il dato a 2, 4 o 8 byte è allineato naturalmente, allora è sufficiente un solo accesso in memoria. Conviene immaginare la memoria come organizzata in righe di 8 byte ciascuna.

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
0									0
1									8
2									16
3									24
					...				
$2^{61} - 1$									$2^{64} - 8$

Una rappresentazione con gli indirizzi più piccoli a destra, che crescono andando verso sinistra, da una parte è conforme alla modalità *little endian* usata nell'architettura, dall'altro ci consente di usare la notazione *big endian* del nostro modo di scrivere i numeri (a sinistra, quindi in posizione minore, mettiamo la parte più significativa, non quella meno significativa). Il fatto di accedere una sola volta in memoria ha a che fare con l'efficienza: le memorie RAM sono piuttosto lente in confronto alla CPU, e un ciclo di lettura richiede circa 100 clock, che il processore potrebbe usare in altro modo. Ovviamente, l'allineamento di un dato è condizione sufficiente, ma non necessaria, affinché quel dato possa essere prelevato con un solo accesso in memoria. Pensiamo a `mov 0xf43, %eax`. Il dato non è allineato, essendo l'indirizzo dispari, ma i 4 byte sono tutti nella stessa regione naturale, quindi vi si accede settando a 0 /b3, /b4, /b5 e /b6.

Quanto detto deve avere un riscontro lato hardware: come è possibile effettuare queste operazioni? Consideriamo un chip di RAM con 2^{k-3} di capacità. Esso avrà un ingresso di *enable* /s, 8 fili di dati e $k-3$ fili di indirizzo. Di per sé, non permette gli accessi paralleli finora esposti. L'idea è di utilizzare 8 moduli RAM di questo tipo, ciascuno dei quali rappresenta una colonna nella visione tabellare della memoria di cui sopra (la memoria complessiva è quindi 2^k). Per questo motivo, dei 64 **fili di dati** che ho del bus, invio gli 8 più significativi alla RAM della colonna 7, i successivi alla RAM di colonna 6 e così via. Rispetto ai fili di indirizzo, **che contengono il numero della regione naturale interessata**, i $k-3$ meno significativi diventano l'indirizzo di accesso per i singoli moduli RAM. All'*enable* dei moduli dovrà contribuire sia il filo /bx, sia i restanti fili di indirizzo del bus. Se coinvolgesse solo il *byte enable*, tutte le volte che imposto un indirizzo, qualunque esso sia, si attiverebbe la locazione di 64 bit corrispondente all'indirizzo indirizzo modulo 2^k . Ma questo sarebbe l'equivalente di ripetere la memoria RAM in tutto lo spazio di memoria, sulle regioni naturali di ampiezza 2^k . Allora, date tutte le regioni naturali di tale grandezza, ne scelgo una in cui inserire il mio modulo RAM. Realizzo una maschera con delle porte *and* in modo che l'uscita valga 0 quando sto selezionando quella porzione di memoria: per farlo uso proprio gli $n-k$ fili restati da quelli di indirizzo (M_r in figura). Questa uscita va in *or* con i singoli *byte enable*, dovendo gestire della logica attiva bassa. I moduli RAM riceveranno l'*enable solo* quando si seleziona la specifica memoria e si occupano di uno dei byte specificati dai fili /bx. Lo stesso ragionamento si può fare prendendo memorie singole di dimensione 2^k , dicendo che sul bus ci sono n fili e che quindi gli indirizzi sono di dimensione $n-3$.



Facciamo adesso alcune riflessioni. Poniamo di voler eseguire `mov 0x3f4, %rax`. Il dato non è allineato, quindi sono necessarie due letture: prima si accede alla regione naturale `0011|1111|0`, e si prendono i 4 byte con indirizzo maggiore, poi a `0011|1111|1`, con i 4 byte di indirizzo minore. A causa del *little endian*, i primi 4 byte prelevati sono i meno significativi del dato d'interesse, i successivi i più significativi. È compito del processore inserirli correttamente dentro `%rax`.

Supponiamo di voler modificare un singolo bit di un byte in memoria. Ad esempio, si porti ad 1 il bit numero 3 di `0x3f4`. Ci verrebbe spontaneo scrivere `orb $0x80, 0x3f4`. Per fare questa operazione, il processore preleva il dato dalla memoria con una lettura, effettua l'operazione di or e lo riscrive in memoria. In tal caso, possiamo dire che l'operazione è svolta dall'hardware. Ma l'operazione è equivalente a scrivere

```
mov 0x3f4, %al
or $0x80, %al
mov %al, 0x3f4
```

In questo caso, si limita ad eseguire la singola istruzione ma, non avendo memoria, non è cosciente di star eseguendo un or di un dato in memoria. Questo significa che l'operazione di or è svolta dal software, essendo l'unico che ci dà la visione d'insieme dell'operazione.

Sappiamo che non è possibile avere un'istruzione che abbia due operandi in memoria. Questa non è una limitazione dell'hardware, ma del formato delle istruzioni: è consentito avere uno solo dei due operandi nel formato *offset(base, indice, scala)*. Tuttavia, quando l'operando in memoria è implicito, è effettivamente possibile effettuare due operazioni che coinvolgano operandi in memoria. È il caso di `push` e `pop`: avendo un dato implicito puntato da `%rsp`, l'altro può benissimo essere in memoria.

Sulla programmazione Assembler e C++

Un primo approccio

A differenza di un assemblatore, che si limita a sostituire byte e definire delle aree con le quali allocare questi valori in memoria, il compilatore effettua numerose modifiche. Tuttavia, strumenti come *g++* e *gcc* non sono altro che front end per più programmi, ciascuno dei quali si occupa di un aspetto differente: in un primo momento si passa ad un file Assembler, che viene assemblato dall'assemblatore e successivamente collegato e caricato. I file oggetto non hanno alcun riferimento al linguaggio di alto livello dal quale provengono: per tale motivo, un oggetto realizzato a partire un codice in C++ può essere collegato con uno da Assembler. Quello che cerchiamo di fare è capire le modalità con cui lavora il compilatore, simulandone il funzionamento. Per prima cosa, vediamo un esempio di programmazione Assembler su più file sorgente.

```

1 .global esamina, alpha, beta
2
3 .data
4     alpha: .byte 0
5     .align 8
6     beta: .quad 0
7
8 .text
9 esamina:
10    push %rax
11    push %rdx
12    push %rcx
13
14    mov alpha(%rip), %cl
15    movabs beta, %rax
16    mov $0, %rdx
17
18 prossimo:
19    test $0b10000000, %cl
20    jz zero
21
22    movb $'1', (%rax, %rdx)
23    jmp avanti
24 zero:
25    movb $'0', (%rax, %rdx)
26    jmp avanti
27
28 avanti:
29    shl %cl
30    inc %rdx
31    cmp $8, %rdx
32    jb prossimo
33
34    pop %rcx
35    pop %rdx
36    pop %rax
37    ret

```

Vogliamo un file principale, codifica, che si occupa di leggere da tastiera dei caratteri, ciascuno dei quali, una volta stampato, viene passato a esamina, che restituisce in un array di 8 caratteri, le codifiche ascii dei bit del carattere, che il primo si occuperà di stampare. Il programma termina quando si va a capo. Affinché le due parti possano comunicare, non conoscendo ancora il passaggio per parametri, inseriamo nella variabile globale alpha il carattere attuale, in beta l'indirizzo della prima locazione dell'array nella quale vogliamo che esamina scriva. Useremo il file ser.s per effettuare l'input e l'output. All'interno di questo file, troviamo una funzione tastiera, che preleva un carattere dallo standard input e lo inserisce in %al, video, che stampa il carattere associato al valore di %al e uscita, che termina l'esecuzione del programma. Cominciamo con la funzione esamina.

Le variabili alpha e beta ho scelto di dichiararle all'interno di questo file sorgente. Affinché il collegatore le possa vedere ed usare anche nell'altro file, è necessario che le dichiari come globali, con la direttiva .global (assieme all'etichetta del sottoprogramma che dovrà essere invocato). beta contiene l'indirizzo del primo elemento dell'array, ed è quindi un puntatore. All'inizio del programma salvo in pila i registri che andrò ad usare. Questa è una convenzione molto importante, perché il programma chiamante si aspetta che i registri che sta utilizzando non siano sporcati alla chiamata di esamina. Alla riga 13, l'uso dell'indirizzamento tramite offset da %rip mi permette di non preoccuparmi degli indirizzi a 64 bit, dando per scontato che il programma stia su 4GiB complessivi. %rdx mi farà da contatore: eseguirò il confronto con il bit più significativo di %cl 8 volte, e a quel punto potrò uscire. L'istruzione test effettua la stessa operazione di and senza però sporcare il registro destinatario, così che non si debba ricaricare ogni volta. Osserviamo che, una volta stabilito se il bit numero %rdx è 0 o 1, voglio andare a mettere la codifica ascii associata della posizione rdx-esima dell'array il cui primo indirizzo sta in beta. Per questo motivo, una scrittura quale

molto importante, perché il programma chiamante si aspetta che i registri che sta utilizzando non siano sporcati alla chiamata di esamina. Alla riga 13, l'uso dell'indirizzamento tramite offset da %rip mi permette di non preoccuparmi degli indirizzi a 64 bit, dando per scontato che il programma stia su 4GiB complessivi. %rdx mi farà da contatore: eseguirò il confronto con il bit più significativo di %cl 8 volte, e a quel punto potrò uscire. L'istruzione test effettua la stessa operazione di and senza però sporcare il registro destinatario, così che non si debba ricaricare ogni volta. Osserviamo che, una volta stabilito se il bit numero %rdx è 0 o 1, voglio andare a mettere la codifica ascii associata della posizione rdx-esima dell'array il cui primo indirizzo sta in beta. Per questo motivo, una scrittura quale

```
mov $'1', beta(%rdx)
```

non solo è sbagliata sintatticamente, non farebbe neanche quanto voluto: questa accede alla rdx-esima posizione successiva a beta, ma a noi non interessa ma le celle da esso puntate! Il valore contenuto in beta viene prima messo in poi si usa la notazione con l'indice per scorrere gli elementi. In particolare, quanto è del tutto equivalente (e quindi sbagliato) a scrivere, alla riga 14, mov \$beta, %rax oppure lea beta, %rax. In entrambi i non osserviamo che ci interessa l'indirizzo contenuto **in** beta, non quello **di** beta. Un'ultima osservazione è che, per come abbiamo scritto la porzione .data, beta risulta allineato. Per far sì che non si abbiano problemi, si possono dichiarare abbastanza che facciano da padding, oppure usare la direttiva .align 8. Inserita prima della dichiarazione di un dato, fa sì che questo sia inserito al primo indirizzo successivo allineato con 2³, ottenendo l'allineamento naturale.

In questo programma principale, per cosa includiamo il file ser.s. L'assemblatore, prima di processare il file, effettuerà un copia e incolla di quello nel nostro, in modo da avere a disposizione tutte le funzionalità in esso dichiarate. Il programma non è nulla di particolare, se non per l'uso di etichette che non sono state dichiarate nel nostro stesso file (riga 14, tra le tante). Quando l'assemblatore si rende conto che l'etichetta non è stata dichiarata, non se ne fa un problema, lasciando il compito di risolvere il riferimento al collegatore: si parla in questo caso di *etichetta esterna*. Si poteva anche esplicitare con la direttiva

```
.extern alpha, beta, esamina
```

Tra le righe 16 e 17, voglio inserire in beta l'indirizzo del buffer che ho dichiarato, in modo che esamina acceda ad una porzione di memoria corretta. Per farlo, devo prima inserire l'indirizzo dentro un registro e poi spostarlo, per non ricadere in una sintassi scorretta. Un aspetto interessante è che la notazione per l'indirizzamento alle righe 16 e 23, seppur molto simili, in realtà non hanno nulla che fare. In 16, il collegatore calcola l'offset tra beta e %rip, e in fase di esecuzione si accede effettivamente a beta tramite %rip + offset; nel secondo caso, vogliamo accedere a buffer + %rcx, e tale indirizzo è memorizzato nell'istruzione.

```

1  .include "ser.s"          ma
2  .global _start
3
4  .data
5  buffer:
6      .fill 8,1
7
8  .text
9  _start:
10     call tastiera
11     cmp '$\n', %al
12     je fine
13     call video
14     mov %al, alpha
15     lea buffer, %rax
16     mov %rax, beta(%rip)    beta,
17     call esamina
18     mov '$ ', %al
19     call video
20     mov $0, %rcx            %rax,
21
22 ancora:                   detto
23     mov buffer(%rcx), %al
24     call video
25     inc %rcx
26     cmp $8, %rcx
27     jb ancora
28
29     mov '$\n', %al
30     call video
31     jmp _start
32
33 fine:                     casi,
34     call uscita

```

non

byte

%rax,

detto

casi,

non

byte

prima

Per collegare i due file oggetto, usiamo `ld` con due input, `esamina.o` e `codifica.o`. Provando a chiamare `objdump -d esamina.o`, ci rendiamo conto che l'assemblatore non si è occupato di assegnare un offset ad `alpha`, `beta` ed `esamina`. Infatti, non ha neanche idea di cosa essi rappresentino, e di quanto spazio quindi richiedano come dati.

Vediamo ora alcuni casi limite. Se in `esamina` mi fossi dimenticato di fare le `pop` dalla pila, avrei ottenuto un `segmentation fault`. Infatti, la `ret` si occupa di prendere dalla pila un indirizzo di ritorno, ma se questo non fosse quello corretto (non avendo svuotato la pila), finiremmo in una porzione di memoria a noi non garantita. Questo errore allora va a nostro vantaggio: se il programma potesse proseguire l'esecuzione in modo tanto errato, danneggerebbe gravemente la memoria, facendo accessi imprevedibili. Un'alternativa sarebbe stata se avessimo estratto i registri dalla pila in un ordine non corretto. In tal caso, nessuno avrebbe avuto nulla da ridire, e anzi, l'esecuzione sarebbe andata a termine correttamente, perché, per puro caso, `codifica` non fa uso dei dati nei registri dopo la chiamata di funzione. Questo bug è comunque un problema nel caso in cui ci fossero nuove versioni di `codifica`, perché quasi sicuramente verrebbero alla luce le conseguenze.

Un esempio di programmazione mista

Finora abbiamo programmato su due file Assembler, ma `g++` ci consente anche di fare una programmazione mista con C++: questo, infatti, non è altro che un'interfaccia per una serie di programmi, ed effettua operazioni differenti sulla base dei tipi di file che riceve come input. Tra le altre cose, esso accetta anche solamente file Assembler. `g++ -o codifica -no-pie codifica.s esamina.s`: nonostante il comando sia corretto, riceviamo un errore che ci dice che l'etichetta `_start` è stata definita più volte. Questo accade in quanto di default `g++` include tutte le librerie standard del linguaggio e alcuni file di inizializzazione. In questi file vi è una funzione `_start`, che si occupa, ad esempio, di inizializzare una serie di oggetti che abitualmente usiamo in C++, come `cin` e `cout`. Dopo aver fatto queste operazioni, invoca la funzione `main`, e con questo nome deve essere chiamata l'etichetta principale del nostro programma. Una volta che si incontra `ret`, `_start` si preoccupa anche di distruggere gli oggetti precedentemente dichiarati, e di riportare il controllo al nucleo con quelle 3 istruzioni viste nel primo esempio. Chiaramente, tali *start files* non

```

1 #include<iostream>
2
3 extern char alpha;
4 extern char* beta;
5 extern void esamina();
6 char buffer[8];
7
8 int main(){
9     while(true){
10         char c;
11         std::cin.get(c);
12         if(c=='\n') break;
13         alpha = c;
14         beta = &buffer[0];
15         esamina();
16         for(int i=0; i<8; i++)
17             std::cout << buffer[i];
18         std::cout << ' ';
19     }
20 }
```

sono indispensabili, soprattutto nel caso in cui non si vogliano usare le funzionalità delle librerie di C++: per usare `g++` nel modo in cui ci aspetteremmo (realizzare un file eseguibile a partire dal sorgente) si usa il comando `nostartfiles`. Una delle funzionalità più importanti di `g++` è quella di fare da compilatore, ossia tradurre il linguaggio ad alto livello in istruzioni Assembler che possano poi essere assemblate e collegate. Questo significa che possiamo usare codice sorgente in Assembler con codice in C++. Riscriviamo il codice di `codifica.s`.

Nel codice devo dichiarare che alcuni nomi si possono usare in questo file in quanto dichiarati in altri. Per farlo, uso la parola chiave `extern`. Questo accade sia per `alpha` che per `beta`, ma anche per la funzione `esamina` che devo andare a

chiamare: il compilatore deve infatti essere certo del tipo e del numero degli argomenti, così come del valore di ritorno. La correttezza dei tipi deve avvenire in ogni caso, perché con questi lavora C++: è vero che `beta` conterrà un indirizzo, ma di fatto stiamo lavorando con un puntatore a `char`, e come tale deve essere dichiarato. La chiamata di una funzione corrisponde ad un salto ad un'etichetta definita altrove.

Con `g++` è possibile creare l'eseguibile a partire da `codifica.cpp` ed `esamina.s`. Tuttavia, ci troveremmo con l'errore per cui l'etichetta `_Z7esaminav` non è stata dichiarata. Cosa è successo? Il compilatore, trovandosi davanti ad una funzione dal nome `esamina` con valore di ritorno `void`, le assegna un'etichetta che comprende tutte queste caratteristiche (7 informa sul numero di caratteri successivi che realizzano l'identificatore). Il motivo di questa scelta è che C++, a differenza di C, ammette l'overloading di funzioni con nomi uguali che accettano parametri in numero e tipo diverso: per capire quale specifica funzione deve essere chiamata, e quindi per distinguere i casi d'esame, usa un'etichetta di questo tipo. Le soluzioni sono due: 1. Modifichiamo di conseguenza il'etichetta della funzione nel file `esamina.s` 2. Scriviamo `extern "C" void esamina();` Una dichiarazione di questo tipo informa il compilatore che il formato dell'etichetta di quella funzione è quello adottato dal linguaggio C, in cui l'overloading non c'era e le funzioni avevano come etichetta lo stesso nome della loro dichiarazione. A questo punto è possibile creare

```

1 char alpha;
2 char* beta;
3
4 extern "C" void esamina(){
5     char c = alpha;
6     for(int i=0; i<8; i++){
7         if(c & 0x80) beta[i] = '1';
8         else beta[i] = '0';
9         c << 1;
10    }
11 }
```

l'eseguibile. Al pari di questo, possiamo anche scrivere in C++ il file esamina.s, in modo che si interfacci con codifica.s.

La funzione in sé è identica al caso del file Assembler. L'uso di `extern` nella dichiarazione ci consente di gestire l'etichetta alla stregua di C, in modo che l'istruzione `call esamina` nell'altro file faccia quanto ci si aspetti.

Usando il comando `-S` con `g++`, ci possiamo fermare al corrispettivo Assembler realizzato dal compilatore. Si osservi che alcuni registri sono utilizzati senza che il contenuto precedente sia salvato in pila e successivamente recuperato. Tali registri sono detti *di scratch* e sono `%rax, %rcx, %rdx, %rsi, %rdi, %r8-11`. È un compromesso tra chiamante e chiamato: tutti vorrebbero far uso di registri, perché con questi le elaborazioni sono più veloci, ma il salvataggio e recupero in pila è dispendioso. Per questo, questi registri *di scratch* possono essere usati liberamente dalla funzione chiamata, mentre gli altri, se usati, devono essere lasciati allo stato precedente. Tale fatto è osservato dal compilatore, ma anche noi, nel realizzare i nostri programmi, ne dobbiamo tener di conto.

Per capire più a fondo l'operato del compilatore, è conveniente avere sott'occhio il formato dei tipi. Lo standard stabilisce una dimensione minima per ciascun tipo, ma è l'implementazione che dobbiamo tenere in considerazione. Nel nostro caso, si farà riferimento alle informazioni presenti [qui](#). Per ciascun tipo, ne conosciamo la dimensione, o `sizeof`, e l'allineamento, `alignof`, che deve essere rispettato al momento dell'inserimento in memoria. Oltre ai tipi fondamentali, conviene considerare i tre tipi derivati fondamentali: puntatori, array e struct.

- *Puntatori*. Dovendo contenere un indirizzo in un'architettura a 64 bit, avranno come dimensione 8 byte, e come tali saranno allineati.
- *Array*. Consideriamo un array di tipo *T* e dimensione *dim*. La sua dimensione è, banalmente, $\text{dim} * \text{sizeof}(T)$. Per l'allineamento, si ha `alignof(T)`. In memoria, gli elementi sono posti uno dopo l'altro, a partire da `array[0]` con indirizzi crescenti.
- *Struct*. Il problema delle struct deriva dalla loro non omogeneità, essendo realizzate da tipi tra loro diversi.

```
struct s{
    tipo1 f1;
    tipo2 f2;
    tipo3 f3;
}
```

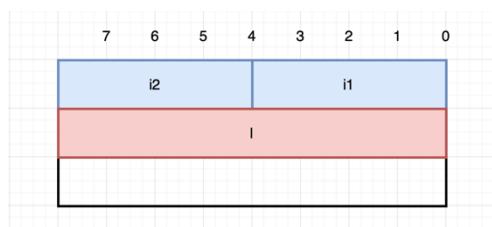
Per l'allineamento, si usa il valore massimo tra i tipi che la compongono: `alignof(s) = max{alignof(tipo_i)}`

Per quanto riguarda la dimensione, bisogna vedere il layout con il quale la struct è memorizzata in memoria. Si usano le seguenti regole: 1. *Ogni campo deve rispettare il proprio allineamento* 2. *L'ordine degli elementi è lo stesso della dichiarazione*. 3. *Alla fine, la dimensione complessiva deve*

essere un multiplo dell'allineamento della struttura; 4. Si deve usare la minima dimensione possibile. Facciamo alcuni esempi.

Per la prima struttura, l'allineamento deve essere quello di un long, deve necessariamente partire dall'estrema sinistra. Inserendo gli elementi nell'ordine e con il proprio allineamento (gli interi, avendo dimensione 4, sono allineati così), non si hanno buchi. Alla fine, la dimensione della struttura è 16, multiplo di 8, per cui non si hanno problemi.

Per la seconda struttura, l'allineamento deve essere di 8, per la presenza di un long. Si inserisce quindi il char, ma dopo non si può mettere il long, perché non sarebbe naturalmente allineato: si deve andare al blocco successivo. Inserito anche c2, si osserva che la dimensione risultante sarebbe di 17, che non è multiplo di 8. Per questo, alla dimensione della struttura si aggiungono anche i byte successivi, arrivando a 24. Questa cosa causa uno spreco, ma il compilatore non interviene in alcun

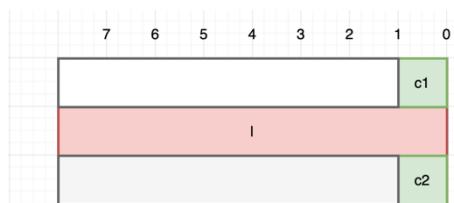


modo (anzi, si attiene solo alle regole di layout), e deve al più essere l'utente a preoccuparsene, inserendo gli elementi in ordine di dimensione decrescente per risparmiare spazio.

```
struct s1{
    int i1;
    int i2;
    long l;
};

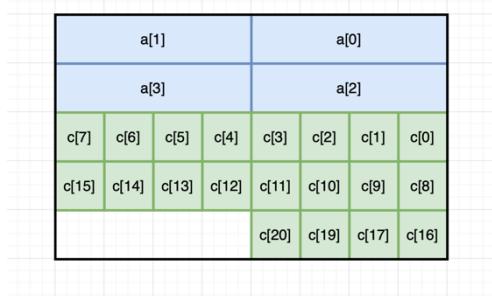
struct s2{
    char c1;
    long l;
    char c2;
};

struct s3{
    int a[4];
    char c[20];
}
```



L'ultimo caso ci mostra come le regole per gli elementi sono ricorsive. L'allineamento della struct è il massimo degli allineamenti dei vari elementi. Quello degli array è lo stesso del proprio tipo, quindi l'allineamento complessivo è quello degli interi, 4. Inserendo gli elementi seguendo il proprio allineamento naturale, si arriva ad occupare 36 byte. È corretto? Sì, in quanto multiplo del proprio allineamento. Una scelta del genere ci permetterebbe di usare i 4 byte dell'ultima riga per inserire una parte di un altro oggetto dello stesso tipo, ed è per questo che viene

adottato questo layout.



Il passaggio di parametri

Per poter fare programmazione mista, è necessario capire come effettuare il passaggio di parametri alla funzione chiamata e come collocare in memoria le variabili locali. La soluzione è standardizzata affinché sia possibile comunicare con librerie, e quindi funzioni, precompilate. L'uso dello stack per il salvataggio dell'indirizzo di ritorno e per le variabili locali non è stata la scelta più immediata. In un primo momento, la funzione veniva direttamente copiata nel codice ogni volta che era necessario chiamarla, un po' come avviene nei linguaggi ad alto livello con le macro. Per i parametri, venivano utilizzate delle aree di memoria statiche, nelle quali la chiamante lasciava i dati e la chiamata li riprendeva. Approcci di questo tipo, tuttavia, rendono impossibile l'annidamento di funzioni e la ricorsione (non si potrebbero passare i parametri se la funzione, chiamata più volte, salva i dati nella stessa porzione di memoria). La pila risulta conveniente anche perché le chiamate di funzione seguono la procedura LIFO: l'ultima che viene chiamata è anche la prima che ritorna, quindi può lasciare lo stack inalterato per la funzione chiamante.

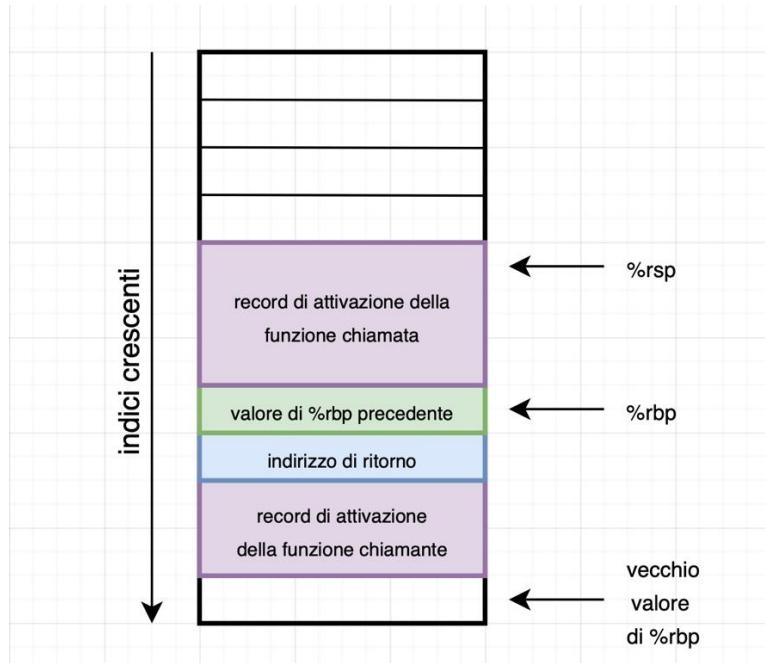
Quando si chiama una funzione, si crea sullo stack il **record di attivazione della funzione**. Questo comprende lo spazio riservato per le variabili locali, per l'indirizzo di ritorno e altri aspetti utili all'esecuzione della funzione. Il suo formato è standard e, nel caso in cui si abbia una nuova chiamata di funzione, basta aggiungere in cima alla pila il record di quest'ultima. Al ritorno della funzione, non serve neanche distruggere il record, basta incrementare lo *stack pointer* in modo che torni al punto in cui si trovava prima della chiamata. L'accesso ai vari campi, e quindi alle varie variabili, sia locali sia argomento della funzione, si fa come offset a partire da un punto di partenza. Tale punto di partenza non conviene che sia memorizzato in %rsp, poiché questo potrebbe essere modificato durante l'esecuzione e rendere complicato il calcolo dell'offset: il registro %rbp è un puntatore al quad sovrastante l'indirizzo di ritorno della funzione, e viene usato proprio in tal senso. Sopra di esso, verrà costruito il record, e gli offset potranno essere calcolati in maniera assoluta, con offset negativi (infatti, andando verso l'alto si decrementano gli indirizzi). Per questo motivo %rbp è fondamentale per ogni funzione, e deve essere preservato in fase di chiamata. Non a caso, ogni funzione **deve** cominciare con le istruzioni

```
push %rbp
mov %rsp, %rbp
sub $costante, %rsp
```

La prima salva il base pointer della funzione precedente, affinché sia recuperato prima di una ret. La seconda istruzione modifica il valore di %rbp ponendolo uguale ad %rsp, che puntava agli 8 byte in cui avevo appena salvato %rbp. Per questo diciamo che %rbp è il base pointer per il record di attivazione: punta al byte immediatamente precedente il suo inizio. Dobbiamo infine lasciare lo spazio sullo stack per quello di cui abbiamo bisogno, così che push e pop non modifichino il record di attivazione. Tale spazio va calcolato guardando la funzione, valutando quanto può servire sia per le variabili locali sia per i parametri. (deve essere multiplo di 16) Prima di ritornare, si devono riportare le cose come stanno, effettuando i passaggi inversi:

```
mov %rbp, %rsp
pop %rbp
```

ret



Adesso, bisogna capire come fa il chiamante a passare dei parametri alla funzione chiamata. Ci sono due modi: nei sistemi a 32 bit, i parametri erano salvati in pila prima dell'indirizzo di ritorno della funzione, in modo che il record di attivazione fosse usato solo per le variabili locali; nei sistemi a 64 bit, si inserisci il contenuto all'interno di registri, seguendo un ordine specifico:

%rdi, %rsi, %rdx, %rcx, %r8, %r9

Se lo spazio dei registri non fosse sufficiente, si ricorrerebbe allo standard del 32 bit, ma, nel nostro caso, non considereremo questa eventualità. È importante che **argomenti diversi vadano in registri diversi**, nonostante la loro dimensione: $f(\text{char } c, \text{ char } d)$ inserisce c in %rdi e d in %rsi. Un approccio del genere si usa anche per il valore di ritorno. Esso viene posto in %rax se 8 byte sono sufficienti, altrimenti in %rax_%rdx. Per restituire oggetti di dimensione maggiore di 16 byte, si ricorre a metodi che vedremo più avanti. Lo stesso per i registri: se l'oggetto (ad esempio una struttura) ha dimensione compresa tra 9 e 16 byte, si usano due registri presi nell'ordine (nel primo ci finisce la parte meno significativa dei byte), altrimenti si usano sistemi più complessi.

La funzione è libera di salvare, nello spazio dedicato, i parametri o meno. Di base questo può non essere conveniente, se ad esempio si effettuano delle semplici elaborazioni, essendo i registri centinaia di volte più veloci della memoria (il compilatore tiene di conto di queste cose). Ci sono però alcuni casi, più o meno banali, in cui è strettamente necessario salvare da qualche parte i dati dei registri. Ad esempio, se dobbiamo chiamare un'altra funzione, e usare quei registri proprio per inserirvi dei dati o, più in generale, se è necessario usare quel dato registro. Consideriamo poi i seguenti casi.

```

struct s{
    int i[4];
};

int f(s x){
    int sum=0;
    for(int j=0; j<4; j++){
        sum += x.i[j];
    }
    return sum;
}

```

La struttura sta su 16 byte, quindi la sua parte meno significativa ($i[0], i[1]$), va in `%rdi`, la più significativa in `%rsi`. Per poter accedere a $i[1]$ e $i[3]$ mi servono i 32 bit più significativi dei due registri, a cui possiamo accedere, per esempio con uno shift o facendo spostamenti tra registri, non direttamente. Per questo, potrebbe essere conveniente salvarli in memoria. Un altro caso piuttosto frequente è

```

void f(int x){
    g(&x);
}

```

Si passa alla funzione `g` l'indirizzo di `x`, ma i registri non hanno indirizzo, quindi si deve necessariamente salvare da qualche parte.

Alcuni esempi di programmazione mista

Consideriamo il seguente esercizio: dato il programma in C++, si vuole riscrivere la funzione elab1 in modo che possa comunicare con il main scritto in C++.

```

1  #include "servi.cpp"
2
3  extern "C" int elab1(int n1, int n2);
4  int alfa, beta;
5
6  int main(){
7      int ris;
8      alfa = leggiint();
9      beta = leggiint();
10     ris = elab1(alfa,beta);
11     scrivint(ris);
12     nuovalinea();
13     return 0;
14 }
15
16 extern "C" int elab1(int n1, int n2){
17     int i, j;
18     i = n1+n2;
19     j = n1-n2;
20     return i*j;
21 }
22
23

```

L'etichetta è quella usata nella funzione, avendola dichiarata con `extern "C"`. Ci sono due variabili locali, `i` e `j`: il compilatore le inserisce in memoria. In questo caso non le inizializziamo perché non le dovremmo leggere mai, ma è un errore leggere qualcosa che non è stato inizializzato: potrà contenere un qualsiasi valore, sulla base di cosa c'era nella pila nelle ultime operazioni di push e pop. Per poter lavorare con il record di attivazione, è necessario disegnarlo e capire quale posizione occupa ciascun dato.

```

1  .global elab1
2  #           +4
3  # +-----+
4  # +   j    |   i    + -16
5  # +-----+
6  # +   n2   |   n1   + -8
7  # +-----+
8  # +   vecchio %rbp  +
9  # +-----+
10
11 elab1:
12     push %rbp
13     mov %rsp, %rbp
14     sub $16, %rsp
15
16     mov %edi, -8(%rbp)
17     mov %esi, -4(%rbp)
18
19     mov %edi, %eax
20     add %esi, %eax
21     mov %eax, -16(%rbp)
22
23     mov %edi, %eax
24     sub %esi, %eax
25     mov %eax, -12(%rbp)
26
27     mov -16(%rbp), %eax
28     imull -12(%rbp)
29
30     leave
31     ret
32

```

Il primo parametro sta nella parte meno significativa di %rdi, e deve essere memorizzato dove è indicato n1. Per accedere a quei 4 byte devo usare l'offset -8 rispetto a %rbp. E per quanto riguarda n2? Non ci dobbiamo **assolutamente** dimenticare che rappresentiamo gli indirizzi crescenti da destra verso sinistra in conformità con il little endian. L'indirizzo di n2 è quindi più vicino a %rbp piuttosto di quello di n1, quindi l'offset sarà -4. Lo stesso vale per i e j: l'indirizzo di i ha offset -16, quello di j -12. I + in alto in figura si usano per capire, sulla base dell'indirizzo di riga, quale valore prendere come offset (es. -8+4 per accedere a n2). In termini di efficienza non è conveniente fare le elaborazioni in memoria, ma fare tutto nei registri e poi copiare nel posto giusto il risultato della computazione. Tuttavia, quando il compilatore non è ottimizzato salva e preleva nuovamente tutti i dati in memoria. In una situazione del genere, anche in assenza di ottimizzazioni, il compilatore avrebbe salvato e modificato il valore di %rbp, ma non avrebbe modificato %rsp per assicurare uno spazio di memoria alla funzione per il suo record di attivazione. Questo dipende dalla funzione stessa: il compilatore si rende conto che lo stack non verrà mai usato fino alla prossima ret, quindi non ha alcun senso spostare %rsp. Questa ottimizzazione della *red zone* si può evitare con il comando `-no-red-zone`

A questo punto, possiamo tradurre la funzione main. In esso vi è una sola variabile locale (alfa e beta sono dichiarati globali, e come tali si inseriranno nel campo .data), quindi il record di attivazione avrebbe dimensione 4 byte. Tuttavia, un incremento del genere comporta %rsp allineato a 4, non a 8: questo fa sì che, volendo fare push di dati a 8 byte, saranno sempre disallineati. Le push allora dovendo inserire in memoria anche singoli byte, come nell'istruzione `push %al`, decrementano sempre %rsp di almeno 8 byte, affinché sia sempre allineato. È preferibile avere dati allineati anche a costo di occupare memoria. Non solo: alcune funzioni che usano lo stack richiedono addirittura i dati allineati sempre a 16 e, in caso contrario, producono un errore che interrompe l'esecuzione. Questa convenzione non è fondamentale, ma sarebbe bene tenerla a mente: da ora in poi, nel riservare spazio per il record di attivazione, accetteremo solo

incrementi di %rsp multipli di 16 byte. Rispetto all'esempio, osserviamo che, una volta lasciato lo spazio per il record di attivazione, la sua gestione è a nostra completa discrezione, e nessuno ci verrà a chiedere di come abbiamo allocato le variabili e in quale ordine. Le variabili globali devono essere dichiarate nella sezione .data e inizializzate con il valore 0, perché questo è ciò che prevede lo standard del C++. Ricordiamo che nell'architettura a 64 bit, una modifica della parte bassa di un registro comporta sempre l'annullamento della parte alta. Infine, possiamo ritenere come standard la sequenza di istruzioni

leave

ret

per la terminazione di una funzione. Infatti, avendo a disposizione molti registri di scratch, raramente accade di dover salvare i registri in pila per poi recuperarli alla fine dell'esecuzione del programma.

```

1  .global main
2  .data
3  alfa:
4  |   .long 0
5  beta:
6  |   .long 0
7
8  .text
9  main:
10 push %rbp
11 mov %rsp, %rbp
12 sub $16, %rsp          #               +4
13 call leggiint          # +-----+
14 mov %eax, alfa(%rip)  # |           |           |
15 call leggiint          # +-----+
16 mov %eax, beta(%rip)  # |           |           |
17 mov alfa(%rip), %edi  # +-----+
18 mov beta(%rip), %esi  # |       xxx    |       xxx    | <- %rsp
19 call elabi             # +-----+
20 mov %eax, -8(%rbp)    # |       xxx    |       ris    | -8
21 mov -8(%rbp), %edi   # +-----+
22 call scrivint          # |           veccio %rbp | <- %rbp
23 call nuovalinea        # +-----+
24 mov $0, %eax           # |           indirizzo di ritorno |
25 leave                  # +-----+
26
27 ret

```

```

1 #include "servi.cpp"
2
3 extern "C" elab3(int& tot, n1, n2){
4     int i,j;
5     i = n1+n2;
6     j = n1-n2;
7     tot = i*j;
8 }
9
10 int main(){
11     int a, b, ris;
12     a = leggiint();
13     b = leggiint();
14     elab3(&ris,a,b);
15     scriviint(ris);
16     nuovalinea();
17     return 0;
18 }
```

Consideriamo una funzione simile in cui uno degli argomenti è un riferimento. Un riferimento non è altro che un puntatore fittizio: il chiamante passa l'indirizzo della variabile locale `ris`, e, usando un indirizzamento indiretto, la funzione chiamata può direttamente modificare la variabile in questione. Si procede quindi disegnando il record di attivazione per avere chiari gli offset e traducendo istruzione per istruzione. Facciamo caso al fatto che per modificare il valore di `ris` lo dobbiamo prima mettere dentro un registro (di scratch), e poi usare l'indirizzamento indiretto. Nel main, quando andiamo a

impostare il valore del primo argomento della funzione in `%rdx`, dovremmo scrivere

```
leaq -16(%rbp), %rdi
```

```

1 .global elab3
2
3 .text
4 elab3:
5     push %rbp
6     mov %rsp, %rbp
7     sub $32, %rsp
8
9     mov %rdi, -8(%rbp)      ##          +4
10    mov %esi, -16(%rbp)    ## |xxxxxxxxxxxxxxxxxxxxxx| <- %rsp
11    mov %edx, -12(%rbp)    ## |           j   |   i   | -24
12
13    mov -16(%rbp), %ea    ## |           |       |
14    add -12(%rbp), %ea    ## |           |       |
15    mov %eax, -24(%rbp)    ## |           n2   |   n1   | -16
16
17    mov -16(%rbp), %ea    ## |           |       |
18    sub -12(%rbp), %ea    ## |           |       |
19    mov %eax, -20(%rbp)    ## |           vecchio %rbp | <- %rbp
20
21    mov -20(%rbp), %ea    ## |           |       |
22    imull -24(%rbp), %    ## |           indirizzo di ritorno | +-----+
23
24    mov -8(%rbp), %rdx
25    mov %eax, (%rdx)
26
27    leave
28    ret
29
```

È bene tenere a mente che, se una cosa viene allocata sullo stack, allora la sua durata non va oltre la fine della funzione, quando `%rsp` torna al punto di partenza. Questo vuol dire che, volendo per esempio gestire una variabile statica in una funzione, è necessario allocarla nel campo `.data`, facendo attenzione al fatto che la sua visibilità è limitata solo alla funzione stessa.

```

1  #include "servi.cpp"
2
3  extern "C" void raddoppia(int a[], int n);
4
5  int main(){
6      int ar[5]; int i;
7      for(i=0; i<5; i++) ar[i] = leggiint();
8      raddoppia(ar,5);
9      for(i=0; i<n; i++) scrivint(ar[i]);
10     nuovalinea();
11 }
12
13 extern "C" void raddoppia(int a[], int n){
14     int i;
15     for(i=0; i<n; i++) a[i] = 2*a[i];
16 }
17

```

Si traduca la seguente porzione di codice. Partiamo da raddoppia. La prima cosa da fare è, come sempre, disegnare il record di attivazione. Un array si passa come puntatore al primo elemento, quindi a in memoria occuperà 8 byte, contenendo l'indirizzo di a[0].

```

1  #          +4
2  #+#-----+
3  #+#   i    |   n    + -16
4  #+#-----+
5  #+#   a    + -8
6  #+#-----+
7  #+#   vecchio %rbp +
8  #+#-----+
9  #+#   indirizzo di ritorno +
10 #+#-----+
11 .text
12     raddoppia:
13         push %rbp
14         mov %rsp, %rbp
15         sub $16, %rbp
16
17         mov %rdi, -8(%rbp)
18         mov %esi, -16(%rbp)
19         movl $0, -12(%rbp)
20     .Lfor:
21         cmp %esi, -12(%rbp)
22         jl .Lcorpo
23         jmp .Lfine
24     .Lcorpo:
25         movslq -12(%rbp), %rdx
26         mov (%rdi,%rdx,4), %eax
27         sal %eax
28         mov %eax, (%rdi,%rdx,4)
29         incl -12(%rbp)
30         jmp .Lfor
31     .Lfine:
32         leave
33         ret

```

Per tradurre il ciclo, conviene mantenere il suo valore semantico in ogni momento, in modo da non fare errori. All'inizio del corpo, si inizializza il valore di i, che sta in -12(%rbp), e si confronta con n, il valore già contenuto dentro %esi. Se i < n, allora entro nel corpo del for, contrassegnato dall'etichetta .Lcorpo, altrimenti vado fuori. Per accedere ad a[i], devo inserire dentro un registro a 64 bit il valore di i, che farà da indice, dentro un altro il valore del puntatore, che farà da base e usare la scala di 4, avendo a che fare con interi. Il problema è che i deve essere esteso con segno su 64 bit! Infatti, questo tipo di indirizzamento mi consente anche di accedere ad elementi che vengono prima del punto di partenza, non solo dopo. Per estendere un operando a 32 bit su 64 ed inserirlo dentro un registro, si usa

movslq source, dest

dove la s sta per ‘estensione con segno’, lq per ‘da long a quad’. Ovviamente, per accedere nel posto dove voglio io, devo usare l’indirizzo **contenuto** negli 8 byte che partono da -8(%rbp), non quelli di questo oggetto. Sarebbe stato un errore scrivere

```
mov %eax, -8(%rbp, %rdx, 4).
```

Adesso possiamo tradurre il main. In questo caso, facciamo uso della direttiva .set. Essa mi permette di definire delle costanti simboliche che saranno poi sostituite con i simboli associati. Sono utili per indirizzamenti nello stack.

Anche in questo caso, per spostare l’indice dalla memoria ad un registro, è stato necessario estendere con segno. Tuttavia, tramite %rbp possiamo accedere proprio alla locazione che ci interessa, quindi usiamo la sintassi vista sopra per indirizzarlo. Allo stesso modo, al momento di chiamare la funzione raddoppia, dobbiamo mettere dentro %rdi l’indirizzo di ar[0], trattandosi, come già visto, di un puntatore. Alla riga 31, sarebbe un errore prendere dalla memoria l’indice e poi chiamare la funzione. %rdx è infatti un registro di scratch, e non abbiamo alcuna garanzia che il suo valore sia preservato a seguito dell’esecuzione di leggiint. Anche in questo caso, è bene attenersi rigorosamente alla sintassi del ciclo for: eseguo il primo statement; eseguo la condizione: se è falsa esco, altrimenti eseguo il corpo; eseguo il terzo statement e verifico nuovamente la condizione.

```

1   #          +4
2   #+-----+
3   #+    xxx    |    i    + -32
4   #+-----+
5   #+    ar[1]  |    ar[0] + -24
6   #+-----+
7   #+    ar[3]  |    ar[2] + -16
8   #+-----+
9   #+    xxx    |    ar[4] + -8
10  #+-----+
11  #+      vecchio %rbp +
12  #+-----+
13  #+      indirizzo di ritorno +
14  #+-----+
15
16 .set ar, -24
17 .set i, -32
18
19 .text
20     main:
21         push %rbp
22         mov %rsp, %rbp
23         sub $32, %rsp
24
25         movl $0, i(%rbp)
26     .Lfor1:
27         cmpl $5, -12(%rbp)
28         jl .Lcorpo1
29         jmp .Lfine1
30     .Lcorpo1:
31         call leggiint
32         movslq -12(%rbp), %rdx
33         mov %eax, ar(%rbp,%rdx,4)
34         incl -12(%rbp)
35         jmp .Lfor1
36
37     .Lfine1:
38         lea ar(%rbp), %rdi
39         mov $5, %esi
40         call raddoppia
41
42         movl $0, i(%rbp)
43     .Lfor2:
44         cmpl $5, -12(%rbp)
45         jl .Lcorpo2
46         jmp .Lfine2
47     .Lcorpo2:
48         movslq -12(%rbp), %rdx
49         mov ar(%rbp, %rdx, 4), %rdi
50         call scrivint
51         incl -12(%rbp)
52         jmp .Lfor2
53     .Lfine2:
54         leave
55         ret

```

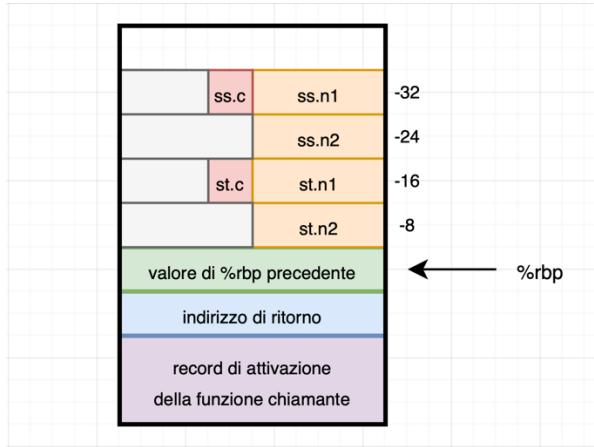
```

1  #include "servi.cpp"
2
3  struct s{int n1; char c; int n2};
4
5  extern "C" s leggis(){
6      s ss;
7      ss.n1 = leggiint();
8      ss.c = leggichar();
9      ss.n2 = leggiint();
10     return ss;
11 }
12
13 extern "C" scrivirsi(s ss){
14     scrivint(ss.n1);
15     scrivchar(ss.c);
16     scrivint(ss.n2);
17     nuovalinea();
18 }
19
20 extern "C" fai(s st){
21     s ss;
22     ss.n1 = st.n1 + 5;
23     ss.c = st.c + 1;
24     ss.n2 = st.n2 + 10;
25     return ss;
26 }
27
28 int main(){
29     s st1, st2;
30     st1 = leggiris();
31     st2 = fai(st1);
32     scrivirsi(st2);
33     return 0;
34 }
```

Si traduca la funzione `fai`. Si vuole quindi passare una struttura per valore. Osserviamo che la struttura sta su 16 byte (in particolare, il suo `alignof` è 4, il suo `sizeof` è 12), quindi si possono usare i registri sia per i parametri che per il valore di ritorno. La regola per l'inserimento nei registri è la seguente:

- Quando una struttura di dimensione compresa tra 9 e 16 è passata per valore, e sono disponibili, nell'ordine, i registri R1 e R2 (nel nostro caso, `%rdi` e `%rsi`), la parte meno significativa va in R1, quella più significativa in R2, dove per **meno significativa** si intende quella che sta più in alto nella rappresentazione della memoria (appunto, ad indirizzo minore).
- Quando una struttura di dimensione compresa tra 9 e 16 è ritornata da una funzione, allora la sua parte meno significativa è inserita dentro `%rax`, quella più significativa dentro `%rdx`.

La rappresentazione della memoria così realizzata, ci mostra che la parte meno significativa della struttura, quella passata dentro `%rdi`, coinvolge i `n1` e `c`, mentre quella più significativa, in `%esi`, al byte successivo per rispettare l'allineamento, comprende solo `n2`. La conoscenza del formato della struct ci è fondamentale per accedere ai vari campi.



```

1   .text
2   fai:
3   |   push %rbp
4   |   mov %rsp, %rbp
5   |   sub $32, %rsp
6   |   mov %rdi, -16(%rbp)
7   |   mov %esi, -8(%rbp)
8   |   #ss.n1 = st.n1 + 5
9   |   mov -16(%rbp), %eax
10  |   add %5, %eax
11  |   mov %eax, -32(%rbp)
12  |   #ss.c = st.c + 1
13  |   mov -12(%rbp), %al
14  |   add %1, %al
15  |   mov %al, -28(%rbp)
16  |   #ss.n2 = st.n2 + 10
17  |   mov -8(%rbp), %eax
18  |   add %10, %eax
19  |   mov %eax, -24(%rbp)
20
21  |   mov -32(%rbp), %rax
22  |   mov -24(%rbp), %rdx
23
24  |   leave
25  |   ret
26

```

Consideriamo il main. Come si traducono le due istruzioni (supponendo che l'allineamento nello stack sia come quello della figura precedente, identificando st con st1 e ss con st2)

```

st1 = leggiris();
st2 = fai(st1);

```

```
# st1 = leggiris
call leggiris
mov %rax, -16(%rbp)
mov %rdx, -8(%rbp)
# st2 = fai(st1)
mov -16(%rbp), %rdi
mov -8(%rbp), %rsi
call fai
mov %rax, -32(%rbp)
mov %rdx, -24(%rbp)
```

Le etichette delle funzioni

Finora abbiamo fatto uso dei `extern "C"` affinché il compilatore adottasse come etichetta l'identificatore della funzione. Tuttavia, in C++ esiste l'overloading delle funzioni: `func1(int)` e `func1(int, int)`, pur avendo lo stesso identificatore, sono funzioni diverse, selezionate dal compilatore a seconda del parametro che si passa loro in chiamata, e come tali devono essere distinte nel codice Assembler. Il collegatore associa le etichette confrontando le stringhe, quindi assieme all'identificatore dobbiamo inserire delle informazioni in riferimento ai parametri della funzione. Non solo: l'algoritmo con il quale si generano tali etichette deve essere standardizzato, in modo che, anche se su file sorgenti diversi, una volta compilati le etichette siano le stesse per il file oggetto dove è stata definita la funzione e dove è stata chiamata. L'algoritmo che vediamo noi, nelle sue funzioni di base, è quello usato dal compilatore g++. Chiaramente, l'overloading si effettua solo sugli argomenti, non sul valore di ritorno, per cui di quest'ultimo non ci sarà alcun riferimento all'interno dell'identificatore.

Data la funzione `func(tipo1, ..., tipon)`, la stringa che sarà la sua etichetta è `_ZfuncS1S2...Sn`, dove `_Z` è standard per le etichette di funzioni, il numero indica quanti caratteri ha l'identificatore della funzione e i caratteri successivi, uno per parametro, sono associati ai diversi tipi secondo certe regole. Bisogna vedere come distinguere i tipi base, i tipi definiti dall'utente (`struct`, `class` e `enum`) e i tipi derivati (`*`, `&`, `const` e `array`).

Per i tipi base, si usa banalmente una tabella.

Tipo C++	Sotto identificatore Assembler
void (o argomenti assenti)	v
short int	s
int	i
long int	l
unsigned short int	t
unsigned int	j
unsigned long int	m
char	c
unsigned char	h

bool	b
------	---

```
miaFunzione(int,char,bool) -> _Z11miaFunzioneicb
```

Per quanto riguarda i tipi derivati, si usa l'identificatore con il quale sono stati dichiarati, preceduti dal numero di caratteri dei quali è composto l'identificatore. Avendo definito la struct miaStruct{...}

```
miaFunzione(miaStruct,int) -> _Z11miaFunzione9miaStructi
```

Per i puntatori, si utilizza una P maiuscola, che sta per Pointer.

```
tipo* -> Ptipo
int* -> Pi
int** -> PPi
miaStruct* -> P9miaStruct
```

Una cosa simile si ha per i riferimenti, con i quali si usa la lettera R. Per collegare modificatori diversi, si leggono da destra verso sinistra, come nel terzo esempio (e come si osserva bene negli esempi successivi).

```
tipo& -> Rtipo
int& -> Ri
mias*& -> RP4mias
```

Per i dati const, è necessario fare più attenzione. Innanzitutto, si considerino i tipi

```
const int *
int const *
int * const
```

Il terzo è un puntatore costante ad un intero. Significa che posso usare il puntatore per modificare il dato a cui punta, ma non posso modificare il puntatore in sé. Il secondo è un puntatore ad intero costante:

non posso modificare il valore del dato a cui punta con quel puntatore, ma posso modificare il dato a cui punta. Il primo è una sintassi antiquata per il secondo. Infatti, come regola generale, quando `const` è la prima cosa che vediamo nel tipo, allora si riferisce all'elemento **immediatamente** successiva. In questo caso, il tipo è `int const *`, e quindi uguale al secondo. In tutti gli altri casi, il `const` è riferito a **quello che viene immediatamente prima** (e infatti, sotto questa luce si interpreta il significato del secondo e del terzo tipo). Riflettiamo poi sul fatto che passare per valore ad una funzione un puntatore costante (e non a costante) non ha senso: infatti le modifiche dentro le funzioni non vanno ad influenzare il dato passato come parametro. Quindi, se `const` è l'ultimo elemento di un tipo, nel dare un identificatore si può ignorare. `funz(int)`, `funz(const int)` e `funz(int const)` sono esattamente la stessa funzione.

```
miaFunzione(int*) -> _Z11miaFunzionePi
miaFunzione(const int *) -> _Z11miaFunzionePKi
miaFunzione(int const *) -> _Z11miaFunzionePKi
miaFunzione(int * const) -> _Z11miaFunzionePi
miaFunzione(int const * const) -> _Z11miaFunzionePKi
miaFunzione(int * const *) -> _Z11miaFunzionePKPi
```

Per quanto riguarda gli array, sappiamo che spesso decadono in puntatori, perdendo, se unidimensionali, le informazioni sulle dimensioni. Per tale ragione

```
miaFunzione(int* a) -> _Z11miaFunzionePi
miaFunzione(int a[]) -> _Z11miaFunzionePi
miaFunzione(int a[10]) -> _Z11miaFunzionePi
```

Le classi

Le classi rappresentano la grande innovazione nel passaggio da C a C++ ma, nella sostanza, non sono altro che un aggregato di dati, al pari delle strutture, e di codice che può essere chiamato su di essi. Bisogna quindi capire come lavorare sui membri, la definizione delle funzioni membro, dei costruttori e distruttori.

Il primo approccio di fronte alla traduzione è stato quello di passare ad una formulazione del codice come avverrebbe in C, e usare questo per il passaggio ad Assembler. Per farlo, al momento della dichiarazione della classe il compilatore non fa altro se non tenersi da parte la formulazione e le funzioni che possono essere invocate su di esso. Queste ultime si possono vedere come delle funzioni globali, che hanno, tra i vari parametri, un riferimento all'oggetto sul quale sono state chiamate. In particolare, nel momento in cui dichiaro l'oggetto `obj` e scrivo `obj.func()`, questa viene tradotta come la chiamata ad una funzione `func` che ha come primo parametro l'indirizzo dell'oggetto su cui è invocata. Per utilizzarlo, all'interno della funzione si usa il puntatore `this`. Con un ragionamento del genere, le funzioni membro sono del tutto equivalenti alle normali funzioni del C++, con la differenza che nell'etichetta dovranno avere un qualche riferimento alla classe alla quale appartengono. Sarà poi standard il fatto che in %rdi finirà l'indirizzo dell'oggetto su cui sono state chiamate le funzioni (questo parametro non è mai dichiarato esplicitamente), nei registri successivi eventuali altri parametri.

```
class C{
    tipo_r func(tipo_1, ..., tipo_n);
}
```

Dopo `_Z` nell'etichetta della funzione, si scrivono tra le lettere N ed E (nested-end) i nomi degli scope al quale appartiene la funzione (con tanto di eventuali namespace), per poi aggiungere il nome della funzione stessa. Dopo la E, si ha la lista dei parametri per come l'abbiamo già vista, senza contare `this`.

```
void miaClasse::func(); -> _ZN9miaClasse4funcEv
```

Se uno dei tipi si rifà alla classe stessa (magari è un oggetto passato per valore, o un puntatore), invece di ripetere il nome della classe si usa `S_`.

```
class miaC{
    void funz1();          -> _ZN4miaC5funz1Ev
    void funz2(int);       -> _ZN4miaC5funz2Ei
    void funz3(miaC*);    -> _ZN4miaC5funz3EPS_
}
```

Se la funzione che vogliamo dichiarare è un costruttore, al posto del nome si usa C1. Per convenzione, i costruttori, pur non avendo un valore di ritorno esplicito, lasciano in %rax l'indirizzo dell'oggetto su cui sono stati chiamati. Per i distruttori, al posto del nome si usa D1.

```

1 void scrivint(int);
2
3 class clai{
4     int ix, iy, iz;
5 public:
6     clai();
7     clai(int,int,int);
8     void stampa();
9     clai somma(int,int,int);
10 };
11
12 clai::clai(int a, int b, int c){
13     ix = a; iy = b; iz = c;
14 }
15
16 clai::clai(){
17     ix = 0; iy = 0; iz = 0;
18 }
19
20 void clai::stampa(){
21     scrivint(ix); scrivint(iy); scrivint(iz);
22 }
23
24 clai clai::somma(int a, int b, int c){
25     clai temp;
26     temp.ix = ix + a; temp.iy = iy + b; temp.iz = iz + c;
27     return temp;
28 }
29
30 int main(){
31     clai c1, c2(1,2,3), c3;
32     c1 = c2.somma(5,10,15); c1.stampa();
33 }
```

Cominciamo con la funzione stampa. Essa non deve far altro che prendere i vari valori e inviarli come parametri a stampaint.

```

#
# +-----+
# +      this      + -8
# +-----+
# +    vecchio %rbp   +
# +-----+


.global _ZN4clai5stampaEv
_ZN4clai5stampaEv:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rsp
    mov %rdi, -8(%rbp)

    mov -8(%rbp), %rax
    mov (%rax), %esi
    call _Z9stampaintv
    mov -8(%rbp), %rax
    mov 4(%rax), %esi
    call _Z9stampaintv
    mov -8(%rbp), %rax
    mov 8(%rax), %esi
    call _Z9stampaintv

    leave
    ret
```

I due costruttori sono estremamente simili. Ricevono l'indirizzo dell'oggetto a che devono costruire in %rdi, e restituiscono lo stesso valore in %rax

```

1   #
2   # +-----+
3   # +      this      +    +8
4   # +-----+
5   # +      vecchio %rbp      +
6   # +-----+
7
8 .global _ZN4claiC1Ev
9 _ZN4claiC1Ev:
10  push %rbp
11  movq %rsp, %rbp
12  subq $16, %rbp
13  mov %rdi, -8(%rbp)
14  movl $0, (%rdi)
15  movl $0, 4(%rdi)
16  movl $0, 8(%rdi)
17  movq %rdi, %rax
18  leave
19  ret
20

#          +4
# +-----+
# +      b      |      a      +    -24
# +-----+
# +      |      c      +    -16
# +-----+
# +      this      +    -8
# +-----+
# +      vecchio %rbp      +
# +-----+
.global _ZN4claiC1Eiii
_ZN4claiC1Eiii:
push %rbp
mov %rsp, %rbp
sub $32, %rbp
mov %rdi, -8(%rbp)
mov %esi, -24(%rbp)
mov %edx, -20(%rbp)
mov %ecx, -16(%rbp)
mov %esi, (%rdi)
mov %ecx, 4(%rdi)
mov %edx, 8(%rdi)
mov %rdi, %rax
leave
ret

```

La funzione `somma` è un po' più sottile delle altre. Infatti, in essa viene dichiarata un'istanza della classe stessa, ma, come tale, è necessario che vi si chiami sopra uno dei costruttori. È bene non dimenticarsi di questo aspetto, così come si deve invocare un distruttore nel momento in cui un'istanza di quella classe termina il suo scope. Un'altra peculiarità è che vorremmo restituire un'istanza della classe `clai`. Anche in questo caso, non vogliamo restituire il `result` che abbiamo allocato sullo stack nel record di attivazione, ma lo dobbiamo prendere ad esempio per chiamare il costruttore di copia rispetto al punto dove vogliamo il risultato. Approfondiremo questo aspetto più avanti. Per ora, visto che il `sizeof` della classe è minore di 16, possiamo restituire il risultato in `%rax` o `%rdx`, aspettandoci che nel main sia gestito di conseguenza. La traduzione di `main` consiste nel mettere assieme tutte queste cose: costruire gli oggetti e chiamare le varie funzioni membro sugli oggetti appena creati.

```

1 #          +4
2 # +-----+
3 # + temp.iy | temp.iz + -40
4 # +-----+
5 # +xxxxxxxxx| temp.iz + -32
6 # +-----+
7 # +     b    | a      + -24
8 # +-----+
9 # +xxxxxxxxx| c      + -16
10 # +-----+
11 # +      this   + -8
12 # +-----+
13 # + vecchio %rbp +
14 # +-----+
15 .global _ZN4clai5sommaEiii
16 _ZN4clai5sommaEiii:
17     push %rbp
18     mov %rsp, %rbp
19     sub $48, %rsp
20     mov %rdi, -8(%rbp)
21     mov %esi, -24(%rbp)
22     mov %edx, -20(%rbp)
23     mov %ecx, -16(%rbp)
24     lea -40(%rbp), %rdi
25     call _ZN4claiC1Ev
26     mov -8(%rbp), %rdi
27     mov (%rdi), %eax
28     add -24(%rbp), %eax
29     mov %eax, -40(%rbp)
30     #Si fa lo stesso per iy e iz
31     mov -40(%rbp), %rax
32     mov -32(%rbp), %edx
33     #Eventuale chiamata del distruttore
34     leave
35     ret
36
37
38
1 #          +4
2 # +-----+
3 # + c3. ix |xxxxxxxxxxxxx+ -40
4 # +-----+
5 # + c3. iz | c3. iy + -32
6 # +-----+
7 # + c2. iy | c2. ix + -24
8 # +-----+
9 # + c1. ix | c2. iz + -16
10 # +-----+
11 # + c1. iz | c1. iy + -8
12 # +-----+
13 # + vecchio %rbp +
14 # +-----+
15 .global main
16 main:
17     push %rbp
18     mov %rsp, %rbp
19     sub $48, %rsp
20
21     lea -12(%rbp), %rdi
22     call _ZN4claiC1Ev
23     lea -24(%rbp), %rdi
24     mov $1, %esi
25     mov $2, %edx
26     mov $3, %ecx
27     call _ZN4claiC1Eiii
28     lea -36(%rbp), %rdi
29     call _ZN4claiC1Ev
30
31     lea -24(%rbp), %rdi
32     mov $5, %esi
33     mov $10, %edx
34     mov $15, %ecx
35     call _ZN4clai5sommaEiii
36     mov %rax, -12(%rbp)
37     mov %edx, -4(%rbp)
38     #...

```

Finora non abbiamo preso in considerazione la possibilità che una funzione abbia come valore di ritorno una classe o una struttura che non stiano su due registri. Più in generale, ogni volta che calcolo un'espressione, il C++ si occupa non solo di crearla, ma anche di allocarla in memoria come oggetto temporaneo. Supponiamo che `a` e `b` siano istanza di una classe `C` che ha una dimensione piuttosto grande, e nel quale è stato ridefinita la funzione `operator+`. Scrivere `a+b` necessita chiaramente di uno spazio in memoria nel quale allocare il risultato, essendo anch'esso grande, nonostante magari il risultato dell'espressione non sia neanche utilizzato. Più in generale, tale valore è di ritorno della chiamata `C::operator+(&a, b)`, quindi ci riconduciamo allo stesso problema. Nello standard C++, lo spazio nel quale inserire tale valore risultato deve essere allocato sullo stack dalla funzione chiamante. La funzione chiamata si occupa di costruirlo facendo uso dell'operatore di copia, dove presente, mentre il chiamante ne chiama il costruttore alla fine del suo scope (per espressioni così calcolate, il distruttore viene chiamato quando è finito lo statement che lo coinvolge). La funzione vuole come primo parametro, in `%rdi`, l'indirizzo di questo oggetto temporaneo (anche nel caso in cui vi sia necessità di `this`, che andrà come secondo parametro in `%rsi`). Al momento in cui si incontra `return x`, lo spazio allocato dal chiamante viene allocato con tale valore.

```
c = somma(a,b);
```

In un caso come questo, il chiamante alloca sullo stack sia lo spazio per l'oggetto `c` sia per quello fittizio nel quale somma andrà a inserire il risultato della chiamata. Dopo, spetta al chiamante effettuare la copia da tale oggetto in `c`, e alla fine distruggere il primo. Vediamo un esempio.

```

1  struct s{int n1; int n2; char c[10];};
2
3  s fstruct(int a, char c){
4      int i; s st;
5      st.n1 = a;
6      st.n2 = 2*a;
7      for(i=0; i<10; i++) st.c[i] = c+i;
8      return st;
9  }
10
11 int main(){
12     s sa;
13     as = fstruct(5,'a');
14     return 0;
15 }
```

Come visto, la funzione `fstruct` alloca lo spazio per la sua variabile locale `st` e lavora direttamente con questa. Poi, al momento del `return`, non è che *restituisce* `st`, ma la usa come espressione con cui costruire lo spazio di memoria allocato dal chiamante e passato alla funzione stessa. Per effettuare la copia, nel caso in cui non sia previsto un costruttore di copia, si possono utilizzare le funzioni `stringa`, e, in particolare, la `movs`. Per convenzione, la funzione restituisce l'indirizzo dell'oggetto fittizio da essa costruita nel registro `%rax`.

```

1 .global _Z7fstructic
2 _Z7fstructic:
3     push %rbp
4     mov %rsp, %rbp
5     sub $48, %rsp
6     mov %rdi, -8(%rbp)
7     mov %esi, -16(%rbp)
8     mov %dl, -12(%rbp)
9     #eventuale costruttore di st
10    #elaborazioni per inizializzare st
11    lea -40(%rbp), %rsi
12    mov -8(%rbp), %rdo
13    mov $5, %rcx
14    rep movsl
15    mov -8(%rbp), %rax
16    leave
17    ret
```

<pre> # +4 # +-----+ # + st.n2 st.n1 + -40 # +-----+ # + st.c[7] ... st.c[0]+ -32 # +-----+ # + i st.c[9]... + -24 # +-----+ # +xxxxxxxxx c a + -16 # +-----+ # + ind result + -8 # +-----+ # + vecchio %rbp + # +-----+ </pre>
--

Così come nella funzione abbiamo copiato il risultato nello spazio il cui primo indirizzo era contenuto in `%rdi`, nel `main` ci dobbiamo occupare di riservare tale spazio e di usare i suoi dati per inizializzare quello che vogliamo.

```

1 .global main
2 main:
3     push %rbp
4     mov %rsp, %rbp
5     sub $48, %rsp
6     lea -48(%rbp), %rdi
7     mov $5, %esi
8     mov $'c', %dl
9     call _Z7fstructic
10    lea -48(%rbp), %rsi
11    lea -24(%rbp), %rdi
12    mov $5, %rcx
13    rep movsl
14    #eventuale distruttore di temp
15    #eventuale distruttore di sa
16    xor %eax, %eax
17    leave
18    ret
#          +4
# +-----+
# + temp.n2 | temp.n1 + -48
# +-----+
# + temp.c[7]... temp.c[0]+ -40
# +-----+
# +           | t.c[9]... + -32
# +-----+
# + sa.n2 | sa.n1 + -24
# +-----+
# + sa.c[7] ... sa.c[0]+ -16
# +-----+
# +           |sa.c[9]... + -8
# +-----+
# +   vecchio %rbp +
# +-----+

```

Questo è il metodo che è nato per quando è necessario avere un oggetto ‘grande’ come valore di ritorno. Qualcosa di affine accade quando lo stesso oggetto deve essere passato per valore, ma non affrontiamo tale visto essendo molto più complicato. Piuttosto, prendiamo in considerazione il costruttore di copia. Quand’è che viene chiamato? I casi sono tre: quando si costruisce un oggetto con un altro dello stesso tipo; quando si passa un oggetto per valore; quando viene ritornato un oggetto. Consideriamo il seguente caso:

```

miaClasse g(){
    return miaClasse(100);
}

...
miaClasse c1 = g();

```

Secondo quanto visto, la chiamante alloca spazio sullo stack per un oggetto temporaneo. La funzione crea l’oggetto `miaClasse`, poi usa un costruttore di copia (eventualmente facendo una copia bit a bit, come nei casi precedenti) per costruire l’oggetto temporaneo. Distrugge quello costruito e ritorna. La chiamata usa il valore dell’oggetto temporaneo per inizializzare `c1`, poi distrugge quello temporaneo. Tuttavia, a partire da C++17 lo standard prevede tutta una serie di ottimizzazioni (anche intuitive) che permettono di evitare tutti questi oggetti temporanei e copie.

La *return value optimization (RVO)* fa sì che, nel caso in cui il valore di ritorno sia usato per inizializzare un oggetto nella funzione chiamate, piuttosto che creare un oggetto temporaneo nello stack si passa direttamente l’indirizzo dell’oggetto che sarà in ultimo inizializzato. Nell’esempio di sopra, `temp` in `main` non esiste più, e a `fstruct` si passa l’indirizzo dello spazio riservato per `sa`, in modo che, alla fine della chiamata, questo sia già costruito.

Un’altra ottimizzazione (non necessaria ma implementata da molti compilatori), è la *named return value optimization (NRVO)*. Supponiamo, come in `fstruct`, che si debba creare un oggetto, `st`, che sarà poi

usato per inizializzare `sa` in `main`. L'idea è quella per cui conviene, già nella funzione, lavorare direttamente con `sa` nel record di `main`, in modo che non si debba poi fare alcuna copia. Nell'esempio, si vede che si accede ai vari dati tramite indirizzamento indiretto.

```

1      #          +4
2      # +-----+
3      # + sa.n2 | sa.n1 + -24
4      # +-----+
5      # + sa.c[7] ... sa.c[0]+ -16
6      # +-----+
7      # +     |sa.c[9]... + -8
8      # +-----+
9      # + vecchio %rbp +
10     # +-----+
11
12 .global main
13 main:
14     push %rbp
15     mov %rsp, %rbp
16     sub $32, %rsp
17     lea -24(%rbp), %rdi
18     mov $5, %esi
19     mov $'c', %dl
20     call _Z7fstructic
21     #eventuale distruttore di sa
22     xor %eax, %eax
23     leave
24     ret
25
26
1      #          +4
2      # +-----+
3      # + |c| a + -16
4      # +-----+
5      # +     ind result + -8
6      # +-----+
7      # + vecchio %rbp +
8      # +-----+
9      .global _Z7fstructic
10     _Z7fstructic:
11     push %rbp
12     mov %rsp, %rbp
13     sub $16, %rsp
14     mov %rdi, -8(%rbp)
15     mov %esi, -16(%rbp)
16     mov %dl, -12(%rbp)
17     mov %esi, (%rdi)
18     shl $1, %esi
19     mov %esi, 4(%rdi)
20     #altre elaborazioni...
21     mov -8(%rbp), %rax
22     leave
23     ret
24

```

Alcune prove d'esame

1. Siano date le seguenti dichiarazioni, contenute nel file cc.h:

```
struct st1 { char v1[4]; };
struct st2 { char v2[4]; };
class cl {
    char v3[4]; int v3[4]; long v2[4];
public:
    cl(st1& ss);
    cl elabi(char ar1[], st2 s2);
    void stampa() {
        for (int i = 0; i < 4; i++) cout << (int)v1[i] << ' '; cout << endl;
        for (int i = 0; i < 4; i++) cout << (int)v2[i] << ' '; cout << endl;
        for (int i = 0; i < 4; i++) cout << (int)v3[i] << ' '; cout << endl << endl;
    }
};
```

Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(st1& ss)
{
    for (int i = 0; i < 4; i++) {
        v1[i] = ss.vi[i]; v2[i] = v1[i] / 2;
        v3[i] = 2 * v1[i];
    }
}
cl cl::elabi(char ar1[], st2 s2)
{
    st1 s1;
    for (int i = 0; i < 4; i++) s1.vi[i] = ar1[i] + i;
    cl cla(s1);
    for (int i = 0; i < 4; i++) cla.v3[i] = s2.vd[i];
    return cla;
}
```

Ad una prova d'esame, il file cc.h contiene le definizioni che andremo ad utilizzare. Nostro compito sarà quello di tradurre alcune delle funzioni dichiarate in questo file, e usare prova1.cpp (compilato insieme al nostro file Assembler) per verificare che l'output ottenuto sia lo stesso di quello in es1.out.

Di fronte ad un esercizio del genere, è bene riflettere sulle strutture dati che abbiamo davanti, e rappresentarle subito in memoria con la propria dimensione e il proprio allineamento. In questo caso, st1 e st2 sono strutture molto semplici con allineamento ad 1 e dimensione complessiva di 4 byte. Per quanto riguarda la classe cl, si usano le regole di layout delle strutture, per cui si inseriscono gli elementi nell'ordine con cui appaiono, usando meno memoria possibile e rispettando l'allineamento di ciascuno. Il risultato è il seguente:

```
#          +4
# +-----+
# +      v3[0] |      v1      +  <- inizio Classe
# +-----+
# +      v3[2] |      v3[1]      +  +8
# +-----+
# +XXXXXXXXXXXXXX|      v3[3]      +  +16
# +-----+
# +          v2[0]      +  +24
# +-----+
# +          v2[1]      +  +32
# +-----+
# +          v2[2]      +  +40
# +-----+
# +          v2[3]      +  +48
# +-----+
# +                               +  +56
# +-----+
```

Si osservi che l'offset del primo byte che va fuori dall'oggetto è anche la dimensione complessiva dell'oggetto. A questo punto possiamo tradurre il costruttore. Si rappresenta in maniera standard il record di attivazione, che comprenderà il puntatore a this, il riferimento, che sappiamo essere implementato come

puntatore, e l'unica variabile locale, i. Come sempre, per un costruttore si inserisce in %rax l'indirizzo dell'oggetto appena costruito, il valore di this.

L'altra funzione, che esegue un'elaborazione, ha come valore di ritorno un'istanza della classe cl, che è non banale occupando 56 byte. Secondo la convenzione, è il chiamante ad allocare la memoria per tale oggetto, e la funzione, ricevendo il suo indirizzo in %rdi, si occupa di costruirlo con il costruttore di copia o, in assenza di questo, effettuando una copia bit a bit dei due oggetti. Per il resto, all'interno della funzione si ha tra le variabili locali un oggetto al quale si dovrà riservare della memoria e sul quale si dovrà chiamare il costruttore. Un'osservazione importante è che, pur avendo due cicli, non possiamo più usare l'etichetta .Lfor, essendo già stata usata nella stessa sezione text.

```

1  # +-----+
2  # +XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX+ -32
3  # +-----+
4  # +XXXXXXXXXXXXXXX| i + -24
5  # +-----+
6  # + &ss + -16
7  # +-----+
8  # + &this + -8
9  # +-----+
10 # + old %rbp + <- %rbp
11 # +-----+
12
13 .global _ZN2clC1EP3st1
14 _ZN2clC1EP3st1:
15     push %rbp
16     mov %rsp, %rbp
17     sub $32, %rbp
18     mov %rdi, -8(%rbp)
19     mov %rsi, -16(%rbp)
20
21     movl $0, -24(%rbp)
22
23 .Lfor:
24     cmpl $5, -24(%rbp)
25     jl .Lcorpo
26     jmp .Lfine
27
28 .Lcorpo:
29     #v1[i] = ss.v1[i]
30     movslq -24(%rbp), %rcx
31     mov -16(%rbp), %rdi
32     mov (%rdi, %rcx), %al
33     mov -8(%rbp), %rdi
34     mov %al, (%rdi, %rcx)

35
36     #v2[i] = v[1]/2
37     mov (%rdi, %rcx), %al
38     sar $1, %al
39     #devo estendere con segno da byte a quad
40     movsbq %al, %rax
41     #uso l'offset di v2 dentro l'oggetto classe
42     mov %rax, +24(%rdi, %rcx, 8)
43     #v3[i] = v[1]*2
44     mov (%rdi, %rcx), %al
45     shl $1, %al
46     movsbl %al, %eax
47     mov %eax, 4(%rdi, %rcx, 4)
48     incl -24(%rbp)
49     jmp .Lfor

50 .Lfine:
51     mov -8(%rbp), %rax
52     leave
53     ret

```

```

1 # +-----+
2 # + v3[0] | v1 + -96
3 # +-----+
4 # + v3[2] | v3[1] + -88
5 # +-----+
6 # +xxxxxxxxxxxxxx| v3[3] + -80
7 # +-----+
8 # + v2[0] + -72
9 # +-----+
10 # + v2[1] + -64
11 # +-----+
12 # + v2[2] + -56
13 # +-----+
14 # + v2[3] + -48
15 # +-----+
16 # +xxxxxxxxxxxxxx| s1 + -40
17 # +-----+
18 # + i | s2 + -32
19 # +-----+
20 # + ar1 + -24
21 # +-----+
22 # + %this + -16
23 # +-----+
24 # + &risultato + -8
25 # +-----+
26 # + old %rbp + <- %rbp
27 # +-----+
28 .global _ZN2cl5elab1EPc3st2
29 _ZN2cl5elab1EPc3st2:
30     push %rbp
31     mov %rsp, %rbp
32     sub $96, %rsp
33     mov %rdi, -8(%rbp)
34     mov %rsi, -16(%rbp)
35     mov %rdx, -24(%rbp)

36     mov %ecx, -32(%rbp)
37
38     movl $0, -28(%rbp)
39
40 .Lfor1:
41     cmpl $4, -28(%rbp)
42     jl .Lcorpo1
43     jmp .Lfine1
44
45 .Lcorpo1:
46     mov -24(%rbp), %rsi
47     movslq -28(%rbp), %rcx
48     mov (%rsi, %rcx), %al
49     movsbq %al, %rax
50     add %rcx, %rax
51     mov %al, -40(%rbp, %rcx)
52     incl -28(%rbp)
53     jmp .Lfor1
54
55 .Lfine1:
56     lea -96(%rbp), %rdi
57     lea -40(%rbp), %rsi
58     call _ZN2cl5elab1EPc3st1
59
60 #traduzione secondo ciclo for
61
62     mov -16(%rbp), %rdi
63     lea -96(%rbp), %rsi
64     mov $7, %rcx
65     rep movsq
66     leave
67     ret
68
69

```

Alcuni appunti di sintassi. È possibile dichiarare diverse etichette numeriche all'interno del campo `.text`, e chiedere, al momento di un salto, di andare all'etichetta alla prima etichetta di numero `x` che si incontra andando avanti o indietro. Questo è particolarmente utile nel momento in cui vogliamo dichiarare alcune etichette per la gestione di strutture auto-contenute, come i cicli for dell'esercizio precedente. Per saltare alla successiva etichetta `x`, si scrive `jmp xf`, per la precedente `jmp xb` (rispettivamente, forward e backward).

Supponiamo di aver eseguito il programma una volta compilato e di aver ottenuto un segmentation fault. Può essere utile sapere in che punto ho ottenuto tale risultato. Per farlo, innanzitutto si deve compilare con l'opzione `-g`, in modo che nell'eseguibile rimangano le informazioni che permettono a `gdb` di mostrare le istruzioni, poi di eseguire `ulimit -c unlimited`. Questo comando ci genera un file dal nome `core` che, dato a `gdb` assieme al file eseguibile, ci permette di risalire all'istruzione esatta dell'interruzione.

E se nell'esempio precedente avessimo avuto un costruttore di copia? Esso sarebbe stato dichiarato come `cl(const & cl)` e, al momento della copia che si effettua per tradurre il return della funzione `elab1`, si sarebbe dovuta utilizzare questo. Per farlo, come `this` inseriamo l'indirizzo dell'oggetto da costruire, e come parametro l'indirizzo dell'oggetto sulla base del quale effettuare la copia.

```
mov -8(%rbp), %rdi
```

```
lea -96(%rbp), %rsi  
call _ZN2clc1ERKS_
```

A questo punto, uscendo dalla funzione, non serve mettere l'indirizzo dell'oggetto costruito in `%rax`, avendoci già pensato il costruttore.

Supponiamo di avere, alla fine di una funzione, `return *this`. Cosa dobbiamo restituire? Dipende, ovviamente, dal tipo di dato restituito dalla funzione. In particolare, `*this` vuol dire prendere l'oggetto da esso puntato. Se si restituisce l'oggetto, allora si devono usare le convenzioni viste per scrivere i dati al posto giusto, se invece si restituisce un riferimento all'oggetto, basterà ritornare in `%rax` l'indirizzo del dato oggetto, e quindi il valore di `this`.

Sugli strumenti di sviluppo

Concludiamo la trattazione a riguardo della multiprogrammazione analizzando in maniera più dettagliata gli strumenti che portano dal file sorgente all'eseguibile. Quando in ingresso ho un file in formato .cpp, esso viene prima elaborato da un preprocessore, il cui risultato passa al compilatore producendo un file Assembler; quest'ultimo, al pari dei file Assembler che scriviamo direttamente come sergenti, viene passato ad un assemblatore che realizza un file oggetto; i file oggetto sono uniti dal collegatore, che realizza un eseguibile nel quale vi sono tutte le informazioni necessarie per essere eseguito dal caricatore.

Supponiamo di avere un file .cpp in cui si include un altro file nella stessa cartella, con qualche commento è una spaziatura discutibile. Il risultato del preprocessore si può vedere con il comando -E di g++. L'output è esattamente quanto vede il compilatore. In particolare, ha aggiunto una serie di informazioni all'inizio del file per dire quello che ha fatto, precedute da un cancelletto per indicare che debbono essere ignorate; dal posto della direttiva #include ha sostituito il contenuto del file corrispondente; ha rimosso i commenti; ha standardizzato la spaziatura. Un tempo il preprocessore era indipendente dal processore, e ciascuno eseguiva un parsing secondo i propri scopi. Tuttavia, col passaggio da C a C++, il parsing del sorgente è diventato più dispendioso, e ad oggi è il compilatore ad eseguire queste elaborazioni. Analizziamo alcune delle direttive del preprocessore. #include permette di copiare e incollare un file nel nostro: usando i doppi apici "", il file viene prima ricercato nella cartella del programma e, se non trovato, in una serie di cartelle standard; se si usano le parentesi angolari <>, si cerca direttamente nelle cartelle predefinite. Questa distinzione permette di creare file con lo stesso nome delle librerie standard, ma riuscendo ad importarli ugualmente. Il preprocessore si occupa anche di sostituire le macro, specificate con #define. Ogni volta che incontra la macro, essa viene sostituita con l'espressione associata. Il preprocessore non si interessa della sintassi, ma effettua solamente una modifica del file sorgente. È possibile fare delle inclusioni condizionali di codici, sulla base che sia definita o meno una certa macro.

```

1  #ifdef DEBUG
2      void debug(const char *);
3  #else
4      #define debug(msg)
5  #endif
6
7  int main(){
8      int a = 10;
9      debug("Questa è una funzione di debug");
10     return 0;
11 }
12
13

```

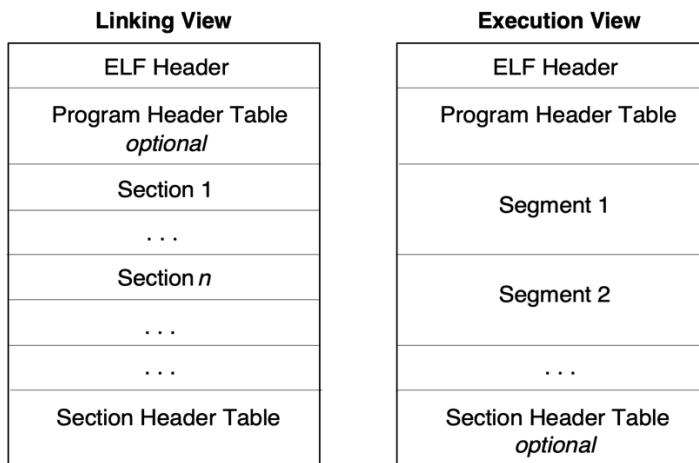
Se la macro DEBUG è definita (e si può fare anche al momento della compilazione, con il comando -D di g++), allora è lasciato nel codice la dichiarazione della funzione sottostante, altrimenti viene definita una macro con un parametro associata a nulla: tutte le volte che incontra la funzione debug con un parametro, la sostituisce con uno spazio vuoto, lasciando il codice senza alcun riferimento alle funzioni di debug.

Il compilatore lo abbiamo visto sostituendoci a lui nelle operazioni più basilari.

Lo scopo dell'assemblatore è quello di preparare il contenuto di sezioni di memoria come .text, .data e .bss (variabili globali che sono di default inizializzate con 0. Per tale motivo, non occupano spazio nel file oggetto, conoscendo già il loro valore). Concettualmente, l'assemblatore opera per due passate successive: nella prima, scorre tutto il file realizzando una tabella dei simboli. In particolare, per ciascuna sezione

del programma si tiene un contatore che incrementa ad ogni byte, in modo da avere l'offset, dall'inizio del programma, del punto che sta leggendo. Quando incontra un simbolo, aggiunge il suo identificativo ad una tabella assieme a tale offset, che quindi non fa riferimento all'intero codice ma solo alla sezione nel quale si trova. Nella seconda passata, si preoccupa di sostituire i byte, tenendo di conto della tavola appena creata. Le due passate servono per non avere problemi se ci si riferisce ad un'etichetta che deve ancora essere dichiarata nel codice, che altrimenti non conoscerei.

Il prodotto dell'assemblatore è un file oggetto, un file binario realizzato secondo il formato ELF, lo stesso dei file eseguibili e librerie dinamiche. A seconda che il file sia risultato dell'assemblatore o del collegatore, varia il suo contenuto, che si può visualizzare usando lo strumento `readelf`. Il primo elemento è sempre un'intestazione/heading, con delle caratteristiche sul nostro sistema e delle indicazioni sulla posizione di ciascun elemento, che si può vedere con il comando `-h` di `readelf`. Il file ELF comincia con un 'magic number' che garantisce che il file che abbiamo tra le mani sia effettivamente nel formato ELF, visto che nei sistemi Unix i file sono solo sequenze di byte, e il loro significato viene interpretato sulla base del programma che si usa.



Nel *linking view*, troviamo una tavola delle sezioni, sul fondo, che indica alcune caratteristiche di ciascuna sezione (quelle presenti nel file sorgente, quali `.text`, `.data...` ma anche altre necessarie al collegatore) e la posizione nel file. Nell'*execution view*, abbiamo invece una tavola dei segmenti, ossia le sezioni di programma che vengono effettivamente caricate in memoria e utilizzate in esecuzione, realizzate dal collegatore. Per ciascuna sezione nella *section header table* ho un nome, un tipo, lo spazio per l'indirizzo al quale cercarla, per ora vuoto (sarà riempito dal collegatore e lasciato nell'eseguibile, visto che la tavola delle sezioni è spesso presente anche nell'eseguibile) e l'offset della sezione rispetto al file oggetto in questione.

There are 8 section headers, starting at offset 0x190:								
Section Headers:								
[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk Inf Al
[0]		NULL	0000000000000000	000000	000000	00	0	0 0 0
[1]	.text	PROGBITS	0000000000000000	000040	000018	00	AX	0 0 1
[2]	.rela.text	RELA	0000000000000000	000128	000030	18	I	5 1 8
[3]	.data	PROGBITS	0000000000000000	000058	000010	00	WA	0 0 1
[4]	.bss	NOBITS	0000000000000000	000068	000000	00	WA	0 0 1
[5]	.symtab	SYMTAB	0000000000000000	000068	0000a8	18		6 6 8
[6]	.strtab	STRTAB	0000000000000000	000110	000015	00		0 0 1
[7]	.shstrtab	STRTAB	0000000000000000	000158	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)

Si possono cercare anche i simboli definiti dall'assemblatore: questi possono essere *Undefined* se non sono stati trovati, oppure avere un offset rispetto all'inizio della sezione.

Symbol table '.symtab' contains 8 entries:						
Num:	Value	Size	Type	Bind	Vis	Ndx Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND
1:	0000000000000000	0	SECTION	LOCAL	DEFAULT	1
2:	0000000000000000	0	SECTION	LOCAL	DEFAULT	3
3:	0000000000000000	0	SECTION	LOCAL	DEFAULT	4
4:	0000000000000000	0	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 foo

Sappiamo che il collegatore non è capace di inserire nelle istruzioni gli indirizzi associati ai simboli, e lascia questo compito al collegatore che, invece, ha una visione globale dei file che costituiranno il programma. Per fare ciò, l'assemblatore produce una *tabella di rilocazione*, in una sezione che prende il nome di .rela.text, in cui ci sono le informazioni per modificare il testo. Ad esempio “al byte xx, inseriscimi un dato a 64 bit corrispondente all'etichetta yyy una volta che questa avrà assunto un valore”.

Relocation section '.rela.text' at offset 0x128 contains 2 entries:						
Offset	Info	Type	Sym.	Value	Sym. Name + Addend	
000000000000b	00020000000b	R_X86_64_32S		0000000000000000	.data + 0	
0000000000012	000200000002	R_X86_64_PC32		0000000000000000	.data + 4	

Il collegatore prende in ingresso diversi file oggetto, ciascun di questi con, ad esempio, le sezioni .data e .text. Esso innanzitutto realizza un'unica sezione per ciascun tipo, mettendo di fila quelle dei diversi file ricevuti. A questo punto, conosce l'indirizzo esatto dell'inizio di ciascuna sezione, e, sulla base degli offset associati ai simboli, modificando quelli delle sezioni messe dopo la prima, può calcolare il valore di ciascun simbolo. Mentre nei file oggetto ci possono essere simboli non definiti, al momento del collegamento tutti devono essere definiti. Il passo finale è l'uso di una tabella di rilocazione complessiva (che tiene conto degli offset aggiornati) per modificare le singole istruzioni del programma.

```
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          0x000000 0x000000000400000 0x000000000400000 0x0000eb 0x0000eb R E 0x1000
LOAD          0x0000eb 0x0000000004010eb 0x0000000004010eb 0x000020 0x000020 RW 0x1000

Section to Segment mapping:
Segment Sections...
 00      .text
 01      .data
```

Il file eseguibile è a sua volta un file ELF, che contiene sia la tavola dei segmenti e, quasi sicuramente, anche la tavola delle sezioni, come lascito del lavoro del collegatore. Nella tavola dei segmenti ci sta scritto di caricare, a partire da un offset nel file, ad un certo indirizzo in memoria, un certo numero di byte successivi, di lasciare in bianco un certo numero di byte e di avere certi privilegi per tale segmento, quali R e RW.

Una libreria è un insieme di file oggetto già pronti e utilizzabili in programmi diversi. In Unix, non è altro che un archivio di file oggetto precedentemente ricavati dall'assemblatore. Essa risolve quanto meno il seguente problema. Supponiamo di avere tre file oggetto, boo.o, main.o e func.o. In main si usa una funzione definita in func.o, ma non quella definita in boo.o. Se dessi i tre file come input a g++, questi si

ritroverebbero in toto nell'eseguibile, senza che tuttavia una parte sia utilizzabile. Vedremo che con le librerie questo non accade. Per creare una libreria, si usa lo strumento `ar`, nella forma

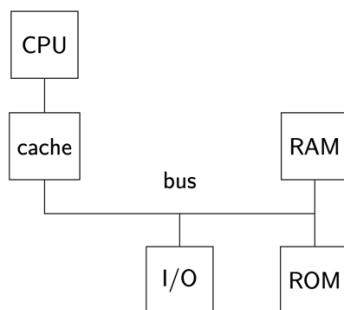
```
ar cr mialib.a file1.o ... filen.o
```

Una libreria con estensione `.a` è gestita da molti dei programmi già visti. Ad esempio, con `readelf` possiamo vedere il contenuto, secondo la codifica ELF, di tutti i file oggetto in essa contenuti. Quando al linker do in ingresso un file oggetto e una libreria, vi va dentro a cercare i nomi che, nella tabella dell'altro file, sono *Undefined*. Fatto ciò, include nell'eseguibile i file oggetto riferiti a tali nomi, escludendo tutti gli altri. Per usare una libreria, posso inserirla nella cartella del programma e usare il suo nome per includerla. Un'alternativa è quella di inserirla in una cartella in cui sono presenti alcune delle librerie standard. La cartella è `/usr/local/lib`, e il file deve cominciare con i caratteri `lib`. Se, ad esempio, vi inseriamo il file `libmiaLibreria.a`, al momento del collegamento devo usare il comando `-lmiaLibreria`. Il fatto che includa una libreria in questo modo non pregiudica che il compilatore richieda una dichiarazione di tutti i nomi che andremo ad utilizzare. Per questo motivo, associato ad un libreria ci deve sempre essere un header file, in formato `.h`, che dichiari quanto andremo effettivamente ad utilizzare. Visto che l'operato del collegatore è sequenziale rispetto ai file che gli passiamo, è necessario che la libreria sia importata per ultima. Infatti, quando la vede cerca i nomi che erano *Undefined* e importa i file oggetto associati, ma non torna indietro alla libreria nel caso questi siano trovati come tali dopo.

Sulla memoria cache

Risulta ormai chiaro che la RAM contiene il programma e i dati ad esso associati: finché il processore non preleva un'istruzione, non riesce a fare nulla. Allo stesso tempo, mentre un'operazione sui registri richiede un ciclo di clock, quelle delle RAM vanno sulle centinaia. In una situazione del genere, non avrebbe senso avere processori più veloci se il tempo di accesso in memoria resta così grande. Allo stesso tempo, dagli anni '70 fino al 2010 si è assistito ad un aumento esponenziale della velocità della CPU, mentre le memorie sono rimaste pressoché lente allo stesso modo, basandosi su tecnologie standard.

Ciononostante, nei nostri computer si osserva che volendo accedere ad un buffer di memoria si impiegano sui 4 cicli di clock per piccole dimensioni, intorno ai 10 fino ad un certo limite oltre il quale si va sui 200 cicli. Questo perché nei calcolatori moderni esistono delle memorie ausiliarie, le *memorie cache*, che rendono l'accesso in memoria efficiente sulle piccole dimensioni. Di base, ci sono varie tecnologie per realizzare una memoria. In generale, si riescono a produrre di veloci solo se piccole, ma allo stesso tempo costose; le memorie grandi si portano dietro un minor costo (sia monetario che energetico), a fronte di minor velocità di accesso. Le prime sono implementate secondo la tecnologia della RAM statica con *flip flop* per ciascun bit, le seconde con dei condensatori, la cui carica è talmente piccola che il tempo impiegato a leggere ciascun valore è abbondante. Tuttavia, una volta che si è effettuata una lettura, quella di altri byte risulta più veloce: questo principio sarà fondamentale per capire come opera la cache.

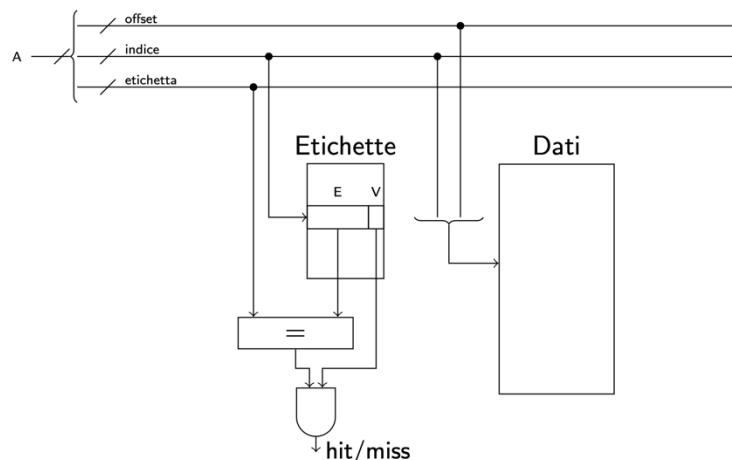


Le memorie cache permette degli accessi dell'ordine di 4 clock, ma hanno dimensioni di 128/256KiB. Nonostante la loro esistenza, il programmatore se ne può disinteressare, in quanto gestite totalmente lato hardware, col vantaggio di avere l'impressione di rapportarsi con una memoria grande e veloce. Quindi, la cache risulta trasparente non solo al programmatore, ma alla stessa CPU: essa esegue le operazioni di lettura e scrittura per come le abbiamo viste, poi se sono portate a termine in meno cicli di clock tanto meglio. Questo è anche il motivo per cui **la cache è gestita esclusivamente lato hardware**: mentre fa le sue cose, la CPU è impegnata ad aspettare la conclusione dell'operazione di lettura/scrittura. La cache ha un controllore che, ogni volta che il processore vuole effettuare una lettura in memoria, effettua l'operazione e salva il dato al suo interno, in modo che sia disponibile per una successiva richiesta. Al primo accesso sono dunque necessari 100 cicli di clock, al secondo 4. Ma ha senso copiarsi qualcosa che è già stato utilizzato? Si osserva sperimentalmente che, volenti o nolenti, i programmi seguono due principi di località:

- *Principio di località spaziale*: In breve tempo, è probabile che si richieda la lettura di dati vicini a quello immediatamente prelevato. Questo accade per come sono allocati in memoria dati e istruzioni, secondo un principio di sequenzialità.
- *Principio di località temporale*: In breve tempo, è probabile che si richieda nuovamente un dato già letto. È quanto accade per le istruzioni dentro un ciclo, o per un dato che viene letto/scritto più volte.

La RAM, come già detto, richiede molto tempo per leggere, ad esempio, 8 byte, ma letture immediate successive non sono altrettanto lente. Per questo motivo, è conveniente leggere 64 o 128 byte tutti insieme. Non solo: prelevando i byte successivi a quello richiesto è probabile che, per la località spaziale, ci troviamo proprio quelli che saranno richiesti nel breve periodo. A questo punto, non dovendo più dialogare direttamente con il processore, il bus può avere un numero di fili di dati maggiore di 8, per rendere tali operazioni più efficienti. Un accesso anticipato di questo tipo ha senso per le RAM, ma non per lo spazio di I/O, per la presenza di effetti collaterali. Il controllore della cache dovrà essere capace di distinguere l'accesso in memoria da quello per elementi legati all'I/O, rendere più efficiente il primo e lasciar passare il secondo.

Nel momento in cui voglio fare una lettura ad un dato indirizzo, il controllore della cache intercetta l'indirizzo. Se è già presente in cache bene, altrimenti va in memoria, identifica il numero della regione naturale di 64 byte (ad esempio) di quell'indirizzo e la preleva per portarla in cache. Questi 64 byte associati ad una regione naturale vengono detti **cache line**. La memoria cache, pur nella sua dimensione ristretta, sarà composta da locazioni di byte, ciascuna delle quali avrà un indirizzo x . Come faccio, dato l'indirizzo y proveniente dal processore (o meglio, numero della regione naturale di ampiezza 8 byte e i *byte enabler*) a capire se è già presente? Il controllore implementa una semplice tabella hash, in modo che dall'indirizzo si risalga, se c'è, alla cache line associata.



Dai bit provenienti dalla CPU, il controllore estrae un offset, la parte meno significativa, e il numero della cache line, dato da un'etichetta e un indice. L'hash implementata dal controllore si basa sull'indice, ovvero la parte meno significativa del numero di cache line. Tale valore viene usato per accedere ad una memoria detta **memoria delle etichette**. Per ciascun indice, si tiene traccia di un *bit di validità*, per sapere se i dati nella porzione di memoria associati a quel numero di cache line sono consistenti o meno, e dell'*etichetta precedentemente usata* per quella porzione. Si fa un confronto tra il valore contenuto e l'etichetta

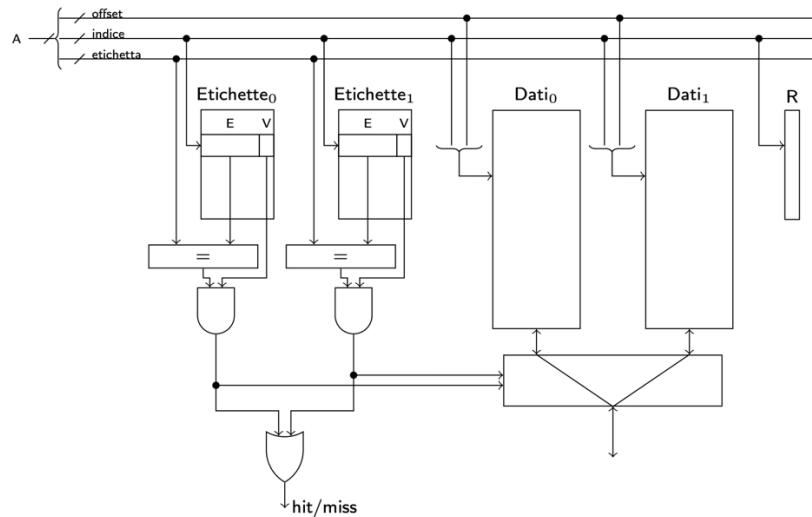
dell'indirizzo proveniente dalla CPU. Tale risultato si mette in and logico con il bit di validità. L'uscita vale 1, **hit**, se l'etichetta è la stessa ed è valida (potrebbe essere la stessa pur contenendo bit casuali), altrimenti vale 0, **miss**. Nel caso di una hit, si possono prelevare i dati dalla memoria cache, all'indirizzo interno dato da *indice+offset*. In pratica, memorizzo dei gruppi di 64 byte consecutivi, ciascuno dei quali ha un numero di cache associato. La parte bassa di questo numero, l'indice, dice qual è il primo indirizzo associato nella memoria cache (da cui ci spostiamo con offset). L'indice serve poi per accedere alla memoria delle etichette, dove vediamo anche qual è l'etichetta della cache line associata. Se è uguale a quella richiesta allora ho una hit, altrimenti devo andare in memoria, leggere la cache line interessata e inserirla in memoria cache al posto di quella, già memorizzata, con lo stesso indice. Questa situazione nasce banalmente dal fatto che le tabelle hash hanno dei conflitti sulle chiavi.

Se ipotizziamo la cache di 128KiB, possiamo dividere lo spazio di memoria in regioni naturali di 128KiB. Per ogni regione naturale non ho conflitti, perché il conflitto si ha con etichette diverse, e quindi nome della regione naturale di 128KiB. L'indice è usato per accedere alla cache line opportuna, l'offset per identificare gli 8 byte d'interesse. Per questo motivo, non solo accedere in locazioni vicine è vantaggioso, ma non crea neanche conflitti, e i dati permangono in memoria cache.

Cosa possiamo fare per quanto riguarda la scrittura? Abbiamo diverse possibilità, a seconda della situazione.

- Nel caso in cui ci sia una **miss**:
 - **write-allocate**: carico la cache line associata e vado nel caso di una hit.
 - **write-no-allocate**: il controllore della cache lascia passare la scrittura, che viene effettuata direttamente in memoria.
- Nel caso in cui si abbia una **hit**:
 - **write-through**: si modifica sia in cache che in RAM.
 - **write-back**: si modifica il valore solo in cache. Questa opzione fa sì che non ci sia allineamento tra RAM e cache. La strategia è quella di copiare la cache line in memoria solo quando si ha una miss su quell'indice, che comporterebbe una sovrascrittura dei dati. In questo modo, posso permettermi di modificare i dati più volte direttamente in cache, e fare poi una scrittura finale in memoria per allineare tutto insieme. In effetti, la strategia adottata dai controllori cache è quella della coppia write-allocate + write-back. Questo comporta l'aggiunta di un'informazione, detta **bit dirty**, dentro la memoria delle etichette, per essere consapevoli che, a seguito di una miss, si dovrà fare una scrittura in memoria.

Il punto debole di questa implementazione della memoria cache sono i conflitti: se anche il resto della cache non è utilizzata, con un conflitto sull'indice la cache line viene sostituita. Per fare di meglio si può pensare di avere più etichette per lo stesso indice, replicando il modello sopra esposto. Vediamo il caso di due sole memoria cache.



Ad ogni operazione in memoria, si tenta di accedere ad una delle due memorie cache, sperando che almeno una contenga l'etichetta corretta. I risultati dei due hit/miss vanno in *or* tra di loro, a dire che, alla fine, se almeno uno dei due fornisce un 1 allora l'etichetta di quell'indice è presente in memoria. Si usa poi un multiplexer per selezionare dalla memoria quale dei dati si deve trasmettere alla CPU. L'accesso alle cache avviene indipendentemente, secondo una modalità detta **associativa a insiemi a N vie** (in questo caso due vie). Questa costruzione si paga sia in termini economici che energetici. Ma è preferibile avere memoria cache più grandi o memorie cache con questo montaggio, a parità di dimensione complessiva? Conviene a tante vie, perché fornisce un'opportunità.

Supponiamo che si voglia accedere ad una locazione per cui, in entrambe le memorie cache, la porzione di quell'etichetta è occupata. È possibile scegliere quale delle due cancellare, in modo da ridurre le miss. Matematicamente, l'approccio migliore sarebbe quello di *cancellare la cache line dell'etichetta che sarà usata più in avanti nel futuro*, ma il controllore non ha la capacità di fare tale previsione. Si usa allora la politica dell'LRU (*last recently used*). Nel caso a due vie, è di immediata implementazione: dentro la memoria delle etichette inseriamo un bit che mi dice qual è stato l'ultimo accesso effettuato, in modo da sostituire l'altro. Per N vie le cose si complicano, in quanto si richiede uno storico delle etichette usate a parità di indice.

LRU è un'euristica, per cui non solo può essere implementata in maniera approssimativa, ma può anche portare a situazioni sgradevoli. Supponiamo di avere una cache a quattro vie. Ciascuna di queste contiene 2^k byte, quindi posso dividere la memoria di regioni naturali ampie 2^k , ciascuna identificata da un'etichetta. Si vuole accedere consecutivamente alle cache line con lo stesso indice di 5 etichette diverse, A, B, C, D ed E. All'inizio la cache è vuota, quindi per le i primi 4 accessi si hanno delle miss, caricando A, B, C e D. Alla lettura su E si ha una miss, e dovendo cancellare quella usata da meno tempo si toglie A. Questa però è quella da leggere immediatamente dopo, quindi carica nuovamente rimuovendo B. Il processo si evolve proprio in questo modo, con il risultato che si hanno solamente miss. Ovviamente abbiamo tra le mani un caso limite, che è poi migliorato utilizzando delle funzioni di prefetch da parte del processore: se la CPU osserva delle regolarità nel prelievo in memoria, allora si anticipa nel prelevare i dati successivi, in modo da averli disponibili se necessari. Questo fa comodo soprattutto per quanto riguarda le istruzioni.

Una strategia altrettanto usata è quella di rimpiazzare una cache line a caso, per non dover neanche gestire una struttura dati. Nelle memorie cache a 4 vie, c'è la possibilità di implementare un'approssimazione

della LRU usando solo 3 bit per ciascun indice. In figura si vede una memoria R. I tre bit di ciascuna riga mi dicono, nel caso ci sia una miss e tutte le vie siano piene, quale andare a rimpiazzare. In particolare:

- b_0 mi dice se rimpiazzare uno tra 1 e 2 oppure tra 3 e 4;
- b_1 mi dice se rimpiazzare 1 o 2 nel caso in cui b_0 valeva 0;
- b_2 mi dice se rimpiazzare 3 o 4 nel caso in cui b_1 valeva 1;

Avendo memorizzato 010, il primo bit mi dice di andare a sinistra dell’ipotetico albero, quindi di osservare b_1 , il secondo che devo rimuovere la cache line dalla seconda via. Ovviamente, è necessario effettuare un aggiornamento per ogni accesso all’indice in questione. L’aggiornamento si fa in modo che si sposti il duo a quale rifarsi la volta successiva e che, rispetto al duo a cui appartiene la via a cui si è fatto l’accesso, si inverta l’ordine. Facciamo l’accesso alla terza via. b_0 deve diventare 0, per dire che l’elemento da cancellare è quello tra 1 o 2; b_1 assume lo stesso valore, mantenendo inalterato l’ordine tra 1 e 2. b_2 diventa 1, a dire che tra i due è stato usato meno di recente il quarto. Osserviamo che tutto è possibile senza neanche effettuare la lettura del valore precedente, basta sovrascrivere i dati. Si può allora implementare in modo efficiente se ciascuno dei tre bit di una riga di R può essere scritto autonomamente dagli altri. Poniamo di leggere A. Il valore da sovrascrivere è $11b_2$. Da un semplice esempio capiamo che, per come si modifica b_0 , questa è solo un’approssimazione di LRU. Supponiamo che l’ordine degli accessi fosse BACD. Effettuano una lettura su C, si otterrebbe CBAD. Con l’implementazione vista invece, il risultato è CDBA: l’ordine relativo tra A e B rimane inalterato, ma D acquista posizione per il solo fatto che l’accesso riguardava il suo compagno.

Con questo si esaurisce quanto c’è da dire sulla memoria cache. Tuttavia, c’è ancora un punto in sospeso da chiarire: come fa la memoria cache a capire se la richiesta di accesso riguarda la RAM o una periferica? In questo caso, infatti, una lettura di gruppo come quella della cache farebbe danni, e la richiesta deve essere invece lasciata passare.

Alcune note sulle dimensioni. Supponiamo di avere indirizzi a 64 bit, che una cache line sia ampia 64 byte e che la memoria cache sia di 128 KiB. Allora:

- **A**, proveniente dalla CPU, è formato da 61 bit ($64 - 3$, per accedere alle regioni naturali di 8 byte e selezionare i byte con i byte enabler).
- L’**offset** mi deve permettere di scegliere quale gruppo di 8 byte voglio nei 64 disponibili. 64 byte sono 8 gruppi da 8 byte, quindi mi servono 3 bit.
- L’**indice** mi permette di accedere a regioni di 128 KiB. Vi sono 2^{17} byte, da cui 2^{14} gruppi di 8 byte. Si tolgono i 3 bit per l’offset, e ne rimangono 11.
- L’**etichetta** mi indica quante regioni naturali ampie 128 KiB posso prendere dalla memoria. Sono $2^{64} / 2^{17} = 2^{47}$, quindi mi servono 47 bit.

Per un primo accesso a basso livello

All'interno di un calcolatore, è il software a comandare, dicendo al processore cosa e come fare cosa. Tuttavia, se proviamo ad eseguire un programma con un loop infinito, la CPU non si dedica solamente alla sua esecuzione, ma ci permette di utilizzare normalmente il nostro calcolatore per fare tutto il resto. Sono tanti i casi in cui qualcuno ci impedisce di fare determinate operazioni. Per esempio, non abbiamo pieno controllo sull'I/O. All'indirizzo 0x60 si trova il registro RBR della tastiera, in cui è presente l'ultimo carattere letto. Se provo a fare una `in` da questo indirizzo, ricevo un *segmentation fault*. La stessa cosa succede se provo ad accedere ad un indirizzo in memoria che non è di mia competenza o in sola lettura, come le istruzioni. Chi è che ci impedisce di fare queste operazioni? Intuitivamente ci verrebbe da dire "il sistema operativo". Ma mentre eseguiamo `in 0x60, %al`, la CPU stava effettivamente pensando al mio programma, con un registro `%rip` impiegato a puntare all'istruzione successiva rispetto a quella che stava eseguendo. Ciò significa che queste limitazioni sono realizzate in hardware. In particolare, il sistema operativo l'ha programmato affinché la CPU ci impedisce di fare determinate cose. C'è quindi un'altra porzione di hardware che esegue i controlli. Come vedremo, abbiamo tre protagonisti in questo processo: le **interruzioni** ci consentono di fare altro mentre il nostro programma è in esecuzione; la **protezione** ci impedisce di usare certe istruzioni, come `in` e `out`; la **memoria virtuale** ci impedisce di accedere alla memoria liberamente.

Il kernel è il primo programma ad essere lanciato in memoria, ed è padrone di tutto. Si occupa di programmare interruzioni, protezione e memoria virtuale in modo che tutto sia poi limitato da ciò. Affinché si possano avere gli stessi privilegi del kernel, il nostro programma deve essere caricato prima ancora di questo. Farlo quando il kernel è già in esecuzione è piuttosto difficile, per questo useremo una macchina virtuale *from scratch*, che si occuperà di caricare solo il programma che abbiamo scritto noi, e con il quale potremmo fare quello che vogliamo, non avendo limitazioni. Il software per fare questo è `qemu`, che ha preinstallato un bootloader che si occupa di caricare nella VM il nostro file di partenza (il quale non potrà usare, per esempio, le librerie standard di C++, che fanno uso del kernel). Molte delle operazioni che useremo sono presenti nella libreria `<libce.h>`. Scriviamo un primo programma.

Per stampare qualcosa, si usa la funzione `printf` presente nella libreria, che si comporta come quella di `stdlib.h`, con la stessa formattazione degli elementi (%d per i numeri in decimale, %x per i numeri in esadecimale, %s per le stringhe...). In `libce` c'è anche la definizione di `start`, che chiama il `main` che abbiamo definito.

```
#include <libce.h>

int main(){
    char* name = "Giacomo";
    printf("Hello world, %s", name);
}
```

Si usa il comando `compile` per compilare tutto quello che è presente nella cartella corrente. Il comando `boot` lancia la macchina virtuale. Questa si spegne nel momento in cui si conclude il `main`, quindi è bene inserire alla fine un'istruzione `pause`, anch'essa nella libreria, per mostrare tutto il contenuto. La finestra che apre `qemu` si deve considerare come l'effettivo monitor della macchina virtuale così creata. In questo modo è possibile fare tutto quello che prima ci era precluso: accedere ad ogni locazione e modificarla liberamente; usare istruzioni di input e output, modificare il codice stesso che viene eseguito...

`compile` produce un file in formato `elf`, che `boot` andrà a caricare. Con `objdump` si può visualizzare l'indirizzo dei byte che costituiscono le istruzioni, e tramite i puntatori di C++ si possono andare a modificare.

Poiché l'esecuzione della macchina virtuale non può essere modificata manualmente (non esistono le interruzioni), è bene usare un Debugger. Per farlo, si usa compile con il comando `-g`, e si lancia il programma. In un'altra finestra di terminale, si lancia il Debugger, con il comando `debug`. Si apre in questo modo `gdb`, che funziona come visto per Assembler, ma con riferimento al programma caricato sulla VM.

Sulle periferiche

Adesso che abbiamo pieno accesso all'hardware, possiamo capire come funzioni il software per le periferiche. In particolare, vedremo tastiera, scheda video, timer e hard disk. Con riferimento all'immagine stilizzata del bus che connette CPU, RAM, ROM e I/O, si tratta di capire come funziona tale spazio, composto da 2^{16} byte (e quindi indirizzabile con 16 bit di indirizzo).

A connettere periferica e CPU c'è sempre un'interfaccia, che rende standard il collegamento tra il processore e periferica, senza che debba tenere un comportamento ad hoc per ciascuna di queste e per ciascuno dei vari modelli che potrebbero essere sviluppati. Noi vedremo le interfacce e le periferiche per come erano state realizzate negli anni 80/90; infatti, oltre ad esserci retrocompatibilità, quelle moderne risultano molto più complesse.

L'interfaccia per la tastiera ha quattro registri, due di lettura e due di scrittura, ma un solo piedino di indirizzo. Di suo quindi, l'accesso tramite indirizzo non consente di avere subito a disposizione tutti i registri; tale situazione si risolve sfruttando gli ingressi di /ior e /iow, tipici di ciascuna interfaccia. L'interfaccia è fisicamente collegata alla tastiera con un cavo seriale. La tastiera non invia alcuna codifica ascii, ma un *codice di scansione* associato univocamente a ciascun tasto, spesso scelto sulla base della sua posizione fisica. Ciascun tasto ha in particolare due codici, *make code* e *break code*, il primo inviato quando il tasto viene premuto, l'altro quando viene rilasciato. Questo è particolarmente utile per tutti quei tasti 'modificatori' che permettono le combinazioni. Ad esempio, per avere la 'A', è necessario premere il tasto 'a' mentre uno shift è premuto, quindi a cavallo tra *make code* e *break code* di uno shift. Il codice è inviato all'interfaccia da un microcontrollore presente nella tastiera, che si occupa di verificare migliaia di volte al secondo se ciascun tasto è premuto. L'interfaccia inserisce i caratteri ricevuti all'interno di un buffer, e l'ultimo non ancora prelevato viene inserito dentro il registro RBR. Logicamente, se il buffer si riempie non si possono ricevere più caratteri; nelle tastiere di un tempo, si emetteva un segnale acustico. Ovviamente, il software non può fare altro che prelevare dati da RBR, ma non vi trova alcuna informazione sul fatto che il valore sia corretto o meno: leggendo due volte 'a', non sa se quel valore è presente perché ho premuto due volte 'a' o perché non ci sono stati aggiornamenti. Per questo si usa un altro registro, STR: il programmatore deve prima leggere STR, che avrà un bit ad 1 nel caso il cui RBR sia valido, a 0 altrimenti. Dentro RBR troviamo solamente i codici di scansione dei tasti: deve essere il software ad associarli ai corrispondenti caratteri ascii, così come a controllare se fosse premuto pure uno *shift*, *alt*...

Vediamo qualche esempio di uso di questa interfaccia. Per prima cosa, prendiamo da RBR un valore nel momento in cui è valido e mostriamo la codifica binaria del codice di scansione.

```

1  #include <libce.h>
2
3  const ioaddr iSTR = 0x64;
4  const ioaddr iRBR = 0x60;
5
6  natb get_code(){
7      natb c;
8      do
9          c = inputb(iSTR);
10     while (!(c & 0x01));
11     return inputb(iRBR);
12 }
13
14 int main(){
15     natb c;
16     for (;;) {
17         c = get_code();
18         if (c == 0x01) break; // make code di ESC
19         for (int i = 0; i < 8; i++) {
20             if (c & 0x80) char_write('1'); // bit piu' significativo di c
21             else char_write('0');
22             c <<= 1;
23         }
24         char_write('\n');
25     }
26 }
```

inputb esegue l'istruzione in ad un indirizzo di 16 byte. ioaddr, infatti, è un tipo definito in libce che ha la dimensione di una word. Continuamente leggiamo un valore da RBR se valido e ne stampiamo la codifica. La cosa interessante è che, essendo l'unico programma eseguito dalla CPU, sappiamo esattamente cosa stia facendo in ogni istante. Il *break code* è uguale al *make code* con il bit più significativo ad 1. Se tengo premuto un tasto, dopo un brevissimo tempo comincio a ricevere moltissimi suoi *make code*. Questa funzionalità non può essere realizzata dal processore, quindi deve essere necessariamente gestita dal microcontrollore della tastiera.

L'interfaccia della tastiera ha anche dei registri per la scrittura: servono, ad esempio, ad impostare i led, o a modificare alcune caratteristiche della stessa (il tempo dopo il quale, se un carattere è tenuto premuto, viene inviato all'interfaccia).

Realizziamo adesso un programma che stampi a video il carattere premuto. Gestiamo inoltre lo shift le maiuscole.

```

1 #include<libce.h>
2 namespace kbd {
3 const natl MAX_CODE = 29;
4 bool shift = false;
5 natb tab[MAX_CODE] = // tasti lettere (26), spazio, enter, esc
6 { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
7 0x1E, 0x1F, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26,
8 0x2C, 0x2D, 0x2E, 0x2F, 0x30, 0x31, 0x32, 0x39, 0x1C, 0x01 };
9 natb tabmin[MAX_CODE] =
10 { 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p',
11  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l',
12  'z', 'x', 'c', 'v', 'b', 'n', 'm', ' ', '\n', 0x1B };
13 natb tabmai[MAX_CODE] =
14 { 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P',
15  'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L',
16  'Z', 'X', 'C', 'V', 'B', 'N', 'M', ' ', '\r', 0x1B };
17 }
18
19 using namespace kbd;
20
21 char conv(natb c){
22  natb cc;
23  natl pos = 0;
24  while (pos < MAX_CODE && tab[pos] != c) pos++;
25  if (pos == MAX_CODE) return 0;
26  if (shift) cc = tabmai[pos];
27  else cc = tabmin[pos];
28  return cc;
29 }
30
31 char char_read() {
32  natb c;
33  char a;
34  do {
35   c = get_code();
36   if (c == 0x2A) shift = true; // left shift make code
37   else if (c == 0xAA) shift = false; // left shift break code
38  } while (c >= 0x80 || c == 0x2A); // make code;
39  a = conv(c); // conv() puo' restituire 0
40  return a; // 0 se tasto non riconosciuto
41 }
42
43 int main(){
44  char c;
45  for (;;) {
46   c = char_read(); // puo' restituire 0
47   if (c == 0x1B) // carattere ASCII esc
48    break;
49   char_write(c); // non effettua azioni se c vale 0
50  }
51  return 0;
52 }

```

Abbiamo bisogno di un array nel quale teniamo salvati i codici di scansione dei caratteri che ci servono. Usiamo la `get_code()` vista sopra per prelevare un codice di scansione. Se è quello di pressione di uno shift, mettiamo la variabile globale `a` a `true`, se è di rilascio a `false`. Dopo, dobbiamo eventualmente stampare qualcosa solo se è un codice di pressione (ignoriamo break code, che hanno MSB ad uno). Si scorre l'array `tab`. Se troviamo il codice associato, entriamo in `tabmin` o `tabmai` per prendere il carattere, e usiamo la funzione `char_write` per stampare a video.

Il cuore della scheda video, nella sua versione più semplificata, è una RAM, nel quale la CPU scrive l'immagine che vuole andare a visualizzare sullo schermo. Ci sono poi dei circuiti che, continuamente, leggono il contenuto di questa memoria, lo decodificano secondo opportune modalità, e lo inviano al monitor che riesce a mostrare esattamente quanto richiesto. Poiché la scrittura su questa RAM deve essere continua-tiva, conviene inserirla nello spazio di memoria, ad un indirizzo specifico. Sapendo l'indirizzo a partire dal quale è posizionata, possiamo modificarne il contenuto. La RAM della scheda video deve essere anche realizzata in modo speciale, perché continuamente è scritta dal software e letta dall'adattatore.

Ci sono due modalità diverse per l'uso della scheda video, *testo* e *grafica*. Nella modalità grafica, è necessario specificare il colore di ogni singola locazione possibile. La modalità testo è quella più facile e immediata, e prevede che la RAM sia visualizzata come una matrice; scrivendo un carattere nella locazione $[0, 0]$ di questa matrice, il carattere viene mostrato in alto a sinistra dello schermo. In particolare, ogni locazione è composta da due byte: il meno significativo contiene la codifica ascii del carattere, l'altro le informazioni che hanno a che fare con il colore della cella sullo schermo. Degli 8 bit:

- Il primo determina se il carattere sarà lampeggiante o meno (non supportato da qemu).
- Il successivo determina l'intensità del colore del carattere, se alta o bassa.
- I tre successivi, il colore del carattere, fondendo rosso, verde e blu.
- I tre successivi, il colore dello sfondo, allo stesso modo.

Nei calcolatori IBM, per default all'avvio si usa la modalità testo, 80x25. Avendo queste informazioni, possiamo scrivere un programma che riempia la RAM della scheda video, in modo da visualizzare il contenuto a schermo. La riempiremo di caratteri 'a'.

```

1 #include <libce.h>
2
3 namespace vid {
4 natw* video = reinterpret_cast<natw*>(0xb8000); // array di 2000 word;
5 }
6 using namespace vid;
7
8 int main(){
9     for(int i = 0; i < 2000; i++)
10         video[i] = 0x4B00 | 'a';    // sfondo rosso (0100), simbolo azzurro chiaro (1011)
11     for (;;) {
12         char c = char_read();
13         if (c == 0x1B) break;    // carattere ASCII esc
14     }
15     return 0;
16 }
```

video è un puntatore alla prima cella della RAM della scheda video, e lo creiamo convertendo a puntatore a word il suo indirizzo. L'uso del puntatore alla stregua di un array consente di modificare locazione per locazione. La dimensione di default consente di gestire 2000 caratteri.

A questo punto, l'idea per usare lo schermo e scrivere tutto quanto è simile a quello della macchina da scrivere. Abbiamo una posizione corrente nella matrice 80x25. Quando premiamo un carattere lo scriviamo in tale posizione e incrementiamo il cursore. Quando siamo arrivati alla fine della riga, vado a quella successiva. Riempita l'intera pagina, mettiamo la seconda riga al posto della prima, la terza al posto della seconda... e lasciamo l'ultima vuota, per dare l'effetto di scrolling. La gestione della posizione del cursore lo lasciamo alla scheda video. Essa, infatti, ha moltissimi registri che mantengono le informazioni più disparate. L'accesso a tali registri si fa con due registri accessibili, IND e DAT: nel primo si specifica l'indirizzo del registro al quale vogliamo accedere, con il secondo leggiamo/scriviamo il valore corrispondente. Nel nostro caso, il cursore è gestito tramite la parte alta e la parte basse della posizione della cella presente, come si vede nella funzione cursore.

```

1 #include "libce.h"
2
3 namespace vid {
4
5 const ioaddr IND = 0x03D4;
6 const ioaddr DAT = 0x03D5;
7 const natl COLS = 80;
8 const natl ROWS = 25;
9 const natl VIDEO_SIZE = COLS*ROWS;
10 const natb CUR_HIGH = 0x0e;
11 const natb CUR_LOW = 0x0f;
12
13 natw attr;           // attributo colore
14 natb x, y;          // coordinate x e y inizialmente a 0
15 natw* video = reinterpret_cast<natw>((0xb8000));
16 // array di 80*25 = 2000 posizioni, ciascuna di una parola
17 }
18
19 using namespace vid;
20
21 void cursore(){
22     natw pos = COLS * y + x;
23     outputb(CUR_HIGH, IND);
24     outputb(pos >> 0x8, DAT);
25     outputb(CUR_LOW, IND);
26     outputb(pos, DAT);
27 }
28
29 void clear_screen(natb col){
30     attr = static_cast<natw>(col) << 8;
31     for (int i = 0; i < VIDEO_SIZE; i++) video[i] = attr | ' ';
32     x = 0;
33     y = 0;
34     cursore();
35 }
36
37 void scroll(){
38     for (int i = 0; i < VIDEO_SIZE - COLS; i++)
39         video[i] = video[i + COLS];
40     for (int i = 0; i < COLS; i++)
41         video[VIDEO_SIZE - COLS + i] = attr | ' ';
42     y--;
43 }
44
45 void char_write(natb c){
46     switch (c) {
47     case 0:
48         break;
49     case '\n': case '\r':
50         x = 0;
51         y++;
52         if (y >= ROWS) scroll();
53         break;
54     default:
55         video[y * COLS + x] = attr | c;
56         x++;
57         if (x >= COLS) {
58             x = 0;
59             y++;
60         }
61         if (y >= ROWS) scroll();
62         break;
63     }
64     cursore();
65 }
66
67 int main(){
68     natb c;
69     clear_screen(0x4B);
70     for (;;) {
71         c = char_read();
72         if (c == 0x1B) break; // carattere ASCII esc
73         char_write(c);
74     }
75 }
76 }
77

```

Nella modalità grafica, la RAM va interpretata come una matrice di pixel, ciascuno dei quali indica il colore di un punto sullo schermo. Per quanto concettualmente sia immediato, il costo per il processore è altissimo: ha senso quindi che la scheda video abbia un processore a sé stante, la GPU, che si occupi di riempire la RAM secondo le indicazioni della CPU. L'uso della scheda video ha anche formalizzato i connettori, secondo lo standard VGA e S-VGA. Accedere alla modalità grafica è abbastanza ostico; noi siamo avvantaggiati utilizzando un emulatore, per cui basta invocare una funzione presente in `libce`. Essa restituisce anche l'indirizzo al quale è allocata la RAM dove scrivere per questa modalità. Vediamo un esempio.

```

1 // file svgaini.cpp
2 #include <libce.h>
3
4 const natl COLS = 1280;
5 const natl ROWS = 1024;
6
7 natb* framebuffer;
8
9 int main(){
10     framebuffer = bochs_vga_config(COLS, ROWS);
11
12     for (int i=0; i<COLS; i++)
13         for (int j=0; j<ROWS; j++)
14             framebuffer[j*COLS + i] = 0x04;      // rosso scuro (vga 0100)
15     char c;
16     do
17         c = char_read();
18     while (c != 0x1B);                  // carattere ASCII esc
19     return 0;
20 }

```

A questo punto, l'idea per riempire lo schermo è piuttosto semplice, e si basa sull'assegnare in modo opportuno i singoli byte (ogni pixel è rappresentato da un byte, nel quale vi sono codificate le impostazioni per il colore).

Il timer è un elemento di conteggio, nella sua versione più astratta. Esso deve poter essere inizializzato via software ad un certo valore, e decrementato via hardware sulla base di impulsi. È opportuno che, nel momento in cui il contatore arriva a 0, si riceva un segnale. Nel computer IBM, il chip che realizzava questa funzionalità era l'[8253](#), funzionalità che adesso è integrata in un chip molto più ampio. Al suo interno, si trovavano tre contatori e un registro di controllo, che consentiva di anticipare le azioni da fare e impostare determinate modalità di funzionamento. Ciascun contatore, era gestito tramite 4 registri, da 8 bit: due, accessibili in sola scrittura, rappresentavano il contatore vero e proprio (parte alta/parte basse), gli altri due servivano per allacciare il valore del contatore prima di essere letto. Infatti, il contatore varia molto spesso, e deve essere allacciato affinché la lettura non sia influenzata da questa dinamica.

Ogni contatore aveva tre piedini dedicati, come si vede in figura: OUT, di uscita, per comunicare con l'esterno, CLK, un segnale di decremento in ingresso, e GATE, anch'esso in ingresso, che a seconda della modalità di utilizzo poteva avere vari significati. Le modalità erano 6. Sulla base della modalità, GATE funziona da trigger, dicendo al contatore quando partire (per esempio, decremente se GATE vale 1, altrimenti sta in pausa). Un problema è che ci sono solo due piedini di indirizzo, A0 e A1, ma i registri a cui dovremmo poter accedere sono 13 (quello di controllo + 4 per ciascuno dei diversi contatori). L'approccio è quello di distinguere l'accesso ai registri di lettura/scrittura sulla base del comando di lettura o scrittura. In questo modo ci siamo ricondotti a due indirizzi per ciascun contatore (7 indirizzi). Poi, si organizza che, sia durante la lettura

che durante la scrittura, si fanno sempre due accessi, uno che riguarda la parte bassa, uno che riguarda la parte alta. Il chip gestisce correttamente il processo, riconoscendo quale è il primo o il secondo accesso. In questo modo, sono sufficienti quattro indirizzi, uno per contatore + il registro di comando, che giustifica i due soli piedini.

All'interno del computer IBM, il contatore 0 era collegato al controllore delle interruzioni, il contatore 1 al refresh della DRAM (come lo schermo di SSEM, la memoria fatta di condensatori doveva essere refreshata molto spesso, e questo contatore inviava un segnale per stabilire quando fare tale operazione; adesso il comando è già presente nel chip della RAM) e il contatore 2 ad uno speaker, l'unico dispositivo audio del computer. In una delle modalità del contatore, era possibile generare un'onda quadra per metà del conteggio dal valore iniziale allo 0, eseguendo l'operazione ciclicamente. In particolare, tale valore determina la frequenza dell'onda, e quindi la nota in uscita. Per eseguire tale operazione, usiamo il terzo contatore e un chip per la gestione dello speaker.

D7	1	24	VCC
D6	2	23	-WR
D5	3	22	-RD
D4	4	21	-CS
D3	5	20	A1
D2	6	19	A0
D1	7	18	CLK 2
D0	8	17	OUT 2
CLK 0	9	16	GATE 2
OUT 0	10	15	CLK 1
GATE 0	11	14	GATE 1
GND	12	13	OUT 1

che riguarda la parte bassa, uno che riguarda la parte alta. Il chip gestisce correttamente il processo, riconoscendo quale è il primo o il secondo accesso. In questo modo, sono sufficienti quattro indirizzi, uno per contatore + il registro di comando, che giustifica i due soli piedini.

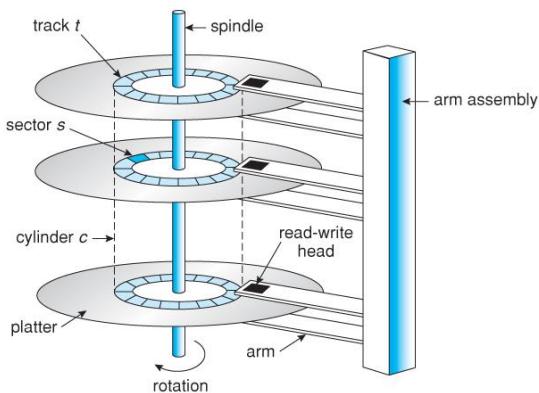
```

1 #include <libce.h>
2
3 const ioaddr iSPR = 0x61;
4 const ioaddr iCWR = 0x43;
5 const ioaddr iCTR2_LSB = 0x42;
6 const ioaddr iCTR2_MSB = 0x42;
7
8 void ini_cont(){
9     natw N = 1190;           // costante per 1000 Hz
10    outputb(0xB6, iCWR);    // contatore 2, modo 3
11    outputb(N, iCTR2_LSB);
12    outputb(N >> 8, iCTR2_MSB);
13 }
14 int main(){
15     ini_cont();
16     outputb(3, iSPR);
17     pause();
18     outputb(0, iSPR);
19     return 0;
20 }
```

`iSPR` è l'indirizzo del registro per la gestione dello speaker. Affinché questo funzioni, si devono impostare il valore a 3. La funzione `ini_cont` si occupa invece di impostare i valori corretti nel timer. Per prima cosa accede a `iCWR`, il registro per il controllo del timer, e ci scrive `10110110`: `10` indica l'accesso al contatore 2, `011` indica la modalità per la generazione dell'onda quadra. Dopo di che, si deve impostare il valore della frequenza, scrivendo prima nella parte bassa dell'indirizzo collegato al registro del contatore 2, poi in quella alta (`iCTR2_LSB` e `iCTR2_MSB` hanno lo stesso indirizzo, proprio per quanto visto prima). Il valore da scrivere nel contatore è legato alla nota che vogliamo sentire, ma anche al clock del computer, essendo il segnale che, collegato a `CLK`, decremente il contatore. Maggiori informazioni sulla pagina Wikipedia, rispetto alle varie modalità e al significato dei singoli bit.

L'hard disk è un dispositivo a blocchi: l'unità di trasferimento è un bocco, storicamente di 512 byte, non un singolo byte, come avviene nelle memorie RAM. Nonostante il pensiero comune, essi non sono indispensabili per il corretto funzionamento del computer: è vero che contengono i programmi, ma la CPU non può lavorare direttamente con un HD, per cui si ha sempre un passaggio sulla RAM. Nell'uso di programmi, la velocità di un HD impatta solo in fase di caricamento in RAM (e nel caso in cui il programma non possa stare completamente in memoria, caso per cui si inserisce una parte per volta secondo necessità, procedura che prende il nome di *swap*). Ciascun blocco ha un identificativo univoco: le interfacce ricevono tale identificativo dalla CPU, e si impegnano a caricare/scrivere nel dato blocco.

Gli HD sono implementati con un ampio apporto meccanico. Ci sono alcuni dischi uno sopra l'altro, impennati in un asse e tenuti in rotazione. Ciascuna faccia dei dischi contiene i dati, che sono accessibili tramite delle testine. Ciascuna faccia è organizzata per tracce concentriche (`t` in figura), divise in settori (`s` in figura). Quindi, per accedere ad un determinato settore mi basta sapere la faccia, la traccia e il numero del settore. Questi elementi in principio erano tutti necessari per poter accedere univocamente ad un blocco: ad oggi, per evitare gestioni inutili, si preferisce assegnare univocamente un *id* a ciascuna blocco, senza preoccuparsi della posizione fisica. Si tratta quindi di dispositivi ad accesso casuale, poiché non era necessario leggere tutti gli elementi precedenti prima di arrivare a quello d'interesse. L'informazione è memorizzata con dei campi magnetici che la testina riesce a rilevare.



Meccanicamente, una volta che conosco la traccia, è necessario che la testina vi si sposti sopra. Questo tempo è detto **seek**, e dipende ovviamente dalla posizione precedente della testina. Per poter fare un'operazione, il settore deve passare sotto la testina. Questo tempo viene detto **rotational latency**, e dipende da dove si trova rispetto alla testina, se subito prima o subito dopo (nel qual caso deve fare tutto il giro). Pur sapendo che questi tempi sono relativi alla posizione di ciascun elemento, in media il seek sta sui 5-10 ms, mentre la latency sui 7-8 ms. Questi sono tempi enormi in confronto alla CPU, che si aggira sui 4 GHz. Il problema sta nel fatto che un HD è un componente meccanico che non può andare oltre una certa velocità, mentre la CPU è elettronica. L'ultimo fattore che influenza è il tempo di accesso è il **read/write**, quanto ci impiega la sezione a passare sotto la testina. Tale tempo diminuisce aumentando la densità delle informazioni, affinché le sezioni siano sempre più piccole. Ciò è possibile avvicinando la testina al piatto.

Secondo questo schema le sezioni esterne sono più grandi di quelle interne, ma contengono la stessa informazioni, ed è uno spreco. Si aggiunge allora la suddivisione per zone, in cui si cerca di avere sezioni della stessa dimensione. Come conseguenza, una *read/write* in un blocco esterno necessiterà di un tempo maggiore rispetto ad uno interno, a causa di una dimensione maggiore a parità di velocità angolare. Il problema sta nel tempo di seek/latency, che però si riduce radicalmente se faccio accessi sequenziali e non casuali. In particolare, anche in computer moderni, un accesso ad elementi ravvicinati porta ad una velocità di 200 MiB/s, mentre volendo accedere qua e là si scende a 2 MiB/s.

A fianco a queste tecnologie, si stanno sviluppando le SSD, unità a stato solido, che non si basano su accessi meccanici ma su fenomeni elettronici. Nonostante siano ancora molto più lenti delle CPU, non hanno alcun tempo di seek e latency. Anch'esse lavorano per blocchi.

Si tratta ora di studiare come funziona l'interfaccia per il loro utilizzo. Lo standard di riferimento è l'ATA. In un sistema PATA, i dati venivano trasmessi con cavi paralleli 8 bit alla volta, mentre in SATA con cavi seriali. Nonostante la differenza strutturale, alla lunga una trasmissione seriale risulta più veloce, perché si compensa aumentando moltissimo la frequenza di trasmissione, cosa che non accade nel caso parallelo. All'interno di qemu, ci rapporteremo con un hard disk virtualizzato come se fosse connesso con un cavo parallelo. In particolare, ad un cavo di questo tipo era possibile collegare due unità, *master* e *slave*, specificando tramite un apposito bit quale delle due richiediamo.

L'interfaccia è realizzata in modo che si possano specificare le coordinate di ciascun settore come visto sopra. In particolare, abbiamo:

- CNH e CNL: cylinder number, permettono di specificare la traccia su cui andare. Si parla di cilindro perché le testine si spostano tutte insieme, quindi senza dover spostare la testina si accede alla stessa traccia anche su facce diverse.
- SNR: sector number, mi dice quale settore prendere.
- SNC: sector count, rende possibili operazioni anche su più blocchi alla volta, specificando quello di partenza e andando ad incrementare.
- HND: i 4 bit più significativi indicano il numero di faccia, gli altri sono di comando (ad esempio, servono per scegliere tra master e slave).
- BR è un registro di buffer a 16 bit.
- STS è un registro di stato, mi permette di sapere quando il blocco è stato prelevato correttamente e pronto per essere letto o quando posso procedere ad una scrittura.
- CMD è un registro di comando, in cui inserisco l'operazione da fare.

Nel caso della lettura, le operazioni da fare saranno le seguenti: 1. Scrivo il settore negli appositi indici 2. Do il comando di lettura 3. Aspetto che il regista di stato mi dia il via libera 4. Leggo 16 bit alla volta tramite il buffer. Infatti, l'interfaccia preleva il blocco e lo porta in una memoria interna. Facendo 256 lettura nel buffer, riesco a recuperare tutti i byte, ed è l'interfaccia che aggiorna il registro con il dato successivo. Avviene qualcosa di simile per la scrittura.

Il problema di questo approccio è che serve concordare tra interfaccia e sistema operativo la struttura dell'HD, in modo che nei registri si scriva la cosa giusta. Questo non è affatto scontato, e per anni ci sono stati problemi di compatibilità. La soluzione è stata quella di usare i registri CNH, CNL, SHR e i 4 bit di HND per scrivere un indirizzo univoco, che prende il nome di *LBA*. In tal modo, il software si occupa solo di dare un identificatore, ed è l'interfaccia a traslitterarlo in una posizione fisica. Avendo 28 bit per LBA, posso accedere

a 512×2^{37} byte, ovvero 128 Gib. Questa è una limitazione, e oggi si usa un LBA su 48 bit, per non avere problemi di spazio. Una volta visto questo, possiamo scrivere dei programmi che leggono e scrivono un HD.

```

1 #include <libce.h>
2
3 namespace hd {
4     const ioaddr iBR = 0x01F0;
5     const ioaddr iCNL = 0x01F5;
6     const ioaddr iCNH = 0x01F5;
7     const ioaddr iSNR = 0x01F3;
8     const ioaddr iHND = 0x01F6;
9     const ioaddr iSCR = 0x01F2;
10    const ioaddr iERR = 0x01F1;
11    const ioaddr iCMD = 0x01F7;
12    const ioaddr iSTS = 0x01F7;
13    const ioaddr iDCR = 0x03F6;
14 };
15 using namespace hd;
16
17 void hd_set_lba(natl lba){
18     natb lba_0 = lba, lba_1 = lba >> 8,
19         | lba_2 = lba >> 16, lba_3 = lba >> 24;
20
21     outputb(lba_0, iSNR);
22     outputb(lba_1, iCNL);
23     outputb(lba_2, iCNH);
24     natb hnd = (lba_3 & 0x0F) | 0xE0;
25     outputb(hnd, iHND);
26 }
27
28 void hd_start_cmd(natl lba, natb quanti, natb cmd){
29     hd_set_lba(lba);
30     outputb(quanti, iSCR);           // numero di settori
31     outputb(cmd, iCMD);            // comando di scrittura
32 }
```

```

34 void hd_wait_for_br(){
35     natb s;
36     do s = inputb(iSTS);
37     while (s & 0x88 != 0x08);
38 }
39
40 void hd_output_sect(natb vettore[]){
41     hd_wait_for_br();
42     outputbw((natw*)vettore, 256, iBR);
43 }
44
45 natb buff[8*512];
46 int main(){
47     natl lba = 1;
48     natb quanti = 2;
49     for (int i = 0; i < quanti*512; i++) buff[i] = 'f';
50     hd_start_cmd(lba, quanti, WRITE_SECT);
51     for (int i = 0; i < quanti; i++)
52         hd_output_sect(&buff[i * 512]);
53
54     str_write("OK\n");
55     pause();
56     // ...
57     return 0;
58 }
```

Per prima cosa si scrive in un buffer di dimensione opportuna il contenuto dei due blocchi da 512 byte che vogliamo andare a riempire. Poi si inizializzano i registri per LBA, il numero di blocchi su cui operare e il comando. Osserviamo che, nonostante possa dire su quanti blocchi fare le operazioni, comunque questi sono prelevati uno alla volta, e per ciascuno di questi devo controllare nuovamente lo stato prima di effettuare l'operazione. Nella funzione `hd_output_sect` si specifica l'indirizzo dal quale prendere la word che dovrò scrivere (faccio una conversione da puntatore a byte a puntatore a word, così da accedere correttamente al valore che mi serve). La copia nel buffer si fa con `outputbw`, che accetta un buffer di source, uno di destinazione, e il numero di locazioni da copiare (nel nostro caso 256 da 16 bit). Per la lettura, il ragionamento è analogo.

```

1 #include <libce.h>
2
3 namespace hd {
4     const ioaddr iBR = 0x01F0;
5     const ioaddr iCNL = 0x01F4;
6     const ioaddr iCNH = 0x01F5;
7     const ioaddr iSNR = 0x01F3;
8     const ioaddr iHND = 0x01F6;
9     const ioaddr iSCR = 0x01F2;
10    const ioaddr iERR = 0x01F1;
11    const ioaddr iCMD = 0x01F7;
12    const ioaddr iSTS = 0x01F7;
13    const ioaddr iDCR = 0x03F6;
14 };
15 using namespace hd;
16
17 void hd_wait_for_br(){
18     natb s;
19     do s = inputb(iSTS);
20     while (s & 0x88 != 0x08);
21 }
22
23 void hd_input_sect(natb vett[])
24 {
25     hd_wait_for_br();
26     inputbw(iBR, reinterpret_cast<natw*>(vett),
27             0x0000);
28     void leggisett(natl lba, natb quanti, natb vetti
29                 hd_start_cmd(lba, quanti, 0x20);
30                 for (int i = 0; i < quanti; i++) {
31                     hd_input_sect(&vetti[i * 512]);
32                 }
33 }
34
35 natb buff[8*512];
36 int main(){
37     natl lba = 1;
38     natb quanti = 2;
39     leggisett(lba, quanti, buff);
40     for (int i = 0; i < quanti*512; i++)
41         char_write(buff[i]);
42     char_write('\n');
43     pause();
44     return 0;
45 }

```

Lo stato prevede che non solo il bit *ready* sia ad uno, ma che un certo bit sia a 0.

Sulle interruzioni

Le interruzioni sono state introdotte nell'ambito dell'I/O. Quando abbiamo studiato la tastiera, per prelevare un carattere dal registro RBR abbiamo scritto un ciclo for che controllasse continuamente il bit di stato fintanto che non si fosse riempito il registro con un bit valido. Questo significa che la CPU non farà altro che controllare quel bit finché non sarà possibile accedere al registro d'interesse, mentre potrebbe spendere lo stesso tempo per realizzare ulteriori elaborazioni.

. Le interruzioni hanno lo scopo di evitare che il processore aspetti tutto questo tempo senza fare nulla. Infatti, nelle applicazioni reali non è possibile stare in attesa in questo modo. Si pensi ad un videogioco: mentre si aspetta che l'utente prema un tasto per proseguire l'azione, si deve continuamente aggiornare lo stato interno, modificare il contenuto dello schermo e far andare avanti la musica. Ciò non può essere fatto se la CPU sta ad aspettare in questo modo. Ovviamente, le interruzioni non sono in tal senso indispensabili (potremmo verificare la pressione del tasto una volta, aggiornare quanto necessario e poi verificare nuovamente il bit di stato della tastiera), ma semplificano molto la programmazione, e in generale rappresentano un primo passo verso la multiprogrammazione.

Dijkstra scrisse un sistema operativo che faceva uso delle interruzioni, e ne spiegò il motivo in questi termini. Supponiamo che un calcolatore faccia delle elaborazioni molto dispendiose e poi si preoccupi, alla fine di ciascuna di queste, di scrivere il risultato per mezzo di una macchina da scrivere usando dei segnali elettrici per attivare i tasti. Ci sarà un'interfaccia con un registro che mantiene il carattere da scrivere, e questo non dovrà essere modificato durante tutta la scrittura dello stesso, per non avere danni a livello meccanico. Si supponga allora che ci sia un bit di feedback per sapere quando la macchina è pronta a ricevere un nuovo carattere. Per come abbiamo programmato finora, dovremmo fare un ciclo in cui si passa un carattere alla volta alla macchina aspettando, per ciascuno, che sia stato scritto correttamente: nel frattempo, avremmo potuto portare avanti la computazione. Volendo scrivere un carattere, proseguire con il codice e controllare periodicamente se si debba andare al carattere successivo, è necessaria un'implementazione lato SW? Se sì, dovrebbe essere il programmatore a scrivere una riga di codice per ogni controllo, benché questo potrebbe avvenire troppo spesso o troppo di rado. Conviene che sia la CPU a capire quando è necessario proseguire con la stampa, effettuando la verifica *alla fine di ogni istruzione*, affinché non ci siano problemi di attese eccessive.

Il comportamento della CPU dovrà essere modificato. Se prima prelevava un'istruzione e la eseguiva, adesso dovrà prelevare, eseguire, e verificare se ci sono delle richieste di interruzioni da gestire. Per farlo, testa un bit che sarà in ingresso alla CPU, che chiameremo INTR (*interrupt request*). Per semplificare, supponiamo che solo un dispositivo possa mandare interruzioni alla CPU. Cosa dovrà fare quest'ultima? Ovviamente, sarà il software a dirlo. Infatti, non avrebbe senso specializzare la CPU per una data periferica o dispositivo, dovendo rimanere più generica possibile. Il programma dirà al processore che, nel caso si presenti una richiesta di interruzione, non deve prelevare l'istruzione successiva ma proseguire verso una certa routine. Più che un salto, si deve fare qualcosa di simile ad una chiamata di funzione, per cui si salva (in pila) anche l'indirizzo al quale dovremo tornare alla fine dell'esecuzione della routine. Potenzialmente, alla fine di ogni istruzione ci potrebbe essere un *salto* causato da un'interruzione: questo fa sì che il programma perda determinismo.

Come si realizza un meccanismo del genere? Supponiamo che INTR sia collegato al bit *busy* della tastiera. Se si trova ad 1, c'è un carattere da prelevare. Una volta che si fa l'istruzione di IN da RBR, il valore del bit torna a 0. Ma questo è un problema: infatti, se ad ogni istruzione chiamo la routine dell'interruzione, poiché quest'ultima avrà altre istruzioni prima della IN, INTR rimarrà ad 1, entro in un loop infinito nel quale

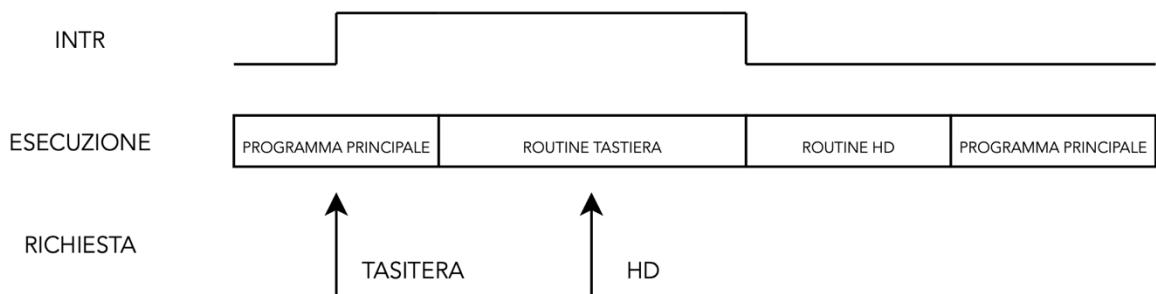
chiamo continuamente la routine. Ci sono due modi per risolvere questo problema. Il primo è quello di non collegare *busy* direttamente ad INTR, ma di intramezzare un latch, il quale viene azzerato dalla CPU al momento dell'accettazione di una richiesta di interruzione. L'altro è quello di avere un flag che mi dice se posso o meno accettare richieste di interruzioni: accetto interruzioni solo se il bit è a 1, e prima di chiamare una routine imposto il bit a 0 per non avere i problemi suddetti. Questa è la soluzione adottata dall'Intel, con il bit IF dentro il registro dei flag.

Una richiesta di interruzione modifica i flag. Per questo motivo, è bene che, insieme al salvataggio nello stack dell'indirizzo di ritorno, si faccia anche il salvataggio del registro dei flag, affinché il programma ritrovi tutto quanto intatto. Per quanto riguarda il programmatore, per uscire da una routine dovrà usare l'istruzione `iretq`, che non ripristina solo `%rip` ma anche RF. Il programmatore deve avere altri accorgimenti per scrivere una routine per un'interruzione. Infatti, a differenza delle chiamate di funzioni, non ha senso parlare di registri di scratch: l'interruzione potrebbe avvenire da un momento all'altro, e il programma principale non può vedersi modificare i registri in modo casuale. La struttura della routine deve essere la seguente:

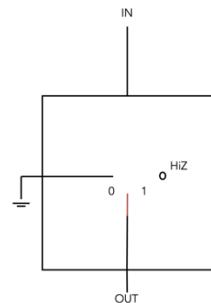
```
s_routine:
    #salvo tutti i registri in pila
    CALL cpp_rutine
    #ripristino i registri
    IRETQ
```

Non c'è un modo standard per dire al compilatore che stiamo scrivendo qualcosa per le interruzioni, e che quindi i registri non devono essere sporcati. Per questo, la routine vera e propria si scrive in Assembler, ma la funzione che effettua l'elaborazione si può anche fare in C++.

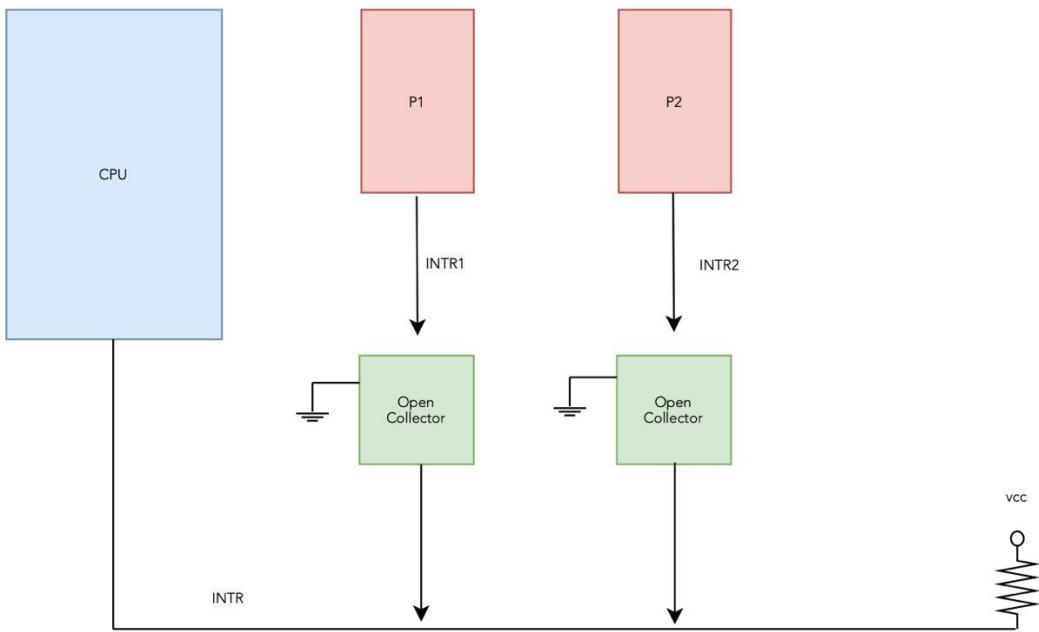
Le interruzioni permettono dunque un avanzamento parallelo tra i programmi. Tuttavia, gestire INTR in questo modo permette di accettare interruzioni da un solo programma, cosa insensata se si pensa a tutte le periferiche collegate al calcolatore: la tastiera ha un bit *busy*, l'hard disk ha un bit *data_ready*, il timer 0 è collegato ad una richiesta di interruzione... Una soluzione istintiva è quella di collegare tanti fili di INTR al processore, uno per ciascuna periferica. Questo però è oneroso e impraticabile all'aumentare delle periferiche. Collegiamo allora le periferiche in modo che, se qualcuna di queste genera un'interruzione, INTR vada ad 1. Chiaramente, si tratta di fare l'*or* tra tutti i bit che generano interruzioni. Ma a quel punto, come sappiamo quale periferica ha generato l'interruzione? La cosa non è affatto scontata, perché finora la CPU non fa altro che chiamare una routine quando INTR vale 1. È necessario che lato software si verifichino una per una le periferiche che potrebbero causare un'interruzione, ed eseguire un programma diverso per ciascuna di queste. Ma ci sono tante cose che potrebbero andare storte. In particolare, il meccanismo con il latch crea problemi. Nel momento in cui si accetta una richiesta, INTR viene posto a 0, anche se l'altro dispositivo continua ad avere una richiesta in sospeso. Con INTR collegato all'uscita della porta *or*, invece, è possibile gestire questa situazione: ad esempio, come in figura, alla fine della routine per la tastiera si esegue immediatamente la routine per l'hard disk. Questo però potrebbe comportare *starvation*: se l'hard esegue moltissime richieste aventi con priorità maggiori di quelle della tastiera, queste non saranno considerate se non raramente. Alla fine dell'interruzione, essendo ripristinato IF, verrà eseguita l'altra interruzione, senza tornare al programma principale.



È necessario che la CPU, alla fine dell'esecuzione di un'istruzione, esegua la verifica di INTR & IF, per sapere se serve gestire un'interruzione. In caso affermativo, si può supporre di saltare ad un indirizzo costante, non prima di aver salvato in pila %rip e RF e di aver posto a 0 IF, per non avere problemi non avendo ancora azzerato INTR. Il problema adesso è quello di poter gestire interruzioni da interfacce diverse. Non possiamo inserire alcuni piedini dentro il processore, perché potrebbero essere in numero limitato, ma, allo stesso modo, non possiamo neanche collegarli tutti ad una porta OR, avendo lo stesso problema. Si usa un nuovo tipo di porta logica, detta **open collector**. Se l'ingresso vale 0, l'uscita è collegata a massa, altrimenti viene messa in alta impedenza, equivalentemente ad un filo staccato.



L'approccio che si utilizza è il seguente:



Se entrambe le uscite INTR delle due periferiche sono a 0, tali lo sono le uscite degli OC. Il filo INTR è collegato a massa, essendo in cortocircuito con questi due valori, e la CPU riceve 0. Lo stesso accade se anche solo uno dei due vale 0: l'altro filo sarà in alta impedenza, il filo INTR riceverà comunque la tensione corrispondente al valore logico 0. Se le uscite delle due periferiche valgono 1, i fili intermedi tra INTR e gli OC sono in alta impedenza, come se fossero staccati: la tensione su INTR è Vcc tramite la resistenza di *pull down*, che corrisponde al valore logico 1. Abbiamo ottenuto l'equivalente di una porta AND, ossia una porta OR nella logica attiva bassa. Questo è il motivo per cui le interruzioni sono gestite in termini di logica attiva bassa. Tale gestione è detta **wired or**.

La scelta della Intel è stata tuttavia differente: c'è un controllore delle interruzioni, APIC (*Advanced Programmable Interrupt Controller*), una sorta di periferica che accetta le richieste, le ordina e le invia al processore. In questo modo possiamo avere un numero limitato di periferiche (24 piedini, ma se ad un piedino usiamo il ragionamento visto sopra risolviamo il problema), ma alcuni vantaggi. Prima fra tutte, realizziamo **interruzioni vettorizzate**. Invece di far sì che la CPU, al momento di un'interruzione, salti ad un indirizzo fisso e inizi a testare tutte le periferiche per capire quale ha generato la richiesta, associamo tramite APIC un tipo a ciascun piedino (256 tipi). Quando c'è una richiesta, il controllore invia INTR e il tipo associato con un handshake; all'interno della memoria c'è una **tabella delle interruzioni**, IDT, nella quale possiamo associare a ciascun tipo un indirizzo, corrispondente alla routine da eseguire al momento di un'interruzione di quel tipo. IDT è gestita dal programmatore, in modo che possa farci tutto quello che vuole. Per facilitare questo compito, IDT sta in memoria, e il suo indirizzo è contenuto dentro un registro del processore, IDTR. Per modificare tale indirizzo, si può usare l'istruzione Assembler *LIDTR* (*load IDT register*).

Anche il controllore può essere gestito dal programmatore; in particolare, si comporta a tutti gli effetti come un'interfaccia, con dei registri che, tra le tante cose, ci permettono di associare un tipo a ciascun piedino. La comunicazione tra APIC e CPU è abbastanza complessa; possiamo avere un'idea intuitiva vedendo come avveniva in una sua versione più primitiva, detta PIC. Tra CPU e PIC erano presenti i fili INTR e INTA, con i quali si effettuava un *handshake*. Quando PIC vuole sollevare un'interruzione, porta INTR ad 1. La CPU, ad un certo punto se ne accorge e l'accetta, portando su INTA. PIC risponde immettendo sul bus il tipo dell'interruzione e portando giù INTR. La CPU preleva il tipo e porta giù INTA per conferma. Come conseguenza di

un approccio del genere, non è più necessario IF per evitare loop durante l'accettazione delle interruzioni. Il flag tuttavia non scompare, e ogni volta viene testato per sapere se eseguire o meno l'interruzione. Il programmatore può anche settarlo/resettarlo, con le istruzioni STI e CLI. Inoltre, nella tabella delle interruzioni ci sono molte informazioni aggiuntive: ad esempio, si dice se, all'esecuzione della routine associata, si voglia che il processore azzeri IF oppure no.

Di per sé, IDT e APIC sono organizzati in modo molto contorto, per questioni di retrocompatibilità. Noi utilizzeremo alcune funzioni presenti in `libce` per effettuare operazioni standard. Tramite la funzione `gate_init(tipo, indirizzo)`, si inizializza l'indirizzo associato ad un tipo in IDT. Infatti, come vedremo, **i tipi sono detti gate**. La funzione `apic_set_VECT(piedino, tipo)` serve per associare ad un piedino un tipo. Sappiamo quale periferica è collegata a ciascun piedino dalla documentazione del calcolatore (per esempio, la tastiera è collegata al piedino 1, il timer al 2). Ogni piedino di APIC ha anche un bit che dice se abilitare o meno le interruzioni provenienti dallo stesso. Per abilitarle, si usa `apic_set_MIRQ(piedino, false)`. Oltre a questo, APIC vuole sapere quando la CPU ha finito di gestire un'interruzione. Per comunicarglielo, alla fine della routine, si usa `apic_send_EOI()`.

Per abilitare un'interruzione, dovremo scrivere, nell'esempio della tastiera:

```
apic_set_VECT(1, KBD_TYPE);
gate_init(1, handler);
apic_set_MIRQ(1, false);
abilita_int();
```

L'ultima è una funzione specifica per la periferica in esame, e le permette di realizzare interruzioni. L'handler che associamo all'interruzione non può essere una normale funzione di C++, perché dovrà preservare i registri e terminare con `iretq`. Scriviamo allora una funzione `a_handler` in C++ per fare le operazioni che vogliamo, e un file Assembler in cui definiamo l'handler come segue:

```
1 #include "libce.s"
2 .text
3 handler:
4     salva_registri
5     CALL a_handler
6     carica_registri
7     IRETQ
8
9
```

Dentro `libce` sono definite le due macro per salvare in pila tutti i registri e riprenderli. Ovviamente, per fare una cosa del genere, la funzione `a_handler` deve essere dichiarata con `extern "C"`. L'ultimo aspetto importante è che, se in C++ ci sono delle variabili che non sono apertamente modificate, le ottimizzazioni del compilatore tendono ad eliminarle. In particolare, magari facciamo uso di una variabile globale per comunicare tra un programma principale e la routine di un'interruzione, ma il compilatore non se ne rende conto, non avendo il concetto di interruzione. Per non avere problemi, queste variabili debbono essere dichiarate con la parola chiave `volatile`, che evita tali ottimizzazioni.

L'APIC può associare ad ogni piedino altre informazioni, che dicono come gestire le richieste. Il flag MASK, se ad 1, disabilita le interruzioni da quel piedino. Ci sono poi l'*Interrupt Polarity* IPOL e il *Trigger Mode* TRGM, che consentono di adattarsi alle singole periferiche. TRGM permette di riconoscere una richiesta sul fronte oppure sul livello: nel primo caso, aspetta solo un fronte, nel secondo caso, accetta la richiesta sul fronte, ma la considera nuovamente inviata se, al momento dell'EOI, trova il segnale del piedino ancora ad 1. IPOL serve a dire quale valore significa interruzione e quale no, tra 0 o 1.

La tastiera agisce con il bit *busy*: genera una nuova richiesta di interruzione quando ha un nuovo codice, poi rimuove la richiesta di interruzione quando vede una lettura in RBR. Cosa succede se, mentre ha un nuovo codice, prima che avvenga la lettura in RBR, si preme un nuovo tasto? La tastiera lo memorizza internamente, e *busy* resta ad 1 anche dopo aver letto RBR. Per questo, conviene usare il trigger mode sul livello, per non perdere mai richieste di interruzioni.

Il timer 0 ha un piedino di uscita collegato ad APIC (in QEMU è collegato al piedino 2). Il segnale che il timer genera può essere usato per richiedere interruzioni, ma, a differenza della tastiera, genera interruzioni senza aspettare una comunicazione per generare la successiva. Se lo mettiamo ad esempio in modalità 4, esso genera un'onda quadra di frequenza nota. Per usare questo segnale come richiesta di interrupt, ci conviene usare il fronte. Se usassimo il livello, potremmo triggerarci due volte se la CPU fosse estremamente veloce ed accettasse l'interruzione prima che il timer abbia portato giù la linea. Di fatto, il tempo che il segnale resta ad 1 è legato al clock del timer, che può essere piuttosto lento in confronto al clock della CPU.

```

1  ↵ #include<libce.h>
2   #include<apic.h>
3
4   volatile int counter = 0;
5 ↵ extern "C" void c_timer(){
6     counter++;
7     apic_send_EOI();
8   }
9
10  extern "C" void a_timer();
11  const ioaddr iCWR = 0x43;
12  const ioaddr iCTR0_LOW = 0x40;
13  const ioaddr iCTR0_HIG = 0x40;
14
15 ↵ int main(){
16   apic_set_VECT(2,0x50);
17   gate_init(0x50,a_timer);
18   apic_set_MIRQ(2,false);           //abilita le interruzioni per quel piedino
19   apic_set_TRGM(2,false);          //false: fronte; true: livello
20   outputb(0b00110111, iCWR);
21   outputb((natb)50000, iCTR0_LOW); // riempiamo il valore del timer 0
22   outputb((natb)50000 >> 8, iCTR0_HIG);
23 ↵ //Nel ciclo non faccio nulla, quindi se il compilatore vuole ottimizzare
24 //potrebbe togliere la i, cosa che non vogliamo
25   for( volatile long i=0; i<1000000; i++ );
26   printf("counter = %d\n", counter);
27   pause();
28 }
```

Nell'esempio, ci facciamo mandare più richieste dal timer che incrementano il contatore. Alla fine, ne stampiamo il valore. Affinché si ricevano richieste, bisogna assegniamo un tipo al piedino (riga 16) e

associare una funzione dentro IDT (riga 17). Usiamo `apic_set_TRGM` per modificare il trigger mode, per cui 0 vuol dire sul fronte, 1 sul livello. Bisogna poi programmare il timer, impostando la correttamente la modalità e il tempo di conteggio. Ovviamente serve anche del codice Assembler, per scrivere la funzione `a_timer`. Usando la modalità 0, il timer invia una richiesta e si ferma nel conteggio. Impostando la modalità 3 (onda quadra) con riconoscimento sul fronte, si ottiene un contatore pari a 30. E se mettiamo il riconoscimento sul livello? Ci aspettiamo che ci possano essere dei problemi, ottenendo `counter > 30`: per tutto il tempo in cui il timer ha tenuto la linea ad 1, il nostro programma ha eseguito la routine, ha inviato EOI, e, trovando la linea ancora ad 1, la considera come una nuova richiesta di interruzione. Per il timer, quindi, conviene riconoscere il fronte.

A questo punto, grazie all'APIC, possiamo gestire più di un'interruzione alla volta: il programma principale può essere interrotto da una routine, ma possiamo lasciare IF ad 1 durante l'esecuzione della routine. Questo permette di avere interruzioni annidate: quando l'ultima termina con una `iretq`, si torna all'esecuzione della routine precedente (gli instruction pointer sono tutti salvati in pila). Non abbiamo più il problema che INTR ad 1 ci porta in loop: INTR viene riportato a 0 da APIC alla fine dell'handshake tra i due. Questa gestione delle interruzioni annidate ha senso se la seconda interruzione ha una priorità maggiore di quella che si stava gestendo. Per esempio, le richieste del timer sono immediate, mentre la tastiera si tiene un codice in RBR che posso elaborare anche in un secondo momento. Se tenessi le interruzioni sempre disabilitate durante la routine della tastiera, perderei le interruzioni del timer, che, al contrario, non si possono accumulare.

L'APIC mi consente di assegnare delle priorità ad ogni piedino, per gestire l'annidamento. Allo stesso tempo, anche IF deve rimanere ad 1. Abbiamo due possibilità: usare l'istruzione STI, oppure chiedere alla CPU di non resettare IF, gestendo un certo bit dentro la IDT. Per ogni entrata, oltre all'indirizzo della routine a cui saltare, ci sono altre informazioni, tra cui una che dice se la richiesta è *Interrupt* oppure *Trap*. Trap significa che, quando si passa attraverso quel gate, l'interrupt flag non deve essere modificato, e rimarrà dunque ad 1 (non poteva essere a 0, altrimenti non avrei accettato la richiesta). Questo è il motivo per cui salva lo stato precedente dei flag e non mi preoccupo di gestire solo IF: dipende tutto dall'interruzione che stava in esecuzione. Per impostare la modalità Trap, si usa la funzione `trap_init` di libce. Se impostiamo la tastiera in modalità *Trap*, la CPU può accettare nuove richieste di interruzioni anche durante l'esecuzione.

A questo punto ci dobbiamo chiedere *quando l'APIC invia le richieste di interruzione*. **Il tipo è una codifica della priorità**: i 4 bit più significativi sono la classe di priorità, gli altri sono la sotto priorità all'interno della classe. L'APIC invia una nuova richiesta solo se **tra quelle pendenti ce n'è una con classe di priorità maggiore della massima classe di priorità che è già stata accettata dal processore**. Affinché questo meccanismo funzioni, l'APIC deve sapere che cosa il processore sta facendo, ovvero quale routine di interruzione sta gestendo. Il meccanismo con cui salva tale informazione è implicito: quando invia una richiesta con un tipo, assume che il processore sia saltato ad una routine di gestione con quella classe di priorità. Si ricorda tutte le interruzioni che gli ha inviato e le loro priorità. Quando il software invia l'EOI, l'APIC assume che il processore abbia terminato la routine di priorità massima. L'assunzione dell'APIC è che le interruzioni siano gestite LIFO, e ciò dipende esclusivamente dal programmatore, che deve ricordarsi di inviare EOI. Se ciò accade, l'APIC mantiene una visione coerente di cosa sta accadendo nel processore.

Ci sono due registri da 256 bit, un bit per ogni tipo, ISR (*interrupt service register*) e IRR (*interrupt request register*). Quando arriva un interrupt da uno dei piedini, trova il bit e mette ad 1 il corrispondente bit di IRR: esso ricorda quali sono le interruzioni che l'APIC ha visto e che non ha ancora passato al processore. Decide di passarlo al processore quando la classe di priorità pendente è maggiore della massima classe di priorità memorizzata in ISR, che invece salva l'attuale esecuzione da parte del processore. Se ciò accade,

l'interfaccia invia la richiesta al processore: nel momento in cui viene accettata, a compimento dell'handshake, il bit si sposta da IRR a ISR. Quando arriva una nuova richiesta da un altro piedino, mette ad uno il corrispondente bit di IRR. Guarda poi la massima priorità che era in esecuzione. **Se è minore o uguale non fa nulla**, altrimenti invia una nuova richiesta alla CPU. Quando il processore e il software scrivono dentro EOI, l'APIC rimuove il bit ad 1 che sta più a sinistra di ISR (assumendo proprio che il software abbia finito di gestire l'ultima interruzione che è andata in esecuzione). Fatto ciò, accede in IRR e verifica se ci fossero altre richieste in attesa, con priorità compresa tra quella che ha appena terminato e quella che deve continuare ad essere eseguita (a parità di priorità si cerca la sottoclasse maggiore). Se il processore la trova, la invia alla CPU, eventualmente surclassando l'ultima non ancora conclusa.

In generale, l'APIC fa passare le richieste che hanno priorità maggiore stretta a quella che sta eseguendo. Questo è il motivo per cui serve EOI, anche nel caso in cui c'è una sola periferica: l'interruzione era salvata dentro IRR, ma se non viene azzerato il corrispondente bit di ISR non si possono ricevere altre interruzioni dalla stessa periferica.

Le routine di interruzioni iniziano e finiscono. Per far sì che possa fare cose *con memoria*, è necessario salvarsi delle variabili globali, affinché la funzione venga conclusa e richiamata più volte, a differenza del programma principale che viene chiamato più volte. La parola chiave volatile aiuta in tal senso, per evitare ottimizzazioni quando sembra che la variabile non sia mai utilizzata.

L'accettazione di una richiesta dipende quindi da molti fattori. Innanzitutto, dalla priorità della richiesta, ma anche dall'attuale valore di IF. Supponiamo che la tastiera abbia priorità 0x50, il timer 0x40. La tastiera richiede un'interruzione, e il processore l'accetta, senza modificare IF. Il timer a questo punto, pur mandando numerose richieste, non può fare nulla, avendo classe di priorità minore. Le cose cambiano nel momento in cui arriva l'EOI dalla CPU: il timer ha l'interruzione con priorità maggiore, e viene messa in esecuzione prima ancora che la tastiera possa fare IRTEQ. Ovviamente, questo comporta che si perdano molte interruzioni dal timer. Si veda l'esempio int-tastiera-2.

```

1  #include <libce.h>
2  #include <apic.h>
3
4  const ioaddr iSTR = 0x64;
5  const ioaddr iTBR = 0x60;
6  const ioaddr iRBR = 0x60;
7  const ioaddr iCMR = 0x64;
8
9  volatile bool stop = false;
10 extern "C" void a_driv_tas();
11 extern "C" void c_driv_tas(){
12     natb c; c = inputb(iRBR);
13     if (c == 0x01) stop = true;
14     for (int i = 0; i < 8; i++) {
15         if (c & 0x80) char_write('1')
16         else char_write('0');
17         c <<= 1;
18         for (int j = 0; j < 10000000;
19     }
20     char_write('\n');
21     apic_send_EOI();
22 }
23
35  int main(){
36      trap_init(0x48, a_driv_tas);
37      apic_set_VECT(1, 0x48);
38      apic_set_MIRQ(1, false);
39      stop = false;
40      enable_intr_kbd();
41      // costante di tempo relativa a 50 ms
42      attiva_timer(59500);
43      trap_init(0x50, a_driv_tim);
44      apic_set_VECT(2, 0x50);
45      apic_set_MIRQ(2, false);
46      int i = 0;
47      while (!stop) {
48          video[40] = (video[40] & 0xff00) | sp
49          i = (i + 1) % sizeof(spinner);
50      }
51  }
52

```

Supponiamo che il tipo del timer sia 0x50, quello della tastiera 0x40. Se durante un'interruzione della tastiera il gate è in modalità Trap, la CPU accetta l'interruzione del timer, inviata avendo priorità maggiore. Se il gate era in modalità Interrupt, la routine del timer non sarà eseguita fintanto che IF resta a 0, quindi fino all'esecuzione di iretq. Cosa succede se si dimentica di inviare EOI nella routine della tastiera, con gate in modalità Trap? L'APIC invia le richieste del timer, avendo priorità maggiore, ma il bit corrispondente a 0x40 di ISR resta sempre ad 1, in quanto il controllore presuppone che la CPU stia sempre eseguendo quella routine. Come conseguenza, non si possono avere altre interruzioni della tastiera. Se invece EOI lo dimentichiamo con il timer, queste non sono accettate per lo stesso motivo, quelle della tastiera rimangono comunque in sospeso avendo priorità minore. Ovviamente, questa situazione non inficia con il programma principale, che continua ad eseguire le sue istruzioni non venendo più interrotto.

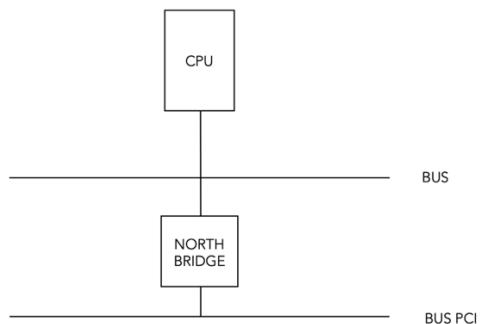
Finora, per accedere alle periferiche abbiamo visto il trasferimento a controllo di programma (è il programma che va a vedere lo stato di ciascuna per sapere se c'è bisogno di effettuare un'operazione) e le interruzioni (la periferica invia un'interruzione quando necessario, e il software è programmato in modo da eseguire una particolare routine). La terza modalità di cui ci dobbiamo occupare è l'accesso diretto in memoria (*direct memory access*, o DMA). In questa situazione, una periferica è istruita per scrivere direttamente in memoria, mentre noi non ci interessiamo delle sue attività. Per le periferiche viste finora, potrebbe essere utile lavorando con l'hard disk, in modo che le sue operazioni non coinvolgano direttamente la CPU. Per trattare di questo modello, è necessario introdurre il bus PCI.

Sul bus PCI

Un'architettura è realizzata in modo da poter essere espandibile. Decenni fa, si acquistavano delle schede in più per avere nuove funzionalità (*per esempio, la scheda di rete per il collegamento ad internet*). Tali periferiche potevano essere realizzate da chiunque ma, in quanto tali, dovevano essere inserite sul bus e convivere non solo con RAM e I/O, ma anche con le altre periferiche. Ciascuna di queste necessitava di **una serie di indirizzi** con i quali effettuare l'accesso ai suoi registri. Ma se questi fossero stati fissati, ci sarebbero stati problemi di sovrapposizione di indirizzi tra periferiche diverse: l'accesso ad un indirizzo avrebbe comportato un segnale di lettura/scrittura per due periferiche, causando problemi di carattere sia logico che elettrico. Allo stesso tempo, il software per dialogare con queste periferiche deve conoscere gli indirizzi a cui sono fissate, che devono essere standard al momento dell'installazione. Per tale ragione, presa una scheda, potrebbe non essere compatibile con quelle già presenti, in termini di sovrapposizioni.

Un altro problema coinvolge invece gli utenti: se questi inserissero male una scheda, come potrebbe il software capire che, al dato indirizzo, non c'è quanto si aspettava? Un metodo standard potrebbe essere quello di scrivere in un registro e leggerne un altro in cui si aspetta un valore prefissato; ma se i due indirizzi corrispondono a periferiche che non gli competono, si hanno altri problemi sul bus.

I costruttori, di fronte a questi problemi, hanno definito lo standard PCI, per connettere periferiche ad un sistema di elaborazione. Esso non è il bus direttamente collegato alla CPU, ma un ulteriore bus che dialoga con il primo attraverso un circuito che fa da controllore, il **ponte ospite/PCI o north bridge**. In genere, ci possono essere anche dei bus PCI collegati ad altri bus PCI, con una struttura ad albero, tra di loro legati da ponti PCI/PCI. In tale struttura, per come qemu emula il sistema, la RAM è collegate al north bridge.



In quanto tale, il bus PCI ha dei collegamenti e dei protocolli per il suo utilizzo. Esso è su **32 bit**, e si eseguono delle *transazioni* ciascuna delle quali ha un **iniziatore** e un **target**, tra i quali avviene lo scambio di dati. Il tipico iniziatore è il north bridge, che agisce per conto della CPU. Ad alto livello, la comunicazione avviene per come visto sul bus del processore: l'iniziatore definisce un indirizzo, e le maschere delle varie periferiche fanno sì che una sola risponda; avviene allora l'operazione richiesta tramite i fili dati.

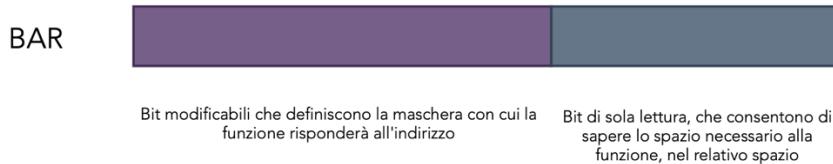
Sul bus ci sono tre spazi di indirizzamento: memoria, I/O e spazio di configurazione. Come vedremo, i primi due entreranno a far parte della memoria e I/O del processore. L'ultimo invece è la caratteristica che evita i problemi di inizio paragrafo. All'avvio, si dialoga con ciascuna periferica in modo **mutuamente esclusivo**, e si assegnano loro sia indirizzi che non conducano a conflitti, sia lo spazio di cui necessitano per le operazioni. Ciascuna periferica ha un registro nel quale scrivere la maschera che le deve attivare, e che può

essere configurabile via software. Una volta conclusa queste operazioni al boot, non è più necessario accedere a tale spazio.

Il primo momento della transazione è detto *fase di indirizzamento*. Il bus ha un clock, molto più lento di quello della CPU. Il segnale che avvia una transazione è **/frame**, che viene portato a 0. Ci sono poi i fili **c/be** e **A/D**, che sono multiplexati: in una prima fase, vengono usati per scambiare, rispettivamente, comando dell'operazione, comprendente di R/W e spazio da usare, e indirizzo del target; dopo che questo ha risposto, sono usati per scambiare i byte enabler e i dati. Avendo a che fare con un bus a 32 bit, sono sufficienti 4 byte enabler (e infatti c/be è di 4 bit). I dispositivi, trovandosi /frame a 0, verificano se l'indirizzo li coinvolge, controllando il valore del registro opportuno. Se le impostazioni sono state fatte correttamente, solo un dispositivo può rispondere, e lo fa portando il filo **/devsel** a 0 (*se ciò non accade entro un certo numero di clock, si ha un abort dell'operazione*). Quando l'iniziatore si accorge che /devsel è a 0, conosce *be* e *D* per comunicare il resto dei dati. Quella che inizia è detta *fase di dati*. In tale fase, l'iniziatore può richiedere anche più operazioni consecutive, senza ricorrere ad una nuova inizializzazione. Per farlo, c'è un handshake tra le due parti con i fili **/irdy** e **/trdy**. Il processore comunica che si sta gestendo l'ultima operazione portando /frame ad 1: il dispositivo risponde al termine del processo alzando /devsel, azione che conclude la transazione.

Occupiamoci ora dello spazio di configurazione. Ogni dispositivo (eventualmente, un altro bus PCI), trovandosi collegato al bus, acquista un indirizzo univoco. In particolare, ogni ulteriore bus ha un indice a partire dallo 0, quello direttamente collegato alla CPU, ad incrementare. I bit dedicati per tale valore sono 8: si possono avere al più 256 bus. Ogni bus permette di collegare al massimo 32 dispositivi (5 bit), ciascuno dei quali con 8 funzioni diverse (3 bit). In tal modo, all'inizializzazione, è possibile dialogare con ogni funzionalità di ciascun dispositivo, aventi dei registri di controllo. Per ogni funzione, ci deve essere uno spazio di memoria di 256 byte, per cui lo standard definisce i primi, e concede ai vendori l'uso dei successivi.

Dei byte associati a ciascuna funzionalità per la programmazione, i primi due sono un identificativo del venditore della periferica PCI, un *vendor id* assegnato dal consorzio PCI. I due che seguono sono un *device id*; grazie a questo, so univocamente con quale dispositivo sto dialogando, e quale driver serve per usarlo. Ci sono poi i **BAR** (*base address register*, di 32 bit) per la programmazione delle maschere. Ogni funzione può avere fino a 6 BAR, e quindi 6 maschere: pensando alla scheda video, ad esempio, ce ne saranno almeno due, una per l'indirizzo dello spazio di memoria per la sua RAM e una per lo spazio di I/O per i suoi registri di controllo. Per ciascuna periferica, il PCI bios deve sapere di quanto spazio hanno bisogno e dove per assegnarglielo. I BAR hanno una doppia funzione: permettono di capire le esigenze del dispositivo e di programmare la scheda in termini di maschera.



Il primo bit è di sola lettura, è codifica se serve spazio di memoria o I/O; seguono una serie di bit tutti in sola lettura, e il loro numero rappresenta lo spazio che la funzione richiede. Per esempio, avendo n bit in sola lettura, sono necessari 2^n bit per la funzione. Per testare se il bit è in sola lettura, prova a scriverci e vede se si è modificato. Gli altri bit rappresentano allora quelli fissi per tutta la regione naturale associata alla funzione, e sono dunque la maschera: la programmazione avviene tenendo conto degli indirizzi associati alle

funzioni già trovate, e modificando di volta in volta le maschere. Queste operazioni sono eseguite da un programma che potremmo chiamare *PCI bios*.

A questo punto, bisogna capire come fa il software a dialogare con questo bus. Per quanto riguarda la memoria e l'I/O, il bus è quasi trasparente, inoltrando la richiesta al ponte (e comportandosi quindi da iniziatore della transazione). Esso poi risponde scrivendo sui fili di dati del front bus. Per distinguere un accesso alla RAM o alla memoria dei dispositivi PCI, si usa la convenzione per cui la RAM (*che nel calcolatore che studiamo è da 1GiB*) si trova nei primi indirizzi disponibili dello spazio di indirizzamento. Per indirizzi maggiori di 1GiB, risponde il ponte PCI.

Il software accede allo spazio di configurazione in modo indiretto tramite due registri del ponte, **CAP** e **CDP**. In CAP è inserito l'indirizzo dello spazio di configurazione della riga (numero di bus + device + funzione + offset del registro) con cui vogliamo interagire; la scrittura/lettura la facciamo agendo su CDP a 32 bit. Per effettuare queste operazioni, si usano alcune funzioni di libce: make_CAP restituisce un valore valido da inserire nel CAP, pci_read_confb si occupa di prelevare il corretto valore di CDP. La stessa funzione esiste anche per lunghezze di word e long, così come per la scrittura.

Scriviamo un programma con cui vedere quali periferiche sono collegate al bus PCI. Il ponte registra un errore nel caso in cui una periferica non abbia risposto in tempi opportuna. Tale errore si realizza nella pratica scrivendo una serie di 1 in CDP. Un'ulteriore convenzione è che il *vendor ID* di un dispositivo non può essere formato di tutti 1: in tale modo, si possono far variare dispositivi e funzioni, leggere i primi byte dello spazio di configurazione di ciascuna funzione e vedere se sono tutti ad 1. In caso negativo, ho trovato un dispositivo di cui posso stampare le informazioni.

```

1  #include <libce.h>
2  int main(){
3      natb bus = 0, c;
4      for (natb dev = 0; dev < 32; dev++) {           // 32 dispositivi per ciascun bus
5          for (natb fun = 0; fun < 8; fun++) {        // 8 funzioni per ciascun device
6              natw vendorID, deviceID;
7
8              // Leggo due byte ad offset 0 del proprio spazio di configurazione
9              vendorID = pci_read_confw(bus, dev, fun, 0);
10             if (vendorID == 0xFFFF) continue;
11             // Se non sono ad 1, prosegua con la lettura del deviceID
12             // Leggo due byte ad offset 0 del proprio spazio di configurazione
13             deviceID = pci_read_confw(bus, dev, fun, 2);
14             printf("%2x.%2x.%2x: %4x.%4x\n",
15                   bus, dev, fun,
16                   vendorID, deviceID);
17         }
18     }
19 }
20 pause();
21 return 0;
22 }
```

Sul DMA

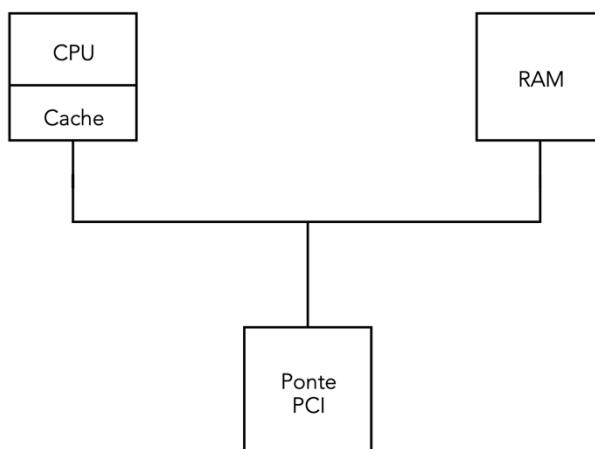
Alcuni dispositivi del bus PCI, tramite particolari settaggi interni, possono fare operazioni di lettura o scrittura direttamente con la RAM, intramezzati dal ponte PCI. Questo è il modo con cui si implementa il Direct Memory Access. Il target della transazione sarà quindi il ponte, che proseguirà l'operazione lato front bus verso la RAM; questa modalità è detta Bus Mastering PCI.

A questo punto, si hanno alcuni problemi di carattere funzionale ed elettrico. Primo fra tutti, ci potrebbero essere più iniziatori di transazioni, sia sul front bus (CPU esegue il software e ponte traduce le operazioni) sia sul bus PCI (ponte e periferiche). Ci deve essere mutua esclusione degli iniziatori all'accesso sul bus.

Nel caso del bus PCI, lo standard prevede la presenza di un **arbitro**. Ciascun dispositivo, ponte incluso, ha un filo di *request* REQ verso l'arbitro e uno di *granted* GNT che riceve da esso. Nel momento in cui vuole fare un'operazione sul bus, invia un segnale di request all'arbitro, che risponderà con un granted. L'handshake può avvenire anche durante l'esecuzione di una transazione sul bus: il dispositivo non può iniziare l'operazione nel momento in cui riceve GNT, ma deve aspettare che /devsel vada a 1 (infatti il filo /frame ad 1 indica che siamo all'ultima operazione di accesso, ma questa potrebbe andare avanti). Se non ci fosse un arbitro, ma ci basassimo solo sul filo /frame, due dispositivi potrebbero iniziare un'operazione contemporaneamente quando questo va ad 1, causando problemi elettrici.

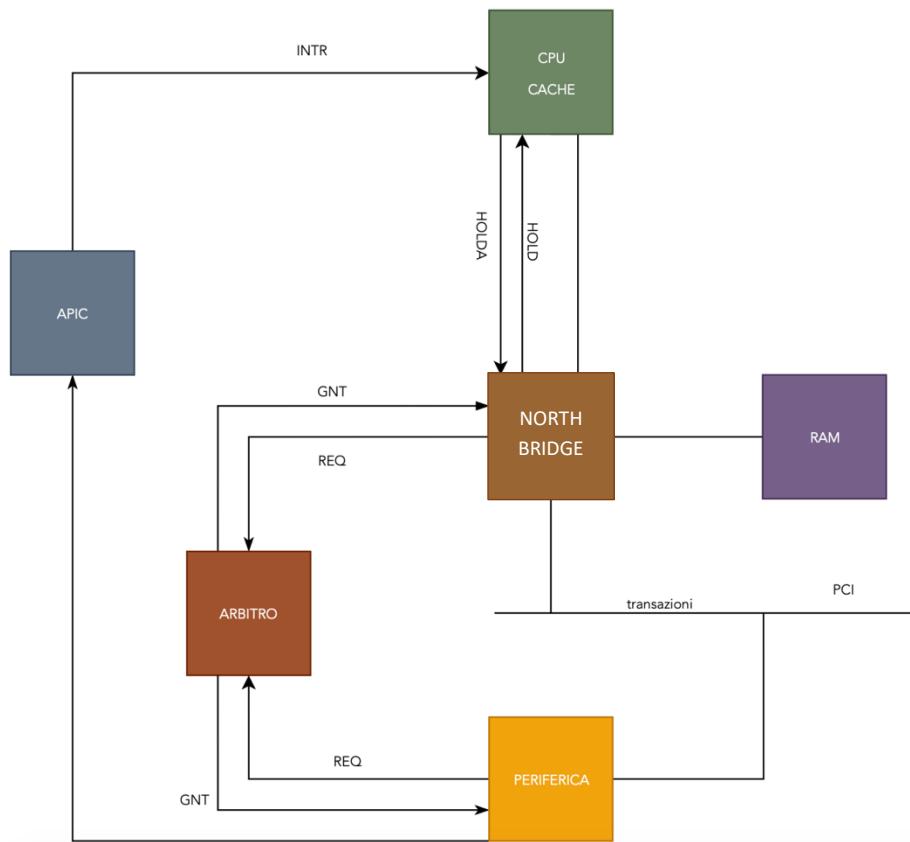
Per quanto riguarda il front bus, ciascuna scheda madre agisce autonomamente. Un metodo standard è quello di avere un handshake tra ponte e CPU, con /hold e /holda, in modo che questa metta i fili in alta impedenza quando il ponte vuole fare delle operazioni. Una soluzione più moderna, utile anche per altri aspetti, come vedremo più avanti, è quella di collegare la RAM direttamente al ponte, in modo che questo intramezzi anche gli accessi della CPU/cache alla RAM.

Dal punto di vista software, il DMA si realizza programmando una periferica, si pensi all'hard disk, affinché sposti dei dati verso un'area di memoria. Alla fine dell'operazione, magari può mandare un'interruzione, per segnalare che ha terminato. Ma, se il ponte scrive in RAM per conto di una periferica troppo spesso, non è che la CPU si trova in uno stato di attesa, per cui non riesce ad accedere alla memoria neanche per prelevare delle istruzioni? Per questo ci viene incontro la cache, che anticipa le operazioni di lettura sui byte immediatamente successivi, spesso necessari nel breve periodo. Per tale motivo, il front bus è quasi sempre libero, e utilizzabile per il DMA.



Allo stesso tempo, la presenza della cache crea dei problemi di carattere funzionale. Consideriamo la lettura. Se una periferica aggiorna la RAM, e in particolare una porzione che la cache ha attualmente memorizzato, ci troviamo con un disallineamento tra le due parti, e la cache non solo fornirà alla CPU dati errati quando le vengono richieste, ma non ne è neppure consapevole. Infatti, quando abbiamo costruito la cache, siamo partiti dal presupposto che la CPU fosse l'unica a poter modificare la memoria, e che quindi il controllore captasse ogni sua modifica, tenendo aggiornate le cacheline. Ci sono dei problemi pure per quanto riguarda il write back. Se voglio scrivere dei dati sull'hard disk, li devo comunicare effettuando una scrittura da qualche parte: ma se la cache non fa passare la scrittura in memoria, aspettando che si abbia un conflitto sull'indice per aggiornarla, non si può fare quanto voluto. Oppure, se c'è una cacheline sporca che si vuole rimuovere per far spazio ad un altro, la scrittura della cache in memoria potrebbero rimuovere il contenuto di qualcosa scritto dal dispositivo quando ancora la cacheline era memorizzata.

Il senso di questo discorso è che il controllore della cache deve comunicare con il ponte PCI, per sapere quali operazioni sono effettuate in RAM e se serve aggiornare una cacheline (o, eventualmente, invalidarla). Una possibile soluzione è quella di far sì che la cache capti le operazioni sul front bus e si adeguhi di conseguenza. Tuttavia, non è sufficiente se il ponte volesse fare delle letture. Se il ponte accede alla RAM, per leggere dei byte, ma questi, modificati dal processore, sono rimasti in cache ancora da trasportare in memoria, il dispositivo si ritrova con dei dati inconsistenti. In conclusione, il ponte, prima di fare un'operazione sulla memoria, interroghi la cache, si faccia dare dei dati da questa se presenti, oppure li vada a prendere, senza problemi in RAM. La struttura davanti a cui di troviamo è la seguente:

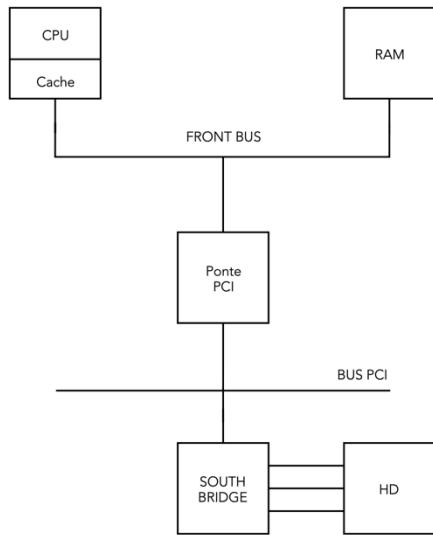


C'è un altro problema. Supponiamo che una periferica stesse effettuando delle operazioni di scrittura in RAM, e che ad un certo punto, avendo finito, mandi la richiesta di interruzione. L'APIC l'accetta e parte la routine associata. Siamo sicuri che i dati siano effettivamente in RAM? Infatti, potrebbe essere che il ponte, dovendo gestire anche altre periferiche, non abbia ancora effettuato alcune delle operazioni verso la RAM. Una soluzione, complicata, potrebbe essere via hardware: l'APIC dovrebbe comunicare con il ponte, per sapere se le operazioni della periferica sono concluse prima di inviare la richiesta al processore. L'altra, più immediata, è software. Lo standard prevede che il ponte abbia un buffer per le operazioni di lettura e scrittura, che esegue con modalità FIFO. Il software, per essere sicuro che le operazioni riguardanti la periferica siano terminate, può chiedere l'accesso ad un registro della stessa, magari uno di stato. Questa richiesta va in coda al buffer del ponte, e prima o poi viene eseguita. A questo punto, il ponte deve aver eseguito tutte le operazioni venute prima (e, in particolare, prima della richiesta di interruzione), per cui i dati si trovano sicuramente in RAM. Tale via è versatile anche perché quasi sempre il processore si dovrà interfacciare con qualche registro della periferica, anche solo per comunicare che la richiesta di interruzione è stata accettata.

L'hard disk di suo già permetteva il DMA, per mezzo di un controllore ad hoc: esso, tuttavia, non è un dispositivo PCI, e deve essere intramezzato da un ulteriore elemento. Nel calcolatore emulato da qemu, troviamo un ponte, detto *south bridge*, che fa da collegamento tra il bus e le periferiche che non erano PCI. Al suo interno, così come avveniva per il controllore DMA, troviamo un bus master PCI, nel quale si specificano la destinazione/sorgente dei byte da scrivere/copiare e il loro numero, poi si imposta il settaggio corretto all'hard disk, così come l'avevamo già visto, e si avvia l'operazione. L'hard disk lancerà un'interruzione al termine dell'operazione. Le specifiche di funzionamento si trovano [qui](#).

Finora non ci siamo accorti che le operazioni di `in` e `out` fossero realizzate dal south bridge, al quale è collegato l'hard disk e le altre periferiche viste, perché tutte sono comunque intercettate dal ponte PCI. Nello specifico, esso distingue alcuni indirizzi che hanno a che fare con le periferiche PCI, con altri indirizzi che rimangono standard per alcune periferiche. Per esempio, l'indirizzo della tastiera nello spazio di I/O resta sempre lo stesso, anche se nel mezzo c'è l'intervento del ponte PCI.

L'uso di una periferica di questo tipo impone numerose limitazioni. Ad esempio, il bus master si dovrà tenere un registro nel quale immagazzinare l'indirizzo nel quale si dovrà scrivere il prossimo dato. Possiamo immaginare l'indirizzo su 32 bit, nel caso in cui siano sufficienti per accedere alla RAM disponibile. Il problema è che, allo stesso tempo, il dispositivo ha un sommatore a 16 bit per incrementare tale registro. Quindi, nel momento in cui i 16 bit meno significativi sono ad 1, si ritorna all'inizio della regione naturale di 64KiB in cui erano contenuti, non si va in quella successiva. Il programmatore ne deve essere consapevole, impostando come destinazioni delle regioni che possano contenere i dati da copiare interamente. Inoltre, questo è uno di quei casi in cui bisogna distinguere il momento in cui viene lanciata un'interruzione e quello in cui viene completata la scrittura in RAM. I dati, infatti, dall'HD devono andare in SB, poi in NB e infine in RAM, ma il tempo che passa dall'inizio alla fine potrebbe essere considerevole. In questo caso lo standard prevede che l'interruzione dell'hard disk sia recepita anche dall'SB. A questo punto, si deve fare una lettura nel registro di stato dell'HD, come *acknowledgment* della richiesta di interruzione proveniente dall'hard disk, per capire lo stato dell'operazione. L'SB non fa proseguire la richiesta finché non sono stati copiati tutti i dati dell'HD in RAM. Questo è possibile perché anche l'SB riceve l'interruzione. In questo modo, la gestione FIFO dei dati fa sì che, quando la CPU ha ricevuto il valore del registro di stato, le operazioni con la RAM siano definitivamente terminate.



Per capire bene il collegamento tra le varie parti, è bene osservare quali soggetti fanno cosa e quali sanno cosa. Ad esempio, la visione globale delle operazioni è concessa solo al software, mentre la CPU non sa nemmeno che sta avendo luogo un'operazione DMA (non solo perché non ne è artefice, ma anche perché è senza memoria). Il ponte PCI vede solo le transazioni che passano sul ponte e che lo riguardano come target.

Sulle eccezioni

Le eccezioni sono un meccanismo simile alle interruzioni, ma presente già da prima. L'idea è la stessa, far eseguire una routine quando si verifica un evento, ma, nel caso delle eccezioni, gli eventi sono interni al processore stesso. Per tale ragione, le interruzioni provenienti dall'APIC sono dette **esterne**, riguardando l'ingresso INTR. Per esempio, mentre il processore fa una divisione, si potrebbe accorgere che il divisore è 0 e sollevare un'eccezione. Parte allora un microprogramma allo stesso modo con cui si verifica un'interruzione esterna: c'è un tipo associato e si trova il gate da attraversare tramite la IDT.

Nelle eccezioni esterne il tipo proviene dall'APIC, nelle interruzioni interne è implicito ed associato all'eccezione. Sul manuale dell'Intel si trova scritto che i primi 32 ingressi di IDT sono riservati per le eccezioni. Questo è il motivo per cui i tipi da associare alle interruzioni esterne devono avere un valore maggiore di 32. In effetti, nell'APIC i primi 32 bit di IRR e ISR non sono usati, essendo tipi riservati alle interruzioni interne. La gestione poi è la stessa di quelle delle interruzioni, con il salvataggio in pila dei registri e le modifiche ai flag dovuti ai bit della riga di IDT. Le eccezioni possono essere di tipo diverso, a seconda del valore di %rip che salvano in pila. Nel tipo è già implicito il tipo di eccezione.

Le eccezioni possono essere di 3 tipi:

1. **TRAP.** Salva l'instruction pointer dell'istruzione successiva. È un'eccezione che si verifica tra due istruzioni, oltre il quale si riprende l'esecuzione.
2. **FAULT.** Si verifica durante un'istruzione: viene salvato proprio il %rip dell'istruzione che ha causato il fault. L'idea per questa modalità è che il problema potrebbe essere risolto: il programmatore salta ad una routine in cui si sistemano le cose, poi si esegue nuovamente l'istruzione sperando che il fault non si verifichi più. Per il processore, implementare tale semantica richiede un certo costo, perché, quando si accorge che c'è un fault, già sono stati modificati dei registri, quindi si deve tornare allo stato precedente all'esecuzione. Una soluzione per il processore è quella di aggiornare i registri in modo definitivo solo alla fine dell'istruzione, usando nel frattempo delle copie di lavoro: se si verifica un fault, scarta le copie di lavoro.
3. **ABORT.** È una situazione più anomala. Il processore non può andare avanti, e quello che accade è indefinito. Non avendo una routine associata, molto spesso il processore si spegne.

Come esempio possiamo scrivere un programma che faccia una divisione per 0 nella macchina virtuale. Nei sistemi operativi, le interruzioni interne sono già captate dal sistema operativo. Nel nostro caso, bisognerà anche associare una routine. Lanciando il programma con qemu, troviamo nel terminale “Eccezione 0”. Questo perché libce già inizializza IDT all'avvio. È stata dichiarata una struttura tabulare con 256 entrate ciascuno da 16 byte. La funzione init_idt la inizializza con una serie di valori opportuni per ciascun gate delle eccezioni, sfruttando una macro. Il nome delle funzioni richiama il nome dell'eccezione presente sul manuale. La divide_error, per esempio, passa un po' di parametri alla funzione che stampa poi un messaggio: c'è il tipo, che tanto è fissato per le eccezioni, e l'errore. Alcune eccezioni salvano in pila una quad contente un errore, con alcune informazioni sul problema. Per uniformare, se non c'è le funzioni di gate passano 0. Si estrae poi %rip dalla pila per mostrarlo. Non essendo più in pila, non si può tornare al chiamante: in effetti, la strategia della libce è quella terminare al presentarsi di un'eccezione. Il messaggio di errore non appare sul monitor, ma sul terminale: è l'output che la macchina virtuale avrebbe lanciato alla sua porta seriale. Per stamparlo, si usa la funzione flog, che ha lo stesso funzionamento di una printf. La divisione per 0 non è di tipo abort, perché salta alla divide_error e non si spegne. Tramite objdump, vediamo che il valore di %rip salvato in pila è quello dell'istruzione di divisione: siamo quindi nel caso di un'eccezione fault.

C'è un'istruzione del processore che ha il solo scopo di generare un'eccezione trap, **int3**. Con objdump vediamo che int3 si trova all'indirizzo 0x200156. Ha codice operativo 0xcc, occupa un byte, ed essendo un'eccezione di tipo trap in pila troviamo %rip con valore 0x200157; come tipo ha 3.

Come si intercetta un'eccezione, in modo che il programma faccia quello che vogliamo? La modalità è la stessa delle interruzioni, con la sola differenza che non possiamo scegliere liberamente il tipo. Vediamo un esempio.

L'eccezione numero 1 (debug) è sollevata alla fine di ciascuna istruzione se il Trap Flag, TF, è settato. Si parla di *single step*, proprio perché una certa azione è eseguita per ogni istruzione.

```

1 #include <libce.h>
2
3 extern "C" void enable_single_step();
4 extern "C" void disable_single_step();
5 extern "C" void a_debug();
6 extern "C" void c_debug(void *rip)
7 {
8     flog(LOG_DEBUG, "rip=%p", rip);
9 }
10
11 int main()
12 {
13     int x = 0;
14     gate_init(1, a_debug);
15     enable_single_step();
16     x++;
17     x++;
18     x++;
19     x++;
20     disable_single_step();
21     pause();
22     return x;
23 }
24
1 #include "libce.s"
2 .global enable_single_step
enable_single_step:
3     pushf
4     orw $0x100, (%rsp)
5     popf
6     ret
7
8 .global disable_single_step
disable_single_step:
9     pushf
10    andw $0xFEFF, (%rsp)
11    popf
12    ret
13
14 .global a_debug
a_debug:
15     salva_registri
16     movq 120(%rsp), %rdi
17     call c_debug
18     carica_registri
19
20     iretq

```

Per abilitare il *single step*, e quindi settare TF, non c'è l'analogo di cli e sti. Per farlo, si usano le istruzioni pushf e popf, che fanno push e pop del registro dei flag: si salva in pila il registro, si modifica il bit opportuno e si fa una pop. Per disabilitarlo, il contrario. La funzione che intercetta le eccezioni deve essere sulla falsa riga di quelle delle interruzioni: deve salvare tutti i registri, riprenderli e terminare con iretq. Con gate_init associamo tale funzione, scritta in Assembler, al gate 1 e, nel momento in cui chiamo enable_single_step, a_debug è eseguita per ogni istruzione, mostrando il valore di %rip. Questo tipo di esecuzione può essere utile per realizzare un Debugger. Essendo un'eccezione di tipo Trap, salva il valore successivo. Più in particolare, l'eccezione viene generata *sulla base del valore di TF all'inizio dell'istruzione*. Quindi popf di riga 6 non fa partire l'eccezione, perché all'inizio TF era a 0; RET, al contrario, si, iniziando con TF già ad 1. Per lo stesso motivo, dentro disable_single_step l'eccezione viene sollevata anche dopo popf. Allo stesso tempo, quando viene sollevata l'eccezione si salta alla nostra routine, ma per queste istruzioni non sono lanciate le relative eccezioni: infatti, il TF viene azzerato passando il gate, senza possibilità di scelta. Il suo valore precedente va a finire nel salvataggio del registro dei flag in pila, e viene ripristinato nella iretq. Questo ha senso sia perché altrimenti entreremmo in un ciclo, sia perché il debug è interessato alle istruzioni del programma, non a quelle che gestiscono il debug. Questo spiega anche come mai TF viene testato prima dell'istruzione: in caso contrario, la iretq di riga 20 lancerebbe un'eccezione, pur facendo ancora parte della routine.

Anche int3 serve al Debugger, per implementare i checkpoint. Dove mettiamo un CP, il Debugger inserisce l'istruzione int3: è di un byte affinché possa sostituire qualsiasi istruzione nel processore. È come se fosse una trappola per l'esecuzione: incontrando int3, il controllo passa nelle mani del Debugger.

Istruzione di partenza: 3e 43 1f 10

Istruzione dopo la modifica del Debugger: cc 43 1f 10

Istruzione al proseguimento del programma: 3e 43 1f 10

Quando il Debugger ha finito, e l'utente fa proseguire il programma, l'istruzione deve essere posta al valore precedente, perché sia eseguita normalmente. Allo stesso tempo, essendo di tipo Trap, devo portare indietro di 1 %rip, per eseguire l'istruzione che avevo sostituito. In questo modo però è sparito il checkpoint, e se il programma ci ripassasse non lo incontrerebbe. Questa situazione si può risolvere con il Trap. Flag. Quando intercettiamo l'eccezione 1, si prende il primo byte dell'istruzione eseguita e si sostituisce con int3. Questo è possibile se il Debugger imposta il *single step* subito prima della fine dell'istruzione, e lo disabilita al termine della sostituzione: facendo tutto questo, la trappola torna dove stava. Naturalmente, serve salvare i punti in cui ci sono i checkpoint.

<pre> 1 #include <libce.h> 2 void foo() { 3 printf("foo()\n"); 4 } 5 6 natb old; 7 extern "C" void a_debug(); 8 extern "C" void c_debug(void *rip){ 9 natb *p = static_cast<natb*>(rip); 10 p--; 11 flog(LOG_DEBUG, "breakpoint all'indirizzo %p", p); 12 *p = old; 13 } 14 15 extern "C" void a_singlestep(); 16 extern "C" void c_singlestep(){ 17 natb *p = reinterpret_cast<natb*>(foo); 18 old = *p; 19 *p = 0xCC; 20 } 21 22 int main(){ 23 gate_init(1, a_singlestep); 24 gate_init(3, a_debug); 25 natb *p = reinterpret_cast<natb*>(foo); 26 old = *p; 27 *p = 0xCC; 28 foo(); 29 foo(); 30 pause(); 31 return 0; 32 }</pre>	<pre> 1 #include "libce.s" 2 .global a_debug 3 a_debug: 4 salva_registri 5 movq 120(%rsp), %rdi 6 call c_debug 7 carica_registri 8 orw \$0x100, 16(%rsp) 9 decq (%rsp) 10 iretq 11 12 .global a_singlestep 13 a_singlestep: 14 salva_registri 15 call c_singlestep 16 carica_registri 17 andw \$0xFEFF, 16(%rsp) 18 iretq </pre>
---	---

Nell'esempio, il puntatore p di main punta al primo byte della funzione foo. Questo viene sostituito con 0xCC per inserire la trappola, salvandosi anche il precedente valore. All'esecuzione di foo, si gate nella trappola passando il gate 3: si ricava dalla pila il valore di %rip, e si passa alla funzione c_debug che ne stampa il valore. Subito dopo, si decrementa il valore di %rip in pila e si crea nuovamente l'istruzione nella

quale era stata messa la trappola. Al termine di a_debug, si importa il TF, che finirà nei flag a seguito della iretq. Viene eseguita la prima istruzione di foo. Terminata, TF è ad 1, quindi si chiama a_singlestep: la trappola viene nuovamente inserita, e TF resettato.

Sulla protezione

La protezione nasce dal voler usare dei grossi computer per eseguire più programmi contemporaneamente. Potremmo eseguire un programma alla volta, ma, se si volesse fare una stampa, il processore starebbe fermo ad aspettare, mentre potrebbe eseguire una nuova computazione. L'idea che viene a molti è dire: è inutile che il processore aspetti durante questa stampa, il nastro è in grado di trasferire dati da solo, magari con DMA; piuttosto, allo stesso tempo potrebbe iniziare a eseguire un altro programma. Perché questa cosa funzioni, vogliamo che mentre il processore è sotto il controllo di P1, in qualche modo passi ad eseguire P2, e, arrivati ad un certo punto, tramite un'interruzione, il controllo ritorni a P1.

Tuttavia, non solo i programmi non si conoscono, ma potrebbero anche effettuare operazioni che compromettono questa procedura: P1 non ha interesse a dare il controllo a P2, che a sua volta farebbe di tutto pur di non lasciarlo a P1. Può il *sistema operativo* (o gli operatori che un tempo caricavano i programmi sul calcolatore) costringere che il passaggio tra l'uno e l'altro accada indipendentemente dalla volontà dei due? Supponiamo che le interruzioni (che permettono tale passaggio) si potessero disabilitare solo con `cli`: l'operatore potrebbe dire che un programma non abbia `cli` al suo interno. Ma non è detto che si sia scritto `cli` perché a run time venga eseguito `cli`. Non è facile cercare queste cose dentro un programma. Dobbiamo allora farci aiutare dall'hardware. Il software, infatti, è sotto il controllo dell'utente, e non ci possiamo lavorare.

Per far ripartire un programma dal punto in cui era arrivato si tratta soltanto di salvare lo stato di tutti i registri e della memoria; supponiamo si saperlo fare. Il problema è convincere P1 a salvare il suo stato e caricare P2, invece che stare ad aspettare la fine. Ovviamente, non dobbiamo far sì che il computer faccia quello che vogliono gli operatori: se il programma facesse dei calcoli, non avrebbe senso interromperlo! Gli utenti devono essere liberi, senza che siano concessi loro determinati privilegi. Allora, l'hardware deve distinguere tra codice degli utenti e codice degli operatori: il primo può fare alcune cose, quello degli operatori può fare tutto. Otteniamo così due modalità del processore, e un meccanismo per poter passare dall'una all'altra.

Quando il codice degli utenti è in esecuzione, dobbiamo essere in modalità utente: alcune istruzioni sono vietate, come `cli`, `in` e `out`. Vietate significa che l'utente le può inserire o cercare di eseguirle, ma il processore solleva un'eccezione di protezione. Tuttavia, vietando `in` e `out` l'utente non può comunicare con l'unità nastro direttamente: se ci vuole parlare, deve usare delle routine che l'operatore fornisce. Nel sistema allora c'è già il codice degli operatori, al pari di una libreria, che l'utente può chiedere di eseguire. Il salto a tale codice può avvenire per tre motivi diversi. 1. Interruzione esterna; 2. Eccezione; 3. Invocazione di una routine del sistema; in tutti questi casi, il processore si porta in modalità sistema, ed esegue il codice degli operatori. Visto che l'utente può eseguire una routine di sistema, allora tanto vale associare questa cosa ad un'interruzione: si usa l'istruzione `int`, che lancia un'interruzione dato il tipo del gate da attraversare.

Queste tre possibilità riguardano IDT: in effetti, passare da un gate, è l'unico modo per innalzare il livello di privilegio. Come mai servono le interruzioni, e non si può fare una `call` (ad esempio, dicendo che il privilegio si alza saltando in una zona protetta)? Nella `call` l'utente sceglie dove saltare, nell'istruzione `int` tipo inserisce solo il gate, non l'indirizzo: l'utente potrebbe saltare a metà di una routine, provocando danni. Nel salire di privilegio, il punto in cui si salta deve essere scelto da chi ha scritto il codice, non da chi sta a livello di privilegio più basso.

La protezione è sempre implementata allo stesso modo nelle diverse architetture, dovendo gestire sempre pressoché le stesse cose. I controlli devono essere aggiunti direttamente alla CPU, non possiamo

pensare di inserirli lato software, e sono sufficienti due modalità: in quella utente ci sono dei limiti sulle istruzioni eseguibili e sugli indirizzi ai quali è possibile accedere. L'utente è limitato su tre elementi fondamentali dell'architettura, la CPU, la RAM e l'IO. Serve anche un modo per passare da livello utente a sistema e viceversa. Se ciò non accadesse, avremmo creato un sistema che non è in grado di fare certe cose per nessuno. L'obiettivo finale non è proteggere il sistema, o il codice scritto dagli operatori: vogliamo proteggere un utente da un altro, senza che il programma P2 si tenga il controllo della CPU impendendo a P1 di proseguire il proprio lavoro.

In prima istanza, per la RAM, si può pensare che nella CPU ci sia un registro che dica che gli indirizzi minori di un certo valore riguardano la modalità sistema, e non vi si può accedere; se siamo in modalità sistema, non si tiene conto di questa limitazione. Allo stesso tempo, se carichiamo *job1* e *job2* degli utenti, P2 potrebbe modificare la memoria di P1, sia volontariamente che involontariamente; dovremmo proteggere anche un utente dall'altro. La soluzione più immediata è quella per cui, nel passaggio da P1 a P2, tutto il contenuto della memoria di P1 sia copiato sull'HD, cancellato e sostituito con la memoria di P2. In questo modo, non ci sono mai contemporaneamente P1 e P2 in memoria.

L'Intel ha introdotto la protezione nel 286, quando ancora era a 16 bit, ispirandosi ad un'architettura derivata da *Multics*, un sistema operativo dell'MIT. Nel 286 c'erano quattro livelli di privilegi e un'architettura segmentata (ogni indirizzo ha un identificatore di segmento e un offset, dove per segmento si intende un intervallo di indirizzi. Lo spazio di memoria di un programma è composto da tanti segmenti, e ogni volta che si vuole riferire un byte si devono specificare i due campi), che non è stata portata avanti se non per compatibilità. L'architettura che si preferisce è quella standard senza segmenti, detta *flat*. I 4 livelli di privilegio sono stati ricondotti a 2 nel 386, introducendo il meccanismo della paginazione. Il processore sa a quale livello di privilegio si trova a seconda del segmento codice che sta usando.

Nel processore Intel c'è il registro CS (*code selector*). Oltre all'identificatore del segmento codice corrente, gli ultimi due bit contengono il livello di privilegio. Non interessandoci della segmentazione, chiamiamo questi bit CPL (*current privilege level*). Inoltre, ci dimenticheremo che sono 4, lavorando con due soli livelli di privilegio: CPL assumerà i valori 00 (sistema) o 11 (utente). L'unico modo per passare questi livelli di privilegio è di passare tramite le interruzioni nella loro generalità (esterne, interne o software con `int`), con un gate di IDT. Allo stesso modo, l'unico modo per tornare da modalità sistema ad utente è quello di usare l'istruzione `iretq`. All'avvio, ci troviamo in modalità sistema.

Il punto di partenza è il tipo di un'interruzione generalizzata, che mi permette di identificare un'entrata della IDT. Le informazioni che vi troviamo sono:

- L'indirizzo della routine da mandare in esecuzione.
- Un bit **I/T** che specifica il tipo del gate, se interrupt o Trap.
- Flag **P**, che dice se l'**entrata è valida oppure no**. Potenzialmente ci sono 256 entrate, ma non tutte sono implementate: le entrate valide hanno il bit P ad 1, altrimenti a 0. In tal caso, non si può attraversare il cancello, e si genera un'eccezione per *gate non implementato*. Ovviamente, questa eccezione è indipendente dall'interruzione per cui si è generata. Il programmatore di sistema avrà implementato questo gate prevedendo un comportamento adeguato.
- Mi serve sapere il **livello di privilegio successivo all'attraversamento del gate**, e quindi il valore da inserire in CPL. Non è scritto esplicitamente, ma troviamo un selettore del segmento codice a cui si trova la routine. Tali segmenti sono descritti in un'altra tabella, GDT. Qua dentro si trova il livello di

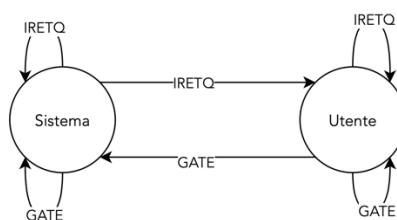
privilegio L. Di fatto, è come se il livello di privilegio si trovasse direttamente nella IDT. Non necessariamente il livello si alza.

- DPL, *descriptor privilege level*, dice il livello di privilegio minimo perché il processore attraversi il cancello con una int. Si tratta quindi del privilegio prima dell'esecuzione della routine. Visto che dentro IDT ci sono le routine di risposta a tutti i tipi di interruzione, niente potrebbe vietare all'utente di chiamare il driver della tastiera con una int. Allo stesso tempo, abbiamo scritto supponendo che fosse la tastiera a lanciare l'interruzione (per esempio quando c'è un carattere da prelevare), non l'utente. Avendo DPL sistema, l'utente non può chiamarle (al più i programmi di sistema, ma non saranno interessati ad operazioni del genere).

Se il livello di privilegio di alza, il processore cambia pila. Questa operazione si verifica confrontando CPL con il valore di privilegio prelevato dalla IDT. In questa nuova pila, salva il **selettore pila vecchio** (legato alla segmentazione), il **vecchio %rsp**, il **registro dei flag**, il **vecchio CS**, che contiene CPL, e il **vecchio %rip**. Salvate queste informazioni, dal gate si preleva il nuovo %rip e il nuovo CS; alcuni flag si modificano, come Trap Flag che si resetta o IF sulla base del tipo di gate. Ma qual è il senso cambiare pila? Il processore fa quest'operazione di default, dando per scontato che ne abbia bisogno prima ancora di una qualunque operazione via SW (che potrebbe comunque cambiare pila). Il problema è che l'utente potrebbe costringere il sistema a scrivere dove vuole lui, modificando banalmente %rsp. È fondamentale che, nel momento in cui il processore si innalza di livello e necessita di usare la pila, non si fidi del valore lasciato dall'utente. Più in generale, l'idea della protezione è che *non ci si può fidare di ciò che accade lato utente*. È possibile che da sistema si rimanga a livello sistema, e in questo caso non si cambia pila, perché è fidata.

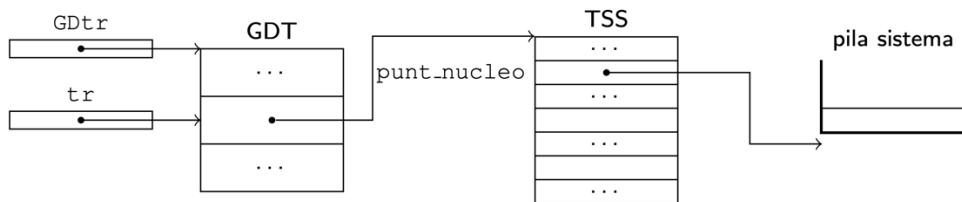
La iretq disfa quello che ha fatto il meccanismo delle interruzioni generalizzato. Prende dalla pila corrente, puntata da %rsp, il %rip a cui saltare, CPL, RF, %rsp e il selettore pila. Confronta CPL prelevato e corrente: il livello a cui si ritorna deve essere minore o uguale di quello corrente. A livello sistema si può tornare sia a livello utente che sistema, ma a livello utente non si può andare a quello utente. In generale, con una iretq il livello di privilegio non può aumentare. Senza questa limitazione, si potrebbe aggirare la protezione modificando %rsp e i valori da esso puntati, chiamando poi una iretq: il CS sarebbe quello impostato da noi, con la possibilità di passare da livello utente a sistema.

Il gate è realizzato in simmetria con la iretq: si può passare da livello utente a sistema, ma non viceversa. La situazione opposta non sarebbe credibile: il gate è come una call, e passando ad un livello di privilegio più basso, non è detto che il controllo ritorni, come invece ci si aspetterebbe. Non considereremo i casi in cui si passa da livello utente a utente con un gate.



Nei flag c'è anche IF. Potremmo usare iretq, popf e pushf per modificarlo e disabilitare le interruzioni. Questo è un difetto di avere IF all'interno del registro dei flag. Si risolve imponendo che tali istruzioni non permettano di sovrascrivere alcuni dei flag del registro.

Soffermiamoci sul meccanismo del cambio della pila al momento di un'interruzione che effettui il passaggio da privilegio utente a sistema. In una prima versione della protezione, il cambio di processo era realizzato interamente lato hardware: in una tabella GDT (*global descriptor table*) sono presenti tutte le informazioni necessarie per riprendere l'esecuzione di ciascun task, in puntatori a campi TSS (*task state segment*). In questi campi c'è anche un `punt_nucleo` che punta alla pila sistema da usare per cambiare. Ragionevolmente, avendo un processo corrente, c'era bisogno di un registro di sistema che puntasse il TSS corrente: questo è TR (*task register*), che contiene l'offset dell'entrata della GDT che lo punta. Nella versione originaria, il sistema aggiorna TR ad ogni cambio di task. Visto che a noi interessa solo l'indirizzo della pila sistema da adottare, useremo un unico TSS, senza mai modificare TR, nel quale caricheremo, ad ogni cambio di processo, l'indirizzo della pila sistema da prendere.



Le istruzioni `in`, `out`, `cli` e `sti` non sono automaticamente vietate. Nel registro dei flag c'è IOPL (*I/O privilege level*), che può assumere i valori 00 o 11. Questo campo dice quale livello di privilegio bisogna avere per eseguire queste operazioni. Per non farle usare, gli utenti devono trovarsi sistema in questo campo. È un'informazione statica, non cambia nel passaggio dei gate, e, al pari di IF, si deve impedire che l'utente la possa modificare.

Trovandoci di default a livello sistema, l'unico modo per passare a livello utente è eseguire una `iretq`. Serve una funzione che modifichi la pila con informazioni che, una volta prelevate con una `iretq`, mi portino in modalità utente. A questo punto, le operazioni che non mi sono autorizzate causeranno il sollevarsi di un'eccezione (ad esempio la 13, per delle istruzioni non autorizzate).

```

1  # modifichiamo la pila in modo da poter eseguire una iretq che porti il
2  # processore a livello utente. Vogliamo tornare comunque al chiamante e senza
3  # passare ad una nuova pila, quindi effettuiamo la seguente trasformazione:
4  #
5  #          +-----+
6  #          | rit   |
7  #          +-----+
8  #          | cs usr |
9  #          +-----+
10 #         ==> | rflags |
11 #          +-----+
12 #          | rsp .----+|
13 # +-----+     +-----+ | |
14 # | rit    |     | ss usr | | |
15 # +-----+     +-----+ | |
16 #                      <-+
17 #
18 .global liv_utente
19 liv_utente:
20     popq %rax      # rit
21     movq %rsp, %rdi
22     pushq $0x1b    # selettore dati utente (ss usr)
23     pushq %rdi     # rsp salvato in precedenza
24     pushq $0x3200  # rflags con IF=1 e IOPL=sistema
25     # provare anche con
26     # $0x3200 (IF=1 e IOPL=utente)
27     pushq $0x13    # selettore codice utente (cs usr)
28     pushq %rax    # rit salvato in precedenza
29     iretq
30
31 # per tornare a livello sistema modifichiamo le informazioni salvate in pila
32 # dalla int 0x70 in modo che 'cs' sia quello sistema (0x8) e non quello utente
33 # (0x13). A questo punto eseguiamo iretq. Questo non causerà eccezioni di
34 # protezione perché il ritorno da livello sistema a livello sistema è ammesso.
35 .global liv_sistema
36 liv_sistema:
37     movq $0x8, 8(%rsp)
38     movq $0, 32(%rsp)
39     iretq
40

```

Sui processi

Per “*sistema multi-programmato*” intendiamo un sistema in grado di eseguire più processi. Chiariamo i termini processo, primitiva e contesto. Un processo è un programma in esecuzione su dei particolari dati in ingresso.

Pensiamo ad un pizzaiolo inesperto che fa la pizza, e che ha bisogno di una ricetta. In vari passaggi passa dalla pasta alla pizza: il programma è la ricetta, un testo di ordini da fare per il pizzaiolo, e può essere usata per creare più pizze, anche contemporaneamente.

Il pizzaiolo è chiaramente il processore, la pizza i dati, manipolati per ottenere il risultato finale. Nella metafora, il processo è il filmato di tutto ciò che accade dall'inizio fino alla fine, la sequenza di tutti gli stati attraverso cui il sistema passa. In ogni istante ci troviamo davanti ai dati semilavorati, e ogni istruzione eseguita modificherà in qualche modo la memoria o i registri. Il processo ha quindi una realtà temporale. Lo stesso programma può essere eseguito da più processi contemporaneamente, così come un processo può eseguire più programmi contemporaneamente. Nella metafora, una pizza è richiesta da più clienti, e il pizzaiolo può fare più pizze contemporaneamente, spostando la sua attenzione da una pizza ad un'altra. Ogni fotogramma di un processo contiene tutte le informazioni necessarie affinché possa proseguire: se il pizzaiolo fa più pizze, di ciascuna si ricorda a che punto è arrivato, anche se deve momentaneamente passare ad un'altra (il processore non è così furbo, se guarda i dati non sa cosa deve fare, ma ha bisogno di alcune informazioni in più, come `%rip` che salva l'istruzione successiva da eseguire). Nello specifico, per il processore il contenuto dei registri e della memoria è sufficiente per portare avanti un processo.

Non si confonda programma e processo: il programma è la sceneggiatura del processo, ma l'esecuzione si adatta a delle varianti, dovute magari ai dati in ingresso. Se nel programma c'è un ciclo scritto testualmente, nel processo le istruzioni sono eseguite tante volte. Allo stesso tempo, un processo non è (solo) un programma in esecuzione, perché potrebbe eseguire più programmi: l'unica cosa certa è in ogni istante esegue uno specifico programma. Un processo è sequenziale, istanziando in tal senso tutti quei costrutti che nel programma prevedevano dei salti in avanti o indietro, come i cicli e le condizioni.

Una primitiva è una funzionalità di base che il sistema fornisce, non ulteriormente scomponibile. Sono come delle funzioni di librerie, utilizzabili ma i cui passi non possono essere scomposti. Sono realizzate da chi ha più privilegi di noi, e usate per costruire qualcosa di superiore. Le primitive si realizzano come chiamate di sistema: il processore è in grado di eseguire del codice a livello sistema tramite una `int`, e una volta passato il gate si può eseguire la primitiva con un privilegio superiore. Il nucleo di cui ci occuperemo fornirà l'astrazione dei processi, eseguendo programmi al loro interno, e alcune primitive basilari: per esempio, potremo creare nuovi processi e gestirli, oppure eseguire operazioni di ingresso-uscita.

Per contesto si intende il significato complessivo dei dati. Pur avendo processi che proseguono simultaneamente, ognuno farà riferimento al proprio stato. Immaginiamo una porzione di programma con `mov 1000, %rax`, in esecuzione da due processi; quando lo eseguire il processo 1, sta parlando del suo `%rax`, così come del suo indirizzo `1000`, dove c'è una variabile che interessa a lui; se l'istruzione la esegue il processo 2, `1000` e `%rax` non riguardano più quelli precedenti. Questa distinzione si può realizzare associando un contesto ad ogni processo: `mov 1000, %rax` si interpreta diversamente sulla base del contesto in cui è eseguita.

La stessa idea si può usare per le primitive: `out` è lecita o illecita? Dipende da chi la esegue: se la usa una primitiva, con il privilegio sistema, va bene, altrimenti no, essendo preclusa in modalità utente. Anche quando un processo invoca una primitiva si può immaginare un cambio di contesto: si parla di **cambio di**

privilegio, per cambiare contesto tra processi abbiamo un **cambio di contesto**. In generale, bisogna essere consapevoli del contesto: lo stesso codice può fare o non può fare delle cose sulla base del contesto in cui quel codice è eseguito (non lo dice il codice stesso). Ad esempio, useremo `libce` sia a livello sistema che a livello utente.

Il sistema che andiamo a studiare è realizzato in moduli, ciascuno dei quali è un programma a sé stante eseguito in un livello di privilegio differente: i moduli *sistema* e *I/O* si eseguono con privilegio sistema, viceversa quello *utente*. I primi due moduli costituiranno il sistema operativo, mentre chi usa il computer scriverà i suoi programmi nel modulo *utente*, facendo uso delle primitive fornitegli. In *sistema*/ ci sono solo due file, *sistema.s* e *sistema.cpp*: una cosa scritta in Assembler o C++ da considerarsi per livello sistema deve finire in uno di questi due file. Lo stesso per quanto riguarda *io/*. Per l'utente si forniscono anche una serie di file che permettono di usare le primitive, assieme ad una libreria d'appoggio.

Siamo abituati a sistemi in cui ci troviamo già dentro il kernel, con gli strumenti per compilare; questo non accade in un sistema tanto semplice, e dobbiamo usare Linux per fare cross-compiling (compiliamo su Linux e lo facciamo girare sulla macchina virtuale). Usiamo il comando `make` per fare tale compilazione. *utente.cpp* sarà fornito dall'utente, scrivendo quanto si vuole fare. La scrittura su video, per come l'abbiamo già vista, richiede una `out`: essendoci preclusa, serve una primitiva, `writeconsole`. `terminate_p()` si usa per terminare il processo, dopo il quale il sistema termina. Una volta compilato, dentro la cartella `build/`, troviamo i moduli che avevamo inserito nella root. Il modulo *sistema* corrisponde al bootloader, che inizializza alcune strutture dati e prepara i moduli *I/O* e *utente*. Quando è tutto terminato, crea un nuovo processo e gli fa eseguire il nostro `main` a livello utente; da questo punto in poi, il modulo *sistema* si comporta come una libreria, pronto a rispondere a chiamate di sistema (primitive), eccezioni o interruzioni. Queste tre operazioni restituiscono il controllo al modulo *sistema*, che fa le sue operazioni con livello privilegio più alto e torna infine all'utente. Lo script per avviare si trova nella cartella nucleo, `./run`.

```

1 #include "all.h"
2
3 ↘ void main(){
4     writeconsole("Hello world\n",14);
5     pause();
6     terminate_p();
7 }
```

Ogni processo ha un identificatore, e i messaggi nel log lo usano per comunicare il processo da cui provengono. Ci rendiamo conto che siamo a livello utente provando a fare delle operazioni sul video senza far uso delle primitive, usando un puntatore a word per l'indirizzo `0xb0000`, la RAM della scheda video. Il compilatore non da errori, ma in esecuzione c'è il lancio dell'eccezione 14, di `page fault`. Stampa poi una serie di informazioni per capire lo stato del processo nel momento in cui si è verificato un errore. Il log è automaticamente decodificato con uno script (associa ad ogni indirizzo la riga nel codice dell'utente).

Il nostro programma è collegato con `libce`, quindi per scrivere su video usiamo lo stesso codice visto a suo tempo. Tuttavia, non è il codice ad essere privilegiato, ma il contesto: chiamare `char_write` a livello utente provoca un'eccezione, perché ha a che fare con una scrittura in un'area riservata; se invece è invocata con una primitiva (privilegio sistema) non si hanno problemi.

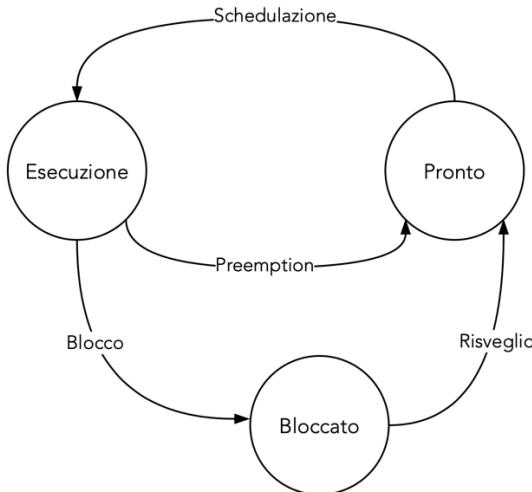
Nell'ambiente utente, è possibile attivare un nuovo processo con `activate_p()`: deve dire che funzione deve eseguire il processo, un parametro intero per la funzione, la precedenza e il tipo. In un sistema completo, si dovrebbe dire il programma da eseguire, ma in questo caso più ristretto si considerano le funzioni, che devono essere di tipo `void` (`int`). Il processo allora partirà eseguendo tale funzione, all'interno del proprio contesto. Il sistema che realizziamo esegue i processi in base alla loro precedenza, data da un intero: quando deve scegliere quale processo far andare avanti, sceglie sempre quello con la massima precedenza. Questa operazione è detta *schedulazione*, ed è una delle prerogative dei Sistemi Operativi. La priorità è uno dei tanti metodi di schedulazione. Il tipo di processo `LIV_UTENTE` serve per generalizzare la funzione, e permetterle, nel modulo sistema, di creare processi di livello sistema. La primitiva restituisce come valore l'identificatore del processo creato. `terminate_p()` termina il processo che la invoca; a sua volta, `main` sarà eseguita da un processo. Se inseriamo `f2` in un altro processo, anch'esso dovrà prevedere `terminate_p()`. `delay(x)` sospende il processo per `x` intervalli di tempo. Per far vedere che sono due processi diversi, `main` crea il processo `6`, questo va in esecuzione dopo la chiamata di `delay` per un po' per un fatto di priorità ma, chiamando `f1`, termina per l'eccezione. Il processo `main` allora riprende ed esegue quanto doveva.

```

1  #include "all.h"
2
3  void f1(){
4      char_write('a');
5  }
6
7  void f2(natq i){
8      printf("%d", i);
9      f1();
10     terminate_p();
11 }
12
13 void main(){
14     natw id = activate_p(f2, 10, 40, LIV_UTENTE);
15     printf("Identificatore processo: %d", id);
16     delay(10);
17     pause();
18     terminate_p();
19 }
```

Dal contesto utente non si può uscire con altre funzioni scritte da livello utente: se vediamo qualcosa sullo schermo, è dovuto alla primitiva che scrive sullo schermo: ciò che richiede un contesto sistema può essere fatto solo con l'uso di primitive.

Un processo può trovarsi in uno tra diversi stati: mentre uno è *in esecuzione*, tutti gli altri processi sono o *pronti* o *bloccati*. Un processo pronto andrebbe avanti, ma il processore è occupato da un altro processo con precedenza maggiore. Bloccato è un processo che non può andare avanti finché un dato evento non si verifica.



Poniamo J1 in esecuzione, con il processore che esegue le sue istruzioni. Ad un certo punto vuole eseguire un'operazione di input da una periferica. Chiama una primitiva e si va nel suo programma con privilegio maggiore. Il sistema avvia l'operazione, ma blocca il processo J1, e rimane in questo stato finché l'operazione non è terminata, per non far perdere tempo. Nel frattempo, si passa ad un processo in stato pronto, J2, e il sistema lo mette in esecuzione. J1 ad un certo punto sarà *pronto*. J2 viene messo in attesa e si riprende J1 se sono soddisfatte le condizioni di precedenza. Questo terminerà con una primitiva di sistema, e farà andare avanti J2.

Il processore, quindi, salta da un processo ad un altro, e ogni passaggio è intramezzato da un intervento con il sistema. Il passaggio da blocco a pronto è detto *risveglio*. J2 è stato forzatamente messo in **coda pronti** mentre era in esecuzione: è avvenuta una *prelazione* dovuta alla priorità. Questo meccanismo è presente in tutti i sistemi multi-processo: dove non c'è, un processo lascia la CPU solo quando è questo a chiamare la primitiva opportuna, e ciò non può accadere in nessun altro caso. Windows 95 e 98 erano sistemi senza preemption: nel caso in cui un processo non lasciasse la CPU autonomamente, tutto il sistema si bloccava, e si doveva usare *ctrl+alt+canc*. Viceversa, in un sistema con *preemption*, è sempre possibile riprendersi il controllo. Quando un processo da bloccato va in pronto, potrebbe avere una precedenza maggiore di quello in esecuzione: in tal caso si esegue una prelazione. Si può passare allo stato terminato per forza da esecuzione, perché serve invocare *terminate_p*. Per passare da *pronto* o *bloccato* a *terminato*, serve un'implementazione nel quale un processo ne può uccidere un altro.

Questi cambi di processo avvengono tramite l'intermediazione del sistema, ovvero con il passaggio al livello sistema dopo l'attraversamento di un gate. Il sistema che vediamo è molto semplificato, usando la schedulazione con priorità e processi che si basano tutti su un solo programma. Un'ulteriore suddivisione si può avere anche sulla base della memoria che condividono tra loro: una distinzione molto ampia è tra sistemi

che non condividono nulla, *sistemi a scambio di messaggi*, e che condividono tutto, *sistemi a memoria comune*. Noi vedremo una realizzazione ibrida, condividendo tutte le cose globali del nostro modulo utente (sezione text, bss, data e heap), ma con una pila per processo, senza che si possa accedere a quella di un altro. Il meccanismo deve essere realizzato completamente dal sistema, visto che l'utente non vede altro che le primitive.

```

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

Ogni processo, dal punto di vista del sistema, è una struct *des_proc* con tutte le informazioni necessarie: un identificatore del processo, la precedenza, il livello

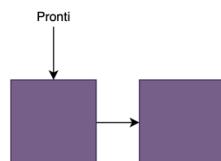
(utente o sistema), il suo ultimo stato (per passare da un processo ad un altro, facciamo una foto dello stato, in termini di contenuto di registri e memoria, e carichiamo lo stato di un altro processo, in modo da poter riprendere quest'ultimo nel proprio contesto. Per ora possiamo limitarci a salvare lo stato dei registri generali) e un puntatore alla pila sistema. Infatti, abbiamo una pila sistema diversa per ciascun processo: mentre è attivo un certo processo, deve essere attiva anche la sua pila sistema (si modifica il TSS corrente), in modo che un qualunque attraversamento del gate provochi il passaggio a questa pila. Ciò accade perché, all'attraversamento del gate, si salvano alcune informazioni necessarie alla prosecuzione del processo (%rip, rflag...), che saranno usate successivamente. In ultimo, c'è un puntatore a des_proc, in modo da poter costruire una coda di processi. Il sistema avrà una variabile globale pronti che punta alla lista. Visto che il sistema lavora per precedenza, la terremo ordinata per questo campo, in modo che la schedulazione avvenga per estrazione della testa. La variabile globale esecuzione tiene traccia del processo in esecuzione. Per i processi bloccati, ci sarà una lista per ciascuno dei possibili eventi che li risveglieranno (hard disk, tastiera...).

Le routine di sistema le possiamo vedere al pari delle routine di interruzione: non sono processi e non hanno memoria; la grande differenza sta nella proprietà dell'atomicità, l'impossibilità di essere interrotte dovendo salvare lo stato e riprendere l'esecuzione successivamente. Quando un processo attraversa un gate, viene scattata la foto, e il processo si ferma. Si passa al contesto sistema che non è più il processo, e dunque non lo fa avanzare. Questa cosa la otteniamo facendo in modo che ogni routine inizi con una call salva_stato, che salva il contenuto dei registri generali nel campo contesto del descrittore di processo di esecuzione. Alcune cose vanno fatte necessariamente in Assembler, dovendo accedere ai registri generali. A questo punto viene eseguito del codice (è il sistema che lavora, non il processo), e si usa carica_stato: questa funzione riprende i registri dal contenuto del processo in esecuzione, che potrebbe essere stato modificato, e modifica %rsp riportandolo a puntare alla pila di sistema del processo che andrà in esecuzione (finora eravamo nella pila del precedente processo in esecuzione). L'uso di una iretq consentirà di riprendere il processo dal punto in cui era rimasto. Sapendo questo, per passare da un processo ad un altro basta modificare il valore della variabile esecuzione a cavallo tra salva_stato e carica_stato.

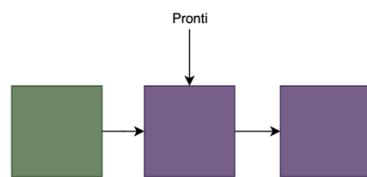
Per quanto riguarda la protezione, tutte le strutture dati, i descrittori di processo, le pile sistema, pronti ed esecuzione devono stare nella memoria non accessibile all'utente, che chiamiamo M1. Il resto apparterrà a M2, dove ci sarà il modulo utente e la pila del processo corrente (per ora, supponiamo che sia in memoria solo una pila alla volta). L'utente, quindi, non può fare alcuna operazione che riguardi M1: attraversando il gate, il livello si innalza e salta ad una delle routine di sistema, con libero accesso a M1. Dal punto di vista della correttezza questo non è sufficiente, dovendo garantire l'atomicità delle routine di sistema. In generale, capita infatti che due variabili siano condivise tra flussi di controllo concorrenti. Per le istruzioni di linguaggio macchina, non ci sono interruzioni a metà, o vengono eseguite completamente o vengono ignorate (eccezione fault). Tuttavia, se un'operazione richiede più di un'istruzione, mentre lavora su una struttura dati globale, potrebbe attivarsi un'interruzione nel momento in cui lascia questa in uno stato inconsistente.

Poniamo di voler inserire un elemento in testa a coda pronti. Il nuovo processo deve puntare la testa della lista, il puntatore di testa deve puntare il nuovo elemento. Questo richiede due istruzioni di linguaggio macchina, come minimo. Cosa succede se nel mezzo a queste due c'è un'interruzione? Sarà chiamata una routine del sistema, che magari vorrà lavorare sulla coda pronti, in quanto globale, aggiungendo un elemento in testa. Una volta fatto questo però, riprendendo la routine precedente, va perso il processo inserito dalla seconda. Infatti, le routine che lavorano su una struttura dati lasciano la lista in uno stato inconsistente, e come tale si suppone di trovarla all'avvio della routine: nel mezzo la lista passa attraverso stati non consistenti. Avere interruzioni che lavorano sulle stesse strutture dati implica che le loro routine potrebbero intervenire proprio trovando le strutture dati inconsistenti, cosa che non deve accadere.

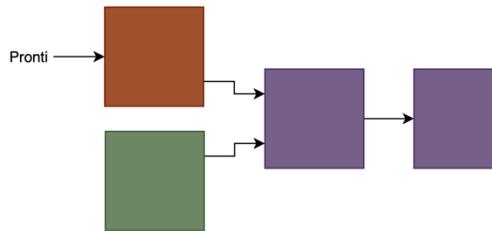
Prima della creazione del processo da parte della routine 1.



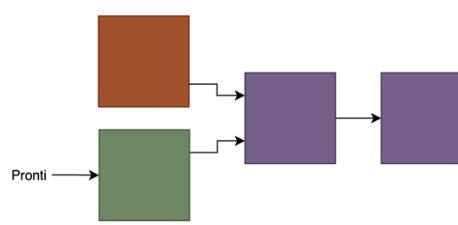
La routine 1 ha creato il processo, lo ha collegato alla lista, ma non ha ancora spostato il puntatore pronti (stato inconsistente).



La routine due ha creato il nuovo processo e lo ha messo in testa, modificando il puntatore pronti.



La routine 1 termina la sua operazione, spostando pronti: un processo viene perso.



Le possibili soluzioni sono due: o far sì che le routine sappiano che le strutture dati potrebbero non essere in stati consistenti, o evitare che ci siano interruzioni. Dato che la prima via è dispendiosa, si assume che una routine di sistema non possa essere interrotta. Per le interruzioni esterne, si può fare con uno dei flag presente nella IDT, ma non è scontato per le eccezioni, dove deve essere il sistema ad evitarle. Lo stesso vale per le interruzioni software, che non devono essere invocate. Infatti, alla chiamata di una routine di sistema corrisponde una `salva_stato`, che sovrascrive in esecuzione il contesto precedente, ma la routine di esecuzione non faceva parte del processo, e si perderebbero le informazioni.

Le routine di sistema diventano in questo modo alla stregua di un'istruzione di linguaggio macchina, in quanto atomiche. Un modo per concettualizzare un nucleo è di pensare che i sistemisti definiscano nuove istruzioni macchina, che il programmatore può usare. Nel modulo IO adotteremo un contesto privilegiato ma rilassato sotto questo punto di vista, ammettendo interruzioni. È comodo che interruzioni interne, esterne e software si comportino allo stesso modo, affinché sia sempre creata una foto del processo nel passaggio ad un privilegio più alto.

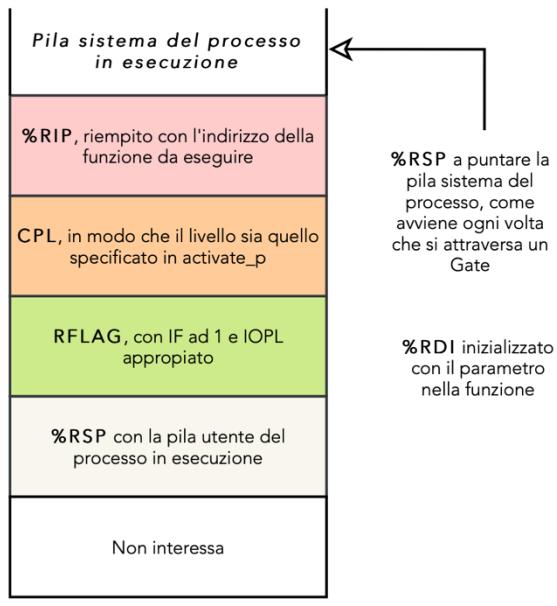
Per i processi appena creati, non è mai stata scattata una foto, visto che il campo contesto è aggiornato attraversamento di un gate. Si deve creare da zero, facendo in modo che inizi eseguendo quanto

specificato in `activate_p`. La cosa più semplice è che `activate_p` crei una foto simile a quella degli altri stati: si inizializza la pila sistema e contesto, affinché sia consistente dentro coda pronti.

```
activate_p (mioProg, parametro, 10, 20, LIV_UTENTE)
```

Oltre a creare la struttura del processo e a riempirla con i campi dei parametri, la primitiva allocherà una pila sistema. Questa sarà inizializzata con le 5 quad word consistenti affinché, alla chiamata di `iretq`, l'esecuzione riprenda normalmente. In `%rip`, ci va l'indirizzo della funzione da eseguire, in questo caso l'indirizzo di `mioProg`; in CPL ci va il codice che seleziona livello utente; nel registro dei flag, IF dovrà essere ad 1, IOPL dovrà basarsi sul livello del processo che creo; `%rsp` deve puntare alla pila utente, che dovrà allocare da qualche parte in M2; l'ultimo campo non ci interessa. Tuttavia, prima di chiamare al `iretq`, si invoca `carica_stato` che riempie i registri generali. Di questi, va riempito `%rdi`, in modo che la funzione `mioProg` possa partire con il parametro appropriato. Anche `%rsp` si deve modificare, in modo che `iretq`

prelevi i dati dalla pila sistema del nuovo processo, inizializzata come visto sopra. Infatti, nel momento in cui viene chiamata `carica_stato`, stiamo lavorando con la pila sistema del processo finora in esecuzione.



Ovviamente la pila sistema non viene recuperata dal descrittore di processo, ma da GDT. Come facciamo a dire al processore di usare tale pila? O facciamo coincidere `punt_nucleo` del TSS che l'Intel aveva progettato con la nostra pila (con la limitazione di averne uno diverso per ogni processo, e un numero di processi limitato alle entrate della GDT), oppure, come effettivamente accade, teniamo TR a puntare un unico TSS, e modifichiamo il suo valore `punt_nucleo` in modo che punti sempre alla pila sistema del processo in corso (è un'operazione che va fatta in `carica_stato`).

Sull'implementazione software dei processi

Lanciando il comando make, sono creati modulo sistema, modulo utente e modulo io dentro build/. Questi sono normali file elf, pronti per essere caricati. All'avvio, il primo programma che parte è il bootloader: il processore Intel deve ripercorrere tutta la storia dei processori Intel: inizia con una modalità compatibile con l'8086 (16 bit senza protezione), si porta in una modalità compatibile con il 386 (32 bit con la protezione) e infine con i 64 bit dell'AMD64. In questo modo, i nostri programmi iniziano assumendo che il processore si rifaccia all'AMD64. Nel passaggio da 8086 a 386 sfruttiamo il fatto che di essere in una macchina virtuale, facendolo fare tutto a qemu. Il salto a 64 bit si realizza con il bootloader visto per gli esempi di IO: a quel punto è possibile caricare in memoria i tre moduli e cedere il controllo al modulo sistema; quest'ultimo va ad interpretare gli altri due moduli, caricandoli opportunamente.

start è l'*entry point* del modulo sistema. Tutte le sezioni .text e .data sono già presenti in memoria, e possiamo inizializzare le strutture dati necessarie, sia in termini hardware (IDT, GDT...) che software (pronti, esecuzione...). La IDT viene inizializzata con le routine da chiamare per ciascun gate, usando la macro carica_gate simile al caso della libce (dobbiamo specificare anche il livello sistema, affinché non possano essere chiamati con una int). I primi gate sono quelli delle eccezioni, seguono quelli per le primitive che il sistema può invocare. Le costanti di TIPO per le primitive sono definite in include/costanti.h, utilizzabili sia da C++ che da Assembler. Nei sistemi seri non si usa un gate per primitiva, perché sarebbe limitante agli ingressi della IDT: si riconduce tutto ad un solo gate, dal quale di scegli la funzione con un valore inserito in %rax prima del suo attraversamento. Una volta riempite tutte le entrate necessarie, viene caricata con l'istruzione lidt. Per semplicità, usiamo la stessa classe per le primitive comune, e le distinguiamo con le sottoclassi. Facciamo la stessa cosa per le primitive di I/O, con classe 3.

```

void dummy(natq i){
    while (processi)
        ;
    end_program();
}

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

```

La GDT si imposta per poter cambiare pila. In tss_punt_nucleo si deve scrivere l'indirizzo della pila sistema che dovremo usare nel prossimo passaggio da livello utente a livello sistema. start inizializza quanto opportuno (ad esempio, chiamando oggetti globali che richiedono la chiamata di un costruttore) e si salta a main: al suo interno, si scrivono i primi processi. Un processo particolare è dummy, che evita di gestire il caso in cui non c'è niente in coda pronti. dummy sarà un processo con priorità minima, che permetterà al processore di fare qualcosa in attesa che altri escano da bloccato (e avendo priorità maggiore, facciano prelazione). La funzione associata a questo processo verifica in continuazione il numero di processi attualmente attivi. Se questo valore è diverso da 0, allora non fa nulla, in attesa che qualcuno lanci un'interruzione e prenda il controllo della CPU; se questo vale 0, allora la CPU non ha più niente da fare, e la spegniamo con la primitiva end_program. Questa carica un puntatore nullo nella IDT e chiama un'interruzione: non trovando consistente il gate, il processore chiamerà una seconda eccezione (per gate non valido), anch'essa non raggiungibile. Alla terza eccezione di fila, la CPU reagisce con un abort che spegne il processore.

Oltre alla lista dei processi, c'è un array **proc_table** che contiene tutti i processi, indicizzati con il proprio identificatore, per accedere immediatamente al processo dato l'id, e per sapere qual è il primo id non ancora utilizzato. Queste strutture dati sono nel modulo sistema, inseriti quindi in M1. Lo schedulatore non dovrà far altro che scrivere dentro esecuzione il primo elemento della lista pronti. Ovviamente, schedulatore sceglie soltanto il processo che andrà in esecuzione al termine della primitiva: scrivere dentro esecuzione non implica il cambio di processo, ma solo stabilire quale sarà usato alla prossima chiamata di `carica_stato`.

`salva_stato` si occupa di inserire, nel contesto del processo attualmente in esecuzione, il valore dei registri generali. Dentro `%rbx` mettiamo il puntatore a `esecuzione` e copiamo tutti i registri opportuni. L'unico accorgimento è che, salvando `%rsp`, ci si inserisca il valore che aveva prima di chiamare la funzione (ci sono state due push nel frattempo). `carica_stato` fa l'operazione opposta. Il suo problema è che ha in pila un indirizzo di ritorno, ma se `esecuzione` è stato modificato dovrà modificare anche la nuova pila, perché la `ret` porta all'indirizzo opportuno. La prima cosa che fa è salvare in `%rcx` l'indirizzo di ritorno per poi rimetterlo nella nuova pila, adesso che `%rsp` è stato caricato. Così la `ret` in fondo permette di tornare normalmente a `iretq` della routine di sistema che l'ha chiamato. Un problema si ha con la `terminate_p`, che distrugge il processo ma non può distruggere la pila, perché la dovrà usare la `carica_stato` per effettuare il cambio di pila. Semplificando, è la `carica_stato` a distruggere la pila, usando una variabile globale a sapere se serve.

La `a_activate_p` non fa altro che salvare lo stato, chiamare una routine scritta in C++ e caricare lo stato del processo: questo è lo schema per qualunque primitiva, con il grosso dell'elaborazione fatta ad alto livello. La `c_activate_p` come prima cosa controlla i dati inseriti dall'utente, affinché non se ne possa approfittare. L'utente non può creare un processo con priorità minore di `dummy` o maggiore della priorità del processo che sta chiamando `activate_p` (lo troviamo puntato da `esecuzione`). In tal modo, `activate_p` non si occupa di cambiare processo. Si verifica poi che il livello sia opportuno rispetto al privilegio di colui che l'ha chiamato. Per capire il livello del chiamante, si guarda CPL nelle informazioni salvate in pila: quello, infatti, è il livello di privilegio in cui si trovava il processore quando è stato chiamata `activate_p`, e non il privilegio del processo chiamante. Per accedere a questo valore, non posso usare `punt_nucleo` di contesto, perché magari salvare lo stato non è stata la prima cosa fatta dopo l'attraversamento del gate (si vedano le primitive di IO): devo usare `%rsp` nel contesto del processo in esecuzione. Se la creazione ha funzionato, si inserisce nella lista pronti, si incrementa `processi` e si restituisce al chiamante l'id del processo creato. Per farlo va inserito nel campo `%rax` del contesto del registro in esecuzione. Infatti, `carica_stato` immediatamente successiva sovrascrive tutti i registri, eliminando il valore restituito (con riferimento alla `a_activate_p`). Per invocare la primitiva si usa una funzione d'appoggio che esegue solo una `int`. La `int` causa un attraversamento del gate, ma i registri non sono modificati, per cui `a_activate_p` se li ritrova come se fossero i parametri nei registri opportuni.

```

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

    if (prio < MIN_PRIORITY || prio > esecuzione->precedenza) {
        flog(LOG_WARN, "priorita' non valida: %d", prio);
        c_abort_p();
        return;
    }

    if (liv != LIV_UTENTE && liv != LIV_SISTEMA) {
        flog(LOG_WARN, "livello non valido: %d", liv);
        c_abort_p();
        return;
    }

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

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

```

Realizziamo una primitiva che mi dica la priorità del processo. Per prima cosa, in costanti.h definiamo un nuovo tipo:

```
#define TIPO_GETPREC 0x27
```

Questo tipo va agganciato ad una routine di sistema, che deve andare in esecuzione quando esegue la int corrispondente. In sistema.s si scrive:

```
carica_gate TIPO_GETPREC a_getprec      LIV_UTENTE
```

Dentro c_getprec scriviamo il cuore della primitiva, facendo uso di tutte quelle strutture dati accessibili solo con i privilegi di sistema.

```

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

```

terminate_p non fa altro che distruggere il processo attualmente in esecuzione, decrementare il contatore globale e chiamare lo scheduler.

Sui semafori

Gli utenti possono creare dei processi e far loro eseguire delle funzioni. Questi processi hanno accesso alla memoria comune, in particolare alla sezione `.data` e allo `heap`. Ma se questi si possono mescolare accedendo a strutture dati condivise, come può l'utente garantire che tutto funzioni correttamente? Nel modulo sistema un problema simile è stato risolto con l'atomicità: non possiamo però permettere all'utente di disattivare interruzioni per avere atomicità, altrimenti non sarebbe possibile il multi-processo. Allo stesso modo, c'è il rischio che non torni più il controllo al sistema. I problemi da risolvere si raggruppano in due categorie: *mutua esclusione* e *sincronizzazione*.

Nella **mutua esclusione**, l'utente deve fare due o più azioni su una stessa struttura dati, ma non vuole che queste azioni si possano mescolare tra di loro. È il caso della manipolazione di una lista per come l'abbiamo già vista: avendo più processi che lavorano su una stessa lista, con inserimenti ed estrazioni, non vogliamo che queste operazioni vengano interrotte a metà lasciando la struttura dati in stato inconsistente (l'operazione può essere interrotta, ma non con un processo che lavori sulla stessa struttura).

Nella **sincronizzazione**, l'utente vuole che una certa azione si verifichi sempre prima di un'altra. In un sistema multi-processo, l'utente non può sapere l'ordine con cui i processi si mescoleranno, essendo presappoco imprevedibile. In alcune occasioni però una certa azione dovrebbe accadere prima di un'altra. Il caso tipico è quello di una struttura dati globale del quale P1 fa da produttore inserendovi un dato, P2 fa da consumatore e vi legge: P2 non può prelevare finché P1 non ha scritto; P1 non può scrivere un nuovo dato finché P2 non l'ha usato.

È bene distinguere queste due situazioni. In un problema di mutua esclusione, non ci interessa l'ordine delle operazioni, ci basta solo che non si mescolino; nella sincronizzazione, imponiamo anche un ordinamento tra le operazioni. Dal punto di vista del sistema, dobbiamo fornire delle primitive che permettano all'utente di risolvere i due tipi di problemi. Le primitive che vediamo sono quelle introdotte da Dijkstra, le primitive semaforiche. Usiamo tuttavia un'altra metafora: forniamo all'utente delle scatole in cui è solo possibile inserire un gettone o prendere un gettone. Non sappiamo quanti gettoni ci sono, e uno non può far altro che cercare di prendere; se non c'è nulla, si blocca e aspetta che qualcuno lo inserisca per riprendere dopo l'esecuzione. Ci saranno allora tre primitive: una per creare la scatola e dire il numero di gettoni che ci sono dentro all'inizio, una per aggiungere un gettone e l'ultima per cercare di prenderne uno. Di per sé queste primitive non risolvono alcun problema, ma ci sono delle regole che permettono di sfruttarle in modo opportuno.

Nella mutua esclusione, si parte con una scatola con un gettone; per fare l'operazione serve prendere un gettone, quando ho finito lo rimetto dentro. C'è un solo gettone, e solo chi ce l'ha può fare un'operazione. Non c'è un problema di mutua esclusione per i gettoni, perché le primitive sono atomiche, e non possono essere eseguite contemporaneamente. Mentre qualcuno è riuscito a prenderlo, tanti altri provano e rimangono congelati: si crea una coda di persone che stanno aspettando un gettone. Quando il processo conclude l'operazione, mette un gettone nella scatola, e uno solo dei processi congelati si risveglia per prendere il gettone. In termini di struttura dati, serve un contatore di gettoni e una coda di processi in attesa. Per i processi, questa è una soluzione cooperativa: non basta fornire le primitive, serve che nel codice vi siano le chiamate opportune. Ciò è immediato in un sistema come il nostro, dove è un solo utente a scrivere tutto quanto, preoccupandosi di rendere le funzioni cooperative, inutilizzabile in un sistema multi-utente.

Per la sincronizzazione, serve una scatola dove all'inizio non ci sono gettoni. P2 non può fare l'operazione finché P1 non ha inserito un gettone nella scatola, in modo che ci sia sincronizzazione. Le cose già si

complicano nella situazione *produttore-consumatore*. Come per gli handshake, servono due scatole, `buff_vuoto` e `buff_pieno`. P1 può procedere se `buff_vuoto` vale 1, altrimenti si ferma; P2 può andare avanti se `buff_pieno` vale 1. È P1 a porre `buff_pieno` ad 1 dopo il suo inserimento. P2 effettua la sua operazione, poi svuota `buff_pieno` e inserisce un gettone in `buff_vuoto`. Le operazioni di estrarre e aggiungere le possiamo interpretare come un `wait` e `signal`. Le operazioni con le scatole si possono interpretare anche in termini di variabili logiche: `buff_pieno` è una condizione che è vera se c'è un gettone, caso nel quale P2 può fare la sua operazione. In questo esempio, abbiamo risolto anche la mutua esclusione, oltre che la sincronizzazione (se il buffer fosse una struttura dati, non vogliamo che il produttore possa cominciare a scrivere mentre l'altro sta leggendo). Questa doppia risoluzione non ha carattere generale. Vediamo la loro implementazione.

Un semaforo contiene un contatore e una lista di processi in attesa. Il numero di semafori necessari lo sa l'utente: noi ci pre-allochiamo un array semafori. Non forniamo una primitiva per distruggere un semaforo, quindi per allocarne uno basta ricordarci in modo incrementale il numero di semafori. Se non abbiamo abbastanza semafori, restituiamo `0xFFFFFFFF`, altrimenti restituisce all'utente l'indice di questo semaforo, al quale si potrà riferire con le altre primitive. Per la gestione, si usano `wait` e `signal`.

```

213     extern "C" void c_sem_wait(natl sem)
214 {
215     // una primitiva non deve mai fidarsi dei parametri
216     if (!sem_valido(sem)) {
217         flog(LOG_WARN, "semaforo errato: %d", sem);
218         c_abort_p();
219         return;
220     }
221
222     des_sem *s = &array_des[sem];
223     s->counter--;
224
225     if (s->counter < 0) {
226         inserimento_lista(s->pointer, esecuzione);
227         schedulatore();
228     }
229 }
```

`wait` significa “prova a prendere un gettone dal semaforo”, usando come parametro l’indice del semaforo; è sempre necessario fare un controllo sull’input, non potendoci fidare dell’utente. Ci creiamo un puntatore al semaforo, e decrementiamo il contatore. Se lo troviamo negativo, il processo che ha chiamato la `sem_wait` non potrà andare avanti, perciò lo mettiamo in coda alla lista di processi in attesa, anch’essa ordinata per priorità, e chiamiamo lo `schedulatore` per mettere in esecuzione la testa della coda pronti. Non facendo alcun controllo sul segno del counter prima del decremento, otteniamo che il suo valore as-

```

struct des_sem {
    int counter;           soluto, se negativo, indica il numero di processi che, in coda, attendono il
    des_proc *pointer;    gettone nella scatola.
};
```

```

233     extern "C" void c_sem_signal(natl sem)
234 {
235     // una primitiva non deve mai fidarsi dei parametri
236     if (!sem_valido(sem)) {
237         flog(LOG_WARN, "semaforo errato: %d", sem);
238         c_abort_p();
239         return;
240     }
241     des_sem *s = &array_dess[sem];
242     s->counter++;
243
244     if (s->counter <= 0) {
245         des_proc* lavoro = rimozione_lista(s->pointer);
246         inspronti(); // preemption
247         inserimento_lista(pronti, lavoro);
248         schedulatore(); // preemption
249     }
250 }
251 }
```

In `sem_signal`, controlliamo di nuovo la validità del dato in input, dopo di che inseriamo un gettone nella scatola. Se il contatore è minore o uguale a 0, c'è qualche processo che aveva cercato di prendere un gettone. Il processo che si è svegliato, in testa alla lista, potrebbe avere precedenza maggiore di quello in esecuzione, e fare preemption. Per verificarlo, inseriamo in lista sia esecuzione (andrà sicuramente in testa, essendo quello di priorità maggiore) che `lavoro`. Entrambi sono nella coda pronti nell'ordine opportuno, e il primo sarà il nuovo processo in esecuzione.

`inspronti` mette forzatamente esecuzione in testa. Non basta inserirlo con la funzione di inserimento in lista, perché a parità di priorità lo metterebbe in coda. In questo caso non vogliamo che torni in fondo, ma solo che si scelga tra il processo appena estratto e quello che era in esecuzione. Non sembra avere senso che il processo in esecuzione perda la sua precedenza FIFO rispetto ad altri nella lista pur a parità priorità. In ogni caso, prima facciamo `inspronti`, poi `inserimento_lista`; nel caso opposto, lo scheduler prenderebbe certamente l'attuale esecuzione, trovandosi in testa di `pronti`.

Con i semafori realizziamo lo stato bloccato: ciascun semaforo rappresenta uno dei possibili stati bloccati, dove ogni attesa dipende dal significato che il singolo utente dà al semaforo.

Dividiamo l'array di semafori in due parti, uno per l'utente uno per l'IO. Quando un utente cerca di fare un'operazione, ci facciamo dire il livello del processore quando all'attraversamento del gate con `liv_chiamante`. Se era in livello utente, impediamo l'accesso alla seconda porzione di array che, avendo a che fare con l'IO in modo automatico, potrebbe compromettere l'esecuzione.

Alcune primitive d'esame

Per le prove d'esame, troviamo il testo, uno zip con il codice del nucleo, la soluzione e l'output che deve essere prodotto. L'esercizio prevede di modificare il nucleo. Consideriamo la prova del 5 giugno 2010.

L'idea dell'esercizio è quella per cui i semafori possono essere migliorati se specializzati in un particolare problema. Nell'ambito della mutua esclusione, possiamo ricordare quale processo ha acquisito la mutua esclusione (finora, il semaforo non si ricorda chi ha il gettone). `mutex` può assumere solo gli stati *libero* e *occupato*, e ricorda l'identità del processo che l'ha richiesto. Si possono allora fare numerosi controlli di errore: è un errore se viene liberato da un processo diverso da quello che l'aveva occupato, ed è errore se lo stesso processo chiede nuovamente il `mutex`. Ci sono tre primitive da definire: `mutex_ini` inizializza un nuovo `mutex`, e ne restituisce un identificatore; `mutex_wait` tenta di occupare il `mutex`, se è già occupato sospende il processo; `mutex_signal` libera il `mutex`, e cerca di svegliare un processo se qualcuno era in coda. Dentro la cartella `es1/` troviamo una versione modificata del nucleo, che dovremo andare a modificare.

In `costanti.h` ci saranno dei tipi per le nuove primitive da aggiungere, così come un tetto massimo al numero di `mutex`. In `sistema.s` ci sono alcuni elementi già inizializzati: nel campo `SOLUZIONE`, serve inserire le nostre cose, come l'inizializzazione dei gate e le funzioni da associare ai gate. Lo stesso succede per `sistema.cpp`, dove è definita la struttura dati e l'array con i valori. Noi dobbiamo fare `mutex_wait` e `mutex_signal`. In `utente/prog/` c'è un programma che cerca di usare le nostre primitive, mostrando poi un output che deve essere come quello indicato nel fine.

Si tratta di riempire le parti mancanti: carichiamo i gate e le funzioni `a_mutex_wait` e `a_mutex_signal`, copiandole dalle altre (con tanto di `.cfi`, che servono per il backtrace). Conviene non omettere mai la chiamata a `salva_stato` e `carica_stato`, a meno che non siamo sicuri che la primitiva implichi un cambio di processo.

<code>carica_gate TIPO_MW</code>	<code>a_mutex_wait</code>	<code>LIV_UTENTE</code>
<code>carica_gate TIPO_MS</code>	<code>a_mutex_signal</code>	<code>LIV_UTENTE</code>

Per scrivere le funzioni in `sistema.cpp`, ci dobbiamo ricordare che non possiamo attraversare dei gate e non dobbiamo pensare che la primitiva stia girando insieme al processo che l'ha generata: sono atomiche, e non possono in nessun modo essere interrotte. La primitiva non si può però sospendere, o fa tutto o non fa nulla. La `mutex_ini` è un esempio di una primitiva che continua ad essere eseguita nel contesto dello stesso processo (non abbiamo salvato lo stato), ma è solo un'ottimizzazione. Innanzitutto, non ci possiamo fidare del `mux` che abbiamo ricevuto dall'utente. Se il processo utente ha fatto un errore in questi termini, è bene abortire il processo. Con `c_abort_p()` cancelliamo il processo in corso e modifichiamo esecuzione di conseguenza (la pila sistema viene cancellata da `carica_stato`). A questo punto sappiamo che `mux` è valido: ci prendiamo un puntatore `m` per lavorarci direttamente. Chiediamoci allora se il processo che ha chiamato la `mutex_wait` è lo stesso che ha adesso il `mutex`.

Una volta fatti i controlli, il processo cerca di occupare il `mutex`, ed eventualmente viene sospeso. Se non è valido, mettiamo il processo in coda `waiting` del `mutex` e chiamiamo `schedulatore`; altrimenti, il nuovo owner diventa esecuzione. Atomico significa che tutto quello che c'è scritto sarà eseguito senza che nient'altro vi si possa infilare in mezzo.

La parte iniziali di mutex_signal è simile, con gli stessi controlli in riferimento agli errori. Gestiamo la preemption allo stesso modo in cui si gestisce nei semafori. Se in coda non c'era nessuno ad aspettare, il mutex va liberato.

```

extern "C" void c_mutex_wait(natl mux){
    if(!mutex_valido(mux)){
        flog(LOG_WARN, "mutex non valido, %d\n", mux);
        c_abort_p();
        return;
    }
    des_mutex * m = &array_desm[mux];

    if (esecuzione->id == m->owner){
        flog(LOG_WARN, "mutex_wait ricorsiva");
        c_abort_p();
        return;
    }
    if(m->owner){
        inserimento_lista(m->waiting,esecuzione);
        schedulatore();
    } else
        m->owner = esecuzione->id;
    return;
}

extern "C" void c_mutex_signal(natl mux){
    if(!mutex_valido(mux)){
        flog(LOG_WARN, "mutex non valido, %d\n", mux);
        c_abort_p();
        return;
    }

    des_mutex * m = &array_desm[mux];

    if(esecuzione->id != m->owner){
        flog(LOG_WARN, "c_signal non valida");
        c_abort_p();
        return;
    }

    if(m->waiting){
        des_proc * w = rimozione_lista(m->waiting);
        inspronti();
        inserimento_lista(pronti,w);
        schedulazione();
        m->owner = w->id;
    } else m->owner = 0;
}

```

Nella prova del 7 giugno 2012, si vuole definire un modo per effettuare uno scambio di messaggi tra processi, conoscendo un identificatore. Bisogna definire le primitive di send e receive. Conviene sempre dare un'occhiata al programma utente e all'output per capire meglio cosa ci si aspetta che facciano le primitive.

La funzione send ha come parametri l'id del destinatario e il messaggio. Volendo inviare un messaggio ad id, non ci possiamo fidare di cosa ci dice l'utente: considereremo errata la condizione `id>=MAX_PROC`. In tal caso, terminiamo il processo. Ci descrittore di processo del destinatario lo prendiamo con la tabella `proc_table`, che punta proprio al descrittore di processo. Se a questo id non corrisponde nessun processo, il puntatore preso da `proc_table` è `nullptr`: in tal caso, si ritorna e si restituisce false. Restituire un valore significa scrivere nel campo `contesto[I_RAX]`, visto che dopo c'è una carica_stato. Se il destinatario sta aspettando, allora gli mettiamo in `I_RAX` il messaggio e si verifica l'eventuale preemption. Se invece il processo non stava aspettando, allora si inserisce nella lista dei senders del destinatario e si chiama `schedulatore()`. È impossibile che `waiting` sia true ma che ci siamo dei senders bloccati. L'effetto di un processo bloccato la facciamo solo chiamando `schedulatore`, non spostandolo in una lista, perché comunque è puntato da esecuzione.

Nella funzione receive, ci dobbiamo chiedere se qualcuno stesse cercando di mandarci un messaggio. In caso negativo, mettiamo a true il campo `waiting` di esecuzione e chiamiamo `schedulatore`. Non è un problema non mettere da nessuna parte esecuzione. Se ci sono dei senders, si estrae la testa della lista dei senders, poi completiamo quello che non siamo riusciti a completare quando il mittente aveva provato a mandare il messaggio: si mette a true il valore di ritorno del mittente, a `msg` il valore di ritorno del processo in esecuzione e si verifica una preemption.

```

1756     extern "C" void c_send(natw id, natl msg){
1757         if(id>MAX_PROC ){
1758             esecuzione->contesto[I_RAX] = 0;
1759             flog(LOG_WARN, "id inserito non valido alla chiamata di send");
1760             return;
1761         }
1762         if(!proc_table[id]){
1763             esecuzione->contesto[I_RAX] = 0;
1764             flog(LOG_WARN, "Non esiste un processo con l'id richiesto alla chiamata di send");
1765             return;
1766         }
1767
1768         esecuzione->contesto[I_RAX] = 1;
1769         des_proc * destinatario = proc_table[id];
1770
1771         if(destinatario->waiting){
1772             destinatario->waiting = false;
1773             destinatario->contesto[I_RAX] = msg;
1774             inspronti();
1775             inserimento_lista(pronti, destinatario);
1776             schedulatore();
1777         }
1778         else{
1779             esecuzione->msg = msg;
1780             inserimento_lista(destinatario->senders, esecuzione);
1781             schedulatore();
1782         }
1783     }
1784 }
```

```
1787     extern "C" void c_receive(){
1788         if(esecuzione->senders){
1789             des_proc * mittente = rimozione_lista(esecuzione->senders);
1790             esecuzione->contesto[I_RAX] = mittente-> msg;
1791             inspronti();
1792             inserimento_lista(pronti, mittente);
1793             schedulatore();
1794         }
1795     else{
1796         esecuzione -> waiting = true;
1797         schedulatore();
1798     }
1799 }
1800 }
```

La primitiva delay

Con la primitiva `delay`, il processo chiede di essere sospeso per un certo numero di intervalli di tempo. Il nucleo sospende il processo e lo risveglia dopo il tempo prestabilito. Il nucleo all'avvio programma il timer 0, in grado di inviare interruzioni all'APIC. Il sistema utilizza il timer o per fare dei controlli, o per gestire una primitiva di questo tipo. L'input della `delay` è il numero di interruzioni dopo il quale risvegliare il processo.

Il sistema si deve ricordare per ciascun processo quanto tempo deve ancora passare; ci sono vari modi: la cosa più ingenua è mettere i processi in una lista con un contatore, decrementare ogni contatore per interruzione ed estrarli una volta arrivati a 0 (finiranno in coda pronti). Noi utilizzeremo una struttura dati un po' più furba. La struttura `richiesta` ha un contatore e un puntatore a richiesta per fare una lista e uno a `des_proc`; ogni contatore conta solo il numero di intervalli di tempo da aspettare dopo che sono trascorsi tutti quelli precedenti. Se i contatori sono a 10, 3, 1, il primo si sveglia dopo 10 intervalli di tempo, il secondo dopo altri 3 intervalli di tempo, il terzo dopo un altro (è equivalente a 10-13-14). In questo modo, la routine che va in esecuzione con il timer deve decrementare la prima richiesta; quando arriva a zero risveglia il primo, e i successivi se hanno 0 (dovevano aspettare lo stesso numero di intervalli di tempo del primo). Ci sarà da spendere un po' di tempo in più per la `delay`, ma sarà immediata la routine del timer, che va in esecuzione molto più spesso dell'altra.

Per inserire un nuovo processo, scorro gli elementi della lista; confronto il valore di attesa nuova con il contatore dell'elemento che ho, ed eventualmente calcolo il residuo; finché è positivo vado avanti, quando è negativo lo inserisco per com'era il residuo e modifco il successivo.

La funzione `main_sistema` si occupa di inizializzare il timer per come abbiamo visto a suo tempo; associato alle interruzioni del timer (che ha priorità massima), c'è la routine `driver_td`, che, dopo aver salvato lo stato (potrebbe risvegliare un processo con precedenza maggiore di esecuzione) chiama `c_driver_td`.

```

330     extern "C" void c_driver_td(void){
331         richiesta *p;
332         inspronti();
333
334         if (p_sospesi != nullptr) p_sospesi->d_attesa--;
335
336         while (p_sospesi != nullptr && p_sospesi->d_attesa == 0) {
337             inserimento_lista(pronti, p_sospesi->pp);
338             p = p_sospesi;
339             p_sospesi = p_sospesi->p_rich;
340             delete p;
341         }
342         schedulatore();
343     }

```

La primitiva `delay` fa uso di `new`, con lo heap di sistema opportunamente inizializzato. La funzione `inserimento_lista_attesa` esegue l'algoritmo visto. Finché c'è qualcosa in lista, se quanto deve aspettare il nuovo processo è maggiore dell'elemento della lista che stiamo guardando, decrementiamo la sua attesa. A quel punto facciamo l'inserimento. L'elemento successivo viene decrementato del tempo della richiesta inserita. Per gli elementi ancora dopo non c'è problema, visto che si basano sul tempo degli elementi precedenti, che abbiamo sistemato.

```
300 void inserimento_lista_attesa(richiesta *p){  
301     richiesta *r, *precedente;  
302  
303     r = p_sospesi;  
304     precedente = nullptr;  
305     ins = false;  
306  
307     while (r != nullptr)  
308         if (p->d_attesa > r->d_attesa) {  
309             p->d_attesa -= r->d_attesa;  
310             precedente = r;  
311             r = r->p_rich;  
312         } else break;  
313  
314     p->p_rich = r;  
315     if (precedente != nullptr) precedente->p_rich = p;  
316     else p_sospesi = p;  
317  
318     if (r != nullptr) r->d_attesa -= p->d_attesa;  
319 }
```

Sulla paginazione

Ci sono alcune cose lasciate in sospeso o risolte in maniera temporanea, tutte legate alla memoria. Nello spazio di indirizzamento (con il buco in mezzo), oltre alla RAM, che ne occupa una piccola parte, ci sono anche delle periferiche (APIC, memoria video in modalità grafica...). Un primo problema era sorto con la cache: si dovrebbe disattivare quando stiamo eseguendo un'istruzione di I/O. Pensiamo alla lettura di un codice di scansione dalla tastiera: accediamo continuamente a RBR, ma se c'è la cache nel mezzo il controllore copia il contenuto del registro in una cacheline, senza che acceda ogni volta al valore corrente. Finché si tratta dello spazio di I/O non ci sono problemi, sapendo distinguere tale porzione tramite l'istruzione usata; se invece i registri sono *memory mapped*, come per l'APIC, non abbiamo alcuna indicazione sul fatto che la cache non debba intervenire.

Altri problemi sono legati alla protezione: il sistema vuole riservarsi una parte di memoria inaccessibile all'utente. In più, Linux impedisce la scrittura su alcuni indirizzi pur facendo parte dell'area utente, come quelli in `.text`. Alcuni indirizzi sono poi completamente inaccessibili anche dal sistema, come lo `0` e `limitrofi`. Nell'ambito della multiprogrammazione, la memoria fa parte dello stato del processo: finora abbiamo ipotizzato di cambiare processo salvando la memoria utente in un hard disk per poi sostituirla con la memoria di un altro processo. Non è una strategia assurda (il primo sistema multi-processo funzionava in questo modo), ma per memorie grandi sarebbe improbabile salvare alcuni GiB per ogni cambio di processo.

Vorremmo che nella memoria ci potesse entrare più di un processo. Un primo problema è di protezione: mentre un processo sta girando, potrebbe modificare la memoria di un altro; il malfunzionamento in questione non sarebbe dovuto ad errori nel codice, ma ad una sovrapposizione aleatoria. Una soluzione potrebbe essere quella di dare un bound alla regione accessibile, `[ind_inf, ind_sup]`: potremmo usare due registri controllati lato hardware e aggiornati al cambio di processo in modalità sistema. Questa soluzione però comporterebbe un costo enorme in fase di collegamento: il collegatore deve decidere a quali indirizzi si trova il programma, ma sarà poi il sistema operativo ad assegnargli una certa porzione di memoria. Se anche facessimo coincidere le due cose, processi diversi non potrebbero usare gli stessi programmi, perché realizzati su specifici indirizzi.

Supponiamo che in RAM ci entrino solo due processi alla volta e che gli altri vadano in HD (la RAM fa da cache dell'hard disk). Se ho P1-P2, poi scarto P2 per far posto a P3, e scarto P1 per far posto a P2, mi ritrovo P2 in porzioni diverse da quella di partenza. Una cosa è modificare gli indirizzi di un programma che deve ancora partire (fase di caricamento), una cosa è modificarli in un programma attivo: qualunque dato potrebbe essere interpretato dall'utente come indirizzo o meno, e non sarebbe intelligibile dal sistema operativo. L'unica soluzione ad una scelta del genere sarebbe quella di inserire un processo sempre nel punto per cui era stato creato, con ovvi problemi di performance.

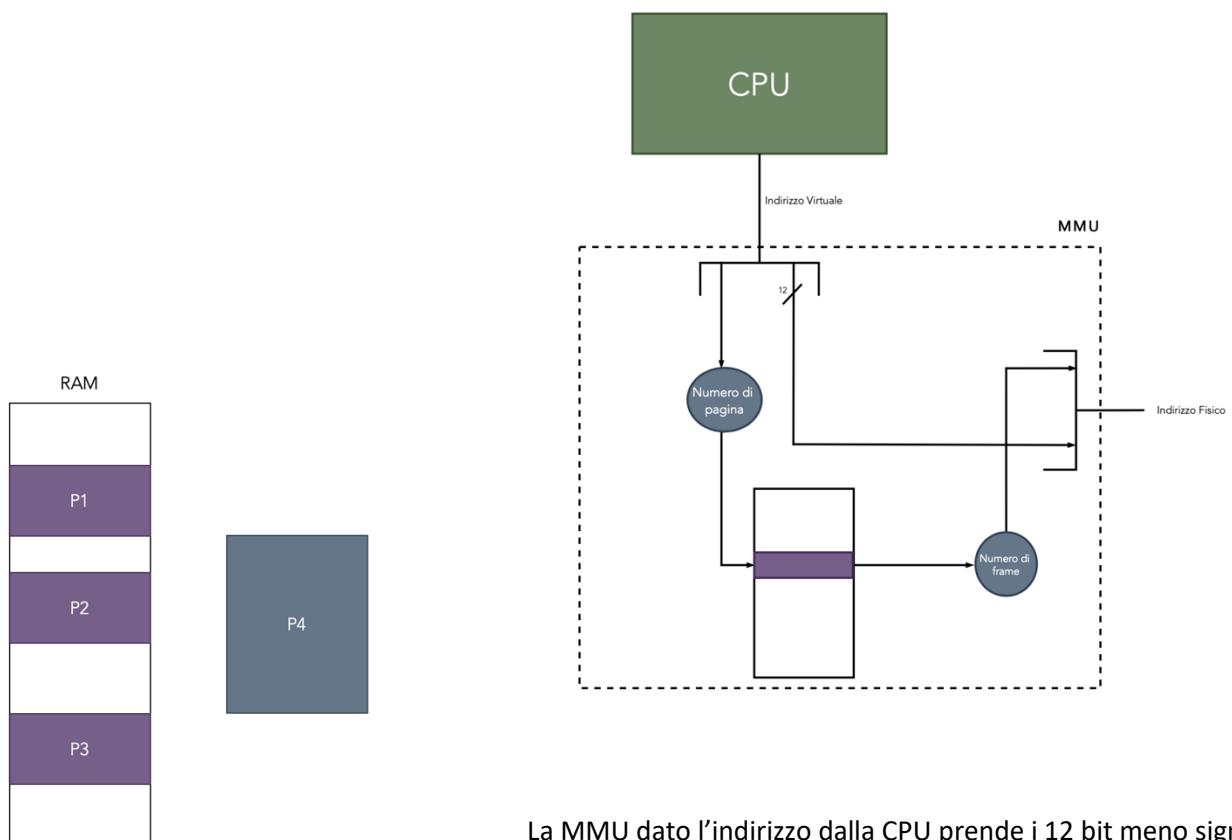
È chiaro che in qualche modo ci deve venire incontro l'hardware. Facciamo in modo che i processi non ci dicano direttamente quale indirizzo di memoria RAM gli serve, ma un offset al quale vogliono accedere, rispetto ad un base pointer che viene loro associato. A run-time viene sommato l'offset al base pointer, e si verifica che non si superi un certo upper bound. Tutti gli indirizzi saranno relativi all'indirizzo di riferimento: al programmatore basta sapere di avere a disposizione gli indirizzi da 0 a ad un certo massimo; il collegatore sfrutterà questi indirizzi fittizi, ma è l'hardware a fare la traduzione, corrispondente ad una traslazione. Non avremo più a che fare con indirizzi fisici, ma con indirizzi virtuali. Ogni processo avrà una propria memoria virtuale, ciascuna indipendente dalle altre; gli indirizzi che generano sono tradotti in qualcos'altro. Nell'ottica della multiprogrammazione, ogni processo avrà a disposizione una CPU virtuale (il contesto salvato nel descrittore di processo) e una memoria virtuale. Questo risolve anche il discorso del posizionamento dei

processi in RAM: i programmi lavoreranno solo con indirizzi virtuali, senza accorgersi delle modifiche al base pointer.

Tale soluzione manca di flessibilità, con le porzioni dedicate ai processi che devono essere contigue. Nell'immagine, servirebbe una rilocazione dei processi in RAM per far spazio a P4, che non entra in nessuna regione disponibile, nonostante lo spazio complessivo sia sufficiente. Si deve quindi cercare una traduzione libera degli indirizzi dei processi: ogni indirizzo virtuale viene tradotto in modo indipendente dagli altri, così che non si debba mantenere una contiguità. Questo permette, tra le altre cose, di condividere alcune porzioni di memoria tra i processi: basterà tradurre alcuni indirizzi sempre allo stesso modo, andando sempre sugli stessi indirizzi fisici. Occupiamoci allora della realizzazione di questa memoria virtuale.

Immaginiamo di avere lo spazio per mantenere tutte le informazioni necessarie per la traduzione. Tra CPU e cache inseriamo la MMU (*memory management unit*). Il suo compito è quello di tradurre tutti gli indirizzi che la CPU genera, che chiameremo virtuali. Sotto MMU, esistono solo indirizzi fisici. Per ora possiamo immaginare la struttura dati usata da MMU come un'enorme tabella, con una riga per ciascun intervallo di indirizzi virtuale.

Tradurre ogni indirizzo indipendente dagli altri è improponibile; si prende allora lo spazio di indirizzamento e lo si divide in regioni naturali gradi di una dimensione fissa. Queste porzioni sono dette *pagine*, e nell'architettura Intel sono di 4096 byte (4KiB, 12 bit). Ogni indirizzo che cade in una pagina è tradotto in un corrispondente indirizzo fisico (base pointer), dal quale ci si sposta con un offset. Poiché ogni processo potrà accedere a ciascuna pagina, ci dovrà essere una tabella del genere per ogni processo, nel quale si mappa una pagina in un indirizzo fisico.



La MMU dato l'indirizzo dalla CPU prende i 12 bit meno significativi, che rappresentano l'offset nella regione; gli altri corrispondono

ad un numero di pagina con cui accedere alla tabella. La traduzione consiste nel generare un indirizzo fisico dove l'offset è lo stesso fornito, mentre il numero di pagina è trasformato in *numero di frame* (equivalenti alle pagine ma per gli indirizzi fisici). Lo spazio di memoria si può pensare come raggruppato in regioni naturali di 4K detti frame: la traduzione consiste nel prendere una pagina e inserirla in una cornice. La parte complicata è la realizzazione di quella struttura dati, con un'entrata per ogni possibile pagina

Lo spazio di indirizzamento fisico è l'unico che si può usare dalla cache in giù. Gli indirizzi virtuali possono essere implementati anche su un numero di bit minori, come 48 o 57; dipende dal modello della CPU, e tale scelta da luogo alla dimensione del 'buco' di cui abbiamo parlato a suo tempo. Lo spazio fisico non presenta buchi, con una dimensione da 52 a 57 bit, consapevoli che 64 sarebbero stati eccessivi.

Tutto lo spazio fisico è diviso in frame (in termini immaginari, operati dalla MMU). Lo spazio virtuale è diviso in pagine della stessa dimensione, e la traduzione della MMU mappa ogni pagina su un frame. Per la protezione, questo meccanismo permette di separare un processo da un altro, visto che può accedere solo ai frame che sono mappati su qualcuno delle sue pagine. Se vediamo l'operato di MMU come la funzione $F(v) = p$, il processo 1 può accedere solo al codominio della sua traduzione; se non esiste un v con cui si ottiene un dato p , tale indirizzo fisico non sarà accessibile. Nello specifico, con indirizzi virtuali uguali i processi accederanno a indirizzi fisici diversi, a meno delle aree di memoria comuni.

Nella tabella che la MMU usa, oltre al numero di frame ci sono altre informazioni significative, che risolvono buona parte dei problemi presentati finora:

- Un bit di presenza P che dice se quell'indirizzo non corrisponde a nulla, indipendente che siamo a livello utente o sistema. Questo è utile perché nessun processo avrà bisogno di tutto lo spazio di memoria virtuale: è meglio ricordarsi che non può accedere ad una certa pagina, causando l'eccezione 14 detta page fault. Questo bit sarà resettato per la pagina 0, in modo da intercettare dereferenziazioni a puntatori nulli.
- Un flag che indica se la pagina è accessibile con privilegio utente o sistema, US. Confrontando questo valore con CPL (che in qualche modo MMU dovrà sapere), verifichiamo la correttezza dell'accesso. Questa soluzione ci risolve anche il problema del *memory mapped IO*: se non vogliamo che un processo acceda alla memoria video con una mov, si imposta la pagina associata in modalità sistema, oppure facciamo sì che, per il dato processo, la pagina non abbia un corrispondente frame a cui accedere.
- Un flag RW che indica se è possibile scrivere sulla pagina.
- PCD (*page cache disable*): per gli indirizzi di questa pagina, disabilita la cache. In questo modo, non ci sono problemi nell'accesso allo spazio di I/O e alle periferiche mappate in memoria.
- PWT: disabilità il write back nella cache. Potrebbe essere comodo per la memoria video, dove una scrittura non si deve fermare in cache, ma proseguire fino alla memoria video.
- I bit D e A servono nell'ambito della paginazione su domanda; nello specifico, ogni volta che usa un'entrata per fare una traduzione, setta ad 1 il bit A dell'entrata, e può servire al sistema per capire a quali pagina gli utenti hanno acceduto. Il bit D viene settato ad 1 quando sta facendo una traduzione all'entrata corrispondente dovuta ad una scrittura. In generale, con questi bit si può capire quali pagine ha usato.
- NX specifica alla MMU se è concessa l'esecuzione di codice da quegli indirizzi. Anche in questo caso, si genera un'eccezione di page fault.
- PS, che vedremo più avanti.

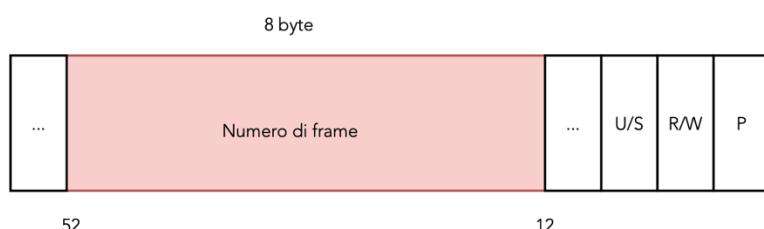
Tra la CPU e il resto del sistema c'è l' MMU che trasforma tutti gli indirizzi da virtuali a fisici, sulla base di una tabella di corrispondenza propria per ogni processo. Queste tabelle sono una di quelle strutture dati condivise tra software e hardware, come IDT, GDT o TSS. È il sistema che ne prepara una per ogni processo, e al cambio, tra le varie cose, rende attiva la tabella di traduzione del processo entrante.

Per limitare il numero di pagine che un processo avrà accessibili, posso non mappare quella porzione con un frame. Ha senso non mappare, per processi utente, la memoria di sistema? Potremmo non metterla proprio, tanto non ci potranno essere accessi. Tuttavia, essa risulta indispensabile per le interruzioni: la CPU dovrà fare una lettura alla IDT per ogni interruzione in arrivo, e quindi un accesso all'area dedicata al sistema; se non fosse mappata, sarebbe impossibile per la CPU accedervi. La presenza di alcuni elementi, come la IDT, il TSS e la pila sistema (nel quale inserire le 5 quadword) sono dunque necessari.

A questo punto, il sistema potrebbe cambiare la tabella di traduzione corrente e metterne una legata al sistema stesso; la cosa più semplice però è mappare tutta la porzione del sistema, la così detta M1, nella memoria virtuale del processo. Così facendo, nel passaggio da livello utente a sistema, la CPU troverà tutto quello che le serve mappato, con codice a livello sistema presente e accessibile. È importante che M1 sia mappata con gli stessi indirizzi virtuali per ogni processo: quando il sistema fa un cambio di processo, si cambia la tabella di traduzione, ma la CPU andrà avanti con gli indirizzi che già aveva (magari con riferimento a %rip, per l'istruzione subito successiva al cambio di tabella). Se il cambio di tabella modifica radicalmente la mappatura degli indirizzi virtuali, la traduzione degli indirizzi già presenti sarebbe completamente differente, causando dei problemi. In conclusione, almeno per quanto riguarda gli indirizzi di sistema, è bene che la traduzione sia sempre la stessa in tutti i processi.

Scrivendo codice sistema bisogna essere consapevoli della distinzione tra i due tipi di indirizzi; gli utenti usano esclusivamente indirizzi virtuali, e non interessa loro quali siano le corrispondenze. Compilatore, collegatore e programmatore ragionano in termini di indirizzi virtuali. In ogni caso, gli ultimi 12 bit, cioè le ultime tre cifre esadecimali, sono le stesse tra indirizzo fisico e virtuale, determinando l'offset nella pagina e nel frame corrispondente. In effetti, facendo girare due volte lo stesso programma, sembrerà che i dati siano sempre agli stessi indirizzi, ma in memoria sono inseriti in porzioni diverse, sfruttando la distinzione dei due processi. Un esempio esplicito di cosa significa cambiare processo si trova [qui](#). I frame non menzionati nella tabella di un processo sono completamente inaccessibili.

La tabella dovrà avere un'entrata per ogni pagina virtuale, contenente, per ciascun ingresso, un numero di frame e un po' di flag, come già visto, per una dimensione di 8 byte. Il numero di frame si trova, per comodità, a partire dal bit 12 di ciascun ingresso, fino ad un massimo di 52, in corrispondenza della porzione di indirizzo virtuale che identifica la pagina; i bit rimanenti sono usati per i flag.



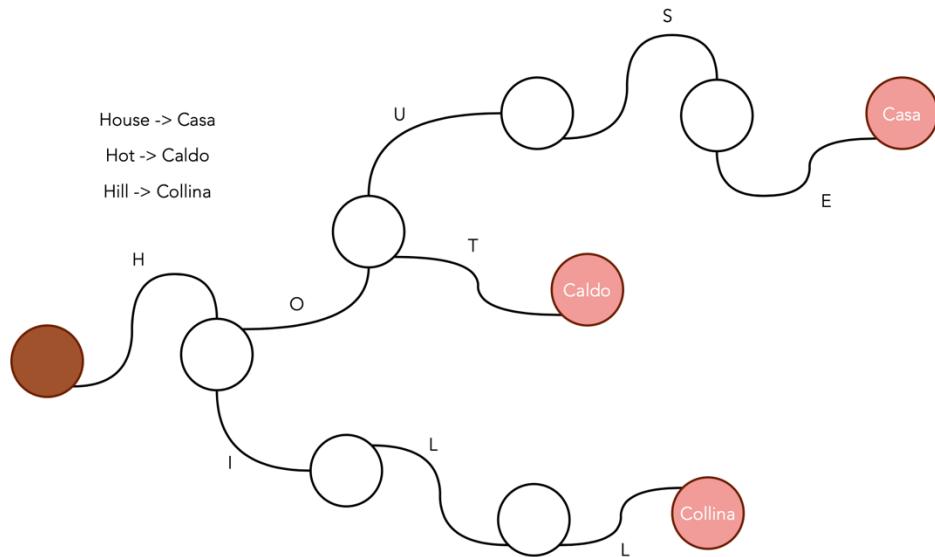
52

12

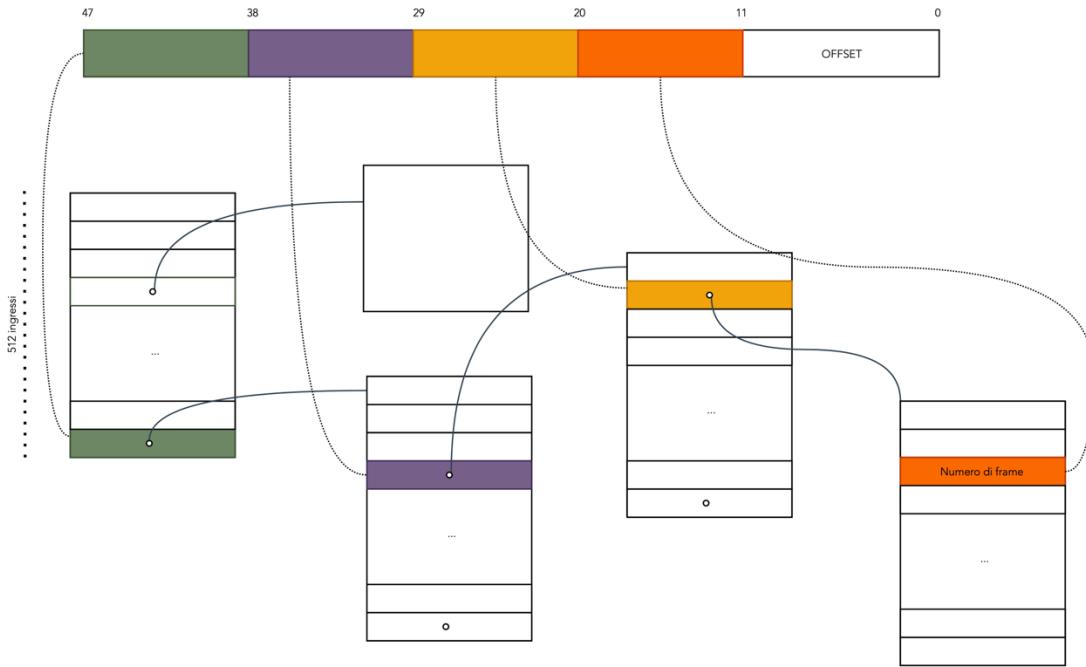
Se supponiamo un indirizzo virtuale di 48 bit (i 12 più significativi devono essere uguali al tredicesimo per avere indirizzi normalizzati), abbiamo 2^{36} ingressi con 8 byte ciascuno: sono 512 GiB per processo. Una tabella di questo tipo sarebbe ingestibile. Una semplificazione si ha osservando che non tutti i processi hanno

bisogno dell'intera memoria virtuale; sarebbe sufficiente allora una struttura dati che mi metta in corrispondenza le (poche) pagine disponibili con i numeri di frame. La struttura dati comunemente usata è il **bitwise trie**. Il trie è una struttura dati ad albero che serve per mappare delle chiavi di tipo stringa; bitwise è una sua generalizzazione dove le chiavi sono sequenza di bit.

Immaginiamo di aver le corrispondenze in figura. Creiamo un albero dal quale ci spostiamo sfruttando la chiave, di nodo in nodo, sulla base della chiave. Se due chiavi hanno le prime k lettere uguali, allora il percorso fino alle rispettive foglie prevede che i primi k nodi siano in comune. Ovviamente, in una struttura del genere, ogni nodo avrà una tabella di puntatori ai figli, tanti quante le possibili lettere.



Nel caso del bitwise trie per la memoria virtuale, le chiavi sono i bit che compongono il numero di pagina, seguendo la medesima regola: si crea un albero in cui la ricerca è guidato dai bit del numero di pagina, dove le foglie contengono il numero di frame corrispondente. Il numero di pagina è scomposto in 4 parti da 9 bit ciascuno (supponendo che l'indirizzo sia di 48 bit, con 12 di offset). Si parte da una tabella detta *tab4*, corrispondente al nodo di partenza dell'albero sopra: i primi 9 bit ci consentono di prelevare il puntatore alla tabella successiva, per un totale di $2^9=512$ ingressi. Una volta usati i primi 9 bit per ottenere il puntatore alla prossima entrata, si prendono i 9 successivi per fare l'accesso alla *tab3*, anch'essa di 512 entrate. Troviamo così il puntatore a *tab2* e, con gli ultimi 9 bit, l'elemento nella *tab1* che contiene il numero di frame.



Una struttura del genere occupa, solo per il livello 1, l'intero spazio della tabella, senza contare gli altri tre livelli. Qual è il vantaggio? Possiamo evitare di avere i sottoalberi per cui il processo che sta usando la tabella non ha mappato nulla. Facciamo qualche conto.

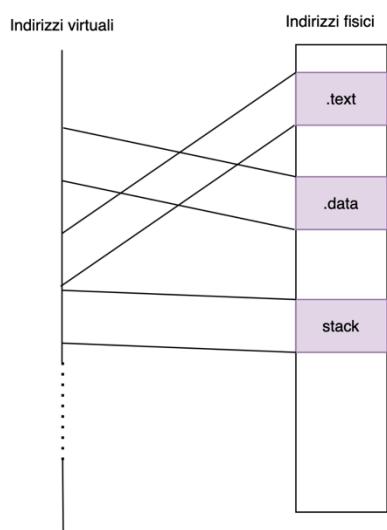
Gli indirizzi che condividono la stessa entrata a livello 4 hanno in comune i primi 9 bit più significativi, e sono quindi $2^{39}=512$ GiB. Se prendessimo tutto lo spazio di indirizzamento virtuale, tutti gli indirizzi che appartengono alla stessa regione naturale di 512 GiB accederebbero allo stesso puntatore nella tabella di livello 4; di queste regioni, ce ne sono 512 (quante le entrate della tabella). Un programma tipico occuperà solo una piccola porzione di una regione di questo tipo: *tab4* non avrà più di due ingressi. Non solo abbiamo sfoltito *tab4*, ma abbiamo anche rimosso tutti le tabelle che appartenevano ai vari sottoalberi. Quanti sono gli indirizzi che passano dalla stessa entrata della stessa tabella di livello 3? Hanno lo stesso prefisso nei primi 18 bit, quindi sono $2^{30}=1$ GiB. Anche in questo caso, i nodi di *tab2* saranno pochissimi. Pensando allo spazio di indirizzamento, un ulteriore prefisso significa prendere delle regioni di 512GiB e suddividerlo in 512 sotto-regioni da 1 GiB ciascuno. I gruppi di 9 byte definiscono la zona in cui andare, restringendo il campo in un range di 1GiB di indirizzi, e così via. Gli indirizzi che prevedono lo stesso ingresso in *tab2* sono $2^{21}=2$ MiB. Gli indirizzi il cui prefisso coincide fino alla tabella di livello 1 sono $2^{12}=4$ K, corrispondenti agli indirizzi di ciascuna pagina.

Un aspetto anti-intuitivo è che queste tabelle hanno tutte la stessa forma, pari alle entrate della tabella finale. Ciononostante, le tabelle di livello 4-3-2 non hanno nulla a che fare con le tabelle di livello 1, visto che le prime contengono indirizzi per le tabelle successive, l'ultima l'effettivo numero di frame. Si suppone che ciascuna tabella sia allineata naturalmente in memoria: ogni tabella è grande 4K, come una pagina, e quindi il loro indirizzo deve terminare con 12 zeri. Così facendo, i puntatori delle tabelle intermedie contengono solo gli altri 36 bit necessari, per accedere alla tabella, da cui ci spostiamo con i 9 bit di livello (ogni riga della tabella è fatta di 8 byte!). In ogni entrata c'è il bit P, e serve, per le tabelle, a capire se l'entrata è valida o meno. Se nell'entrata 3 di *tab4* c'è P=0, la regione di 512GiB della memoria virtuale del processo non è mappata. La MMU genera un page fault per indirizzo non valido non appena incontra P=0. Anche R/W sta in tutti i livelli: la scrittura è permessa se è permessa da tutti i livelli, non basta avere R/W=1 solo all'ultimo

livello. La stessa cosa accade per U/S: l'accesso a livello utente è consentito se lo è per tutto il percorso. In tutte le tabelle c'è il bit A, che viene messo ad 1 ogni volta che si tocca quell'entrata.

Il compito del sistema è gestire questa struttura dati, e cambiarla ad ogni cambio di processo. La MMU deve percorrere la struttura dati per tradurre un indirizzo virtuale, quindi gli indirizzi intermedi presenti nelle tabelle non possono essere virtuali, ma necessariamente fisici, corrispondendo ad effettivi accessi in RAM (altrimenti avremmo un loop). La traduzione dell'indirizzo virtuale sta solo all'ultimo livello dell'albero.

Sotto MMU, per riferirsi a qualcosa si usano indirizzi fisici; la cache, il north bus e la RAM usano indirizzi fisici. La MMU avrà in ogni istante un *trie* attivo che permette di associare ad indirizzi virtuali degli indirizzi fisici; è una grossa struttura dati che sta in RAM, anche se concettualmente la possiamo immaginare dentro MMU. CR3, uno dei registri di controllo, contiene il punto di partenza per il trie corrente: a partire dall'indirizzo virtuale di 48 bit, si visita l'albero in modo opportuno così che, alla *tab1* di arrivo, si riesca a sostituire il numero di pagina con il numero di frame. Ogni gruppo di 9 bit del numero di pagina serve per attraversare l'albero. Con questa struttura dati, è possibile evitare di associare un frame ad ogni pagina: quelle che non hanno un corrispondente frame avranno in qualche punto del loro percorso il bit P a 0, ad indicare che tutta la porzione successiva non è mappata con un indirizzo fisico.



Affinché il software riesca ad accedere ad un'entità, questa deve avere un indirizzo fisico, deve essere mappato in uno virtuale e il software lo deve conoscere; senza queste tre condizioni, è impossibile accedervi. Sarà il sistema che, sapendo dal file ELF realizzato dal collegatore quali indirizzi virtuali saranno necessari, li mapperà creando la corrispondenza del trie. In generale, il collegatore sceglie dove mettere la sezione `.text` e `.data` nella memoria virtuale; il sistema, dopo averle caricate in sistema, crea la corrispondenza opportuna. Allo stesso modo, sceglierà una porzione di RAM per lo stack, e, prima di cedere il controllo all'utente, scriverà dentro `%rsp` l'indirizzo della base.

Per l'utente le cose funzionano bene, ma per il sistema deve essere forte la distinzione tra indirizzi virtuali e fisici, (per esempio perché i dispositivi che fanno bus mastering PCI lavorano con indirizzi fisici). Ma il sistema è software, il trie è una struttura dati in memoria: per poter scriverci qualsiasi cosa, ha bisogno di un indirizzo virtuale che venga tradotto in quello fisico opportuno; deve poi sapere qual è l'indirizzo virtuale in questione, in modo da realizzare le tabelle.

Essendo strutture dati di sistema, avrebbe senso metterle in M1, ma sono grandi, soprattutto se un processo richiede tante pagine. Allora prendiamo M2, la dividiamo in frame le usiamo sia per contenere le pagine che per contenere le tabelle; in effetti, questa cosa ha senso perché sia le tabelle che le pagine sono di 4K come la dimensione associata ad un frame, ed entrambe le entità sono allineate a 4K. Per ogni tabella mi basterà scegliere il frame nel quale allocarla. L'allocazione è semplice: basta avere una lista di frame liberi e prendere volta il primo disponibile.

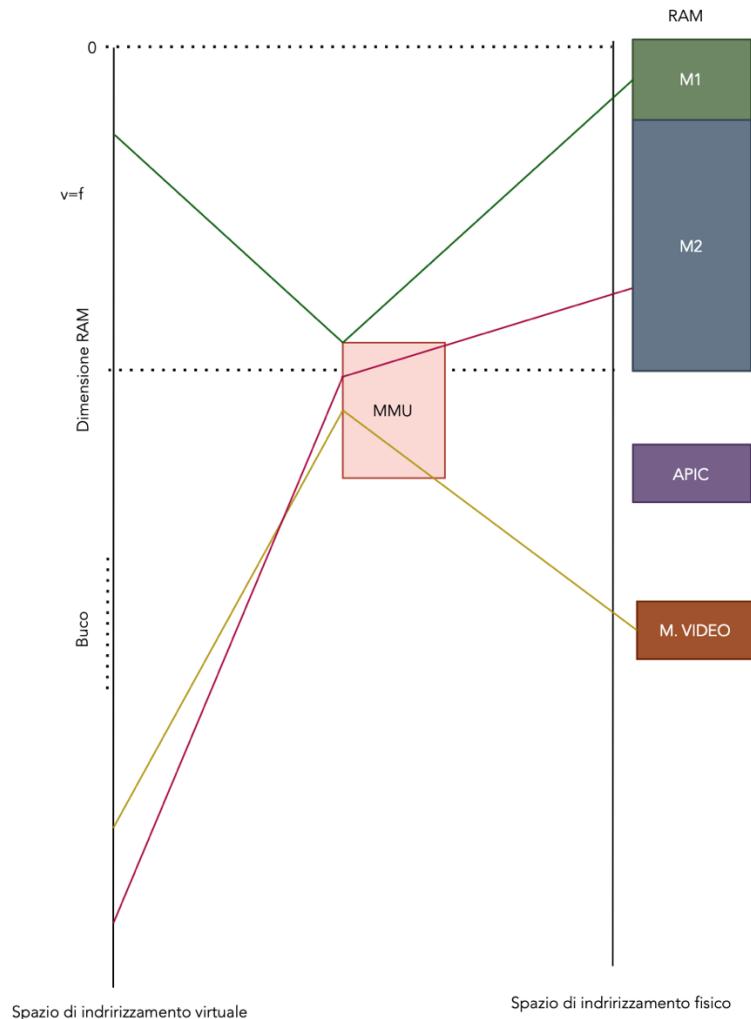
La struttura dati in questione contiene degli indirizzi; da una parte devono essere fisici affinché la MMU non entri in un loop; sotto un altro punto di vista, è il software a scrivere questi indirizzi, ma come fa a conoscerli se non sono virtuali? Usiamo la seguente convenzione.

Dato lo spazio di indirizzamento (virtuale) di ogni processo, si divide in due parti, dove la prima è dedicata al sistema. Avendo un buco, gli indirizzi che cominciano con 0 sono relativi al sistema, gli altri all'utente: il collegatore è consapevole di questa distinzione e sceglie gli indirizzi di conseguenza. Si prendono gli indirizzi virtuali del sistema e li mappiamo in tutta la RAM, per fare sì che tutti gli indirizzi fisici della RAM abbiano a priori un corrispondente indirizzo virtuale. In questo modo si rende raggiungibile l'intera RAM.

Il software deve conoscere v , non f : basterà prendere $v=f$ per non avere problemi. Qualunque sia il processo attivo, nella sua parte bassa dello spazio di indirizzamento c'è sempre una mappatura con l'intera RAM: per accedere ad un dato indirizzo fisico, uso l'indirizzo virtuale numericamente uguale. In questo modo, trattiamo gli indirizzi virtuali come se fossero fisici: per poter accedere alla tabella di livello 4 che sta in RAM uso l'indirizzo in CR3, virtuale, identico a quello fisico. Cambiare processo consisterà nel cambio del valore di CR3.

Avendo $v=f$, si semplifica anche l'inizializzazione del sistema. La paginazione si attiva settando un bit in CR0, con un semplice or. Nel momento in cui esegue, la paginazione non è attivata e il processore usava indirizzi fisici; la corrispondenza 1:1 di questo tipo fa sì che ci sia continuità con gli indirizzi finora utilizzati.

La MMU consente di tradurre ciascun numero di pagina con un numero di frame; allo stesso tempo, più indirizzi virtuali saranno tradotti nello stesso frame. Noi faremo esattamente questo: alcuni indirizzi sono usati dal nucleo per vedere la RAM (e tutto quello che è necessario accedere, come i registri dell'APIC), e sono uguali a quelli fisici; ogni processo avrà pagine mappate in vari frame della parte M2 della memoria fisica (M1 e M2 sono suddivisioni della memoria fisica). In mezzo a tutti i frame, ci sono quelli che contengono i nodi dei trie di ogni processo; questi non saranno mappati per l'utente, ma avranno un indirizzo virtuale per il sistema, e vi si potrà accedere usando l'equivalenza virtuale=fisico.



Ancora sulla paginazione

Il sistema che stiamo studiando parte come un 8086 a 16 bit; dopo abilita insieme la protezione, la segmentazione e i 32 bit. Per passare a 64 bit, si deve abilitare la paginazione con CR0. Abilitare la paginazione significa che ci deve essere un albero già pronto, perché a quel punto MMU inizia subito a tradurre (non possiamo lavorare a 64 bit senza la paginazione). Finora abbiamo sempre usato la paginazione, e non ce ne siamo accorti perché il bootloader, prima di cederci il controllo, già realizza una tabella con una traduzione 1:1 che abilita 4GiB di indirizzi. Questa dimensione copre tutta la RAM della macchina e anche la parte dove si trovano i registri dell'APIC e la memoria video in modalità grafica. Se non ci fosse stata questa traduzione, non avremmo potuto accedere all'APIC con gli indirizzi fisici esatti. Proviamo a realizzare il trie in questione in Assembler, mappando i primi 8MiB.

Dobbiamo trovare lo spazio per un albero di traduzione. Usiamo una sezione .data, con una tabella di livello 4. Tutte le tabelle devono essere grandi 4096 byte e allineate con tale valore. Una tabella di livello 3 copre 512 GiB ($2^{9+9+9+12}$), quindi me ne basta 1; una tabella di livello 2 mi copre 1 GiB (2^{9+9+12}), quindi ne basta 1; una tabella di livello 1 copre 2MiB (2^{9+12}), quindi me ne servono 4. CR3 punterà alla testa della tabella di livello 4, quindi dovremo inizializzare i campi con questa nozione. Ci conviene rappresentare il numero di pagina in ottale, dato che 9 bit sono tre cifre in ottale. In questo modo, conosciamo immediatamente l'indice che MMU usa per ciascun livello.

Nella riga 0 della tab4 ci deve essere l'indirizzo di tab3. Non ci saranno problemi con la distinzione di indirizzo fisico e virtuale, perché già stiamo lavorando con la tabella inizializzata dal bootloader. Non basterà spostare l'indirizzo di tab3 nella prima riga di tab4, perché dobbiamo impostare anche i bit della parte bassa. Infatti, se P è a 0, quell'entrata non è valida; se R/W è a 0, non si può scrivere; se U/S è a 0, non ci si può accedere in modalità utente. Si usa allora un'or con 0b00000111 (non tocchiamo i bit dell'indirizzo di tab3, visto che i suoi ultimi 12 sono 0). Usiamo lo stesso meccanismo per far puntare la prima entrata di tab3 a tab2, e le prime entrate di tab2 a tab1. A mano ci occupiamo delle righe della tabella di livello 1, che dovranno essere completamente occupate con il primo indirizzo del frame, non (la seconda riga è inizializzata con 0x2000, non 0x0001). Una volta riempita la prima tabella di livello 1, avremo mappato gli indirizzi da 0 a 2MiB-1.

```

1
2 # | 12 |
3 #+-----+
4 #| numero di frame | ...SWP|
5 #+-----+
6
7 .global setup_vm
8 setup_vm:
9     movq $tab3, tab4
10    or 0b00000111, tab4
11
12    movq $tab2, tab3
13    or 0b00000111, tab3
14
15    movq $tab1_0, tab2
16    or 0b00000111, tab2
17
18    # 000 000 000 000 xyzw -> 000 000 000 000 xyzw
19    movq $0, tab1_0
20    or 0b00000111, tab1_0
21
22    # 000 000 000 001 xyzw -> 000 000 000 001 xyzw
23    movq $x1000, tab1_0+8
24    or 0b00000111, tab1_0+8
25
26    # 000 000 000 002 xyzw -> 000 000 000 002 xyzw
27    movq $x2000, tab1_0+16
28    or 0b00000111, tab1_0+16

```

```

.data
.balign 4096
tab4:
.space 4096, 0
tab3:
.space 4096, 0
tab2:
.space 4096, 0
tab1_0:
.space 4096, 0
tab1_1:
.space 4096, 0
tab1_2:
.space 4096, 0
tab1_3:
.space 4096, 0

```

In una versione ‘estesa’, l’esercizio si può impostare come segue (tralasciando la dichiarazione delle 7 tabelle):

```

1  .text
2  .global vm_init
3  vm_init:
4      movq $tab3, tab4
5      movb $0b0000111, tab4
6      movq $tab2, tab3
7      movb $0b0000111, tab3
8
9      xor    %rsi, %rsi
10     movabs $tab2, %rdi
11     movabs $tab1_0, %rbx
12     xor    %rax, %rax
13
14 .Lfor1:
15     cmp $4, %rsi
16     jb  .LforCorpo1
17     jmp .LforFine1
18
19 .LforCorpo1:
20     mov    %rbx,(%rdi,%rsi,8)
21     movb $0b0000111, (%rdi,%rsi,8)
22     xor    %rcx, %rcx
23
24 .Lfor2:
25     cmp $512, %rcx
26     jb  .LforCorpo2
27     jmp .LforFine2
28
29 .LforCorpo2:
30     mov    %rax, (%rbx, %rcx, 8)
31     movb $0b0000111, (%rbx, %rcx, 8)
32     add $0x1000, %rax
33     inc  %rcx
34     jmp .Lfor2
35
36 .LforFine2:
37     inc  %rsi
38     add $4096, %rbx
39     jmp .Lfor1
40
41 .LforFine1:
42     movabs $tab4, %rax
43     mov    %rax, %cr3
44     ret

```

Ogni indirizzo da scrivere in un ingresso delle tabelle di livello 1 deve essere incrementato di $0x1000$, cioè 4K in esadecimale. Preparato tutto quanto, possiamo scrivere l’indirizzo della tabella di livello 4 dentro CR3: da questo punto in poi non usiamo la traduzione del bootloader, ma la nostra. Chiamando la funzione in questione, tutto funziona correttamente finché la libce non si riprende il controllo per chiudere il programma; in quel caso abbiamo un page fault, un’eccezione 14. Visto che può capitare per varie ragioni (si incontra P=0, in modalità utente si incontra U/S=1...), il processore salva in pila un dato che permette di capire quale tipo di errore è capitato. Il primo bit dell’errore serve a dire se P era 0, il secondo ci dice se l’istruzione che l’ha causata aveva a che fare con la scrittura; il terzo bit ci dice il livello del sistema quando è capitato l’errore (1 per l’utente, 0 per il sistema).

Nel nostro caso, conviene associare al page fault una routine che ci consente di capire a quale indirizzo stesse cercando di accedere MMU quando ha subito page fault: esso viene salvato nel registro CR2. Associamo una routine di interruzione per mostrarci da dove nasce il problema.

```

46  .global a_page_fault
47  .extern c_page_fault
48  a_page_fault:
49      mov    %cr2, %rdi
50      call   c_page_fault
51      hlt
52      iretq
53
54  extern "C" void a_page_fault();
55
56  extern "C" void c_page_fault(natq addr){
57      printf("L'indirizzo che ha causato l'errore era : %x", addr);
58  }
59
60  extern "C" int main(){
61      gate_init(14,a_page_fault);
62      //...

```

Il risultato è che ci da problemi l’accesso al registro ioregsel dell’APIC, che si trova ad indirizzo fisico $0xfec00000$. Effettivamente nelle funzioni per il dialogo con APIC usiamo lo stesso indirizzo virtuale

per accedervi, ma nella mappatura appena realizzata non c'è, poiché $0xfec00000 > 8\text{MiB}$. Associamogli allora lo stesso indirizzo virtuale.

$$(fec00000)_{16} = (000000000\ 000|000|011\ 111|110|110\ 000000000\ 0000\ 0000\ 0000)_2 = (000\ 003\ 766\ 000\ 0000)_8$$

Le cifre rosse corrispondono all'offset, che non ci interessa. Questa divisione mi dice che l'ingresso della tabella di livello 4 è lo stesso; poi dobbiamo accedere all'indice 03 della tabella 3, al 766 della tabella 2 e allo 0 della tabella 1. In questo punto, inseriamo il valore opportuno, che corrisponde all'indirizzo fisico che vogliamo associare a $0xfec00000$, ossia sé stesso.

```

movq $tab2_1, tab3 + 3 * 8
movb $0b00000111, tab3 + 3 * 8

movq $tab1_4, tab2_1 + 0766 * 8
movb $0b00000111, tab2_1 + 0766 * 8

movabs $0xfec00000, %rax
movq %rax, tab1_4
movb $0b00001111, tab1_4

```

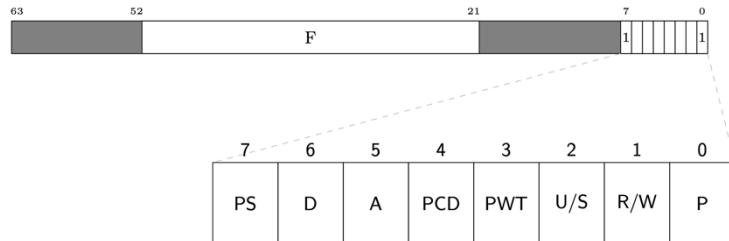
In questo caso, si tratta di un oggetto che ha a che fare con lo spazio di IO, quindi ha senso disabilitare la cache: lo facciamo settando anche il quarto bit meno significativo. Osserviamo che, oltre alla forza della notazione ottale (avendo offset di 9 bit), quando vogliamo spostarci dentro la tabella dobbiamo sempre moltiplicare l'offset per otto, avendo elementi di 8 byte ciascuno.

Allo stesso indirizzo fisico è possibile associare più di un indirizzo virtuale, e lo possiamo fare modificando il numero di pagina opportunamente al punto in cui vogliamo arrivare.

Le pagine da 4K non sono di forzatamente tali, e l'uso del trie ci garantisce flessibilità. Sappiamo che l'offset di 12 bit fornisce la dimensione della pagina, porzione che non viene tradotto ma concatenato al numero di frame; gli altri 4 gruppi di 9 bit guidano la traduzione. Una pagina più grande avrà un offset maggiore e un numero di pagina più piccolo: in tal senso potremmo considerare un albero di tre livelli, e un offset di 21 bit, con pagine da 2MiB. Sicuramente c'è un vantaggio di spazio e di tempo (non serve accedere a 4 tabelle ma solamente a 3), ma allo stesso tempo ci potrebbe essere uno spreco di memoria per le porzioni dei processi: se fossero necessari 2MiB e 1KiB, dovremmo comunque adoperare due pagine e sprecare buona parte della seconda.

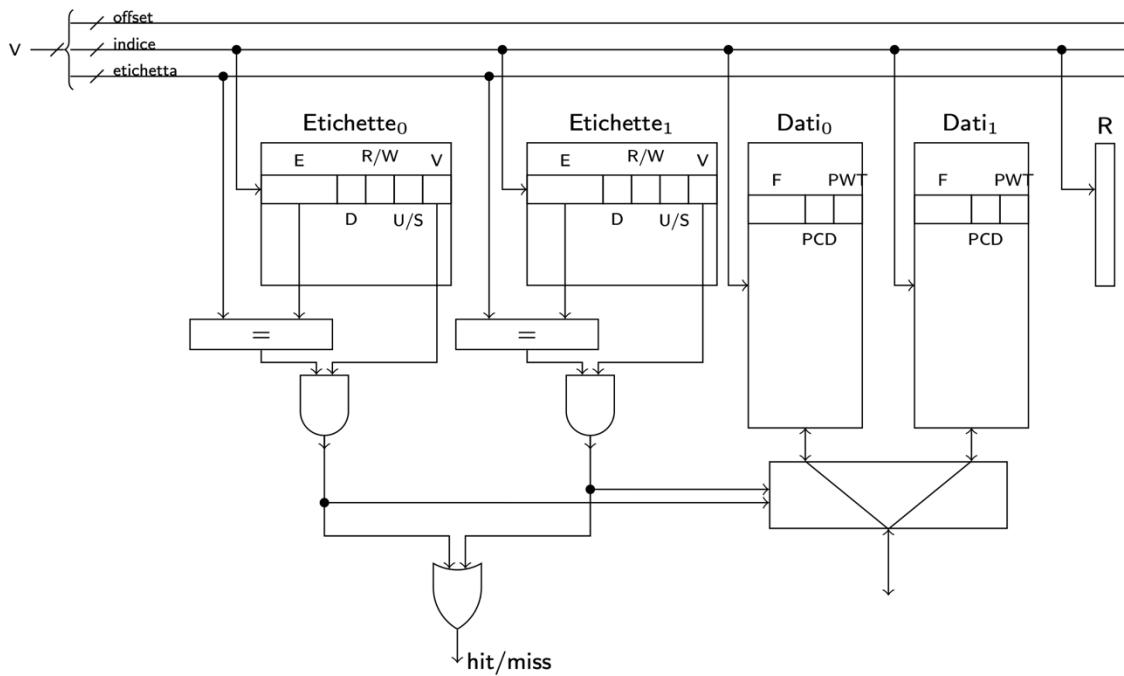
Con il trie non si deve scegliere a priori la dimensione delle pagine: dentro ogni entrata dell'albero c'è un flag PS (*page size*) che dice a MMU se in quel percorso la pagina è grande quanto la dimensione appena percorsa nell'albero (se termino al livello 2, ho pagine di 2MiB). Sostanzialmente, il flag PS dice alla MMU che la traduzione finisce lì. Tale traduzione usa come numero di pagine gli indici che hanno portato a selezionare quell'ingresso, tutto il resto è offset. In quasi tutti i processori è anche possibile usare il bit PS al livello 3, con un numero di pagina di 18 bit e offset di 30 bit. Nella traduzione identità della porzione sistema, è conveniente avere offset grandi, perché ci permette di usare meno tabelle. Indicativamente, le pagine da 2MiB sono usate anche dai programmi normali, le pagine da 1GiB sono usate dal kernel per fare traduzioni identità per accedere allo spazio fisico.

Senza il bit PS, servirebbero molte tabelle di livello 1 per mappare con l'identità il primo GiB di memoria; con PS, basta settare anche quel bit, il più significativo del byte meno significativo di una riga della tabella.



Dal punto di vista dell'hardware, ci resta da capire come compensare il numero di accessi in memoria in termini di prestazioni. Evitiamo quattro accessi per ogni richiesta con una memoria cache **TLB** che si occupa delle traduzioni degli indirizzi, non del contenuto. Serve solamente come alternativa al trie, ricordando l'associazione numero di pagina e numero di frame. È una normale cache associativa a N vie: si effettua l'accesso alla memoria delle etichette per accedere poi alla memoria dati nel caso di una hit, che conterrà il numero di frame e alcuni degli 8 bit significativi. TLB si usa in modo che MMU possa ottenere le stesse informazioni che avrebbe ottenuto durante l'intro percorso del trie. Sappiamo che se nel percorso si incontra P = 0, la pagina non è valida: questa cosa viene compensata non salvando mai questa pagina, e causando una miss ogni volta; MMU a questo punto causerà un page fault incontrando P a 0. Le altre informazioni significative per l'accesso sono RW, US, A e D, mentre PWT e PCD si possono inserire nella memoria dati avendo a che fare con l'accesso.

Nel trie di ciascun bit del tipo US e RW ce n'è uno per livello, quindi 4. In TLB ci va messo l'and logico di questi quattro valori: l'accesso è vietato a livello utente se almeno un US era 0, in scrittura se almeno un RW era 0. Nel caso in cui qualcosa non funzioni, è la tabella delle etichette a generare una miss, nonostante sia presente il frame associato alla pagina. Sotto questo punto di vista, miss ha due significati: o si deve prelevare il dato dal trie, o l'accesso avrebbe in qualche modo causato un page fault.



A e D sono delle informazioni che MMU lascia nel trie in modo che il sistema le possa usare. Il bit A non è riportato in TLB, perché se è finito in TLB vuol dire che c'è stato un accesso (sarebbe ridondante, e lo può settare nel momento in cui attraversa il trie per prelevare il dato). Il bit D è usato dal sistema per sapere se c'è una scrittura. Magari il primo accesso è in lettura, MMU setta A e preleva l'informazione portandola in TLB: da quel punto in poi non va più in TLB, e non può aggiornare D. Per evitare questo, nelle informazioni memorizzate al prelievo del dato, si ricorda anche quanto valeva il bit D. Se c'è un accesso in scrittura che causa una hit, e D vale 0, allora è necessario aggiornare D: si lancia un miss, si usa MMU per prelevare il dato e allo stesso tempo aggiornare D. Se D vale già 1, TLB può dare hit, non essendo necessari aggiornamenti. Ricordarsi quanto valeva D permette di mantenere la consistenza del trie senza andare tutte le volte a verificare il dato. Il bit V indica se la riga nella tabella delle etichette è significativa o meno.

Il TLB è una cache, e come tutte le cache ha il problema di dover cercare di restare consistente con la copia che si era fatto. Per la cache standard, la consistenza era un problema nel momento in cui qualcuno scriveva in RAM senza comunicarglielo. Il TLB si può accorgere da solo che il trie è cambiato? No, e nessun costruttore implementa un TLB che automaticamente effettua questa verifica. Si dovrebbero ricordare gli indirizzi fisici di tutte le entrate che MMU ha consultato per ottenere una traduzione; se vede una scrittura in uno dei 4 indirizzi fisici, allora l'entrata in questione non è valida. La responsabilità si scarica sul programmatore di sistema: la modifica del trie deve essere seguita da un'opportuna invalidazione.

Muovere qualcosa dentro cr3 invalida tutto TLB: è la stessa cosa che succede quando si cambia trie, ed ha senso perché a quel punto non c'è più nessuna corrispondenza tra gli indirizzi fisici e i precedenti virtuali; questo metodo funziona anche se il cr3 si scrive lo stesso valore. Nei casi in cui modifichiamo uno specifico indirizzo virtuale, c'è un'istruzione che dice al TLB di invalidare questo accesso. Ovviamente non sappiamo via software cosa contiene TLB, quindi l'istruzione è del tipo *"invalida la traduzione di questo indirizzo se c'è"*. L'istruzione è invlpg, che vuole un operando indirizzo da passare a TLB. Con libce, si usa la funzione `invalida_entrata_TLB`. Questa funzione si chiama subito dopo aver modificato il trie; che valore gli dobbiamo passare? L'indirizzo virtuale dell'entrata che abbiamo modificato. L'unico aspetto è che per

lavorare con la memoria virtuale la libce usa i `typedef` `vaddr` e `paddr` equivalenti a `natq`, quindi serve un cast.

```

1  #include<libce.h>
2  #include<vm.h>
3
4  // L'indirizzo di var è 0x20f000
5  // 0x20f000 ==
6  // 0000 0000 0000 0000 0000 0010 0000 1111 0000 0000 0000 ==
7  // 000 000 001 017 | 0000
8  // numero di pagina | offset
9  int var;
10
11 // Con questa funzione creiamo la mappatura identità tra i primi
12 // 8MiB di memoria virtuale e la corrispondente memoria fisica.
13 extern "C" void setup_vm();
14 extern "C" natq tab4[], tab3[], tab2[], tab1_0[],
15     | | | | tab1_1[], tab1_2[], tab1_3[], tab1_4[];
16
17
18 int main(){
19     var = 1;
20     printf("Il valore di var è: %d\n", var); //stampa 1
21     pause();
22     setup_vm();
23
24     // Avendo una traduzione identità, facciamo puntare a p l'indirizzo fisico 0x400000
25     // usando l'indirizzo virtuale 0x400000
26
27     int *p = (int*) (0x400000);
28     *p = 3;
29
30     // TLB ha fatto l'accesso in memoria, e si è ricordata che il numero di frame associato
31     // alla pagina 0x400 è 0x400. Modifichiamo ora tale numero di frame.
32
33     tab1_0[0] = 0x20f000 | 0x07;
34
35     // Se non comunichiamo a TLB che abbiamo modificato il trie, accedendo all'indirizzo virtuale
36     // 0x400000 ci fornirà lo stesso frame.
37
38     invalida_entrata_TLB((vaddr)p);
39     *p = 2;
40     printf("Il valore di var è: %d\n", var); //stampa 2
41     pause();
42 }
43

```

Dentro `vm.h` di `libce` ci sono alcune funzioni di utilità per lavorare con la memoria virtuale. Tra le molte, si possono fare dei calcoli sulle dimensioni delle pagine, trovando l'indirizzo della regione di appartenenza; possiamo, dato un indirizzo virtuale, sapere a quale entrata della tabella di livello 4 corrisponde, o accedere ad un ingresso di una tabella di livello 1.

Molto spesso serve visitare il trie. In particolare, nell'ambito del nucleo è necessario inizializzarlo, scorrendolo e assegnando a ciascun'entrata di un indirizzo virtuale i `paddr` opportuno. Per fare questo, in `vm.h` c'è la classe `tab_iter`, che si occupa proprio di fare una visita anticipata. Come parametri al costruttore richiede l'indirizzo fisico della tabella di livello 4, l'indirizzo virtuale dal quale iniziare ad esplorare e il numero di posizioni da verificare. Una volta fermo su un'entrata di una tabella, possiamo chiedergli di modificare alcuni valori o di mostrarceli.

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

Sulla paginazione nel nucleo

Il modulo sistema del nucleo deve gestire sia la memoria fisica (i frame scelti per i dati) che quella virtuale (gli indirizzi con cui accedervi) di ciascun processo. M1 sarà la porzione di RAM ad uso esclusivo del modulo sistema: è il collegatore ad inserirvi del contenuto, restituendoci la terminazione di questa porzione puntata da end; dividiamo M2, la parte restante, in frame, e allochiamo in ciascuno sia le pagine che le tabelle, essendo entrambe di 4K e allineate tali. Non tutto quello che è in M2 è accessibile all'utente, ma solo quanto mappato in un indirizzo virtuale. Le tabelle, per esempio, non saranno mappate per l'utente (presentando magari $US=0$). Il sistema deve poi creare gli spazi di indirizzamento per ogni singolo processo. A priori decidiamo che gli indirizzi virtuali da 0 fino al buco sono dedicati al sistema (che in tal senso comprenderà anche il modulo IO, e in generale quello che ha a che fare con il privilegio sistema). La porzione sotto il buco ha indirizzi virtuali utilizzabili dall'utente.

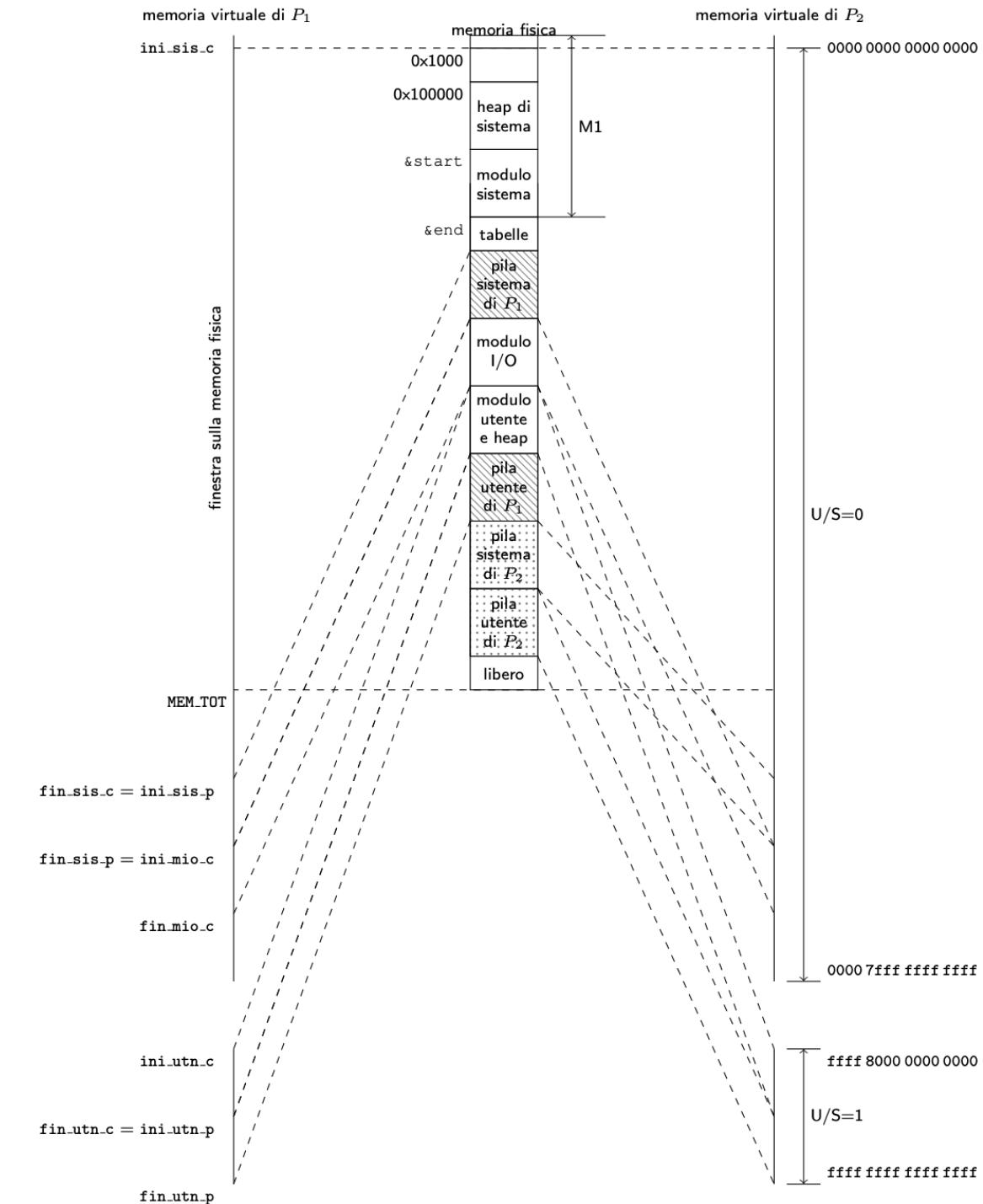
La porzione del livello sistema la dividiamo in tre parti: la prima parte è dedicata al modulo sistema, la seconda al modulo IO, la terza alla pila sistema del processo. Nella porzione del modulo sistema mettiamo le traduzioni che traducono gli indirizzi virtuali in sé stessi, sia per la RAM che per quello che deve essere accessibile essendo *memory mapped* nello spazio di indirizzamento fisico. La traduzione identità, detta anche finestra, coinvolgerà sia M1 che M2. Nella porzione del modulo IO e pila sistema ci saranno degli indirizzi le cui traduzioni portano comunque a frame di M2. In tal senso, non sarebbe necessario fare traduzioni per il modulo IO e la pila sistema, perché comunque già tutti i loro frame sono indirizzati dalla finestra nel modulo sistema; conviene tuttavia che ci sia tale traduzione. I frame della pila non sono contigui, ma allocati dove c'era spazio. Questo significa che gli indirizzi identità non mi permettono di scorrerli direttamente senza una struttura dati che li ricordi in progressione. Conviene allora usare indirizzi virtuali ad hoc per garantire continuità: per quanto ciascun frame della pila sistema sia un differente punto di M2, al sistema sembrerà tutto contiguo.

Oltretutto, usare la finestra corrisponde a disattivare la MMU, senza la possibilità della protezione: se sto usando indirizzi della finestra, non mi rendo conto se ho finito lo spazio della pila, mentre con una nuova traduzione posso porre $P=0$ quando non posso più scrivere.

Anche il livello utente si divide in due: nella parte bassa c'è la pila utente, in quella alta .text, .data e .lo heap. Queste pagine saranno mappate in qualche punto di M2. Questo è solo quello che riguarda un singolo processo; un altro processo avrà a sua volta uno spazio di indirizzamento, con opportune traduzioni. In particolare, le traduzioni del modulo sistema e del modulo IO sono esattamente le stesse, così come la prima parte della porzione utente, in cui ci saranno le variabili globali e le istruzioni del codice. Ciascun processo avrà invece come propria una pila sistema e una pila utente. Questo ci permetterà di risparmiare spazio nella realizzazione della traduzione: se tre delle cinque porzioni sono uguali, basta assegnare le stesse tabelle di livello 3 alle stesse entrate della tabella di livello 4 propria di ciascun processo.

```
// 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
```



Non è la MMU che sceglie quale livello di privilegio usare: è il software che, richiedendo un indirizzo virtuale, permette la verifica tra il livello di privilegio della CPU e quello presente nel trie per la traduzione della pagina. È importante essere consapevoli del contesto del codice che stiamo scrivendo; finora ci siamo preoccupati solo dell'atomicità, adesso bisogna anche pensare al punto in cui si trova il codice che stiamo leggendo e che traduzioni sta usando. Ovviamente questo problema ha a che fare con il livello sistema, non per l'utente. Poniamo che P_1 voglia modificare la pila sistema del processo P_2 con un'apposita primitiva. Non può scrivere agli indirizzi virtuali che userebbe P_2 per accedere alla propria pila, visto che durante

l'esecuzione della primitiva è attivo l'albero di P1. Deve allora usare il trie di P2 per trovare i frame nei quali è presente la pila sistema di P2, e modificarli con la finestra nel suo modulo sistema.

La suddivisione tra le 5 porzioni dello spazio di indirizzamento è fatta a priori. In particolare, ciascuna porzione avrà almeno un'entrata della tabella di livello 4, per un totale di almeno 512GiB a porzione. Le costanti che iniziano con 'I' indicano l'entrata di tab4, quelle con 'N' il numero delle entrate successive che le appartengono. Come vediamo dal prefisso, le uniche porzioni private di ciascun processo sono la pila sistema e la pila utente.

Sia lo spazio virtuale che lo spazio fisico sono stabiliti al momento della creazione del processo. In `des_proc punt_nucleo` è l'indirizzo virtuale della pila sistema del processo, mentre `cr3` è il puntatore (indirizzo fisico) alla tabella di livello 4 nella traduzione del processo in esame. Quando un processo viene caricato con `carica_stato`, carichiamo anche il valore di `cr3`, per spostarsi al suo spazio di indirizzamento. Evitiamo di sovrascrivere `cr3` se è uguale a quello che c'è già, per non invalidare TLB. Svuotare TLB, dal punto di vista delle prestazioni è un problema visti i numerosi accessi necessari per la traduzione. Modificando `%cr3`, non ci sono problemi perché avendo mappato la finestra sempre agli stessi indirizzi. Prima di sovrascrivere `%cr3`, in `%rip` c'è un indirizzo che ha senso nello spazio di indirizzamento del processo uscente; dopo, sarà interpretato nello spazio di indirizzamento del processo entrante, ma le due traduzioni sono uguali, garantendo quindi consistenza.

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

La fine di M1 è stabilita dal linker, che sa esattamente qual è la porzione di M1 necessaria, passandocela in `&end`. La traduzione attiva all'avvio è quella del bootloader, che ci consente di ignorare l'esistenza di MMU usando gli effettivi indirizzi fisici. I frame sono gestiti con una struttura dati `des_frame`; in vdf teniamo i descrittori di ogni possibile frame, calcolati dividendo la memoria disponibile per 4K. Un descrittore conterrà informazioni diverse sulla base che sia libero o meno, e per risparmiare spazio usiamo una union. Se il frame è libero, conterrà l'indice del frame libero successivo: quando si dovrà allocare, conosceremo anche il prossimo che potremo allocare. Nel caso in cui non fosse libero, ci interessa, se contiene una tabella, il numero di `P=0` che ci sono tra tutte le entrate: se tale contatore fosse a 0, allora la tabella sarebbe inutilizzata, ed è possibile deallokarla. Ovviamente terremo il contatore in uno stato consistente, aggiornandolo quando inseriamo un'entrata, decrementandolo quando poniamo `P` a 0 per un'entrata.

Il numero di frame in M1 è `N_M1`; i restanti sono i frame di M2. La funzione `init_frame` si occupa di inizializzare la tabella costruendo l'array di descrittori in modo consistente per il suo successivo uso. Ci sono poi due funzioni per allocare e deallocare un frame. Per allocare, ci prendiamo da una variabile globale il primo frame libero, modifichiamo tale variabile globale, sistemiamo alcuni valori e restituiamo l'indirizzo fisico del frame appena allocato (in modo che, magari, possa essere inserito come traduzione di una pagina).

```

498 // estreia un frame libero dalla lista, se non vuota
499 paddr alloca_frame(){
500     if (!num_frame_liberi) {
501         flog(LOG_ERR, "out of memory");
502         return 0;
503     }
504     natq j = primo_frame_libero;
505     primo_frame_libero = vdf[primo_frame_libero].prossimo_libero;
506     vdf[j].prossimo_libero = 0;
507     num_frame_liberi--;
508     return j * DIM_PAGINA;
509 }

```

Una cosa simile accade quando rilasciamo un frame: non possiamo rilasciare un frame di M1, per il resto si tratta di modificare le variabili globali per lasciare le cose in modo consistente.

```

512 // rende di nuovo libera il frame descritto da df
513 void rilascia_frame(paddr f){
514     natq j = f / DIM_PAGINA;
515     if (j < N_M1) {
516         panic("tentativo di rilasciare un frame di M1");
517     }
518     vdf[j].prossimo_libero = primo_frame_libero;
519     primo_frame_libero = j;
520     num_frame_liberi++;
521 }
522

```

Nell'ambito della paginazione, serve in un primo momento calcolare gli indirizzi di partenza e di fine di ciascuna delle 5 porzioni dello spazio di indirizzamento (stabilite a priori dal sistema tramite le macro in costanti.h). Le prime due funzioni che si incontrano servono per allocare un frame destinato a contenere una tabella o per rilasciarne uno che contiene una tabella (è importante distinguere l'allocazione per pagina o per tabella, dovendo andare a modificare i dati nella union del descrittore di frame). set_entry modifica l'entrata j-esima della tabella tab con il valore se; copy_des effettua la copia di n ingressi di una tabella a partire da i da una tabella src ad una dst.

map è la funzione più utile: gli si dice di partire dal trie di radice tab, e di creare una traduzione per tutti gli indirizzi per l'intervallo [begin, end), con certi flag. Per rendere generica la funzione, non sa dove prendere gli indirizzi fisici per ciascun virtuale, e richiede un tipo template getpaddr che ammetta una sintassi getpaddr(v). Posso ad esempio passare una funzione, una lambda o una classe in cui è ridefinito l'operatore parentesi. Per ricordarci l'indirizzo fisico che dobbiamo restituire, potremmo usare una classe con tutta una serie di strutture dati opportune. Si può anche settare page_size a livello maggiore di 1, per creare pagine di 2MiB.

```

// crea tutto il sottoalbero, con radice tab, necessario a tradurre tutti gli
// indirizzi dell'intervallo [begin, end]. L'intero intervallo non deve
// contenere traduzioni pre-esistenti.
//
// I bit RW e US che sono a 1 nel parametro flags saranno settati anche in
// tutti i descrittori del sottoalbero. Se flags contiene i bit PWT e/o PWT,
// questi saranno settati solo sui descrittori foglia.
//
// Il tipo getpaddr deve poter essere invocato come 'getpaddr(v)', dove 'v' è
// un indirizzo virtuale. L'invocazione deve restituire l'indirizzo fisico che
// si vuole far corrispondere a 'v'.
//
// La funzione, per default, crea traduzioni con pagine di livello 1. Se si
// vogliono usare pagine di livello superiore (da 2 a MAX_PS_LVL) occorre
// passare anche il parametro ps_lvl.
//
// La funzione restituisce il primo indirizzo non mappato, che in caso di
// successo è end. Un valore diverso da end segnala che si è verificato
// un problema durante l'operazione (per esempio: memoria esaurita, indirizzo
// già mappato).
template<typename T>
vaddr map(paddr tab, vaddr begin, vaddr end, natl flags, T getpaddr, int ps_lvl = 1)
{

```

map la usiamo per creare tutte le parti dello spazio di indirizzamento. Quelle condivise sono sempre le stesse, quindi le dobbiamo creare una sola volta. Se l'entrata 0 di tab4 di P1 è dedicata alla finestra, con un'opportuna tab3, alla creazione di P2 posso far puntare l'entrata 0 della propria tab4 alla stessa tab3, per non dover ripetere la finestra. Faremo questa cosa per tutte le parti condivise: modulo sistema, modulo IO, parte utente condivisa: punteranno tutti agli stessi sottoalberi. In pratica le creiamo per il processo dummy che è sempre vivo e ogni volta ci limitiamo a copiare nella nuova tabella di livello 4 le entrate corrispondenti del processo che ha chiamato activate_p.

La traduzione finestra la facciamo con crea_finestra_FM. Si passa la lambda identity_map per avere lo stesso indirizzo fisico rispetto all'attuale indirizzo virtuale. map creerà tutte le traduzioni identiche. Visto che non ho problemi nella traduzione identità, lo mappiamo con pagine grandi 2. La prima regione però non posso mapparla con pagine grandi 2, perché non vogliamo mettere P=1 alla prima pagina e intercettare dereferenziazioni a NULL. In più, serve impostare PWT per la memoria video testo, che si trova in M1. Mappiamo anche la parte dell'APIC settando PWT e PCD, oltre che RW.

Alla fine, ci troviamo tutte le entrate condivise caricate opportunamente in tab4. Ogni volta che viene creato un processo, si alloca una tab4 che funge da radice del trie. Si inizializza copiando dalla radice corrente le entrate di tutte le porzioni condivise. Serve poi creare la pila sistema e, eventualmente, la pila utente: anche in questo caso si usa map. La funzione unmap fa le cose opposte, deallocando le porzioni private.

L'I/O nel nucleo

Bisogna capire come gestire l'ingresso e uscita nel nostro sistema. Effettivamente era l'esempio dal quale eravamo partiti, ossia il motivo per cui poteva aver senso bloccare un processo nel momento in cui attendeva la fine di un'operazione di IO. Per come abbiamo realizzato la memoria virtuale, gli utenti non hanno accesso diretto ai registri delle periferiche. Sarà necessario offrire loro delle primitive per fare ingresso e uscita. La schematizzazione tipica di un'operazione di IO è una trasmissione di N byte da/a una periferica. Una generica chiamata avrà un riferimento alla periferica da usare, l'indirizzo di un buffer (dove scrivere o da dove leggere) e il numero di byte da trasferire. La primitiva dovrà iniziare l'operazione di lettura/scrittura dalla periferica. Questo tipo di approccio fa sì che l'utente non si preoccupi se la periferica era usata da un altro processo: è il sistema a gestirlo. Essenzialmente il processo chiamante non si rende conto che c'è stata un'attesa: il tutto avverrà al pari di una chiamata di funzione; un comportamento di questo tipo si definisce sincrono.

Da questa descrizione capiamo che sarà necessario bloccare un processo e schedularne un altro; dovremo avviare un'operazione di ingresso o di uscita, ma anche questo sappiamo come realizzarlo. Ovviamente l'operazione potrebbe richiedere più interruzioni da parte della periferica: volendo prelevare N byte dalla tastiera, servono N interruzioni successive.

Questo è il meccanismo di base, a cui contribuiscono le primitive (quelle chiamate dall'utente) e le interruzioni esterne (che fanno avanzare l'operazione, gestite dal modulo sistema). Nello specifico, per ogni periferica si dovrà scrivere un *driver* opportuno che ne gestisca l'interruzione.

Ci sono due problemi da risolvere. Finché P1 sta usando una periferica per leggere N byte, nessun altro può attivare una nuova operazione di lettura sulla stessa: si tratta di un problema di mutua esclusione. Vogliamo poi che P1 si risvegli dopo che l'operazione si è conclusa: è un problema di sincronizzazione. La soluzione sono, per entrambi i casi, i semafori.

Per dialogare con la periferica 0, serve prelevare un gettone dal semaforo di mutua esclusione; quando ha finito ci rimette dentro il gettone; se non trova nulla, si congela lì. Il processo, acquisita la mutua esclusione, avvia l'operazione e prova a prendere un gettone da un semaforo di sincronizzazione. Quando il driver della periferica ha concluso l'operazione, mette un gettone nel semaforo della sincronizzazione: il processo chiamante, riprendendo l'esecuzione, metterà un gettone nel semaforo della mutua esclusione e riprenderà l'attività. La struttura sarà la seguente:

```
sem_wait(mux)
avvio operazione
sem_wait(sync)
sem_signal(mux)
```

I driver che vanno in esecuzione ogni volta che c'è un'interruzione devono leggere un byte dalla periferica, scriverlo da qualche parte e ricordarsi quanti byte avevano ricevuto; se l'operazione è conclusa fanno `sem_signal(sync)`.

Una primitiva di questo tipo non è atomica, perché `sem_wait` e `sem_signal` possono mandare in esecuzioni altri processi. Per avere questa cosa, non possiamo salvare lo stato all'inizio dell'operazione, altrimenti non potrebbero essere interrotte in questo modo. In generale, la porzione Assembler di una primitiva non atomica non fa altro che chiamare la parte di C++ per poi fare `iretq`. Concettualmente non cambiamo il contesto del processo ma solo il privilegio (che, tra le altre cose, comporta il passaggio alla pila sistema).

I driver restano atomici, e in questo caso non possiamo fare altrimenti. Questi, infatti, stanno temporaneamente usando le risorse di un processo P2 che non c'entrano nulla, non essendoci stato nel frattempo un cambio di contesto. Se fossero interrotti, `salva_stato` salverebbe nel contesto di P2 alcuni valori dei registri che non hanno a che fare con l'esecuzione di P2; la `carica_stato` non riprenderebbe il processo per come era prima dell'interruzione. Ma se non può essere interrompibile, come facciamo a fare `sem_signal` alla fine del driver? Una via è quella di rimettere in coda pronti il processo che stava nel semaforo, modificando direttamente le strutture dati. Dentro un driver, possiamo chiamare funzioni che modifichino il valore di esecuzione, ma nulla che, generando un'interruzione esterna, provochi la chiamata di `salva_stato`. Sempre nel driver, bisogna inviare l'EOI all'APIC, per far sapere che l'interruzione è stata gestita.

```
a_driver:
    call salva_stato
    call c_driver
    call APIC_send_EOI
    call carica_stato
    iretq
```

Vediamo come deve essere realizzato `c_driver`. Immaginiamo di avere una periferica con RBR per la lettura e un registro CTL per abilitare o meno le interruzioni. Supponiamo che la periferica abbia un handshake, ossia non mandi un nuovo dato prima che sia stata fatta una lettura da RBR. Bisogna far sì che i parametri richiesti dalla primitiva arrivino alle routine di interruzione tramite variabili globali. Allo scopo di far dialogare la primitiva che avvia l'operazione e il driver prevediamo dei *descrittori di operazioni di IO*, in cui raccogliamo tutte le informazioni sull'operazione su una data periferica.

```
struct des_io{
    ioaddr iRBR;
    ioaddr iCTL;
    char* buf;
    natq quanti;
    natl mutex;
    natl sync;
};
```

Gli ultimi due sono gli indici dei semafori di mutua esclusione e di sincronizzazione. Facciamo un array di questi oggetti, `array_desio`, e usiamo l'identificatore numerico della chiamata della primitiva come un indice della periferica. Ogni periferica avrà un suo semaforo di mutua esclusione e di sincronizzazione. In effetti, abbiamo visto che i semafori allocati dagli utenti sono separati dai semafori del sistema, in modo che non ci siano problemi in tal senso. La primitiva scrive nel descrittore di operazione i parametri passati dall'utente, e lavora poi con i semafori. In questo modo possiamo avere tante istanze di una stessa periferica, distinguendole per indice.

`quanti` sarà un contatore del numero di byte che devono ancora essere trasferiti. Il carattere prelevato lo scriveremo dentro `buf`; essendo un puntatore, si potrà incrementare per arrivare alla posizione

successiva in cui scrivere. `c_sem_signal` non cambia lo stato, al massimo modifica il valore della variabile esecuzione.

```
void c_driver(int i){
    des_io * d = &array_desio[i];
    d->quanti--;
    if(d->quanti==0){
        c_sem_signal(d->sync);
    }
    char c = inputb(d->iRBR);
    *d->buf = c;
    d->buf++;
}
```

Il primo problema sta in `*d->buf = c`, perché, all'esecuzione del driver, il mapping attivo è quello del processo interrotto dal driver, non del chiamante P1: l'indirizzo potrebbe non essere mappato correttamente. Questo problema non lo abbiamo se `buf` è condiviso tra P2 e P1, e quindi mappato allo stesso modo nei due alberi di traduzione (questo accade se fa parte dello spazio utente condiviso, e quindi è una variabile globale); tuttavia, se `buf` è locale alla funzione, il driver scrive nella pila di P2, essendo quell'indirizzo associato a questa. Ci sono diverse possibilità per risolvere questa cosa: nel nostro caso, vietiamo che l'utente possa offrire un buffer locale. Faremo questi controlli dentro la primitiva.

In generale non ci dobbiamo fidare le buffer dell'utente: potrebbe tranquillamente passarci indirizzi di sistema che andremmo a sovrascrivere. Sarebbe un problema perché il sistema ha la possibilità di scrivere dove vuole. L'hardware non fa controlli, perché passando un indirizzo non stiamo accedendo, è solo un valore che passa di registro in registro. Questo problema prende il nome di *cavallo di Troia*. I controlli vanno fatti via software, prima dell'utilizzo degli indirizzi. Dobbiamo controllare che questo indirizzo fosse scrivibile dall'utente: se non aveva il diritto di farlo prima, non deve guadagnarla passandolo attraverso una primitiva. Il controllo va fatto anche sugli indirizzi successivi se `quanti>1`. In più non ci deve essere overflow, non dobbiamo avere un `quanti` talmente grande da tornare al punto di partenza.

Siamo sicuri che il driver non generi eccezioni? Magari l'utente ci ha fornito degli indirizzi che non sono mappati, oppure mappati lato utente su byte non scrivibili. Generare un eccezione nel driver è un problema, perché si perde l'atomicità: anche questi controlli devono essere effettuati. Nel modulo sistema si usa `c_access` per questo scopo. Si passa un indirizzo di partenza e una dimensione, con un booleano per sapere se si debba scrivere o meno. Appena troviamo qualcosa che non va restituiamo false.

```
// primitiva utilizzata dal modulo I/O per controllare che i buffer passati dal
// livello utente siano accessibili dal livello utente (problema del Cavallo di
// Troia) e non possano causare page fault nel modulo I/O (bit P tutti a 1 e
// scrittura permessa quando necessario)
extern "C" bool c_access(vaddr begin, natq dim, bool writeable)
```

Il processo chiama `read_n(id, buf, quanti)`; si passa a `a_read_n` e `c_read_n`. Il controllo sui parametri va fatta in `c_read_n`, in modo da non far avviare il driver nel caso in cui i parametri non siano validi. Controlliamo anche l'ID del descrittore di operazioni di IO. Se nel buffer si scrive per fare un'operazione di lettura, il flag `writeable` deve essere 1. Se la primitiva è non atomica, non dobbiamo chiamare `c_abort_p` ma `abort_p` per salvare lo stato.

Normalmente abilitiamo le interruzioni dalla periferica solo quando qualcuno vuole fare un'operazione di IO. Ma quando le disabilitiamo? Appena abbiamo `d->quanti` a 0. Infatti, la lettura da `d->iRBR` potrebbe comportare una nuova richiesta di interruzione da parte della periferica, e questa richiesta potrebbe non avere nessun dispositivo ad attenderla. Nel caso in cui ciò accada, viene chiamato il driver un'altra volta, e si eseguono operazioni non corretto, come scrivere nuovamente nel buffer, sporcando una porzione di memoria dell'utente non predisposta.

```

463     extern "C" void c_read_n(int id, char* buf, natq quanti){
464
465         //controllo parametri
466
467         des_io * d = &array_desio[id];
468         sem_wait(d->mutex);
469
470         //a questo punto è l'unico ad usare la periferica
471
472         d->quanti = quanti;
473         d->buf = buf;
474
475         //abilita la periferica a generare richieste di interruzioni
476         outputb(1, d->iCTL);
477
478         //aspettiamo che il driver abbia finito
479         sem_wait(d->sync);
480     }
481
482     void c_driver(int i){
483         des_io * d = &array_desio[i];
484         d->quanti--;
485         if(d->quanti==0){
486             outputb(0, d->iCTL);
487             c_sem_signal(d->sync);
488         }
489         char c = inputb(d->iRBR);
490         *d->buf = c;
491         d->buf++;
492     }

```

Un processo per eseguire un'operazione di IO invoca una primitiva non completamente atomica: le interruzioni sono disabilitate, sono vietate le eccezioni ma sono ammesse le int. In questo modo è possibile usare le primitive semaforica per gestire la mutua esclusione e la sincronizzazione (il processo si sospende in attesa che le operazioni di IO siano terminate). I driver interromperanno i processi in esecuzione, e al momento della conclusione risveglieranno P1 (che si trova dentro la primitiva di sistema) che riprende la sua esecuzione. Questo è il motivo per cui la primitiva non è atomica: finché non è scattata la foto del processo con `salva_stato`, supponiamo che sia il processo stesso a proseguire.

Un driver realizzato in questo modo è stringente, perché deve essere atomico, quindi non permette interruzioni annidate. Allo stesso tempo, non ha un posto dove salvare il proprio stato, non essendo un processo: non ha risorse destinate a lui, ma usa quelle del processo interrotto.

Introduciamo un nuovo modulo, cioè un binario non collegato con gli altri due. Finora, modulo utente e modulo sistema non sono stati collegati, non condividendo simboli. Il modulo IO comprende `io.cpp` e

io.s: gli diamo la responsabilità di eseguire le operazioni che finora abbiamo espresso in termini di primitive e driver.

Il fatto che una primitiva di IO lavora a livello sistema ma può essere interrompibile è un problema: lavorando su una struttura dati, potrebbe lasciarla inconsistente a causa di un'interruzione. Supponendo che il modulo sia completamente separato, ci vengono incontro collegatore e compilatore. Nel modulo IO non posso usare i simboli del modulo sistema, come esecuzione e pronti. Il codice del modulo IO lo eseguiremo a livello di privilegio sistema, in ogni caso, perché con solo due livelli di privilegio non è facile isolare un modulo intermedio. Sicuramente il livello utente non ha la possibilità di accedere alle periferiche. In generale le operazioni di IO non sono disabilitate a livello utente, ma dipende da IOPL dentro il registro dei flag. Potremmo allora eseguire il codice del modulo IO con il campo IOPL messo ad utente; così facendo il modulo IO potrebbe usare anche cli e sti, e non è proprio il caso, perché il sistema perderebbe il controllo del processore. Si potrebbe lavorare sulla memoria virtuale, ma sarebbe complicato. Per facilitare, il modulo IO sarà eseguito a livello sistema, con l'obbligo di fidarsi del suo comportamento. In linea di massima, possiamo pensare che il modulo IO sia scritto dai costruttori delle periferiche: il sistema lascia loro i privilegi senza farli accedere alle strutture dati di sistema, e ciò è possibile solo se c'è fiducia di consistenza tra i due.

Rendiamo le primitive di IO non atomiche, in modo che siano interrompibili. Per quanto riguarda il driver, li rendiamo dei processi, in modo che siano interrompibili. Non possono essere dei processi utenti, avendo dei privilegi in più; li chiameremo ‘esterni’: fanno parte del sistema e lavorano a tale privilegio, ma riguardano l’IO. Associato ad un processo abbiamo anche un piedino dell’APIC: il processo esterno si risveglia ogni volta che c’è un’interruzione da questo piedino. Il corpo del processo esterno è un ciclo infinito in cui fa le sue cose con la periferica e alla fine chiama *wait for interrupt* wfi.

```
void extern_proc(int i){
    //inizializzazione
    for(;;){
        //elaborazione
        wfi();
    }
}
```

La primitiva di IO blocca il processo chiamante e abilita le interruzioni, in modo che il processo esterno possa agire. Il vantaggio, rispetto al driver per come lo abbiamo visto è che tutto viene realizzato senza atomicità, con la possibilità di interruzioni. Inoltre, il processo esterno può usare le primitive di sistema, come la sem_signal per risvegliare colui che aveva invocato un’operazione di IO.

Aggiungiamo al modulo sistema alcune primitive per il modulo IO: activate_pe, wfi e handler. wfi la chiama un processo esterno quando si vuole bloccare in attesa di una nuova interruzione. In wfi possiamo anche inviare l’EOI. Il processo esterno sarà messo in esecuzione da un handler quando arriva l’interruzione. Il descrittore di processo esterno non deve essere inserito da nessun’altra parte, ci basta inserire il puntatore a des_proc in una tabella a_p

```

444    a_wfi:
445        call salva_stato
446
447        call apic_send_EOI
448
449        call schedulatore
450
451        call carica_stato
452
453        iretq

```

Ogni possibile interrupt ha un handler, che interrompe un processo alla ricezione di un'interruzione. Assumiamo che tutti i processi esterni abbiano una priorità maggiore dei i processi utente. L'handler dovrà salvare lo stato del processo interrotto e fare una schedulazione forzata: chiama `inspronti` e va nella tabella di descrittori di processi esterni per inserirlo in esecuzione.

```

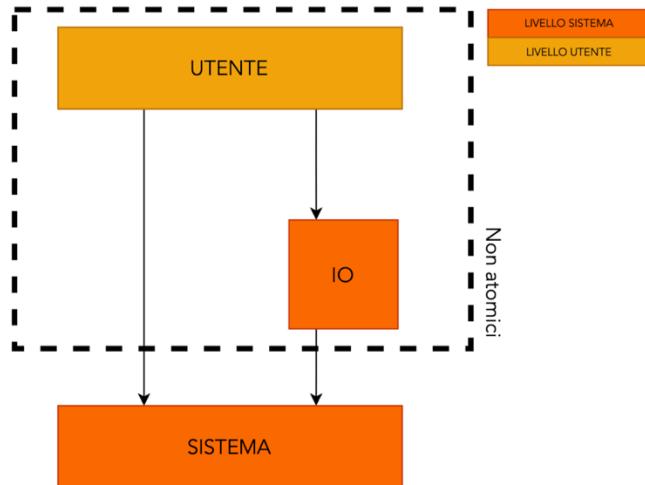
887    a_handler_i:
888        call salva_stato
889        call inspronti
890
891        // esecuzione = a_p[i]
892        movq $i, %rcx
893        movq a_p(, %rcx, 8), %rax
894        movq %rax, esecuzione
895
896        call carica_stato
897        iretq
898

```

Con la `iretq`, riprende il processo esterno all'inizio del ciclo. In particolare, aveva chiamato `wfi`, che ha eseguito `int $TIPO_WFI`. Lo stato salvato dalla `salva_stato` era quello, quindi l'istruzione immediatamente successiva è la `ret` in `wfi`.

Ma il processo esterno gira ad interruzioni abilitate, e può chiamare delle primitive che lo bloccano. In generale, se prendiamo un'istante qualunque della vita del sistema, e ci chiediamo in che stato sono i processi esterni, non è detto che siano tutti bloccati in attesa di un'interruzione: potrebbero essere stati interrotti a metà per molte ragioni che coinvolgono la `preemption`. È vero però che la `iretq` di `a_handler_i` è riporta il processo alla fine di `wfi`: infatti, per far partire l'handler, il processo esterno deve aver inviato l'`EOI`, e ciò è fatto proprio dentro `wfi`.

Adesso il nostro sistema ha un modulo sistema, che viene eseguito con livello di privilegio sistema, un modulo IO eseguito con livello di privilegio sistema e un modulo utente. IO e utente sono eseguiti con interruzioni abilitate. È importante a questo punto tenere conto che quello che accade nel modulo IO non è atomico.



Il modulo sistema mette a disposizione alcune primitive, sia per il modulo utente, come già visto, che per il modulo IO. Il modulo IO fornisce poi delle primitive al modulo utente. Tra quelle del sistema sono destinate esclusivamente al modulo IO, non devono essere usate dal modulo utente, come `activate_pe` e `wfi`. Come facciamo a far sì che solo IO li usi? Impostiamo il flag di sistema nella IDT. Lo abbiamo già usato per far sì che le entrate della IDT in riferimento a eccezioni e interruzioni esterne non fossero chiamate con una `int`. Questo è il motivo principale per cui è complicato realizzare un modulo IO che giri a livello utente: a quel punto serve un altro modo per avere delle primitive invocabili solo dal modulo IO. Con tale argomento in un primo momento i livelli di protezione erano 4.

Oltre a quelle viste, al modulo IO potrà far comodo una primitiva per scrivere dentro la IDT e associare un'entrata ad una primitiva per l'utente. Si realizza con `fill_gate`. Se si passa un buffer, dovrà verificare che il suo uso sia lecito, quindi avrà bisogno della primitiva `access`. In generale, i buffer forniti dall'utente devono stare nell'area utente condivisa, ma questi controlli non sono fatti in IO perché richiederebbero quelle costanti che invece fanno parte del modulo sistema. In ogni caso, se un indirizzo appartenesse alla pila utente, non darebbe problemi, perché un processo esterno, così come uno di livello sistema, non ha una pila utente.

Dentro `include`, ci sono le dichiarazioni nelle primitive offerte: `sys.h` sono quelle offerte dal modulo sistema, che vanno bene sia per utente che per IO; `io.h` contiene quelle che il modulo IO implementa per il modulo utente; `sysio.h` sono le primitive che il modulo sistema implementa per il modulo IO.

```
extern "C" natl activate_pe(void f(int), int a, natl prio, natl liv, natb type);
extern "C" void wfi();
extern "C" void abort_p();
extern "C" void io_panic();
extern "C" paddr trasforma(void* ff);
extern "C" bool access(const void* start, natq dim, bool writeable);
extern "C" void fill_gate(natl tipo, vaddr f);
```

Non essendo atomiche, le primitive di IO devono usare i semafori quando fanno uso di strutture condivise. Nel sistema non c'è questo problema perché sono atomiche, e girano ad interruzioni disabilitate.

Per far tornare un valore al chiamate non si può usare esecuzione->contesto perché inaccessibile: si può usare return, tanto non c'è salva_stato nel mezzo.

I processi esterni possono essere interrotti da interruzioni di privilegio ancora maggiore, ma questo ha a che fare con i privilegi dell'APIC, non del nostro sistema. Allora, tutti i processi esterni avranno una priorità minima, maggiore di quella massima utente, e tra di loro ci sarà una priorità diversa che coincide con il tipo dell'handler associato. activate_pe quindi associa anche un handler all'entrata della IDT corrispondente a quel piedino. In questo modo l'idea di precedenza nell'APIC coincide con quella dei processi nel modulo IO.

In fase di inizializzazione il sistema crea un primo processo IO e gli fa eseguire un main. Le primitive del modulo IO sono impostate come trap: le interruzioni non vengono disabilitate.

È possibile configurare qemu per aggiungere delle periferiche nuove; immaginiamo di dialogare con una periferica con RBR per leggere un dato, STS di stato e CTL per il controllo. Scriviamo le opportune funzioni.

```

985  bool ce_init(){
986      //conoscendo vendorID e deviceID, cerco tutte le periferiche di quel tipo
987      for (natb bus = 0, dev = 0, fun = 0; pci_find_dev(bus, dev, fun, 0xedce, 0x1234);
988          pci_next(bus, dev, fun)){
989          if (next_ce >= MAX_CE) {
990              flog(LOG_WARN, "troppi dispositivi ce");
991              break;
992          }
993          //nuovo descrittore di operazione IO
994          des_ce *ce = &array_ce[next_ce];
995          //leggo il base address, da cui rimuovo l'ultimo bit che indica IO
996          //gli altri bit sono gli indirizzi ai quali sonoallocati i registri
997          ioaddr base = pci_read_conf1(bus, dev, fun, 0x10);
998          base &= ~0x1;
999          ce->iCTL = base;
1000         ce->iSTS = base + 4;
1001         ce->iRBR = base + 8;
1002         //inizializzo i semafori opportunamente
1003         ce->sync = sem_ini(0);
1004         ce->mutex = sem_ini(1);
1005         //trovo scritto anche irq
1006         natb irq = pci_read_conf1(bus, dev, fun, 0x3c);
1007         activate_pe(extern_ce, next_ce, MIN_EXT_PRIO+irq, LIV, irq);
1008         flog(LOG_INFO, "ce id %2x:%1x:%1x base=%4x IRQ=%d", next_ce, bus, dev, fun, ce->iSTS, irq);
1009         next_ce++;
1010     }
1011     return true;
1012 }
```

Si tratta di una periferica PCI, quindi le dobbiamo trovare conoscendo, vendorID e deviceID. Abbiamo un array di descrittori di periferiche e lo riempiamo opportunamente. Leggiamo il registro base, togliendo l'ultimo bit dal valore ottenuto (stava ad indicare IO) per ottenere gli indirizzi dei registri. Nello spazio di configurazione c'è scritto anche irq, il numero di piedino dell'APIC a cui è collegato. Si suppone che il PCI bios lo abbia scritto correttamente. Attiviamo poi il processo esterno. Gli passiamo anche l'identificatore della periferica da gestire, poiché ogni periferica avrà un processo diverso a gestirla. Associamo la primitiva per la lettura ad un ingresso della IDT perché l'utente la possa usare.

```
fill_io_gate    IO_TIPO_CEREAD  a_ceread_n
```

Quando l'utente chiama `read_n` ci passa l'id dell'istanza della periferica di tipo CE da cui vuole leggere, il buffer e il numero di elementi da leggere.

```

946     extern "C" void c_ceread_n(natl id, char *buf, natl quanti){
947         if (id >= next_ce) {
948             flog(LOG_WARN, "ce non riconosciuto: %d", id);
949             abort_p();
950         }
951         des_ce *ce = &array_ce[id];
952         //aspettiamo di avere la mutua esclusione su quella periferica
953         sem_wait(ce->mutex);
954         ce->buf = buf;
955         ce->quanti = quanti;
956         //abilitiamo le interruzioni
957         outputb(1, ce->iCTL);
958         //aspettiamo il termine dell'operazione
959         sem_wait(ce->sync);
960         //rilasciamo il semaforo di mutua esclusione
961         sem_signal(ce->mutex);
962     }

```

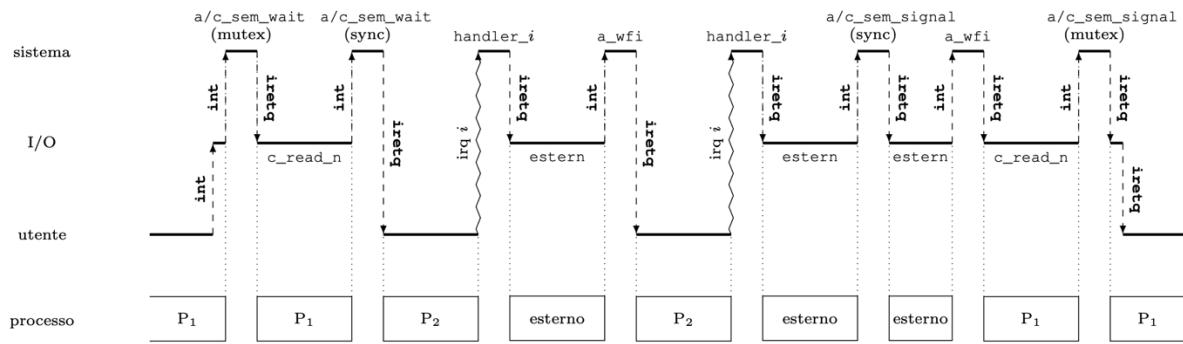
Da questo punto in poi il processore sarà assegnato ad altri processi, ma questi processi saranno interrotti quando la periferica ha dei dati da trasferire. Manda in esecuzione l'handler, che mette in esecuzione il processo esterno.

```

968     extern "C" void estern_ce(int id){
969         //Eseguito sono la prima
970         des_ce *ce = &array_ce[id];
971         natb b;
972         for (;;) {
973             ce->quanti--;
974             if (ce->quanti == 0) outputb(0, ce->iCTL);
975             b = inputb(ce->iRBR);
976             *ce->buf = (char)b;
977             ce->buf++;
978             if (ce->quanti == 0) sem_signal(ce->sync);
979             wfi();
980         }
981     }

```

A differenza del driver, con la `ce->sync` stiamo segnalando che l'operazione è conclusa, senza avere l'atomicità. Non ci saranno problemi perché il processo esterno ha priorità maggiore di quelli utente, ma questa è una convenzione, e in una versione futura del sistema potrebbe venir meno. Quindi, nonostante la convenzione, non dobbiamo dimenticare la semantica dell'operazione: si segnala la fine dell'operazione solo quanto è effettivamente conclusa. Piuttosto, non è comunque necessario disabilitare le interruzioni prima di leggere l'ultimo byte, per non riceverne una nuova.



Le linee orizzontali sono i codici in esecuzione di un certo modulo. Poniamo che P1 voglia fare la `read_n` di due byte: la chiamata con una `int` ci porta al modulo IO con `c_read_n`. Non è stata eseguita una `salva_stato`, quindi siamo ancora nel processo. Con `sem_wait`, il processo P1 si blocca. Tutto quello che riguarda il modulo sistema è eseguito in maniera atomica, avendo istruzioni esterne mascherate e imponendoci di non usare `int`. Le interruzioni esterne sono riabilitate con `iretq`, con il prelievo dalla pila del registro dei flag salvato. Con la `carica_stato`, si continua l'esecuzione della `c_read_n`, eseguita anch'essa da parte di P1. Nel momento in cui invochiamo `sem_wait` del semaforo di `sync`, si blocca fino al termine delle operazioni di IO, poi riprende a quel punto alla fine dell'operazione. Durante l'esecuzione di P2, si ha un'interruzione esterna, con un handler che mette in esecuzione forzatamente il processo esterno. Questo chiama `a_wfi`, che schedula P2 e così via. Nella fase finale dell'operazione di IO, il processo esterno sveglia il processo chiamante (ma non va in esecuzione, perché ha priorità maggiore). P1 riparte da `c_read_n` da dove si era fermato; con la `sem_signal`, rilascia la mutua esclusione, termina la `read_n` e si torna a livello utente.

La primitiva di IO potrebbe non bloccarsi al semaforo se il dato era già stato preparato nel tempo tra l'avvio dell'operazione e il semaforo. Questo è possibile perché la primitiva è interrompibile.

È bene tenere a mente anche quale sia lo spazio di indirizzamento di un processo (quale albero di traduzione è puntato da `%cr3`). Quando è in esecuzione il sistema in modo atomico, si sfrutta l'albero del processo in esecuzione. Si sta ancora usando lo spazio di indirizzamento dell'ultimo processo in esecuzione, cosa possibile sfruttando la finestra presente in ogni albero di traduzione.

Vediamo come fare operazioni DMA. Abbiamo, collegata al ponte PCI, una periferica che consente di fare Bus Master. Supponiamo che ci siamo delle primitive di sistema che permettano di effettuare questo tipo di operazione: `dma_read_n(int id, char* buff, natl quanti)`. Data anche la struttura, l'idea è la stessa delle operazioni già viste: la chiamata della funzione comporterà il passaggio ad una primitiva di IO con la sequenza 'richiesta mutex' - 'avvio operazione' - 'attesa del semaforo di sync' - 'rilascio del mutex'. L'avvio corrisponde a comunicare opportune informazioni alla periferica; nel momento in cui tutto è pronto, l'operazione si svolge lato hardware, mentre la CPU esegue tutt'altro. Al termine, ricevendo l'interruzione dalla periferica, si fa una lettura da un registro di stato: la politica FIFO del bus PCI fa sì che, terminata la lettura dal registro, anche le operazioni precedenti siano terminate. Il processo esterno, quindi, deve: - fare l'acknowledgment della richiesta di interruzione, che ha anche l'effetto di un flush per le operazioni di trasferimento tra il Bus Master e la RAM - fare `sem_signal` per svegliare il processo utente che era bloccato dentro la `dma_read_n`.

Queste cose vanno composte con la presenza della memoria virtuale. L'indirizzo del buffer che l'utente ha passato alla `dma_read_n` è virtuale e sta nella parte utente (condivisa se il buffer era dichiarato

globale, privata se l'ha dichiarato in una funzione). Dal punto di vista software, noi trasferiamo le informazioni, ma i trasferimenti sono svolti solamente dal Bus Master che vive nello spazio di indirizzamento fisico. Per adeguare le due parti, serve percorrere l'albero di traduzione per conoscere l'indirizzo fisico associato al buffer. Il modulo IO potrebbe fare il percorso del trie, avendo privilegio sistema; tuttavia, per rendere netta la distinzione tra modulo IO e sistema, forniamo la primitiva `trasforma`, che restituisce il `paddr` o `0` in caso di errore. Il modulo IO può chiamare questa primitiva, ed è il sistema ad occuparsene.

Tuttavia, se il buffer fosse grande, servirebbero più pagine, ma queste potrebbero essere associate a frame non contigui! Incrementando l'indirizzo di partenza, non proseguiamo nella porzione di memoria del buffer, ma in frame che non aveva nulla a che fare con quello di partenza. La periferica non ha idea di cosa sia la memoria virtuale, e continua degli indirizzi contigui. Otteniamo quindi due effetti dannosi: chi ha fatto la richiesta non ha tutti i byte, e abbiamo sporcati dei frame che potevano contenere qualsiasi cosa.

Per risolvere questo problema, dipende cos'è in grado di fare la periferica. Per l'hard disk, si poteva preparare in memoria una tabellina con cui fare vari trasferimenti. Gli si comunica dove si trova questa tabella, e da sola legge la tabella secondo un certo formato per capire quali trasferimenti fare (da quale indirizzo e quanti byte). L'ultima doveva avere un flag che comunicava il termine. Se la periferica non è in grado di fare una cosa del genere, facciamo il trasferimento in più passi: in ognuno gestiamo un frame, e ricevendo un'interruzione sappiamo che c'è da cambiare frame da comunicare (tramite la pagina a cui siamo arrivati).

Nei trasferimenti non in DMA, a scrivere o a leggere nel buffer è il processo esterno, ma in questo momento è attiva la memoria virtuale del processo esterno, ed è lui che fa il trasferimento. Non potendo allora accedere alla memoria utente privata del processo che ha avviato l'operazione di IO, gli stessi indirizzi virtuali non hanno significato. In questo caso non è più necessario che sia nella parte condivisa, perché il bus master lavora con indirizzi fisici, che sono per loro natura condivisi. Se il buffer di P1 stava in pila, continuerà a corrispondere agli stessa indirizzi fisici anche se in esecuzione c'era un altro processo. L'unica cosa che può dar fastidio è che il sistema decida di togliere un processo dalla memoria perché magari bloccato e la memoria è pieno (swap). Che succede se, mentre è in corso un'operazione di bus master, il sistema scegli di togliere tale processo dalla memoria e mettere al suo posto altro? Il bus master non sa questa cosa, quindi continuerà a scrivere a certi indirizzi fisici. Se al sistema viene in mente di togliere P1 dalla memoria e mettercene un altro. Bisogna allora impedire che un processo che fa bus master possa essere rimosso dalla memoria.