Università degli Studi di Milano

Dipartimento di Informatica

Il Sistema operativo JOS: appunti per il corso di SISTEMI OPERATIVI AA 14/15

Danilo Bruschi, Andrea Di Pasquale

0.1 Introduzione

Scopo della presente dispensa è descrivere in dettaglio le componenti principali del sistema operativo JOS con l'obiettivo di fornire le conoscenze necessarie per poter comprendere il funzionamento di un sistema operativo, almeno nelle sue componenti fondamentali.

Il sistema operativo utilizzato in questo senso è il sistema operativo JOS usato nell'ambito del corso 6.828 del MIT, e progettato per scopi puramente didattici. Per la ricchezza delle funzionalità di cui dispone, crediamo che questo sistema operativo sia un buon punto di partenza per chiunque sia interessato a conoscere approfonditamente un qualunque sistema operativo. JOS è sviluppato su architettura Intel IA-32, l'apprendimento delle sue funzionalità non possono quindi prescindere dalla conoscenza dell'architettura sottostante.

Capitolo 1

L'architettura IA-32: cenni

L'architettura Intel è una architettura che nasce nel 1979 con i processori 8086, 8087, 8088 e 80186, ancora usati in applicazioni che non richiedono grosse risorse computazionali.

Nella sterminata famiglia dei processori Intel ricordiamo il processore 80286, con il quale Intel introduce la modalità di gestione della memoria nota come protected mode, che studieremo più approfonditamente in seguito e che è la modalità con cui operano i principali sistemi operativi come Windows e Linux. Il Protected mode del 80286 impone una gestione segmentata della memoria, questo significa che l'HW provvede attraverso la MMU alla traduzione degli indirizzi seguendo uno schema di segmentazione e quindi presuppone la presenza di una tabella dei segmenti per effettuare la traduzione dello spazio di indirizzamento logico di un processo al suo corrispondente spazio di indirizzo fisico. Con questo schema quindi lo spazio di indirizzamento di un processo è protetto da accessi non autorizzati provenienti da altri processi, ed è possibile implementare la memoria virtuale.

Va poi menzionato il processore 80386, primo processore a 32 bit, attraverso il quale Intel introduce il supporto per la paginazione, o più precisamente le segmentazione paginata poiché sui processori Intel a 32 bit non è possibile disabilitare la segmentazione.

Seguono poi le famiglie Pentium, XEON, Celeron, Itanium, ICore, che aggiungono incrementalmente nuove caratteristiche per rendere sempre più potente in termini computazionali il processore. Va però ricordato che la strategia Intel è basata sulla back compatibilità dei prodotti, quindi ogni nuovo prodotto consente l'esecuzione di tutti i programmi eseguibili su processori dipendenti. Vediamo ora alcuni dettagli sull'architettura hw e alcuni richiami sul linguaggio assembler di riferimento.

1.1 Modalità operative del processore

L'architettura Intel IA-32 (a partire da Intel 386) fornisce una serie di funzionalità per la realizzazione di software di sistema. Queste funzionalità variano in funzione delle modalità operative con cui opera il processore. Operare in modalità diverse significa eseguire istruzioni e sfruttare o meno componenti HW presenti sul processore. L'uso delle diverse modalità operative consente ad un stesso processore Intel di poter operare con diversi instruction set e quindi garantire la retro compatibilità. Le modalità operative dei processori Intel sono:

1. Real-address mode: è la modalità operativa in cui il processore fornisce le stesse funzionalità presenti nell'Intel 8086 con l'aggiunta di alcune estensioni (la possibilità di cambiare modalità operativa in Protected o System Management mode). Quindi quando un qualunque processore opera in Real-address mode, lavora esattamente come un 8086 che è in grado di indirizzare 640 KB di memoria centrale ed esegue istruzioni a 16 bit. Il Real mode è la modalità di avvio di un qualunque processore Intel.

2. Protected mode: è la modalità nativa del processore e fornisce un insieme di funzionalità predefinite per la realizzazione di sistemi multiprogrammati garantendo la retro compatibilità. Quindi la modalità in cui il processore offre le maggiori funzionalità a 32 bit e quindi il sistema HW è in grado di supportare segmentazione e paginazione

3. Virtual-8086 mode: è la modalità che consente al processore di eseguire software predisposto per l'architettura 8086 in un ambiente multiprogrammato e protetto.

4. SMM mode: introdotto nel 1990 sul processore 386SL, è una modalità di esecuzione assolutamente protetta che viene attivata in risposta ad un particolare segnale di Interrupt (chiamato SMI) a seguito del quale viene "congelata´´ l'attività del processore che procede con l'esecuzione di codice memorizzato in una zona di memoria riservata della memoria nota come SMRAM (System Management RAM). zona di memoria assolutamente non visibile al resto del sistema. Quindi è una modalità di esecuzione estremamente potente, tant'è vero che il codice memorizzato in questa porzione di memoria viene eseguito ad un livello di priorità estremamente elevato, ovvero Ring -2. Ricordiamo che Intel mette a disposizione 4 livelli, ovvero Ring 0, Ring 1, Ring

2 e Ring 3 ma i sistemi operativi Windows e Unix usano solo Ring 0 (kernel) e Ring 3 (user code).

5. IA-32e mode: Con l'avvento delle architetture a 64 bit, è stata introdotta la nuova modalità operativa IA-32e mode che consente di sfruttare appieno le potenzialità di queste architetture. Più precisamente il software può essere eseguito in due sotto modalità:

   - 64 bit mode per il supporto di applicazioni e sistema operativo a 64 bit
   - Compatibility mode che consente ad un sistema operativo a 64 bit di gestire applicazioni a 64 e 32 bit

   Riportiamo di seguito il diagramma delle transizioni tra le diverse modalità operative dei processori Interl, ricordando che la modalità iniziale è sempre il Real Address Mode.

1.2 Protected Mode

La modalità Protected è stata introdotta per facilitare la soluzione di problemi di "sicurezza" derivanti dalla multiprogrammazione, quindi protezione e separazione dei vari componenti di un sistema. A tal fine l'architettura IA-32 include meccanismi di protezione e separazione che si prefiggono i seguenti obiettivi:

- Impedire a un processo di modificare l'area di morena occupata da un altro processo (restrizione sulla visibilità della memoria tra processi);
- Impedire a un processo di accedere direttamente alle risorse del kernel del sistema operativo (nuovamente restrizione sulla visibilità della memoria tra processi e kernel);
- In caso di errore in un processo, assicurare la sopravvivenza del sistema operativo e non compromettere la funzionalità degli altri processi.

I meccanismi predisposti dall'HW per risolvere i suddetti problemi sono basati su due principi:

isolamento: ad ogni processo utente è assegnato uno spazio di indirizzamento diverso, quindi processi diversi non hanno modo di interferire tra loro, se non attraverso sistemi controllati di IPC (Inter Process Communication); ovviamente la porzione di memoria dedicata al sistema operativo non può essere acceduta da processi utente;

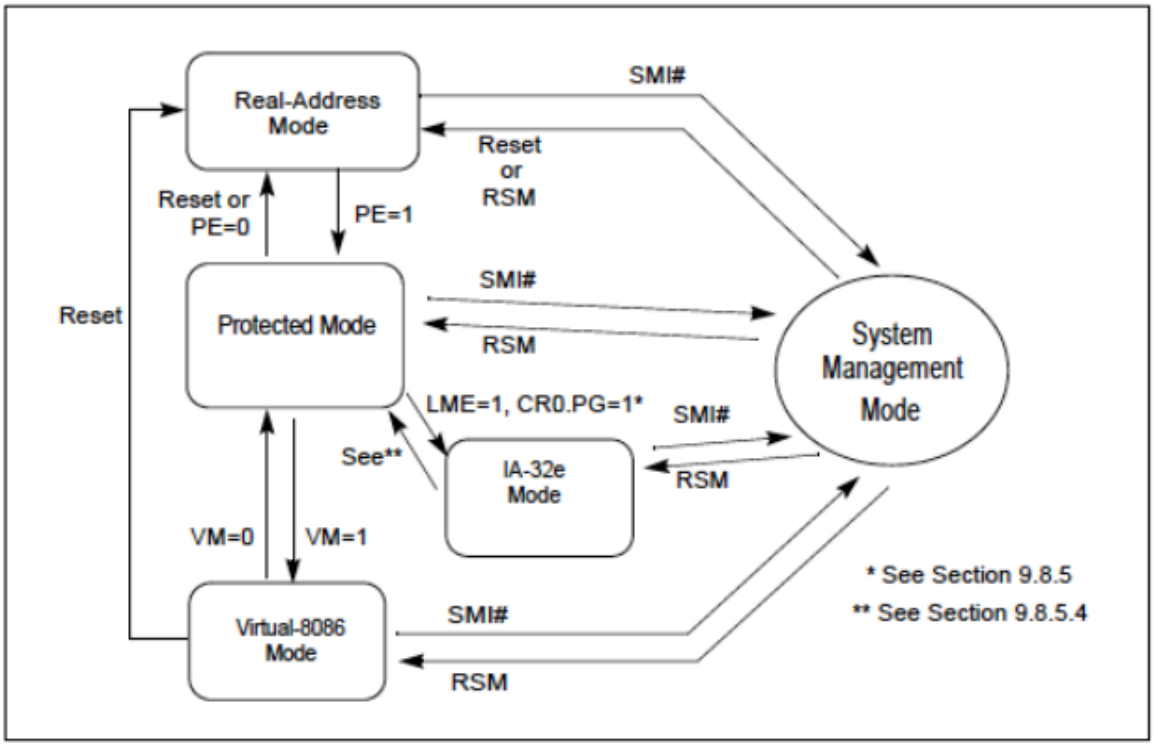

Figure 2-3. Transitions Among the Processor's Operating Modes

Figura 1.1: Diagramma delle Transizioni

Protezione: ai diversi oggetti che popolano il sistema sono assegnati livelli di protezione detti Ring (Ring -2, Ring -1, 0, Ring 1, Ring 2, Ring 3), e a livello HW viene controllato che oggetti con livello di protezione Ring k possano accedere solo ad oggetti con un livello di protezione uguale o maggiore a k, sono ovviamente previsti meccanismi di privilege escalation che consentono la modifica, solitamente temporanea, di tali valori.

1.2.1 I segmenti

Il principio di Isolamento è implementato attraverso la segmentazione della memoria centrale, cioè porzioni di codice di memoria disgiunte in cui sono caricati i processi, in cui distinguiamo solitamente almeno tre segmenti codice, data, stack (quest'ultimo spesso assimilato al segmento dati). Sull'architettura IA-32 un processo è rappresentato dai seguenti segmenti:

1. un segmento codice;
2. un segmento dati/stack;
3. un apposito segmento di sistema che studieremo successivamente e noto come Task State Segment (TSS);
4. Un ulteriore segmento (opzionale) che è costituito da opportune tabelle, con l'elenco degli oggetti accessibili a un processo, noto come Local Descriptor Table (LDT).

Ogni segmento è definito da un'apposita struttura dati nota come *descrittore*. Un descrittore di segmento fornisce al processore informazioni sulla dimensione e l'indirizzo di partenza di un segmento, e per il controllo degli accessi al segmento. I descrittori di segmento hanno un formato specificato dal progettista HW ma devono essere generati dal sistema operativo. Più precisamente un descrittore di segmento contiene:

- Base address: è un numero di 32 bit che definisce l'indirizzo di partenza del segmento;
- Segment Limit: è un numero di 20 bit che specifica la dimensione del segmento. La dimensione del Segment Limit è specificata in numero di pagine (4 KB);
- Diritti di accesso che specifica:
  - Se il segmento contiene Codice o Dati;

– Se i dati sono Read-only e quindi in sola lettura o anche scrivibili
– Il livello di privilegio del segmento.

Riportiamo qui di seguito lo schema dettagliato di un segment descriptor e la corrispondente dichiarazione in linguaggio C dichiarata facendo ricorso al bit-field del C che ci consente di specificare grandezze la cui dimensione è qualche bit.

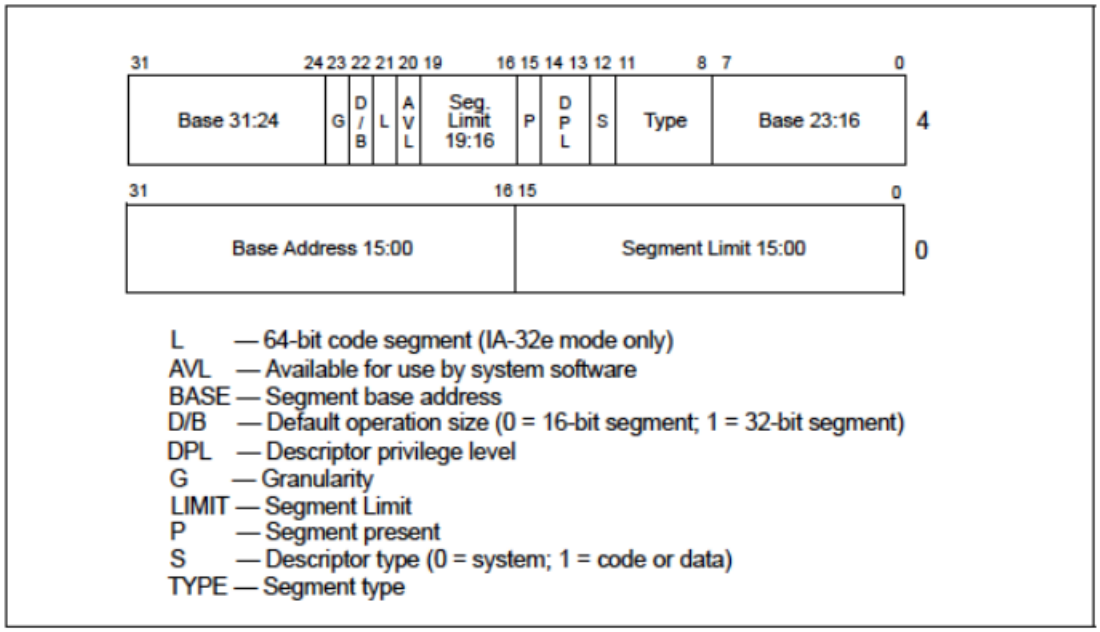

Figure 3-8. Segment Descriptor

Figura 1.2: Struttura Descrittore di Segmento

I descrittori di segmento vanno memorizzati in apposite tabelle dei segmenti note come Descriptor Table. Queste sono le tabelle principali:

GDT o Global Descriptor Table è una tabella contente i descrittori dei principali segmenti presenti all'interno del sistema. La GTD non è di per se un segmento, perché è una struttura primitiva e quindi non ha descrittori che la descrivono. Questo vuol dire che quando il sistema prepara la GTD, il sistema non si deve preoccupare di preparare un descrittore per la GTD, perché l'HW non va a cercare la GTD, ma accede direttamente alla GTD tramite GTDR.

```
// Segment Descriptors
struct Segdesc {
    unsigned sd_lim_15_0 : 16;  // Low bits of segment limit
    unsigned sd_base_15_0 : 16; // Low bits of segment base address
    unsigned sd_base_23_16 : 8; // Middle bits of segment base address
    unsigned sd_type : 4;       // Segment type (see STS_ constants)
    unsigned sd_s : 1;          // 0 = system, 1 = application
    unsigned sd_dpl : 2;        // Descriptor Privilege Level
    unsigned sd_p : 1;          // Present
    unsigned sd_lim_19_16 : 4;  // High bits of segment limit
    unsigned sd_avl : 1;        // Unused (available for software use)
    unsigned sd_rsv1 : 1;       // Reserved
    unsigned sd_db : 1;         // 0 = 16-bit segment, 1 = 32-bit segme
    unsigned sd_g : 1;          // Granularity: limit scaled by 4K when
    unsigned sd_base_31_24 : 8; // High bits of segment base address
};
```

Figura 1.3: Dichiarazione C del Descrittore di Segmento

LDT o Local Descriptor Table è una tabella di descrittori di segmenti locali e visibili solo dal processo a cui la LDT è associata.

IDT o Interrupt Descriptor Table è una tabella contenente i descrittori speciali di particolari segmenti chiamati Gates introdotti per abilitare la privilege escalation. é la struttura dati che useremo per la gestione degli interrupt e trap.

L'accesso ai descrittori di segmento presenti in una tabella dei segmenti avviene attraverso apposite strutture dati note come **selettori**, che posseggono la seguente struttura:

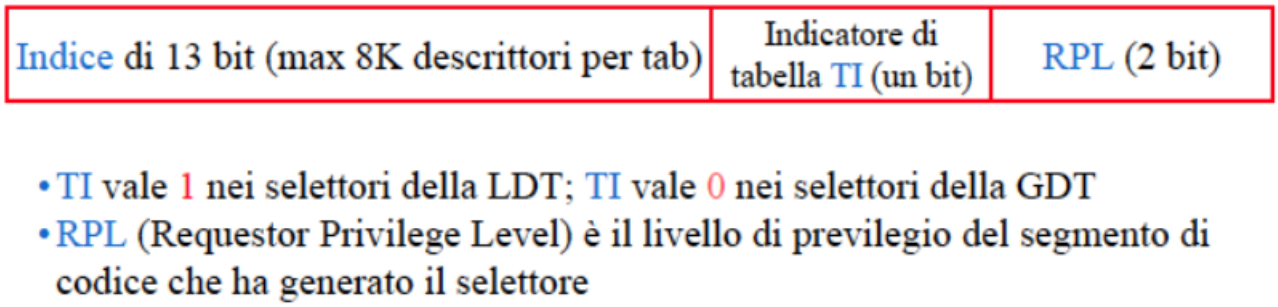

Figura 1.4: Tracciato di un Segment Selector

1.3 I registri di IA-32

Dopo aver velocemente elencato le principali tabelle di sistema che ricordiamo essere:

1. Global Descript Table (GDT)
2. Local Descriptor Table (LDT)
3. Interrupt Descriptor Table (IDT)

introduciamo i registri che l'architettura mette a disposizione per accedere a queste tabelle:

- GDTR o Global Descriptor Table Register: un registro che punta all'indirizzo di memoria della GDT;

- LDTR o Local Descriptor Table Register: un registro aggiornato ad ogni operazione di Context Switch che punta all'indirizzo di memoria della LDT (quando presente) del processo in esecuzione;
- IDTR o Interrupt Descriptor Table Register: un registro che punta all'indirizzo di memoria della IDT;
- TR o Task Register: un registro che punta all'indirizzo di memoria della TSS del processo in esecuzione sulla CPU.

Quindi registri sono accessibili in scrittura solo attraverso opportune istruzioni privilegiate. Esistono poi una serie di altri registri usati dal processore per controllare e gestire l'intero sistema. Generalmente sono registri accessibili (almeno in parte) dal sistema operativo con istruzioni privilegiate. I principali di questi registri sono:

1. EIP o Extended Instruction Pointer: è il Program Counter o PC su architettura IA-32 e punta all'indirizzo di memoria che contiene la prossima istruzione da eseguire;
2. EFLAGS: è il registro che contiene una serie di informazioni rispetto sullo stato del processo in esecuzione;
3. Una serie di Control Register (CR0, CR2, CR3 e CR4): sono registri che contengono informazioni che usa il processore.

Esistono poi 6 registri che contengono i settori dei segmenti su cui il sistema su operando. Si tratta di registri costituiti da una parte accessibili di 16 bit, ed una parte riservata all'hardware di 64 bit che contiene l'intero descrittore di un segmento. Qui di seguito sono riportati questi registri:

Un altro registro di particolare interesse è il registro **EFLAGS** a 32 bit, il cui contenuto riportiamo qui di seguito

EFLAGS contiene una serie di informazioni utili ma quelle che ci interessa di più è il flag **IF** (Interrupt Enable Flag) quando questo flag vale 1 gli interrupt sono abilitati, varrà 0 in caso contrario. Per disabilitare gli interrupt, è possibile utilizzare l'istruzione privilegiata **cli** ovvero Clear Interrupt. Mentre per abilitare gli interrupt è possibile utilizzare l'istruzione privilegiata **sti**.

Esistono poi i registri general purpose, si tratta di registri a 32 bit, in cui è consentito anche l'accesso al singolo byte o word in funzione del nome utilizzato. Ad esempio con EAX accediamo all'intero registro di 32 bit, con AX solo ai 16 bit meno significativi di EAX, con AL al byte meno

Visible part	Invisible part	
Segment selector	Segment base address, size, access rights, etc.	CS
Segment selector	Segment base address, size, access rights, etc.	SS
Segment selector	Segment base address, size, access rights, etc.	DS
Segment selector	Segment base address, size, access rights, etc.	ES
Segment selector	Segment base address, size, access rights, etc.	FS
Segment selector	Segment base address, size, access rights, etc.	GS

Figura 1.5: Segment Selector Register

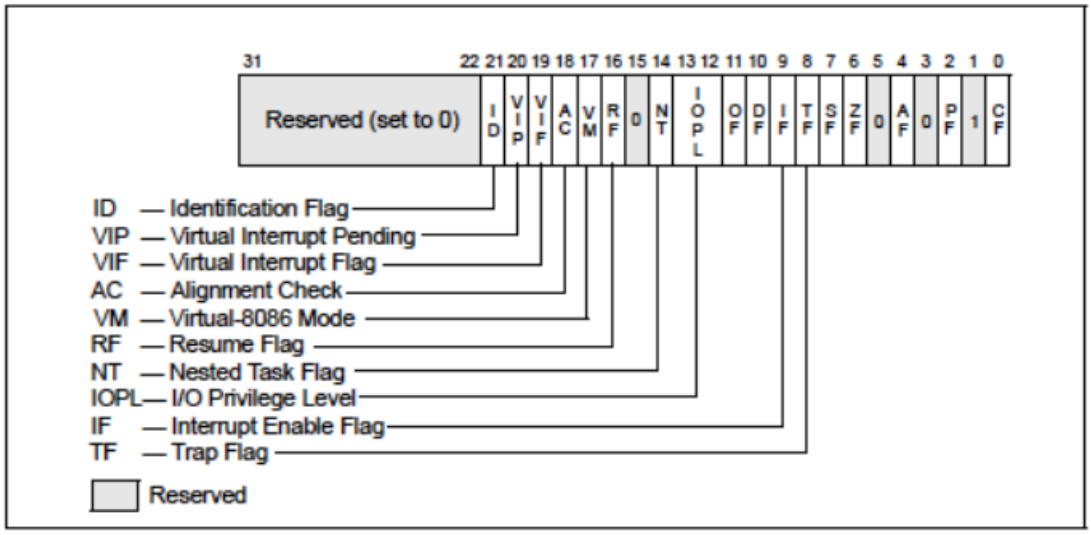

Figure 2-4. System Flags in the EFLAGS Register

Figura 1.6: Contenuto di EFLAGS

significativo e con AH al byte più significativo della word meno significativa di EAX.

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Figura 1.7: Registri General Purpose

Infine esistono particolari registri usati come indici da particolari istruzioni assembler, è il caso dei registri ESI e EDI; e registri usati invece come puntatori a locazioni di memoria si tratta degli registri ESP e EBP utilizzati per la gestione dello stack, il cui scopo tratteremo diffusamente a breve.

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Figura 1.8: Index and Pointer Register

Capitolo 2

Introduzione all'Assembly AT&T

Dopo una breve descrizione delle principali caratteristiche dell'architettura Intel IA-32, introduciamo gli elementi fondamentali del sottostante linguaggio assembler, l'unico linguaggio di programmazione che ci consente di operare direttamente sui registri del processore e dei device e che risulta quindi necessario per la realizzazione di alcune funzionalità di un kernel. Esistono due grandi famiglie di linguaggio assembler per l'architettura IA-32, che si distinguono per la sintassi adottata nella definizione del linguaggio: AT&T e Intel-Microsoft. Noi ci rifaremo alla sintassi AT&T.

2.1 Data Sizes

La dichiarazione delle variabili su cui deve operare un programma avviene utilizzando principalmente 4 pseudo istruzioni dette direttive, più precisamente

1. **.byte**: dichiara variabili la cui dimensione è un byte, quindi nel caso di variabili numeriche senza segna potrà contenere numeri non superiori a 255;
2. **.word**: dichiara variabili la cui dimensione è 2 byte, quindi nel caso di variabili numeriche senza segna potrà contenere numeri non superiori a 65535;
3. **.long**: dichiara variabili la cui dimensione è 4 byte, quindi nel caso di variabili numeriche senza segna potrà contenere numeri non superiori a 4294967295;

4. **.ascii**: dichiara variabili contenti stringhe di caratteri. Ad esempio la pseudo istruzione `msg: .ascii "Hello there "` dichiara una variabile di nome `msg` il cui contenuto è una stringa di 12 byte inizializzata al valore "Hello there´´.

Va ricordato che per quanto riguarda la rappresentazione dei dati numerici Intel adotta la notazione LITTLE ENDIAN. Little Endian significa Little end goes first cioè il byte meno significativo di un numero (cioè quello che contiene le cifre più a destra) è memorizzato nelle locazioni di memoria, destinate a contenere il numero, con gli indirizzi di memoria più bassi (tipicamente quelle che in una stringa di byte appaiono più a sinistra). Se quindi abbiamo una variabile dichiarata dalla presente pseudo istruzione `num: .long 9` la stessa sarà caricata in memoria nel seguente formato:

`00001001 00000000 00000000 00000000`

L'altro sistema usato per rappresentare dati numerici in memoria ,è il BIG ENDIAN e viene usato da altri costruttori di hardware come Motorola. Big Endian significa Big end goest first che sostanzialmente significa che il byte più significativo di un numero in memoria centrale, è memorizzato nelle locazioni di memoria, destinate a contenere il numero, con gli indirizzi di memoria più bassi (tipicamente quelle che in una stringa di byte appaiono più a sinistra). Se quindi abbiamo una variabile dichiarata dalla presente pseudo istruzione `num: .long 9` la stessa sarà caricata in memoria nel seguente formato:

`00000000 00000000 00000000 00001001`

Ancora, nel caso Little Endian il numero binario 0x003377ff sarà memorizzato come 0xff773300

Byte 0: ff → indirizzo memoria più basso, Byte 1: 77, Byte 2: 33, Byte 3: 00 → indirizzo memoria più alto

Nel caso Big Endian il numero 0x003377ff sarà invece modernizzato come 0x003377ff

Byte 0: 00 → indirizzo memoria più basso, Byte 1: 33, Byte 2: 77, Byte 3: ff → indirizzo memoria più alto.

2.2 IA-32 Formato delle istruzioni

Nella sintassi AT&T un'istruzione è generalmente costituita da quattro campi di cui solo il terzo obbligatorio:

1. una label opzionale, con suffisso il carattere :;
2. un prefisso opzionale;

3. un codice operativo obbligatorio;

4. uno o più operandi

Ovvero:
`[label:] [prefisso] opcode operands`

Gli operandi di un'istruzione possono essere zero, uno o due, nel secondo caso il secondo operando è quello su sui sarà memorizzato il risultato dell'operazione svolta dall'istruzione. Questa ,è una delle differenze dei due formati Assembler, difatti nell'assembler Intel nel caso di istruzioni a più operandi, il risultato dell'istruzione viene posto nel primo operando.

Un ulteriore differenza è data dal fatto che nell'assembler AT&T il codice operativo di un'istruzione deve avere come suffisso la dimensione degli operandi, su cui operare. In particolare si userà il suffisso **b** quando gli operando hanno dimensione un byte, **w** word, **l**long. Quindi ad esempio useremo come codice operativo le stringhe `addb, addw, addl`.

2.3 L'istruzione mov

In un sistema i dati sono memorizzati nei Registri del processore cioè speciali locazioni direttamente connesse al processore (di fatto la componente di morena più veloce all'interno di un sistema) e in variabili memorizzate all'interno di locazioni di memoria centrale. Per costruzione il processore può manipolare solo dati all'interno dei Registri, cioè operazioni aritmetiche, logiche possono essere fatte solo ed esclusivamente su dati che sono stati preventivamente memorizzati all'interno dei registri. Per questo motivo ogni architettura presenta una o più istruzioni che consentono di spostare dati dalla memoria centrale verso i registri e viceversa. Questa istruzione è la **mov** il cui formato è:

`mov src, dst`

Dove `src` e `dst` possono essere due registri o un registro e una variabile in memoria centrale. Gli operandi di un'istruzione **mov** possono assumere significati diversi in funzione delle modalità di indirizzamento che il programmatore decide di adottare e che sono esplicitate attraverso il ricorso a particolari simboli. Vediamo quelle principali.

Register Addressing : in questo caso i due operandi sono registri e semplicemente la **mov** sposta il contenuto del registro specificato come primo operando nel secondo registro, ad esempio l'istruzione `movl %edx,%ecx` si occupa di copiare i dati contenuti dal registro EDX (src)

al registro ECX (dst). Ovviamente la scelta dei registri dipende dall'istruzione utilizzata, quindi ad esempio la **movl** che si occupa di spostare 32 bit va bene con l'utilizzo dei registri EDX e ECX a 32 bit, ma non va bene con i registri DX e CX a 16 bit, in questo caso l'istruzione da utilizzare è `movw` che si occupa di spostare 16 bit.

Immediate Addressing : in questo caso è possibile caricare un valore immediato ovvero una costante all'interno di un registro del processore. Il formato generale dell'istruzione è `movl $12, %eax`. La modalità di indirizzamento è identificata dal ricorso al carattere $. Quindi la suddetta istruzione muove il valore costante 12 all'interno del registro EAX del processore. Se noi non avessimo messo il prefisso $, questa istruzione avrebbe mosso il contenuto della locazione di memoria centrale presente all'indirizzo 12, all'interno del registro EAX. Quindi un numero messo come operando src senza il $, viene interpretato come un indirizzo della memoria centrale e da origine alla modalità direct addressing qui sotto descritta.

Direct Addressing : la terza modalità di indirizzamento è chiamata Direct Addressing e si occupa di muovere il contenuto di una particolare locazione di memoria all'interno di un registro del processore. Ad esempio `movl 2000, %ecx` muove il valore contenuto all'interno della locazione di memoria centrale 2000 all'interno del registro ECX del processore e più precisamente muove 4 byte poiché il codice operativo dell'istruzione ha il suffisso l, quindi il contenuto delle locazioni di memoria dall'indirizzo 2000 all'indirizzo 2003 è copiato in ECX. Se volessimo muovere meno di 4 byte lo dobbiamo specificare nell'istruzione, ad esempio `movw 2000, %cx`. Per facilitare il lavoro del programmatore è possibile sostituire gli indirizzi di memoria con indirizzi simbolici. Quindi ad esempio invece di usare l'indirizzo 2000, possiamo utilizzare il nome di una variabile ad esempio `numero` a cui poi l'assemblatore sostituirà l'indirizzo assegnato alla variabile, in questo caso avremmo `movl number, %ecx`

Indirect Addressing : in questo caso gli operandi (il cui nome sarà racchiuso tra parentesi tonde) viene interpretato dall'assemblare come un puntatore, quindi l'istruzione `movl (%eax), %ecx` viene interpretata come accedi alla locazione di memoria il cui indirizzo è presente nel registro EAX ((%eax) indica proprio un puntatore) e carica il contenuto di questa locazione in ECX.

Base Pointer Addressing : è complementare alla modalità vista sopra e consente di usare degli spiazzamenti da sommare al registro usato come puntatore, ad esempio l'istruzione `movl 8(%eax), %ecx`, supponendo che EAX stia puntando alla locazione 2000 opera nel seguente modo:

- legge il contenuto di EAX = 2000:
- gli somma 8
- accede alla locazione di memoria 2008
- trasferisce il contenuto della locazione di indirizzo 2008 nel registro ECX per 4 byte, quindi sono trasferite le locazioni 2008-2011.

Indexed Addressing : sempre basata sull'indirect addressing consente però di operare sullo stesso in maniera più sofisticata. Vediamo un esempio: `movl a(,%eax, 4), %ecx`. Questa istruzione calcola un indirizzo di memoria applicando la seguente formula

$$\textit{indirizzo di memoria} = \%eax + a * 4 \tag{2.1}$$

quindi muove il contenuto di questa locazione di memoria a parte dall'indirizzo sopra calcolato e per la lunghezza di 4 byte all'interno del registro ECX. Più in generale la sintassi della modalità di indirizzamento Indexed Addressing è (comprese le virgole)

```
movl baseaddress(, %indexregister, multiplier)
```

tipicamente questa modalità di indirizzamento viene usata all'interno di cicli for o equivalenti.

2.4 Arithmetic Instructions

Le istruzioni che useremo sono le seguenti:

Addizione : `add(b,w,l) source, dest` il cui risultato è `dest = source + dest` ;

Sottrazione : `sub(b,w,l) source, dest` il cui risultato è `dest = dest - source` ;

Incremento : `inc(b,w,l) dest` il cui risultato è `dest = dest + 1`

Decremento : `dec(b,w,l) dest` il cui risultato è `dest = dest -1`

Confronto : `cmp(b,w,l) source1, source2` questa istruzione calcola il valore di $source2 - source1$ e modifica di conseguenza alcuni flags nel registro EFLAGS, senza modificare il valore originario degli operandi. Sul valore di questi registri si basano poi le istruzioni di jump.

mul-imul : si tratta dell'operazione di moltiplicazione che nel primo caso (`mul`) opera su dati senza segno e nel secondo caso su dati con segno. In entrambi i casi la sintassi è uguale. Il formato dell'istruzione è `mul, imul %ebx` . Questa istruzione effettua una moltiplicazione tra il valore contenuto in EAX e quello in EBX e il risultato viene salvato sempre in EAX (quindi implicitamente il secondo operando viene inteso come EAX). Per maggior precisione il risultato (che può supere i 32 bit) ,è memorizzato nei registri EDX e EAX che vengono concatenati (EDX:EAX) a formare un registro di 64 bit. Ovviamente se il risultato può essere contenuto in 32 bit rimarr,à memorizzato solo in EAX.

div-idiv : come nella caso della moltiplicazione useremo div su dati senza segno e idiv su dati con segno. Il formato dell'istruzione è `div, idiv %ecx`. In questo caso il valore del registro a 64 bit EDX:EAX viene diviso per il valore contenuto in ECX, il quoziente delle divisore viene posto in EAX ed il resto in EDX. Quindi prima di usare le operazioni di moltiplicazione e divisione, dobbiamo sempre preoccuparci di gestire correttamente il valore di EDX.

2.5 Bitwise logic Instructions

Useremo le seguenti istruzioni logiche:

And : `and (b,w,l) source, dest` → dest = source & dest ;

Or : `or (b,w,l) sourcce, dest` → `dest = source | dest` ;

Xor : `xor (b,w,l) source, dest` → `dest = source ^dest`;

Not : `not (b,w,l) dest` → `dest =` $\sim$ `dest`;

Shift a sinistra : `sal(b,w,l) source, dest` → `dest = dest << source`;

Shift a destra : `sar{b, w, l} source, dest` → `dest = dest >> source`;

dove i suffissi indicano rispettivamente:
b → byte → 8 bit
w → word → 16 bit

l → long → 32 bit

2.6 Istruzioni di Control Flow

Il Control Flow, ovvero la gestione del flusso di controllo all'interno di un programma assembler viene fatto tramite l'uso di due istruzioni:

`cmpl $0, %eax` → confronta i due operandi e setta EFLAGS
`je end_loop` → in base al contenuto di EFLAGS salta oppure no.

La prima istruzione `cmpl` sottrae il valore del secondo operando al primo ed in funzione del valore del risultato predispone alcuni bit all'interno dello Status Register EFLAGS. L'struzione `jump` effettua o meno il salto alla label che compare come suo operando, dopo avere verificato i bit dello Status Register EFLAGS predisposti dalla precedente istruzione cmp ed in funzione della condizione espressa dal programmatore. Esistono diversi tipi di `jmp` in funzione della condizione che si vuole testare.

- `je` → Salta se i valori degli operandi della `cmp` sono uguali;
- `jg` → Salta se il secondo valore è più grande del primo;
- `jge` → Salta se il secondo valore è più grande o uguale del primo;
- `jl` → Salta se il secondo valore ,è più piccolo del primo ;
- `le` → Salta se il secondo valore ,è più piccolo o uguale del primo;
- `jmp` → Salta sempre.

2.7 Esercizio

Scrivere un programma in assembler AT&T che computa in ECX la somma dei primi 1000 numeri naturali:

```
_start:
                movl    $1, %eax        # setto a 1 come primo numero naturale
                movl    $0, %ecx        # setto a 0 il sommatore in ECX

loop:
                addl    %eax, %ecx      # in ECX sommo il numero contenuto in EAX
```

```
incl    %eax          # incremento il numero naturale in EAX
cmpl    $1000, %eax   # Control Flow EAX ≤ 10000 ?
jl loop               # Se si loop, altrimenti esco

movl    $0, %ebx      # valore di ritorno 0 → ok
movl    $1, %eax      # system call → sys_exit()
int     $0x80         # chiamo la sys_exit(0)
```

2.8 Compiling & Linking

Scritto un programma in Assembler si procede con la sua traduzione in linguaggio macchina attraverso un Assemblatore. In ambiente Linux l'assemblatore è incluso nel package `binutils`, si chiama `as` e può essere eseguito a linea comando nel seguente modo:

```
$ as name.s -o name.o
```

dove `name.s` è il file contenente il codice sorgente mentre `name.o` è il file contenente il codice oggetto creato dall'assemblatore. Per rendere eseguibile un file `.o` dobbiamo usare il Linker chiamato `ld`:

```
$ ld name.o -o name
```

In ambiente Linux il file eseguibile possiede un formato ELF e per eseguirlo è sufficiente digitare il comando

```
$ ./name
```

dove `./` indica alla shell di ricercare il file `name` nella directory corrente. Possiamo così eseguire l' esercizio svolto precedentemente. Questo programmino però non ha alcun output, quindi noi lo eseguiamo ma non sappiamo cosa ha fatto. Al massimo, l'unico output che possiamo riscontrare è un segmentation fault che si verifica se abbiamo usato male i registri. Per vedere se il programma è corretto oppure per analizzare un ipotetico errore dobbiamo fare Debugging.

2.9 Debugging

In ambiente linux il debugging di un programma viene effettuato ricorrendo al programma `gdb` ovvero il GNU Debugger. Questo software permette di analizzare a runtime l'esecuzione di programmi scritti in molti linguaggi di

programmazione, soprattutto per C ed Assembler. Per eseguire sotto il controllo di `gdb` un programma di nome `name` è sufficiente digitare il comando

```
$  gdb name
```

`Gdb` consente di fare molto cose per analizzare l'esecuzione di un programma. Uno dei comandi più comodi di `gdb` è il Break Point, tramite il comando `break n` è possibile bloccare l'esecuzione di un programma prima dell'esecuzione della n-esima istruzione, e conseguentemente verificare lo stato della macchina prima dell'esecuzione della stessa, utilizzando appositi comandi come ad esempio:

```
(gdb) info register     /*visualizza il contenuto di tutti i registri del processore.
(gdb) print/   $eax   /* Visualizza il contenuto di eax in esadecimale
(gdb) print/d \%eax /* visualizza il contenuto di eax in decimale
(gdb)  x/  ADDR        /* visualizza il contenuto della locazione di indirizzo ADDR
(gdb) help /* Per consultare tutti i possibili comandi di gdb
```

RICORDARSI: per poter eseguire un programma all'interno di un debugger è necessario che lo stesso sia assemblato/compilato con un'opportuna con l'opzione `- gstabs`, quindi nel caso in questione il comando da digitare è:

```
$ as -gstabs name.s -o name.o
```

Riportiamo il listato di una esecuzione del programma `name` riportato in esempio con il setting di un break point dopo la jump e la verifica dell'esecuzione del programma.

```
$ gdb name
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /path/name...done.
(gdb) break 11
Breakpoint 1 at 0x40008b: file name.s, line 11.
(gdb) run
```

```
Starting program: /path/name

Breakpoint 1, loop () at name.s:11
11 jl loop # Se si loop, altrimenti esco
(gdb) print/d $eax
$1 = 2
(gdb) print/d $ecx
$2 = 1
(gdb) continue
Continuing.

Breakpoint 1, loop () at name.s:11
11 jl loop # Se si loop, altrimenti esco
(gdb) print/d $eax
$3 = 3
(gdb) print/d $ecx
$4 = 3
(gdb) continue
Continuing.

Breakpoint 1, loop () at name.s:11
11 jl loop # Se si loop, altrimenti esco
(gdb) print/d $eax
$5 = 4
(gdb) print/d $ecx
$6 = 6
(gdb)
```

Completato lo sviluppo del programma, l'opzione per abilitare `gdb` va rimossa in modo tale da ottimizzare il codice eseguibile.

2.10 Lo Stack

Una componente particolarmente critica da gestire e anche particolarmente importante ,è lo Stack. Lo stack è un area dati particolare che viene usata come memoria temporanea per la memorizzazione di dati non persistenti cioè che non servono per un intera computazione. Ogni processo all'interno del sistema deve avere uno stack così come anche il kernel deve avere uno stack. Lo stack è un area organizzata LIFO (Last In First Out), il dato rimosso è sempre l'ultimo che è stato caricato nello stack ed è talmente

importante che il processore fornisce delle istruzioni a livello macchina per operare su porzioni di memoria gestite a stack.

I dati sullo stack sono quindi caricati a partire dagli dagli indirizzi più alti verso gli indirizzi più bassi, quindi il contenuto più recente sullo stack è nella locazione con l'indirizzo più basso, in gergo si dice che lo stack cresce verso il basso.

Non solo, il processore fornisce anche dei registri di supporto per gestire in maniera ottimale queste aree di memoria. Il principale di questi registri è ESP (Extendend Stack Pointer), un registro che contiene l'indirizzo della locazione di memoria in cui è memorizzato il dato presente sulla cima (top) dello stack. Poiché, come abbiamo visto, in un sistema sono presenti più stack ma ovviamente in un dato istante deve essere attivo sul sistema un solo stack, si procede con l'individuazione dello stack attivo attraverso un ulteriore registro di sistema ed in particolare del segment register SS (Segment Segment Register), in cui è memorizzato l'indirizzo di inizio dello stack attivo.

Quindi il registro SS indirizza una particolare area di memoria che è l'oggetto su cui operano le seguenti istruzioni: Call, Ret, Push, Pop, Enter, Leave. Sono tutte istruzioni che operano direttamente sullo stack usando in qualche modo il contenuto del registro ESP. I dati sullo stack vengono solitamente messi in multipli di 4 byte, se dobbiamo caricare dati di dimensione minore ci dovremmo occupare del padding dello stack, cioè se dobbiamo caricare 1 byte sullo stack, dobbiamo caricarlo sullo stack come locazione di 4 byte e poi quando lo dobbiamo scaricare dallo stack, buttare via i 3 byte che abbiamo aggiunto e usare il byte che ci interessa.

Vediamo graficamente la struttura dello stack e le principali istruzioni che operano sullo stack.

2.10.1 L'istruzione Push

L'istruzione push è un istruzione con un unico operando, `pushl src` carica 4 byte a partire dall'indirizzo `src` all'interno dello stack.

Quale stack? Quello il cui indirizzo di partenza è contenuto nel registro SS il cui valore viene sommato con il contenuto di ESP per determinare l'indirizzo della locazione su cui deve operare l'istruzione `push`. Quindi l'istruzione `pushl src` è equivalente a:

Si tenga presente che il valore di SS è implicito, anche se non appare viene usato per calcolare l'indirizzo effettivo della locazione su cui operare. Esiste anche un'istruzione `pusha` che viene utilizzata tipicamente nei context switch e che da sola carica sullo stack tutti i registri general purpose del

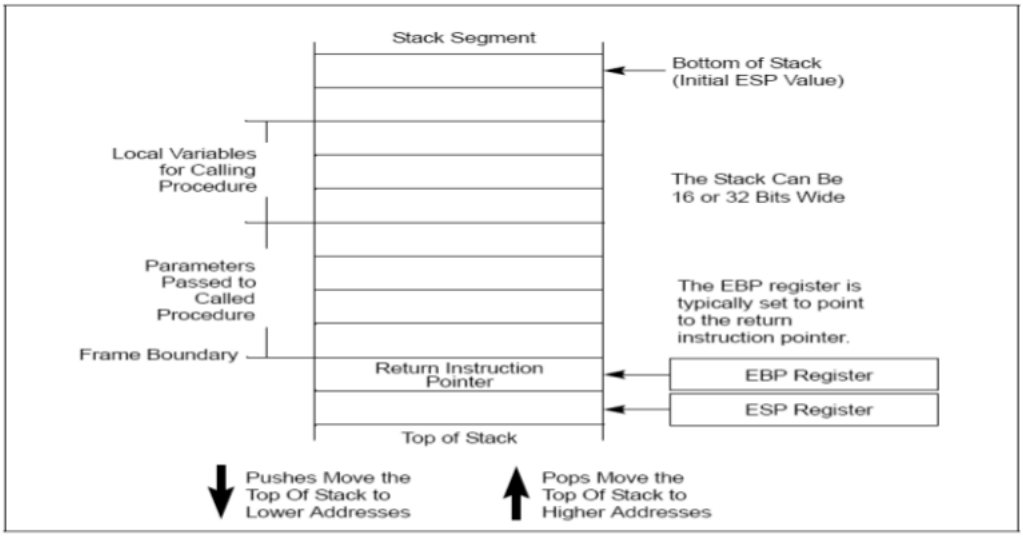

Figure 6-1. Stack Structure

Figura 2.1: Struttura dello Stack

```
pushl src          →        subl $4,%esp
                            movl src,(%esp)
```

Figura 2.2: Semantica istruzione Push

processore (EAX, EBX, ECX, EDX, ESI, EDI e EBP Registers) (non per forza in questo ordine). Quindi `pusha` vuol dire: prendi il contenuto di tutti i registri general purpose all'interno del processore e mettili sullo stack.

2.10.2 L'istruzione Pop

L'istruzione reciproca della `Push` è la `Pop`.

L'istruzione `pop` è un istruzione con un unico operando. L'istruzione `popl dest` carica nella locazione di indirizzo `dest` il contenuto della locazione di memoria il cui indirizzo è caricato in ESP (che sta puntando all'ultimo dato che abbiamo caricato sullo stack). Quindi l'istruzione popl dest è equivalente a:

```
popl dest        →   movl (%esp),dest
                     addl $4,%esp
```

Figura 2.3: Semantica istruzione Pop

2.10.3 Stack e Function call

Lo stack viene usato pesantemente nelle chiamate di funzione. Più precisamente viene usato per memorizzare i seguenti valori:

Return Address : a seguito di una chiamata di funzione è necessario memorizzare l'indirizzo dell'istruzione successiva alla chiamata, a cui ritornerà il controllo al termine della funzione stessa;

Parametri : sono i dati effettivi su cui deve operare una funzione o procedura e che sono preventivamente memorizzate stack;

Variabili Locali : durante l'esecuzione di una funzione o procedura è necessario disporre di una zona di memoria in cui memorizzare le variabili locali che la procedura deve usare, zona individuata da tutti i compilatori all'intero del stack;

Valori di ritorno : eventuali valori di ritorno, che non siano variabili globali, possono essere rilasciati dalla funzione all'interno dello stack.

Molte delle suddette funzionalità sono svolte dalle istruzioni `call, ret`. La `call` è l'istruzione assembler che viene usate per la chiamata di una procedura e svolge le seguenti operazioni:

- salva sullo stack l'indirizzo dell'istruzione successiva alla `call`, quindi l'indirizzo di ritorno dalla funzione chiamata;
- effettua un salto incondizionato alla funzione chiamata.

L'istruzione `ret` invece svolge le seguenti operazioni:

- ripristina dallo stack l'indirizzo della prossima istruzione da eseguire, caricando il suo indirizzo all'interno del program counter (il registro EIP)
- salto incondizionato all'indirizzo di cui sopra.

Va notato che le istruzioni sopra indicate sono a puro titolo esplicativo, poiché EIP è un registro manipolabile esclusivamente dall'HW e il programmatore può modificarne il valore solo attraverso le istruzioni `jump, call e ret`.

2.11 Il passaggio di parametri

Come già anticipato anche eventuali parametri utilizzati da una procedura sono caricati sullo stack, e la convenzione è che i parametri siano passati nell'ordine inverso in cui sono dichiarati all'interno della procedura stessa. Quindi l'n-esimo parametro sarà il primo ad essere posto sullo stack, mentre il primo parametro sarà l'ultimo. A seguito di una `call` il layout dello stack è quello rappresentato in figura:

L'indirizzo del primo parametro è esprimibile come 4(%esp). Più in generale la funzione chiamata può accedere ai parametri sullo stack facendo riferimento ad ESP e sapendo che i parametri vengono caricati sullo stack in un modo ben preciso.

Quindi:

1. L'indirizzo di ritorno si trova sullo stack all'indirizzo contenuto in ESP $\rightarrow$ (%esp);
2. il parametro numero 1 si trova sullo stack all'indirizzo contenuto in ESP + 4 $\rightarrow$ 4(%esp);

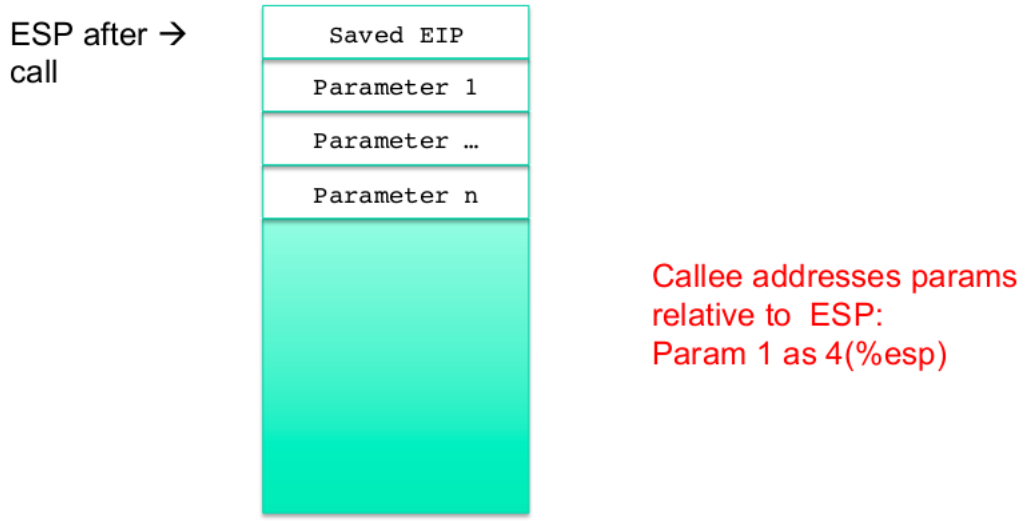

Figura 2.4: Contenuto dello stack dopo una call

3. il parametro numero 2 si trova sullo stack all'indirizzo contenuto in ESP + 8 → 8(%esp);
4. il parametro numero 3 si trova sullo stack all'indirizzo contenuto in ESP + 12 → 12(%esp);
5. E cosi via...

Resta ancora un piccolo problema da risolvere. Una volta eseguita la `ret` sullo stack restano dei parametri che non usiamo più. É buona norma al termine di ogni procedura chiamata ripulire lo stack di tutti i parametri utilizzati, in questo caso è sufficiente incrementare lo stack pointer di una quantità di byte equivalente a quella occupata dai parametri che si vogliono rimuovere, come nel seguente esempio:

Come si ripulisce lo stack? Sommando allo Stack Pointer il valore (n * 4) dove n è il numero di parametri passati, quindi nel caso in questione usiamo l'istruzione `addl $12, %esp`.

2.11.1 Il registro %ebp

Al fine di facilitare la gestione dello stack si può fare riferimento ad un ulteriore registro di supporto: il registro EBP. Il ruolo di EBP è quello di

```
…                          sub:
# Push parameters          …
pushl $5                   movl 4(%esp),var1
pushl $4                   movl 8(%esp),var2
pushl $3                   movl 12(%esp), var3
call sub                   …
# Pop parameters           ret
addl $12, %esp
```

Figura 2.5: Esempio di ripristino stack

mantenere, durante la gestione di una procedura, una posizione costante all'interno dello stack consentendo ad ESP di muoversi liberamente all'interno dello stesso. In particolare, durante una chiamata di una procedura P l' indirizzo di rientro di P si troverà sempre all'indirizzo 4(EBP) Questa caratteristica di EBP viene sfruttata dal compilatore per poter velocemente stabilire la posizione all'interno dello stack dell'indirizzo di ritorno da una procedura, per effettuare questa funzione il compilatore inserisce in ogni procedura due particolari porzioni di codice chiamate `Prologo e Epilogo` il cui contenuto è il seguente.

```
pushl %ebp
movl  %esp, %ebp
(sub  Local_bytes, %esp)
```

Figura 2.6: Prologo di una funzione

Queste sono le prime istruzioni che vengono eseguite da ogni sotto procedura, il cui scopo è:

- salvare il vecchio contenuto di EBP;
- inserire in EBP il valore corrente di ESP che attualmente punta alla locazione successiva a quella contente il return address a causa dell'istruzione di cui la punto precedente;

- predisporre spazio nello stack per ospitare le variabili locali della procedura, questa operazione viene fatta decrementando il valore di ESP di una quantità opportuna che viene calcolata dal compilatore.

A questo punto il return address della sotto procedura chiamata sarà presente nella locazione EBP + 4, ed ESP potrà muoversi sullo Stack come meglio crede, purché prima della ret provveda a ripristinare in ESP l'indirizzo dell'istruzione di ritorno, a questo pensa sempre il compilatore ricorrendo all'epilogo:

```
movl %ebp, %esp
popl %ebp
ret
```

Figura 2.7: Epilogo di una funzione

Prende il contenuto di EBP e lo carica in ESP, riportando quindi ESP all'indirizzo che contiene il vecchio valore di EBP che si trova immediatamente sotto il Return Address. Con la `pop` ripristina il vecchio valore di EBP e contemporaneamente sposta ESP sulla locazione che contiene il RA, a questo punto la rete può essere eseguita correttamente.

Lo Stack ovviamente può essere usato anche per salvare i registri. Tipicamente quando passiamo da una procedura ad una sotto procedura, e vogliamo preservare lo stato della procedura di partenza, dobbiamo anche preoccuparci di salvare sullo stack eventuali registri che saranno utilizzati dalla sotto procedura. In sostanza facciamo un mini context switch gestito da noi. Se ad esempio abbiamo una procedura che chiama una sotto procedura e la procedura principale usa EAX e EBX, o siamo sicuri che durante la sotto procedura, non usiamo EAX ed EBX oppure salviamo i suddetti registri prima di chiamare la sotto procedura e li ripristiniamo al termine della stessa.

2.11.2 Stack Frame

Tutte le informazioni che sono salvate sullo stack durante una chiamata di procedura compongono lo Stack Frame. Quindi lo Stack Frame contiene:

1. Eventuali parametri da passare alla funzione chiamata;

2. Return Address (EIP salvato);

3. Vecchio EBP;

4. Eventuali valori dei registri salvati;

5. Variabili locali.

Lo Stack Frame è delimitato da ESP che punta al top dello Stack Frame corrente (zona bassa di memoria) e da EBP che punta al bottom dello Stack Frame corrente (zona alta di memoria) a cui vanno aggiunte le locazioni necessarie per la memorizzazione dei parametri.

2.12 Assembler Inline

Prima di iniziare a vedere il codice è necessario fare una breve introduzione sul linguaggio assembler inline, cosi da renderne più semplice la comprensione. L'assembler inline è una modalità per poter eseguire codice assembler all'interno di un programma C. La sintassi è molto semplice: `tasm(\codice assembly")`. Il codice assembler viene inserito in una stringa (che eventualmente può essere suddivisa in più righe); le varie istruzioni che compongono il codice devono essere separate da un delimitatore che può essere `\n` oppure ';'.

```
asm ( ''movl %eax, %ebx;" ); // ; come delimitatore
asm ( ''movl %eax, %ebx\n" ); // \n new line come delimitatore
asm ( ''movl %eax, %ebx\n\t" ); // \n + \t new line più tabulazione come delimitatore
asm ( ''movl $5, %eax;"
         '' addl  $12, %ebx;"
         ''subl  %eax, %ebx;"
);
```

La sintassi di base vista prima è molto limitata infatti non è possibile accedere alle variabili definite nel C. Per poter accedere a queste variabili occore usare la Sintassi estesa.

```
asm(
''codice assembly"
:  // lista registri output
: // lista registri input
: // lista registri sporcati
);
```

Le regole per la scrittura del codice assembler sono le stesse viste prima, con la sola differenza che se vogliamo riferirci ad un registro, dobbiamo porre due % davanti al suo nome. Di seguito un esempio di sintassi estesa in cui si assegna il valore di varA a varB.

```
int varA=2, varB;
asm(
''movl %1, %%eax;"
  ''movl %%eax, %0;"
: ''=r" (varB) // output
: ''r" (varA) // input
: ''%eax" // registri sporcati
);
```

Questo codice definisce un parametro di output associato alla variabile `varB`. GCC assegna ai nostri parametri un numero progressivo; `varB` è il primo parametro che definiamo a cui viene associato il numero 0, che nel codice indicheremo con %0. Stesso discorso per i parametri di input: ne abbiamo uno associato a `varA`, a cui accederemo nel codice mediante %1 . Quindi la prima `movl` mette in `eax` il contenuto del secondo parametro (%1), che corrisponde a `varA`, e la seconda `mov` lo sposta nel parametro %0 (varB). Abbiamo scelto `eax` come registro temporaneo in modo del tutto arbitrario, ma come facciamo a sapere che non contenga dati che servivano al compilatore? Non possiamo saperlo, ma possiamo dire al compilatore che noi lo usiamo, che lo "sporchiamo" con i nostri dati. E' a questo che serve la terza lista.

La parola `volatile` dopo `asm` sta ad indicare al compilatore di non eseguire nessuna ottimizzazione sul codice finale.

Capitolo 3

Il Bootstrap

I dettagli definiti in questo paragrafo fanno riferimento all'architettura Intel 8088 e pur rimanendo concettualmente identici, i loro dettagli implementativi sono stati nel tempo modificati per adattarsi all'evoluzione delle diverse architetture.

Vediamo prima brevemente come funziona il bootstrap di un sistema, facendo riferimento all'architettura Intel 8088. Quando si accende un PC tutti i circuiti sono alimentati ed il microprocessore resta in attesa di un segnale di power good che gli sarà fornito quando tutti i circuiti sono correttamente in tensione. Più precisamente, il segnale di "power good" è ricevuto dal clock (timer chip) che nel frattempo continua a resettare il processore non consentendogli l'avvio. Alla ricezione del segnale di Power Good il clock asserisce il pin RESET# sul processore che avvia la fase di inizializzazione.

In questa fase il processore inizializza i registri a dei valori predeterminati, ci interessano per ora solo i valori dei registri CS : 0xf000 ed IP: 0xfff0. Questi due valori infatti determinano l'indirizzo della prima istruzione che sarà eseguita dal processore, valore che si ottiene applicando la seguente formula:

$$16 * CS + IP$$

Quindi nel nostro caso l'indirizzo ottenuto è 0xffff0, che è un indirizzo del BIOS (gli indirizzi del BIOS sono compresi tra 0xf0000 e 0xfffff) in cui per convenzione viene caricata la prima istruzione che sarà eseguita dal processore che è l'istruzione:

$$\texttt{ljmp}\ \$0xf000, \$0xe05b$$

L'istruzione punta quindi ad un indirizzo di memoria (0xfe05b) sempre relativo al BIOS che è di fatto un programma di sistema il cui compito è

verificare il corretto funzionamento dei dispositivi che compongono il sistema e caricare in memoria il boot loader. Più precisamente il BIOS inizializza tutti i dispositivi di input/output (tra cui anche il controller IDE del disco) ed esegue un test della memoria RAM detto POST, che viene eseguito ogni volta che viene fatto un riavvio hardware (cold boot) del PC. Ci sono due tipi di boot:

- Caldo (Warm boot) → il warm boot attivato via software quando ad esempio si riavvia del sistema operativo e il computer quindi non viene spento;
- Freddo (Cold boot) → quando premiamo il tasto di accensione e quindi avviene durante l'avvio del sistema operativo a computer spento.

Una volta terminata questa fase, il BIOS recupera dall'area CMOS i riferimenti del dispositivo di boot cioè del dispositivo da cui caricare il boot loader che a sua volta si occuperà del caricamento del Kernel. Il boot loader la cui dimensione è limitata è nel nostro caso caricato nel primo settore del disco (detto boot sector 512 byte) ed è caricato dal BIOS in RAM nell'area compresa tra gli indirizzi 0x7c00 e 0x7dff. Terminato il caricamento il BIOS setta setta CS:IP a 0x0000:0x7c00, (cioè l'indirizzo della prima istruzione del bootloader) a cui cede il controllo, da questo momento il BIOS non sarà più utilizzato, sino al prossimo riavvio del sistema.

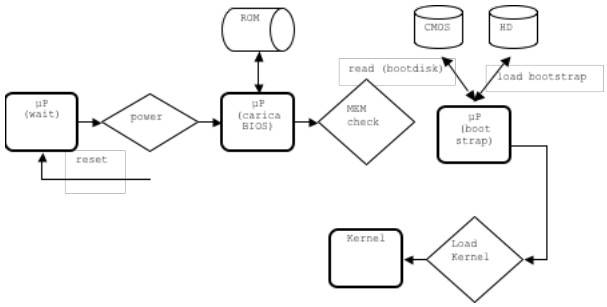

Figura 3.1: Il processo di Bootstrap

3.1 Memory Layout

Questo è il layout della memoria che il sistema si trova a gestire dopo il boot. Va ricordato che dopo il boot, il processore è in Real-address mode, quindi

di fatto indipendentemente dal processore montato sulla nostra macchina, Quad core o altro, subito dopo il boot, la macchina è sempre un 8086 e questa è la configurazione della memoria:

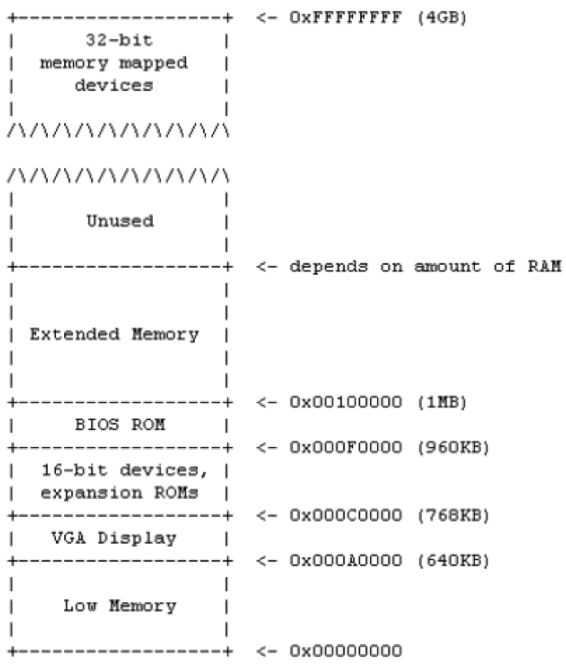

Figura 3.2: Layout di memoria durante la fase di boot

8086 è un processore a 16 bit ed è in grado di indirizzare attraverso il meccanismo di traduzione degli indirizzi spiegato prima consente di accedere solo al primo Mega di memoria centrale. Di questi però solo i primi 640 KB (basse locazioni di memoria) sono disponibili all'utente, i restanti 380 KB sono utilizzati dal sistema. Il BIOS occupa gli ultimo 40 KB di questa porzione di memoria mentre gli altri sono occupati dai dispositivi di I/O. In gergo i primi 640 KB di memoria vengono chiamati **low memory** mentre la parte di memoria che si estende dal 1MB sino a 4GB è chiamata **extended memory**.

3.2 Il Bootloader

Il BIOS per ovvie limitazioni è in grado di caricare dal primo settore del disco un programma di 512 bytes, poiché non è pensabile poter scrivere un sistema operativo usando solo 512 byte si sfruttano questi byte per scrivere un programma che a sua volta dovrà caricare il resto del sistema operativo. Questo programma è boot loader. Compito fondamentale del bootloader è caricare in memoria centrale il Kernel del sistema operativo, per fare quest'operazione deve però predisporre l'hardware a poter ospitare questo programma. Quest' operazione deve però essere preceduta da alcune attività preliminari, specifiche dell'architettura Intel, prima fra tutte l'abilitazione delle linea XXI del bus indirizzi (A 20). Questo il macro schema del boot loader:

```
{
enable A20 line;
enable 32-bit protected mode;
read kernel from disk;
jmp to first kernel instruction;
}
```

3.2.1 Abilitazione linea A20

Il 8088 aveva solo 20 linee di bus indirizzi, sufficienti per indirizzare 1 MB . La formula usata per il calcolo dell'indirizzo fisico

$$indirizzo\_fisico = (CS * 16) + offset$$

poteva però produrre indirizzi fisici di 21 bit, se infatti supponiamo che sia CS che l'offset contegano entrambi il valore lecito $0xffff$, l'applicazione della suddetta formula porta al seguente risultato:

$$indirizzo\_fisico = (CS * 16) + offset = 0xffff0 + 0xffff = 0x10ffef$$

quindi un indirizzo di 21 bit che non può trovare alcuna corrispondenza su un'architettura di 20 bit. Su 8088 il 21 bit veniva eliminato, facendo quindi in modo che il calcolo degli indirizzi venisse effettuato modulo 2^{20}. Quando venne introdotto il 286 (con 24 linee di indirizzo) aveva un real mode progettato per essere al 100 % compatibile con il 8088. Tuttavia non aveva previsto il troncamento sopra descritto, e si scoprì che esistevano dei programmi che in usavano questo troncamento. Cercando di ottenere la

compatibilità perfetta, IBM ha introdotto un meccanismo per abilitare/disabilitare il bit 0x100000 di un indirizzo . Dal momento che il controller della tastiera 8042 aveva un pin non utilizzato, si è utilizzato questo pin . Il segnale è chiamato A20 , e se è zero, il 21 bit di tutti gli indirizzi è rimosso. LA disabilitazione di questo bit non elimina ovviamente tutti gli accessi a memoria superiori ad 1 MB ma solo quelli nei range 1MB - 2MB, 3MB - 4MB, 5MB-6MB, ecc. Il software che opera in real mode però si preoccupa solo di generare eventuali indirizzi nel range 1MB-2MB controllati da A20.

Per garantire la back compatibilità con architetture precedenti la linea A20 del bus è disabilitata in fase di boot. Questo consente in real mode di avere solo indirizzi inferiori a 0xFFFFF e quindi esprimibili con 20 bit. Nel momento in cui ci si pone il problema di estendere lo spazio degli indirizzi da 20 a 32 bit è necessario come prima cosa abilitare la suddetta linea. Il metodo tradizionale per l'abilitazione delle linea A20 prevede di operare direttamente sul controller della tastiera, questo perché in fase progettuale il controller della tastiera 8042 di Intel. Il valore di questo pin viene settato predisponendo il secondo bit della porta di uscita del controller della tastiera, in particolare se questo bit vale zero il 21 bit del bus indirizzi è sempre zero, se vale 1 il 21 bit manterrà il valore che gli è stato originariamente assegnato.

Per scrivere sulla porta di uscita (indirizzo 0x60) del controller 8042 devono essere svolte queste operazioni:

1. Deve essere inviato un comando Write output port cioè 0xD1 al registro comandi del controller cioè alla porta 0x64

2. I dati da inserire nella porta di uscita devono essere scritti sulla porta 0x60.

Prima di eseguire i comandi di cui sopra, è necessario introdurre un ciclo bi busy waiting per verificare se il controller della kyboard è pronto a ricevere i comandi. Qui di seguito il codice assembler per eseguire la suddetta funzione:

```
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default.  This code undoes this.

seta20.1:
inb    $0x64, %al  # Wait for not busy
testb  $0x2,%al
```

```
jnz    seta20.1

movb   $0xd1, %al  # 0xd1 → port 0x64
outb   %al, $0x64

seta20.2:

inb    $0x64, %al  # Wait for not busy
testb  $0x2, %al
jnz    seta20.2

movb   $0xdf, %al  # 0xdf → port 0x60
outb   %al, $0x60
```

3.2.2 Abilitazione a 32-bit protected mode

Inizialmente il boot loader abilita il protected mode con la sola segmentazione, la paginazione verrà abilitata più avanti dal kernel con la memoria virtuale. Per attivare questa modalità è necessario:

- predisporre una tabella dei segmenti (GDT) che contenga almeno i descrittori dei segmenti codice e dati relativi al kernel, specificando nella struttura segdesc il campo sd_db a 1 che indica che i segmenti in questione sono relativi ad una modalità a 32-bit,
- caricare l'indirizzo di questa tabella in GDTR,
- settare a uno il bit 0 del registro CR0
- sovrascrivere il registro CS con il descrittore del segmento codice caricato in GDT.

Figura 3.3: Layout del registro CR0

3.2.3 Caricamento GDT

Queste le itruzioni presenti nel boot loader che provvedono a dichiarare e inizalizzare la GDT:

```
# Bootstrap GDT
.p2align 2              # force 4 byte alignment
# indirizzo gdt contiene 3 descrittori di segmento, ogni
# descrittore è lungo 8 byte
gdt:
  SEG_NULL # segmento nullo
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # segmento codice
  SEG(STA_W, 0x0, 0xffffffff)       # segmento dati

# Il campo da caricare in GDTR è formato da 6 byte. I primi 2
# indicano la dimensione della GDT. Il secondo l'indirizzo

gdtdesc:
  .word   0x17                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

Dove per le macro e le costanti valgono le seguenti definizioni:

```
/* Macros to build GDT entries in assembly */
#define SEG_NULL \
.word 0, 0; \
.byte 0, 0, 0, 0

#define SEG(type,base,lim) \
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
.byte (((base) >> 16) & 0xff), (0x90 | (type)),\
(0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

// Application segment type bits

#define STA_X 0x8     // Executable segment
#define STA_W 0x2     // Writeable (non-executable segments)
#define STA_R 0x2     // Readable (executable segments)
#define STA_A 0x1     // Accessed
```

Come possiamo constatare ogni Descrittore di Segmento occupa 8 byte:

```
// Segment Descriptors
struct Segdesc {
    unsigned sd_lim_15_0 : 16;  // Low bits of segment limit
    unsigned sd_base_15_0 : 16; // Low bits of segment base address
    unsigned sd_base_23_16 : 8; // Middle bits of segment base address
    unsigned sd_type : 4;       // Segment type (see STS_ constants)
    unsigned sd_s : 1;          // 0 = system, 1 = application
    unsigned sd_dpl : 2;        // Descriptor Privilege Level
    unsigned sd_p : 1;          // Present
    unsigned sd_lim_19_16 : 4;  // High bits of segment limit
    unsigned sd_avl : 1;        // Unused (available for software use)
    unsigned sd_rsv1 : 1;       // Reserved
    unsigned sd_db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
    unsigned sd_g : 1;          // Granularity: limit scaled by 4K when set
    unsigned sd_base_31_24 : 8; // High bits of segment base address
};
```

Figura 3.4: Campi di un descrittore di segmento

- Le prime due locazioni sono due word quindi 16 bit o 2 byte ciascuna;
- Le altre quattro locazioni sono quattro byte quindi 8 bit ciascuna.

La macro SEG_NULL definisce il 1° segmento nullo: le prime due locazioni sono word inizializzate a 0, le altre quattro locazioni sono byte inizializzati a 0.

La macro SEG() carica invece i vari bit all'interno del Descriptor Segment, in particolare a partire da tre parametri `type, base, lim` costruisce un descrittore di segmento nel seguente modo. L'istruzione

```
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff);
```

si occupa di costruire le prime due word, che sono rispettivamente inizializzate con i 16 bit meno significativi di `lim` espresso però non come indirizzo assoluto ma come numero di frame (quindi diviso per 4096), ed i 16 bit meno significativi dell'indirizzo di partenza del segmento (base address) espresso invece come indirizzo assoluto di 32 bit.

l'istruzione successiva:

```
.byte (((base) >> 16) & 0xff), (0x90 | (type))
```

provvede invece a predisporre i due byte successivi. In particolare il primo di questi byte contiene i bit da 16 a 23 del base address, mentre il byte successivo conterrà il risultato dell'operazione $10010000 \vee type$, ad esempio nel caso di

$$type = STA\_X | STA\_R = 00001000 \vee 00000010 = 00001010$$

avremo:

$$10010000 \vee 00001010 = 10011010$$

il che corrisponde (ricordandoci che stiamo operando su un'architettura little endian) a predisporre:

$$sd\_type \rightarrow 0101; sd_s \rightarrow 1; sd\_dpl \rightarrow 00; sd\_p \rightarrow 1$$

Analogamente l'ultima parte della macro SEG provvede ad inizializzare gli ultimi 2 byte del descrittore di segmento

```
(0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

si noti che viene setta il campo `sd_db` a 1 ad indicare che il segmento fa riferimento ad un sistema di indirizzamento a 32-bit.

Le istruzioni che provvedono invece a caricare il registro GDTR e il registro CR0 con i dati necessari ad eseguire la modifica di stato sono

```
lgdt    gdtdesc # carica GDTR
movl    \%cr0, \%eax # carico il contenuto di cr0 in eax
orl     $CR0_PE_ON, \%eax # setta a 1 bit 0 di \%eax
movl   \%eax, \%cr0 # muovo il valore di \%eax in cr0

  # A questo punto non siamo ancora in protected mode, bisogna
  # caricare nel registro CS il descrittore di un segmento
  # a 32 bit, per modificare CS usiamo l'istruzione ljmp

ljmp    $PROT_MODE_CSEG, $protcseg
```

Abbiamo detto che il bootloader è in esecuzione con la CPU che opera in real mode. La predisposizione del bit PE di CR0 a 1 non abilita automaticamente il Protected mode. Affinché avvenga questa transizione è necessario che sia carico nel registro CS il descrittore che abbiamo predisposto nella GDT.

Siccome non è possibile operare direttamente sui selettori di segmento come CS attraverso l'istruzione `mov` l'unico modo per modificare il contenuto di CS è quello di utilizzare long jump (`ljmp`) che si comporta come una jump, ma che consente di esprimere come suo primo operando l'indirizzo in GDT di un descrittore di segmento, il cui contenuto viene caricato in CS.

3.2.4 BOOT.s

Analizziamo ora il nostro programma di boot che è un primo esempio di boot loader ed è quello che hanno pensato gli sviluppatori di JOS. Si tratta di un programma Assembler che va assemblato e poi caricato nel primo settore o boot sector dell'hard disk. dove verrà prelevato dal BIOS per essere caricato in memoria a partire dall'indirizzo 0x007c

```
#include <inc/mmu.h>

.set PROT_MODE_CSEG, 0x8          # kernel code segment selector
.set PROT_MODE_DSEG, 0x10         # kernel data segment selector
.set CR0_PE_ON,      0x1           # protected mode enable flag

.globl start
start:  # start è entry pointer del codice
  .code16           # segmento codice a 16 bit

  cli                  # Disabilita interrupt
  cld                  /* istruzione che serve per indicare l'incremento,
      /* il decremento sulle operazioni di stringhe (es:
      /*  movs )

/* Azzera i segment register.*/

  xorw    %ax,%ax             # Azzera segment register
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment


  # ABILITA A20

seta20.1:
  inb     $0x64,%al   # porta 64 è pronta a ricevere ?
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al   # 0xd1 (comando di scrittura) -> %al
  outb    %al,$0x64   # commando di scrittura porta 0x64
```

```
seta20.2:
  inb     $0x64,%al   # Verifico con lo stesso ciclo se il
  testb   $0x2,%al    # dispositivo è pronto
  jnz     seta20.2    # Salta se non è zero

  movb    $0xdf,%al   # 0xdf è il valore da mandare
  outb    %al,$0x60   # sulla porta 0x60

# Fase per passare da real mode a protected mode.
# gdt e gdtdesc sono aree dati del bootloader

  lgdt    gdtdesc # carica la GDT
  movl    %cr0, %eax # carico il contenuto di cr0 in eax
  orl     $CR0_PE_ON, %ex # eseguo OR con la costante
  movl    %eax, %cr0 # muovo il valore di eax in cr0
# il motivo per cui vengono fatti questi 3
# comandi invece di fare una
# movl $1, %eax
# movl %eax, %cr0
# è per preservare il contenuto di cr0

# A questo punto non siamo ancora in protected mode, per farlo occorre
# eseguire una ljmp a un segment descriptor che sia relativo a un
# segmento a 32 bit.
# Il comando ljmp carica un dato in CS e poi ci aggiunge lo spiazzamento.
# Questa è l'ultima istruzione eseguita in real mode.

ljmp    \$PROT_MODE_CSEG, \$protcseg
```

Al termine dell'esecuzione della suddetta porzione di codice nella memoria del sistema è stata costruita una GDT con il seguente contenuto, ed il sistema inizia ad operare in protected mode a 32 bit, eseguendo il codice sotto riportato.

indirizzo 0	SEG_NULL	segmento nullo
indirizzo 8	`SEG(STA_X | STA_R, 0x0, 0xffffffff)`	segmento codice
inidirizzo 16	SEG (STA_W, 0x0, 0xffffffff)	segmento dati

```
  .code32                     # Assemble for 32-bit mode

# inizializza segment register con l'inidirizzo del Data segment
```

```
protcseg:
  movw    $PROT_MODE_DSEG, %ax    # segmento dati
  movw    %ax, %ds
  movw    %ax, %es
  movw    %ax, %fs
  movw    %ax, %gs
  movw    %ax, %ss

  # Adesso quello che ci resta da fare è caricare il kernel.
  # I sistemisti di JOS hanno pensato di farlo con una routine in C.
  # Si crea uno stack che verrà usato successivamente dalla routine C
  # 'bootmain'.

  movl    $start, %esp   # start = 0x00007c00 è l'indirizzo
      #  della prima istruzione eseguibile del
    #  bootloader

  call bootmain
# bootmain è una procedura che legge il kernel
# da disco e lo copia in memoria e poi fa partire il kernel,
# quindi il bootloader non rientra mai

# Il bootmain non dovrebbe mai rientrare

spin:
  jmp spin


# Bootstrap GDT
.p2align 2            # force 4 byte alignment
# gdt contiene 3 descrittori di segmento,
# ogni descrittore è lungo 8 byte
gdt:
  SEG_NULL # segmento nullo
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # segmento codice
  SEG(STA_W, 0x0, 0xffffffff)          # segmento dati

gdtdesc: # Campo formato da 6 byte da caricare in GDTR
 .word   0x17                            # sizeof(gdt) - 1
```

```
 .long   gdt                                    # address gdt
```

L'istruzione `.globl start` è una direttiva al linker e serve a dichiarare l'indirizzo simbolico `start` come globale cioè visibile a tutti i moduli che comporranno il codice finale. ed è l'entry point del codice. Tutti i programmi assembler devono avere un'istruzione con label `start` riferita alla prima istruzione eseguibile del codice (entry point).

L'istruzione `.code 16` è una direttiva all'assemblatore con cui si specifica allo stesso di generare codice per ambienti a 16 bit. Segue poi l'istruzione `cli` che disabilita gli Interrupt. Si è scelto di disabilitare gli interrupt per evitare che la fase di caricamento del kernel sia interrotta da eventi esterni che il kernel non è ancora pronto a gestire. Sarebbe possibile in questa fase provvedere alla gestione degli interrupt sfruttando il BIOS, e quindi facendo convivere il boot loader a parte del BIOS, i progettisti di JOS hanno deciso di operare come se il sistema di riferimento fosse completamente vuoto, questo da una parte rende didatticamente più completo il boot loader dall'altra lo rende un po' meno efficiente. Ovviamente nel momento in cui il sistema è sordo agli Interrupt il dialogo con tutte le periferiche può venire solo in modalità a controllo programma .

L'istruzione `cld` è un istruzione che azzera il flag di direzione. Il flag di direzione viene utilizzato per determinare la direzione in cui operano alcune delle istruzioni che utilizzano il prefisso REP. Le istruzioni

```
xorw \%ax, \%ax
movw \%ax, \%ds
 movw \%ax, \%es
  movw \%ax, \%ss
```

servono per azzerare i segment register, operazione che non può essere fatta con un'istruzione del tipo `movw $0, $ds` perché i segment register possono essere caricati solo con valori di altri registri e non è possibile modificare i valore dei segment register direttamente.

3.3 BOOTMAIN

Bootmain è l'entry point della procedura C che ha come scopo quello di leggere da disco il kernel che è stato memorizzato in formato ELF. Il formato ELF è stato adottato da diversi sistemi per la memorizzazione di programmi in formato oggetto o eseguibile, e sarà illustrato successivamente. Ci preoccupiamo invece ora di analizzare le modalità con cui la procedura bootmain

legge e trasferisce il kernel dal disco alla memoria. La procedura bootmain assume che il disco di riferimento abbia un interfaccia IDE e il controller adotti come modalità di indirizzamento dei blocchi del disco la modalità LBA mode che prevede che il disco sia visto come un array consecutivo di blocchi da 512 byte, ciascuno indirizzabile da un numero intero di 28bit. Questa scelta viene fatta dal BIOS durante l'inizializzazione. Più precisamente, con il metodo LBA, introdotto con lo standard ATA-2, i settori del disco sono numerati da 0 a $2^{28} - 1$ assegnando il valore 0 al primo settore della prima traccia del primo cilindro, procedendo poi lungo tutti i settori della stessa traccia, poi lungo tutte le tracce (corrispondenti a tutte le superfici) dello stesso cilindro per poi spostarsi al cilindro adiacente, continuando così fino all'ultimo settore dell'ultima traccia dell'ultimo cilindro. La comunicazione con il controller IDE avviene utilizzando le seguenti porte ed i relativi indirizzi:

indirizzo	*significato*
0x1F0	Data Port (Dati letti)
0x1F1	ERROR —
0x1F2	Sector Count (numero dei settori da leggere)
0x1F3	LBA low byte
0x1F4	LBA mid byte
0x1F5	LBA hi byte
0x1F6	1B1D Top4LBA
0x1F7	Command/Stauts register

Status Register (0x1F7)

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRO	CORR	IDEX	ERROR

Error Register (0x1F1)

7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	TONF	AMNF

Dove:

- BBK = Bad Block
- UNC = Uncorrectable data error
- MC = Media changed
- IDNF = ID mark not found
- MCR = Media Change Requested

- ABRT = Command Aborted
- TONF = Track 0 Not Found
- AMNF = Address Mark Not Found

3.4 Come si usa un dispositivo IDE

Descriviamo brevemente quali sono i passi da intraprendere per poter utilizzare un dispositivo IDE.

1. Prima di eseguire un qualunque operazione bisogna aspettare che il controller sia in uno stato di ready (bit RDY nel registro di stato)
2. Si caricano i parametri del comando che si vuole eseguire nei registri appositi. Per i comandi di lettura/scrittura significa scrivere nei registri l'indirizzo del settore interessato
3. Si inoltra un comando di lettura o scrittura.
4. Si attende fino a quando il dispositivo è pronto per il trasferimento dati (DRQ nel registro di stato).
5. Alimentare i dati del dispositivo (per la scrittura) o ottenere i dati dal dispositivo (per la lettura).
6. In caso di una scrittura si può attendere il completamento dell'operazione e leggere dal registro di stato l'esito dell'operazione.

I principali comandi che si possono impartire al controller sono:

20H : Leggi il settore con retry. NB: 21H = lettura senza retry. Per questo comando è necessario caricare preventivamente l'indirizzo del settore che si vuole leggere. Una volta completato il comando (DRQ viene attivato) si possono leggere 256 word dal registro dati del disco.

30H : Scrivere un settore con retry, 31H = senza retry. Anche in questo caso è necessario caricare preventivamente l'indirizzo del settore su cui si vuole scrivere. Quindi attendere che DRQ diventi attivo e alimentare il disco attraverso il dta register con 256 word di dati. Successivamente il disco inizia a scrivere. Quando BSY va a zero si può leggere lo stato dal registro di stato.

Più in dettaglio questi sono i dati da inviare al controller IDE per leggere un settore da disco:

- Send a NULL byte to port 0x1F1: outb(0x1F1, 0x00);
- Send a sector count to port 0x1F2: outb(0x1F2, 0x01);
- Send the low 8 bits of the block address to port 0x1F3: outb(0x1F3, (unsigned char)addr);
- Send the next 8 bits of the block address to port 0x1F4: outb(0x1F4, (unsigned char)(addr ¿¿ 8);
- Send the next 8 bits of the block address to port 0x1F5: outb(0x1F5, (unsigned char)(addr ¿¿ 16);
- Send the drive indicator, some magic bits, and highest 4 bits of the block address to port 0x1F6: outb(0x1F6, (addr ¿¿ 24) — 0xE0));
- Send the command (0x20) to port 0x1F7: outb(0x1F7,0x20).

3.4.1 Analisi del codice

La procedura bootmain è inclusa nel file /boot/main.c, e referenzia codice contenuto nei file inc/x86.h (per le procedure outb e insl) e nel file inc/elf.h per la definizione delle strutture Elf e Proghdr.

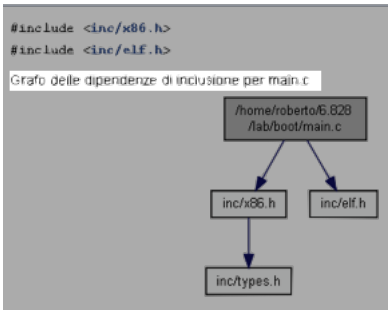

Figura 3.5: Grafo delle dipendenze di Bootmain

Le funzioni definite in /boot/main.c sono: `readsect, readseg, bootmain e waitdisk`, che sono chiamate esattamente in questa sequenza. La prima funzione che analizziamo è la waitdisk la cui funzionalità è quella di verificare lo stato del controller, in particolare `waitdisk` verifica attraverso la porta

0x1F7 e utilizzando un ciclo di busy waiting, se il controller è READY. La comunicazione con il controller avviene utilizzando il busy waiting perché in questa fase gli interrupt sono disabilitati ed è quindi l'unico modo per poter interrogare le periferiche.

```
void
waitdisk(void)
{
        while ((inb(0x1F7) & 0xC0) != 0x40)
                /* do nothing */;
}
```

Il valore esadecimale 0xC0 corrisponde in binario a 11000000 e affinché il ciclo venga interrotto il valore della porta 0x1F7 messo in and logico con la maschera 0xC0 deve valere 01000000, che equivale ad affermare il il READY bit dello status bit del controller è on.

3.4.2 La funzione readsect

La funzione `readsect` legge un settore (512 byte) dal disco. La comunicazione con il controller avviene attraverso le porte da 0x1F0 a 0x1F7 usando le istruzioni `outb e insl` definite nel file (inc/x86.h) .

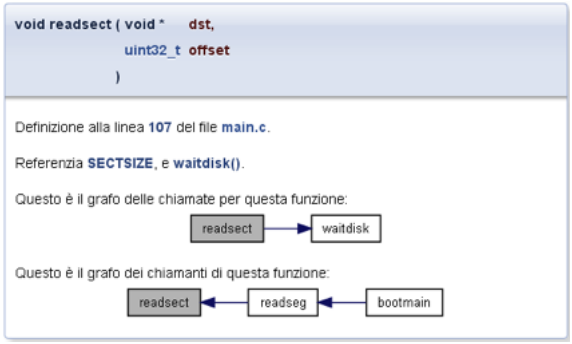

Figura 3.6: Grafo delle dipendenze di Bootmain

```
void
readsect(void *dst, uint32_t offset)
{
waitdisk();    //busy waiting sul controller
```

```
outb(0x1F2, 1);   // scrive sulla porta 0x1F2 il valore 1
outb(0x1F3, offset);
outb(0x1F4, offset >> 8);
outb(0x1F5, offset >> 16);
outb(0x1F6, (offset >> 24) | 0xE0); // Invia 0xE0 per HardDisk

outb(0x1F7, 0x20); // comando 0x20 { lettura di un settore.
// Dopo questo comando il controller parte e si
// mette in busy. Quando ha letto tutto il
// settore si mette in ready.
waitdisk();
// read a sector
insl(0x1F0, dst, SECTSIZE/4); // Legge SECTSIZE/4 volte un long dalla
       // porta 0x1F0 e lo scrive in dst
      // (memoria centrale)
}
```

3.4.3 La funzione readseg

La funzione `readsect` vista precedentemente è usate per realizzare la funzione di più alto livello `readseg`, il cui scopo è quello di trasferire in memoria centrale ad un indirizzo dato (I parametro), un certo numero di settori del disco (II parametro fornito alla procedura), a partire da un certo settore del disco (III parametro). Qui di seguito riportiamo la definizione della procedure `readseg` ed il relativo codice.

```
#define SECTSIZE 512
void
readseg(uint32_t pa, uint32_t count, uint32_t offset)
{
uint32_t end_pa;
end_pa = pa + count;
// Allinea pa alla dimensione del blocco
pa &= ~(SECTSIZE - 1); // ~(512 { 1) = ~(511) = ~(111111111)
// pa &= 000000000

// Identifica il blocco dal quale iniziare a leggere considerando che il byte
// 0 si trova nel settore 1, poichè il settore 0 contiene il bootloader
offset = (offset / SECTSIZE) + 1;
// ciclo per leggere tutti i settori richiesti
```

```
while (pa < end_pa)
  {
    readsect((uint8_t*) pa, offset);
    pa += SECTSIZE;
    offset++;
  }
}
```

3.5 Il caricamento del Kernel

Il kernel è caricato su disco nel formato `.elf`. Si tratta di un formato standard per gli eseguibili, codice oggetto, librerie condivise e core dump pubblicato nella specifica System V Application Binary Interface, e poi nel Tool Interface standard. Nel 1999 è stato scelto come formato di file binario standard per i sistemi Unix e Unix-like su x86. Un file in formato elf contiene tutte le informazioni necessarie al caricamento di un eseguibile in memoria centrale, ed è l'output di un processo di compilazione o linking. Un file di tipo ELF è composto da un header iniziale a lunghezza fissa che contiene un insieme di informazioni generali sul file, seguito da un program header a dimensione variabile che contiene informazioni sulle diverse sezioni/segmenti che compongono il programma. Il formato elf viene usato per rappresentare codice oggetto e codice eseguibile, nel primo caso diciamo che il programma è composta da sezioni, che saranno poi opportunamente accorpate per formare i segmenti nell'ambito di un eseguibile.

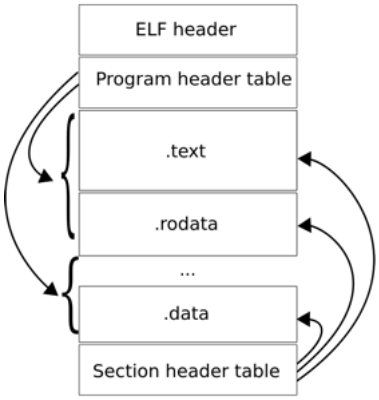

Figura 3.7: Contenuti di un file .elf

L'elf Header è così definito:

```
#define ELF_MAGIC 0x464C457FU   /* "\x.ELF" in little */
struct Elf {
        uint32_t e_magic;       // must equal ELF_MAGIC
        uint8_t e_elf[12];
        uint16_t e_type;
        uint16_t e_machine;
        uint32_t e_version;
        uint32_t e_entry;
        uint32_t e_phoff;
        uint32_t e_shoff;
        uint32_t e_flags;
        uint16_t e_ehsize;
        uint16_t e_phentsize;
        uint16_t e_phnum;
        uint16_t e_shentsize;
        uint16_t e_shnum;
        uint16_t e_shstrndx;
};
```

I campi principali hanno il seguente significato:

- e_entry: fornisce l'indirizzo virtuale delle prima istruzione eseguibile del programma a cui viene trasferito il controllo in fase di esecuzione. Se il file non ha un entry point associato a questo campo vale zero.
- e_phoff: contiene la posizione in termini di offset in byte rispetto all'inizio del file, del program header . Se il file non ha un program header, questo campo vale zero.
- e_ehsize: contiene le dimensioni in byte dell' ELF header.
- e_phentsize: contiene la dimensione in byte di un elemento del program header, tutti gli elementi hanno la stessa dimensione in byte,
- e_phnum: contiene il numero di elementi contenuti nel program header.

Questo invece il contenuto di un elemento del program header:

```
struct elf_ph_entry {
        u32 p_type;     /* Type of segment */
        u32 p_offset;   /* Offset in file */
        u32 p_va;               /* Virtual address in memory */
```

```
        u32 p_pa;       /* Reserved */
        u32 p_filesz;   /* Size of segment in file */
        u32 p_memsz;    /* Size of segment in memory */
        u32 p_flags     /* Segment Attributes */
        u32 p_align;    /* Alignment of segment */
} __attribute__ ((packed));
```

Riportiamo di seguito un esempio di section e program headers realtivi al kernel di JOS

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        f0100000 001000 004f35 00  AX  0   0 16
  [ 2] .rodata           PROGBITS        f0104f40 005f40 0016d0 00   A  0   0 32
  [ 3] .stab             PROGBITS        f0106610 007610 00882d 0c   A  4   0  4
  [ 4] .stabstr          STRTAB          f010ee3d 00fe3d 002c01 00   A  0   0  1
  [ 5] .data             PROGBITS        f0112000 013000 06be1b 00  WA  0   0 4096
  [ 6] .bss              NOBITS          f017de20 07ee1b 000ef0 00  WA  0   0 32
  [ 7] .comment          PROGBITS        00000000 07ee1b 000059 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 07ee74 00004c 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 07f078 000c60 10     10  65  4
  [10] .strtab           STRTAB          00000000 07fcd8 000af6 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xf0100000 0x00100000 0x11a3e 0x11a3e R E 0x1000
  LOAD           0x013000 0xf0112000 0x00112000 0x6be1b 0x6cd10 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .stab .stabstr
   01     .data .bss
   02
```

Figura 3.8: Header di un seguibile contente il kernel di Jos

Assumono particolare rilievo i campi `VirtAddr` (VMA) e `PhysAddr` (LMA) o indirizzo di caricamento della sezione di testo. L'indirizzo di caricamento di una sezione definisce l'indirizzo fisico di memoria in cui tale sezione deve essere caricato in memoria. In un file di tipo ELF, questo dato è memorizzato nel campo `p_pa` del program header relativo alla sezione text.

Il campo VMA di una sezione `(p_va)` è invece l'indirizzo di memoria virtuale che viene assegnato al codice contenuto nella sezione. Quindi il linker genererà un codice eseguibile a partire dall'indirizzo `(p_va)`. Il mapping tra gli indirizzi virtuali e fisici sarà poi realizzato avvalendosi dei meccanismi di memoria virtuale. Ad esempio, nel caso del sistema operativo Jos il Kernel del sistema operativo viene collocato ad un indirizzo virtuale molto

alto, come 0xf0100000 (gli ultimi 256 MB si spazio in una memoria di 32 GB), in modo da lasciare la parte inferiore dello spazio di indirizzamento virtuale della memoria ai programmi utente. Ovviamente, molti sistemi non hanno alcuna memoria fisica all'indirizzo 0xf0100000, quindi non possiamo contare sul fatto di poter memorizzare il kernel lì. Useremo quindi la gestione della memoria hardware del processore per mappare indirizzo 0xf0100000 virtuale all'indirizzo fisico 0x00100000 (dove il boot loader caricato il kernel in memoria fisica).

Analizziamo ora la funzione bootmain il cui scopo è leggere il file .elf che contiene il kernel (che ricordiamo in JOS risiede su HD a partire dal settore 1), estrarre dall'header le informazioni necessarie e con queste caricare il kernel in memoria centrale a partire dall'indirizzo 0x00100000.

```
void bootmain(void)
{
// inizializza due puntatori alle  strutture di elf e program header
// leggi in memoria a partire dall'indirizzo 0x00100000 i primi 4KB
// del kernel presente su disco a partire dal settore 1 (offset 0)

struct Proghdr *ph, *eph;

readseg((uint32_t) ELFHDR, SECTSIZE*8, 0);

if (ELFHDR->e_magic != ELF_MAGIC) // verifica che il formato del
goto bad;  // header sia corretto


// recupera tra i dati appena letti il program header e il numero di
// elementi che contiene

ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
eph = ph + ELFHDR->e_phnum;

// Carica in memoria tutti i segmenti  presenti nel kernel
for (; ph < eph; ph++)
readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

// Cedi il controllo all'entry point
((void (*)(void)) (ELFHDR->e_entry))();
```

```
bad:
outw(0x8A00, 0x8A00);
outw(0x8A00, 0x8E00);
while (1)
/* do nothing */;
}
```

Commentiamo brevemente il suddetto codice:

1. l'istruzione `ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFDR --> e_phoff);` carica nella variabile `ph` l'indirizzo fisico del primo program header da leggere, che si trova all'indirizzo ELFHDR + ELFHDR → e_phoff.

2. l'istruzione `eph = ph + ELFHDR -> e\_phnum`; carica nella variabile `eph` l'indirizzo fisico dell'ultimo program header da leggere, si noti che stiamo operando con l'aritmetica dei puntatori e `eph` è un puntatore ad una struttura di tipo `Proghdr`;

3. il ciclo `while (; ph < eph; ph++) { readseg(ph->p_pa, ph->p_memsz, ph->p_offset) }` provvede a portare in memoria tutte le sezioni del kernel, negli indirizzi specificati all'interno dei rispettivi program header;

4. Ultimata la lettura del kernel di JOS dal disco, eseguiamo una chiamata di funzione usando come puntatore a funzione il valore dell'entry point specificato nell'header del file elf che contiene il kernel di JOS, tramite l'istruzione `((void (*) (void)) (ELFHDR -->e_entry)) ();` ricordiamo che in questo caso la variabile `e_entry` punta all'indirizzo 0x0010000c, che è l'indirizzo della prima istruzione del kernel che eseguiremo. L'indirizzo 0x0010000c è un indirizzo logico che però in questa fase del sistema coincide con l'indirizzo fisico, in quanto il sistema sta operando in Protected mode con solo la segmentazione attivata (senza paginazione), e in base ai valori caricati nell GDT dal bootloader, l'indirizzo fisico viene ottenuto sommando il base address del segmento codice (0x00000000) all'indirizzo logico creando di fatto il mapping identità.

Quindi riepilogando:

1. Il BIOS una volta terminati tutti i controlli, carica e avvia il boot loader presente nei 512 bytes del 1° settore o boot sector del primo disco IDE;

2. Il boot loader a questo punto, attiva il Protected mode, carica e legge il kernel a partire dal II settore del disco IDE, si tratta in questo caso di una specifica esemplificativa, in caso contrario sarebbe stato necessario indicare al boot loader dove andare a localizzare il kernel (informazione solitamente fornita con la Partition Table)

3. Il boot loader carica il kernel in memoria a partire dall'indirizzo fisico 0x00100000

4. a caricamento terminato manda in esecuzione l'entry point di JOS all'indirizzo fisico 0x0010000c

A questo punto va in esecuzione il kernel di JOS.

Capitolo 4

Il Kernel

Quando la prima istruzione del kernel va in esecuzione stiamo ancora operando su un sistema in Protected Mode con la sola segmentazione attivata, e ci troviamo in memoria un eseguibile che è stato creato per operare nello spazio di memoria virtuale che parte dall'indirizzo 0xf0100000, e che fisicamente si trova in memoria a partire dall'indirizzo 0x00100000. Se fosse attiva la paginazione non ci sarebbero problemi, basterebbe mappare gli indirizzi logici a partire da 0xf0100000 sugli indirizzi fisici a partire da 0x00100000 e sarebbe la MMU a risolvere tutti i nostri problemi di indirizzamento. Il problema è che in questa fase non abbiamo ancora la paginazione e quindi la rilocazione degli indirizzi deve essere fatta a mano, almeno fino a che il kernel non ha predisposto il passaggio dalla segmentazione alla paginazione.

Per svolgere questo tipo di operazione viene introdotta la macro `RELOC` così definita:

```
#define KERNBASE  (0xf0000000)
#define RELOC(x)     ((x) { KERNBASE)
```

com'è facile intuire questa macro sottrae ad ogni indirizzo che le viene passato come parametro il valore 0xf0000000. Quindi ad esempio:

$$RELOC(0xf010000c) = 0x0010000c$$

semanticamente trasforma un indirizzo virtuale di kernel nel suo corrispondente fisico. La funzione RELOC lavora nella prima parte del kernel, ed in questa prima parte il kernel predispone le strutture di VM necessarie ad operare correttamente sul resto del codice, più precisamente

```
# '_start' specifies the ELF entry point.  Since we haven't set up
```

```
# virtual memory when the bootloader enters this code, we need the
# bootloader to jump to the *physical* address of the entry point.
.globl _start
_start = RELOC(entry)

.globl entry
entry:
movw $0x1234,0x472 # warm boot

# We haven't set up virtual memory yet, so we're running from
# the physical address the boot loader loaded the kernel at: 1MB
# (plus a few bytes).  However, the C code is linked to run at
# KERNBASE+1MB.  Hence, we set up a trivial page directory that
# translates virtual addresses [KERNBASE, KERNBASE+4MB) to
# physical addresses [0, 4MB).  This 4MB region will be
# sufficient until we set up our real page table in mem_init
# in lab 2.

# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl $(RELOC(entry_pgdir)), %eax
movl %eax, %cr3
# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov $relocated, %eax
jmp *%eax
```

Come si può constatare RELOC è applicata solo a due nomi simbolici `start` e `entry_pgdir`. A `start` il linker assegnerà, *su esplicita richiesta dei progettisti*, il valore 0xf010000c, che ricordiamo è l'entry point del file eseguibile che contiene l'eseguibile del kernel, ma attraverso la funzione RELOC tale valore sarà ricomputato in 0x0010000c cioè l'indirizzo di memoria dove si trova la prima istruzione eseguibile del kernel. La variabile `entry_pgidr` contiene invece l'indirizzo in cui è caricata la nuova page direc-

tory che vogliamo attivare per avviare la traduzione degli indirizzi virtuali del kernel siano mappati nei corrispondenti indirizzi fisici. Anche ad essa il linker assegnerà un valore situato al di sopra di 0xf0100000 ed in questa fase del kernel questo indirizzo deve essere tradotto dalla RELOC nel corrispondente indirizzo fisico.

Le suddette istruzioni servono quindi a fare in modo che la MMU provveda automaticamente a mappare gli indirizzi virtuali del kernel nei corrispondenti indirizzi fisici, cioè effettui il seguente mapping:

$$[KERNBASE, KERNBASE + 4MB) \rightarrow [0, 4MB)$$

nei prossimi paragrafi spieghiamo come può essere fatta questa trasformazione.

4.1 Il supporto hardware alla VM

Riportiamo qui sotto lo schema adottato sull'architettura IA-32 per la traduzione degli indirizzi logici in indirizzi fisici.

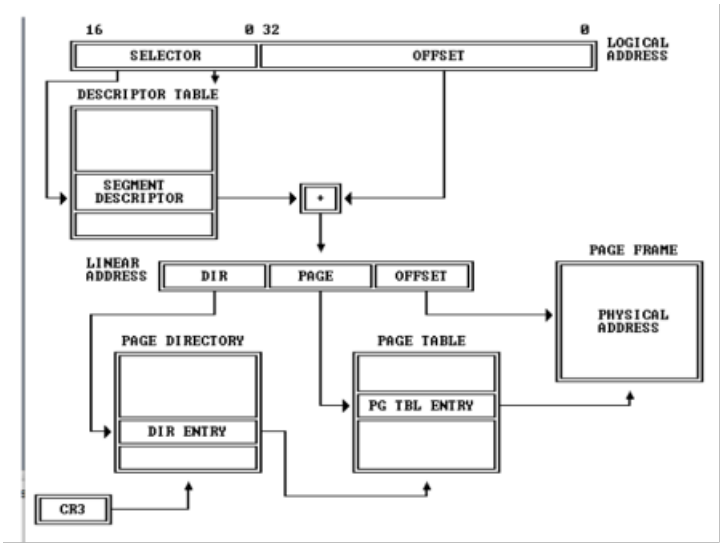

Figura 4.1: Schema di traduzione di un indirizzo logico in fisico

Com'è possibile vedere un indirizzo logico subisce due trasformazioni la prima da indirizzo logico in indirizzo lineare e la seconda da indirizzo lineare in indirizzo fisico.

4.1.1 Da indirizzo logico a lineare

All' indirizzo logico di 32 bit generato dal compilatore, viene sommato l'indirizzo d'inizio del segmento il cui indice in GDT è contenuto in un registro selettore. Nel caso in cui l'indirizzo in questione sia relativo ad un'istruzione il selettore sarà il registro CS, nel caso si riferisca a dati sarà il registro DS. Quindi come funziona? Concettualmente questo il modo con cui si procede alla generazione di un indirizzo logico:

1. la CPU riceve un indirizzo logico da 32 bit;
2. l'indirizzo viene passato alla MMU;
3. la MMU accede al segment descriptor contenuto nella GDT, nella posizione definita da CS (se istruzione) o DS (se dato);
4. Il base address contenuto nel segment descriptor viene sommato all'indirizzo logico, dando origine all'indirizzo lineare.

Nel caso particolare di JOS i descrittori di segmento della GDT sono stati caricati tutti con `base address`= 0, annullando di fatto ogni possibile effetto della segmentazione, poiché sia il segmento codice che il segmento dati operano sull'intera memoria a disposizione senza alcuna separazione hardware. L'indirizzo lineare viene a sua volta interpretato dall'hardware come composto da tre campi, i 10 bit più significativi rappresentano il Page directory Index, i successivi 10 il Page table Index, ed i rimanenti 12 bit il page offset. Come sono usati questi campi per generare un indirizzo fisico?

1. Tramite l'indirizzo fisico contenuto nel registro CR3 si accede alla Page Directory o tabella delle pagine di primo livello spiazzandosi del valore indicato dal Page Directory Index; quindi in sostanza si accede a Page Directory[Page Directory Index] il valore così ottenuto è l'indirizzo fisico della Page Table o tabella delle pagine di 2° livello a cui l'indirizzo lineare si riferisce;
2. a questo indirizzo sommiamo i secondi 10 bit dell'indirizzo lineare accediamo quindi a Page Table [Page Table Index] che conterrà l'indirizzo fisico del Page Frame a cui l'indirizzo lineare si riferisce;
3. all'indirizzo fisico del Page Frame sommiamo agli ultimi 12 bit dell'indirizzo lineare ottenendo così il corrispondente indirizzo fisico ricercato.

Ora che abbiamo visto come funziona il meccanismo di memoria virtuale dobbiamo fare in modo che il kernel di JOS costruisca le necessarie strutture dati che consentano di mappare gli indirizzi virtuali da KERNBASE a (KERNBASE + 4 MB) sugli indirizzi fisici da 0 a 4 MB, e gli indirizzi virtuali da 0 a 4MB sugli indirizzi fisico da 0 a 4MB. Per fare questo è però necessario un ulteriore approfondimento sulle strutture dati, in particolare è necessario conoscere il formato degli elementi della Page Directory e della Page Table, il cui layout è sotto riportato (gli elementi di page directory e page table hanno lo stesso formato).

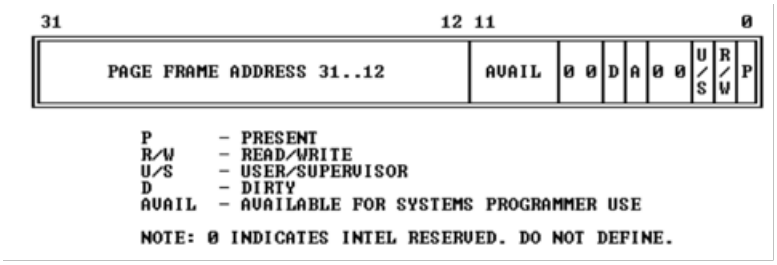

Figura 4.2: Layout di elemento di Page Directory

Questa la descrizione dei campi più importanti:

bit 1 : Pagina presente in memoria? Si ok. No page fault;

bit 2 : diritti di accesso alla pagina; 0 solo lettura, 1 lettura/scrittura;

bit 3 : user/system page, 1 se la pagina è accessibile solo dal kernel;

bit 4 : dirty bit, 1 se la pagina è stata modificata dopo il suo caricamento in memoria;

bit 31 -12 : indirizzo fisico della page table (o frame nel caso di tabelle delle pagine)

Procediamo ora con la costruzione delle struttura dati interessate in modo tale che consentano alla MMU di mappare l'indirizzo KERNBASE (0xf0000000) nell'indirizzo fisico 0. Ovviamente è il kernel, in base al contenuto che caricherà nella PD e PT, ad indirizzare il comportamento della MMU. Vediamo come fare usando come esempio l'indirizzo 0xf0000000 = 1111 0000 0000 0000 00000 0000 0000 0000. Nella fase di traduzione di questo indirizzo la MMU userà i 10 bit più significativi per accedere alla PD, in

particolare accederà all'elemento di indice $0x1111000000 = 960$ dove preleverà l'indirizzo di una PT, che dovrà contenere gli indirizzi fisici dei frame di memoria che devono essere mappati sugli indirizzi virtuali KERNBASE - (KERNBASE + 4MB), cioè i frame i cui indirizzi vanno da 0x00000000 a 0x003ff000 Ovviamente spetta sempre al kernel costruire questa tabella delle pagine che avrà il seguente contenuto:

0x00000000
0x00001000
0x00002000
0x00003000
0x00004000
0x00005000
0x00006000
0x00007000
0x00008000
0x00009000
0x0000a000
0x0000b000
0x0000c000
0x0000d000
0x0000e000
0x0000f000
0x0001000
...

La suddetta tabella viene caricata attraverso il seguente codice C (file kern/entrypgdir.c).

```
typedef uint32_t pte_t;
typedef uint32_t pde_t;

pte_t entry_pgtable[NPTENTRIES];

 // Entry 0 of the page table maps to physical page 0, entry 1 to
// physical page 1, etc.
__attribute__((__aligned__(PGSIZE)))
pte_t entry_pgtable[NPTENTRIES] = {
        0x000000 | PTE_P | PTE_W,
        0x001000 | PTE_P | PTE_W,
        0x002000 | PTE_P | PTE_W,
```

```
        0x003000 | PTE_P | PTE_W,
        0x004000 | PTE_P | PTE_W,
        0x005000 | PTE_P | PTE_W,
        0x006000 | PTE_P | PTE_W,
        0x007000 | PTE_P | PTE_W,
        0x008000 | PTE_P | PTE_W,
        0x009000 | PTE_P | PTE_W,
        0x00a000 | PTE_P | PTE_W,
        0x00b000 | PTE_P | PTE_W,
        0x00c000 | PTE_P | PTE_W,
        0x00d000 | PTE_P | PTE_W,
        0x00e000 | PTE_P | PTE_W,
        0x00f000 | PTE_P | PTE_W,
        0x010000 | PTE_P | PTE_W,
 ecc. ecc.
 };
```

Come abbiamo visto, l'indirizzo di questa tabella (entry_pgtable) deve essere inserito nella tabella di primo livello (entry_pgdir) nell'elemento di posizione 0x1111000000 e nell'elemento di posizione zero. Questo viene fatto con l'esecuzione di questo codice C:

```
 // Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.  Also, because of restrictions
// related to linking and static initializers, we use "x + PTE_P"
// here, rather than the more standard "x | PTE_P".  Everywhere else
// you should use "|" to combine flags.
__attribute__((__aligned__(PGSIZE)))
pde_t entry_pgdir[NPDENTRIES] = {
 // Map VA's [0, 4MB) to PA's [0, 4MB)
[0] = ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P,
// Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
[KERNBASE>>PDXSHIFT]
= ((uintptr_t)entry_pgtable - KERNBASE) + PTE_P + PTE_W
 };
```

Queste strutture dati vengono incluse attraverso il linker nell'eseguibile del kernel e quindi caricate insieme al kernel dal bootloader. Il kernel attra-

verso le seguenti istruzioni provvede ad inizializzare il sistema e ad operare in protected mode paginato

```
 # Load the physical address of entry_pgdir into cr3.
# entry_pgdir is defined in entrypgdir.c.
movl $(RELOC(entry_pgdir)), %eax
movl %eax, %cr3
# Turn on paging.
movl %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov $relocated, %eax
jmp *%eax
```

dopo l'esecuzione dell'istruzione `movl %eax, %cr0` la paginazione è attivata ma EIP punta ancora ad un indirizzo di memoria relativo ad una zona di memoria bassa, quindi ad un'indirizzo del tipo $0x001000f4$. Poiché la paginazione è attivata questo indirizzo sarà processato dalla MMU che userà i primi dieci bit dello stesso per accedere al corrispondente elemento della page directory il cui indirizzo è stato carico in CR3. Nel nostro caso i primi 10 bit sono 0000 0000 00. Quindi la MMU accede al primo elemento della Page Directory, che abbiamo precedentemente predisposto in modo da effettuare il mapping identità tra indirizzo logico e indirizzo fisico.

Nel nostro caso il valore di EIP sarà tradotto nell'indirizzo dell'istruzione `mov $relocated, %eax`, che sarà quindi regolarmente eseguita. Da notare che senza il mapping in questione non sarebbe stato possibile per il processore eseguire l'istruzione `mov $relocated, %eax`. Lo stesso discorso vale per l'istruzione successiva `jmp *%eax`.

Ma cosa succede quando vengono eseguite queste due istruzioni? Poiché `$relocated` è l'indirizzo simbolico di un'istruzione che avrà un valore calcolato a partire da KERNBASE, quindi superiore a $0xf0000000$. Con l'istruzione `jmp *%eax` forziamo il valore di `relocated` all'intero di EIP (cosa che non si può fare in altro modo) e quindi incominceremo ad utilizzare indirizzi virtuali in memoria alta dove è virtualmente posizionato il kernel.

A questo punto abbiamo eseguito le prime 10 istruzioni del kernel e quindi il primo pezzettino di kernel ha abilitato la paginazione, ha predisposto le

prime Page Directory e Page Table che dovranno poi essere completate. Quest'ultima fase di inizializzazione del sistema viene lasciata ad una procedura C che a sua volta richiama tutta una serie di procedure per l'inizializzazione dei dispositivi, l'inizializzazione della memoria e cosi via...

Quindi ancora una volta si deve eseguire una routine in C e va quindi predisposto uno stack per consentirle di operare correttamente. Di fatto c'è ancora lo stack che aveva preparato il boot loader a partire da $0x07c0$ (zona bassa della memoria) per poter eseguire la procedura bootmain, siamo però interessati a predisporre un nuovo stack che opera nella zona di memoria virtuale in cui risiede il kernel, e che di fatto opererà come stack del kernel ognivolta che lo stesso andrà in esecuzione. Per predisporre l'area di memoria è sufficiente usare le seguenti istruzioni dichiarative presenti in entry.s.

```
.data
###################################
# boot stack
###################################
.p2align PGSHIFT # force page alignment
.globl bootstack
bootstack:
.space KSTKSIZE
.globl bootstacktop
bootstacktop:
```

Questa è la definizione dello Stack il cui top address è l'etichetta bootstack e che ha una dimensione prefissata che è `KSTKSIZE` ovvero $8 * 4096 = 32768 bytes = 8$ pagine.

```
#define KSTKSIZE (8 * PGSIZE)
#define PGSIZE (4096)
```

L'inizializzazione dei relativi registri è effettuata dalle istruzioni:

```
# Clear the frame pointer register (EBP)
# so that once we get into debugging C code,
# stack backtraces will be terminated properly.
 movl $0x0,%ebp # nuke frame pointer
# Set the stack pointer
 movl $(bootstacktop),%esp
# now to C code
call i386_init
```

Terminata la preparazione dello stack eseguiamo l'istruzione `call i386_init` che richiama la parte di kernel che si occuperà di fare l'inizializzazione del sistema. Di fatto da questa call non ritorniamo più, si noti che l'indirizzo di ritorno della `call i386_init` è la prima informazione che inseriamo nello Stack del kernel, anche se non sarà mai utilizzata.

4.2 kern/init.c

Questo file contiene l'altra parte del kernel che provvede ad effettuare l'inizializzazione del sistema su un sistema Intel IA-32. Questo il codice che ci interessa:

```
void
i386_init(void)
{
extern char edata[], end[];

// Before doing anything else, complete the ELF loading process.
// Clear the uninitialized global data (BSS) section of our program.
// This ensures that all static/global variables start out zero.
memset(edata, 0, end - edata);

// Initialize the console.
// Can't call cprintf until after we do this!
cons_init();
cprintf("6828 decimal is %o octal!\n", 6828);

// Lab 2 memory management initialization functions
mem_init();

// Lab 3 user environment initialization functions
env_init();
trap_init();
```

Questa procedura fa l'inizializzazione della tastiera e del video tramite la funzione: `cons_init();` poi fa l'inizializzazione della memoria tramite la funzione: `!mem_initi();!`successivamente procede con l'inizializzazione del process manager la funzione `!env_init();!`, dopo di che predispone l'ambiente per la gestione degli Interrupt e delle eccezioni tramite la funzione: `!trap_init();!`.

Capitolo 5

La memoria virtuale in JOS

5.1 Introduzione

Nel precedente capitolo abbiamo visto che nella fase iniziale il kernel predispone page directory e relativa tabella delle pagine per poter svolgere le prime funzionalità. Ricordiamo che la Page directory costruita conteneva due entry

- la prima che effettua il mapping identità agganciando una Page Table che mappa i primi 4 MB di memoria logica con i primi 4 MB di memoria fisica a partire dall'indirizzo 0x00000000,
- la seconda che attraverso la corrispondente Page Table effettua il seguente mapping:

$$[KERNBASE, KERNBASE + 4MB] \rightarrow [0x00000000, 0x00400000]$$

Vediamo ora come questa struttura dati viene ulteriormente arricchita e le diverse procedure di sistema realizzate per operare su page directory e page table per facilitare la gestione della memoria fisica e virtuale. Vediamo quindi le componenti più importanti del Memory Manager di JOS. Introduciamo la convenzione di riferirci a procedure e dati che interessano Page directory usando il prefisso PD, mentre useremo il prefisso PT per riferirci a quanto compete le Page Table.

Riprendiamo il formato degli elementi della Page Directory e della Page Table, il cui layout è sotto riportato (gli elementi di page directory e page table hanno lo stesso formato) e riportiamo il significato dei campi più importanti.

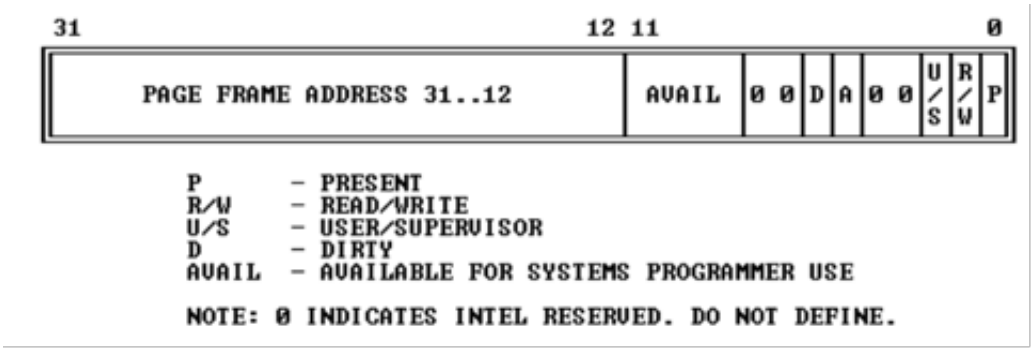

Figura 5.1: Layout di elemento di Page Directory

Address gli indirizzi presenti nelle Page Directory, nelle Page Table così come quelli presenti nel registro CR3 sono sempre esclusivamente indirizzi fisici di 20 bit e quindi allineati alle pagine. Gli indirizzi presenti nella Page Directory contengono gli indirizzi fisici delle corrispondenti Page Table. Gli indirizzi contenuti nella Page Table sono invece indirizzi fisici di Page frame.

Present Bit Indica (se posto uguale a 1) la presenza in memoria del corrispondente elemento, è quindi il bit che controlla il page fault.

R/W e U/S bit Questi bit sono legati alle protezioni delle pagine in memoria centrale, rivediamoli un altra volta: $R/W = 1$ se la pagina può essere modificata, in caso contrario siamo in presenza di una pagina di sola lettura (configurazione tipica per una pagina che contiene codice). U/S = 0 siamo in presenza di una struttura dati o zona di memoria riservata al kernel.

Poniamoci ora il problema di realizzare un Memory Manager che sia in grado di creare a gestire le diverse struttura dati e procedure che consentano l'implementazione di un sistema, seppur primitivo, di memoria virtuale.

5.2 Funzioni di base

Nell'ambito del Memory Manager sono individuabili alcune funzioni elementari che operano sugli elementi di page directory/page Table e sugli indirizzi al fine di estrarne informazioni utili allo svolgimento di diverse operazioni.

Nell'ambito del sistema operativo JOS queste funzioni sono implementate nel file `incmmu.h` e sono qui di seguito descritte, facendo riferimento allo schema qui sotto riportato.

```
// A linear address 'la' has a three-part structure as follows:
//
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table    | Offset within Page  |
// |      Index     |      Index      |                     |
// +----------------+-----------------+---------------------+
//  \--- PDX(la) --/ \--- PTX(la) --/ \---- PGOFF(la) ----/
//  \---------- PGNUM(la) ----------/
//
// The PDX, PTX, PGOFF, and PGNUM macros decompose linear addresses as shown.
// To construct a linear address la from PDX(la), PTX(la), and PGOFF(la),
// use PGADDR(PDX(la), PTX(la), PGOFF(la)).
```

Definiamo quindi la macro `PDX(la)` che prende come input un indirizzo lineare di 32 bit e restituisce i primi 10 bit ovvero l'indice dell'elemento di Page Directory a cui si riferisce l'indirizzo `la`,

```
#define PDXSHIFT 22 // offset of PDX in a linear address
// page directory index
#define PDX(la) ((((uintptr_t) (la)) >> PDXSHIFT) & 0x3FF)
```

Questa macro semplicemente opera uno shift a destra di 22 posizioni di `la` , dopo avere preventivamente effettuato un'operazione di cast a intero di 32 bit senza segno. Il risultato dello shift sarà una stringa del tipo $0000000000000000000000xxxxxxxxxx$ che viene poi messa in `and` con la stringa 00000000000000000000001111111111 al fine di garantire la selezione dei soli 10 bit più significativi di `la`.

Analogamente a quanto fatto per `PDX(la)` si definiscono le seguenti macro.

PTX(la): prende in input un indirizzo lineare di 32 bit e restituisce i secondi 10 bit di mezzo ovvero l'indice dell'elemento di Page Table a cui `la` si riferisce, ed è definita come:

```
#define PTXSHIFT 12 // offset of PTX in a linear address
// page table index
#define PTX(la) ((((uintptr_t) (la)) >> PTXSHIFT) & 0x3FF)
```

PGNUM(la) che dato un indirizzo lineare restituisce il numero della pagina logica a cui si riferisce, ed è definita come

```
// page number field of address
#define PGNUM(la) (((uintptr_t) (la)) >> PTXSHIFT)
```

PGOFF(la) che restituisce l'offset relativo all'indirizzo `la` cioè lo spiazzamento all'interno della pagina in cui è posizionato l'elemento indirizzato.

```
// offset in page
#define PGOFF(la) (((uintptr_t) (la)) & 0xFFF)
```

Accanto alle suddette macro sono definite una serie di costanti utili per la manipolazione dei vari oggetti riferibili alla gestione della memoria:

```
// Page directory and page table constants.
#define NPDENTRIES 1024 // page directory entries per page directory
#define NPTENTRIES 1024 // page table entries per page table

#define PGSIZE 4096 // bytes mapped by a page
#define PGSHIFT 12 // log2(PGSIZE)

#define PTSIZE (PGSIZE*NPTENTRIES) // bytes mapped by a page directory entry
#define PTSHIFT 22 // log2(PTSIZE)

#define PTXSHIFT 12 // offset of PTX in a linear address
#define PDXSHIFT 22 // offset of PDX in a linear address

// Page table/directory entry flags.
#define PTE_P 0x001 // Present
#define PTE_W 0x002 // Writeable
#define PTE_U 0x004 // User
#define PTE_PWT 0x008 // Write-Through
#define PTE_PCD 0x010 // Cache-Disable
#define PTE_A 0x020 // Accessed
#define PTE_D 0x040 // Dirty
#define PTE_PS 0x080 // Page Size
#define PTE_G 0x100 // Global
```

Analizziamo le principali:

- PGSIZE = 4096 bytes = $4KB \rightarrow$ la quantità di memoria contenuta in una pagina;
- NPTENTRIES = 1024 $\rightarrow$ Number Page Table Entries, cioè numero di elementi di una page table;
- PTSIZE = $(PGSIZE{*}NPTENTRIES) = 4KB{*}1024 = 4096KB = 4MB \rightarrow$ che è la quantità di memoria indirizzabile da una Page Table;
- PTE_P $\rightarrow$ è la maschera 000000000001 che useremo per abilitare il primo bit meno significativo di un elemento di Page directory/page table, bit che ricordiamo indica la presenza o meno di una pagina in memoria centrale;
- PTE_W $\rightarrow$ è la maschera che useremo per abilitare il secondo bit meno significativo di un elemento di Page directory/page table, che ricordiamo indica se la pagina è scrivibile (bit a 1) o solo leggibile (bit a 0)
- PTE_U $\rightarrow$ è la maschera che useremo per abilitare il terzo bit meno significativo di un elemento di Page directory/page table, bit che indica se la pagina è accessibile anche dall'utente oltre che dal kernel (bit a 1) o accessibile solo dal kernel (bit a 0).

5.3 La gestione della memoria fisica

Uno degli obiettivi del Memory Manager è quello di tenere traccia dei frame di memoria disponibili ed occupati. Quindi una delle prime cose da fare è predisporre una struttura dati che ci consenta di capire quali pagine della memoria fisica possiamo usare e quali pagine della memoria fisica non possiamo usare e poter così far fronte alle richieste di memoria che perverranno dai vari processi durante la loro esecuzione. La memoria fisica viene gestita in JOS in maniera molto elementare, attraverso un lista linkata di elementi del tipo `PageInfo`, gli elementi linciati tra loro appartengono a frame liberi, elementi che non sono linkati alla lista appartengono a frame occupati. Esiste un puntatore dedicato ad indicare il primo frame disponibile della lista chiamato `page_free_list`.

```
/ * Each struct PageInfo stores metadata for one physical page.
 * Is it NOT the physical page itself, but there is a one-to-one
 * correspondence between physical pages and struct PageInfo's.
 * You can map a struct PageInfo * to the corresponding physical address
```

```
 * with page2pa() in kern/pmap.h.
 */
struct PageInfo {
// Next page on the free list.
struct PageInfo *pp_link;

// pp_ref is the count of pointers (usually in page table entries)
// to this page, for pages allocated using page_alloc.
// Pages allocated at boot time using pmap.c's
// boot_alloc do not have valid reference count fields.

uint16_t pp_ref;
};
```

Quindi per ogni frame esiste un elemento di questo tipo che rispetto al frame contiene le seguenti informazioni:

1. Un puntatore che serve a collegare tra loro i frame liberi;
2. Un contatore che indica il numero di processi che stanno usando quella pagina fisica.

Questa struttura dati viene inizializzata dalla procedura `page_init` qui di seguito illustrata:

```
void page_init(void)
{
// The example code here marks all physical pages as free.
// However this is not truly the case.  What memory is free?
//  1) Mark physical page 0 as in use.
//     This way we preserve the real-mode IDT and BIOS structures
//     in case we ever need them.  (Currently we don't, but...)
//  2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE) is free.
//  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must  never be allocated
//  4) Then extended memory [EXTPHYSMEM, ...).
//     Some of it is in use, some is free. Where is the kernel
//     in physical memory?  Which pages are already in use for
//     page tables and other data structures?
//
size_t i,j;
struct PageInfo *pages; // Physical page state array
physaddr_t physadd;
```

```
page_free_list = NULL;
for (i = 1; i < npages_basemem; i++) {
pages[i].pp_ref = 0;
pages[i].pp_link = page_free_list;
page_free_list = &pages[i];
}
physadd = PADDR (nextfree);
j = (int) (physadd) / PGSIZE;
for (i = j; i< npages; i++) {
pages[i].pp_ref = 0;
pages[i].pp_link = page_free_list;
page_free_list = &pages[i];
}

}
```

Come si può notare la lista dei frame liberi viene creata a ritroso con il puntatore `page_free_list` che punta sul frame libero di indirizzo più alto, questa scelta facilita significativamente le gestione della lista dei frame liberi. Questa struttura dati sarà usata ogni volta che si dovranno allocare frame a processi che ne fanno richiesta o recuperare frame da processi che sono terminati. A questo scopo si usano le seguenti procedure.

```
//
// Allocates a physical page.  If (alloc_flags & ALLOC_ZERO), fills the entire
// returned physical page with '\0' bytes.  Does NOT increment the reference
// count of the page - the caller must do these if necessary (either explicitly
// or via page_insert).
//
// Returns NULL if out of free memory.
//
// Hint: use page2kva and memset
struct PageInfo * page_alloc(int alloc_flags)
{  struct PageInfo * allocated_page;
if (page_free_list == NULL) return NULL ;
allocated_page = page_free_list;
page_free_list = page_free_list->pp_link;
if (alloc_flags & ALLOC_ZERO)
  memset(page2kva(allocated_page), 0, PGSIZE);
return allocated_page;
}
```

```
//
// Return a page to the free list.
// (This function should only be called when pp->pp_ref reaches 0.)
//
void
page_free(struct Page *pp)
{
if (pp->pp_ref == 0) {
memset(page2kva(pp), 0x00, PGSIZE);
pp->pp_link = page_free_list;
page_free_list = pp; }
}
}
```

Attraverso la struttura dati `pages` ogni frame può essere univocamente identificato oltre che dal suo indirizzo fisico di memoria anche dalla posizione che occupa all'interno della struttura dati, in particolare l'i-esimo frame di memoria occuperà la i-esima posizione all'interno della struttura `pages`. Questo consente di poter facilmente risalire da uno dei due dati all'altro. Queste le procedure che effettuano questo mapping:

```
static inline physaddr_t
page2pa(struct PageInfo *pp)
{
return (pp - pages) << PGSHIFT;
}

static inline struct PageInfo*
pa2page(physaddr_t pa)
{
if (PGNUM(pa) >= npages)
panic("pa2page called with invalid pa");
return &pages[PGNUM(pa)];
}
```

La funzione page2pa() dato in input un puntatore alla struttura dati `pages` ritorna l'indirizzo fisico del corrispondente frame. Come opera `page2pa()` ? Attraverso l'operazione (pp – pages) otteniamo la posizione della struttura dati indirizzata da `pp`. Successivamente con lo shift di 12 bit a sinistra, che equivale ad una moltiplicazione per 2^{12} otteniamo l'indirizzo della pagina ricercata.

Nel caso della procedura `pa2page` si tratta di risalire dall'indirizzo fisico di un frame alla sua posizione nella struttura `pages`. Questa trasformazione avviene recuperando dall'indirizzo fisico del frame, il suo numero che viene poi usato come indice all'interno della struttura dati `pages`.

5.4 Indirizzi Logici e Indirizzi Fisici

Una volta che il sistema è stato predisposto ad operare in Protected mode, tutti gli indirizzi acquisiti dal processore sono considerati indirizzi logici e sono trasformati dalla MMU in indirizzi fisici. A questo meccanismo non si può sottrarre neanche il Sistema operativo, quindi dal momento in cui l'hardware entra in protected mode tutte le istruzione decodificate dal processore subiscono, quando necessario, il processo di rilocazione dinamica degli indirizzi da parte della MMU. A maggior ragione significa che tutte le variabili puntatori sono interpretate come indirizzi virtuali.

Il kernel può però avere la necessità di manipolare direttamente indirizzi fisici di alcune strutture dati di kernel, per esempio per il caricamento di page directory e page table. Diventa allora necessario disporre di un meccanismo che consenta facilmente di risalire dall'indirizzo logico di un elemento che si trovi nell'area del Kernel al suo corrispondente indirizzo fisico e viceversa. In JOS il gioco è facile, perché abbiamo visto che il kernel si trova fisicamente nella parte bassa della memoria (a partire dalll'indirizzo fisico 0) che viene mappata all'indirizzo virtuale 0xf0000000 e cosi via a salire. Quindi banalmente possiamo ottenere l'indirizzo fisico di un elemento di kernel, a partire dal suo indirizzo lineare e applicando la seguente formula:

$$\texttt{indirizzo\_fisico = indirizzo\_lineare} - 0xf0000000$$

Questa formula viene usata da due macro in JOS che ci consentono dato un indirizzo fisico di una componente di Kernel di risalire al suo indirizzo logico o viceversa. Le due macro sono:

1. `KADDR(pa)` ovvero dall'indirizzo fisico si ottiene quello virtuale $ka = pa + 0xf0000000$;
2. `PADDR(ka)` ovvero dall'indirizzo virtuale si ottiene quello fisico $pa = ka – 0xf0000000$.

Qui sotto è riportata la loro implementazione:

```
/* This macro takes a kernel virtual address -- an address that points above
```

```
 * KERNBASE, where the machine's maximum 256MB of physical memory is mapped --
 * and returns the corresponding physical address.  It panics if you pass it a
 * non-kernel virtual address.
 */
#define PADDR(kva) _paddr(__FILE__, __LINE__, kva)

static inline physaddr_t
_paddr(const char *file, int line, void *kva)
{
if ((uint32_t)kva < KERNBASE)
_panic(file, line, "PADDR called with invalid kva %08lx", kva);
return (physaddr_t)kva - KERNBASE;
}

/* This macro takes a physical address and returns the corresponding kernel
 * virtual address.  It panics if you pass an invalid physical address. */

#define KADDR(pa) _kaddr(__FILE__, __LINE__, pa)

static inline void*
_kaddr(const char *file, int line, physaddr_t pa)
{
if (PGNUM(pa) >= npages)
_panic(file, line, "KADDR called with invalid pa %08lx", pa);
return (void *)(pa + KERNBASE);
}
```

Ricordiamo che il kernel JOS è compilato a partire dall'indirizzo 0xf0100000, quindi tutti gli indirizzi che vengono generati dal kernel in esecuzione partono da 0xf0100000. Però in realtà sappiamo esattamente dov'è caricato fisicamente il kernel, ovvero sappiamo che fisicamente il kernel è in memoria centrale a partire dall'indirizzo fisico 0x00100000 ovvero dal 1° MB di memoria centrale. Quindi la situazione è questa: 1) Il Kernel di JOS fisicamente si trova in memoria centrale all'indirizzo fisico 0x00100000 e quindi dal 1° MB di memoria centrale 2) Però il Kernel di JOS logicamente si trova in memoria virtuale all'indirizzo logico 0xf0100000 e quindi dal 3° GB di memoria virtuale. Il mapping tra i due spazi di indirizzamento viene effettuato dalla MMU utilizzando le strutture dati che il kernel deve opportunamente predisporre.

5.5 mem_init

`i386_init()` è il kernel di JOS, richiamato dal boot loader, è un modulo che si occupa di inizializzare tutti i moduli di sistema, compresa ovviamente la memoria virtuale di JOS tramite la procedure `mem_init()`. I compiti principali di mem_init() sono:

- riservare lo spazio per le strutture dati del kernel;
- costruire una nuova Page Directory e le relative Page Table che si occupino di mappare il Memory layout effettivo utilizzato dal kernel di JOS e qui sotto riportato, Queste nuove strutture dati rimpiazzeranno poi la Page Directory e le Page Table in uso.

Analizziamo ora la prima porzione della procedura `mem_init()` qui sotto riportata:

```
void
mem_init(void)
{
uint32_t cr0;
size_t n;
char * dummy;

// Find out how much memory the machine has (npages & npages_basemem).
i386_detect_memory();

// Remove this line when you're ready to test this function.
// panic("mem_init: This function is not finished\n");

////////////////////////////////////////////////////////////////////
// create initial page directory.
kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
memset(kern_pgdir, 0, PGSIZE);

////////////////////////////////////////////////////////////////////
// Recursively insert PD in itself as a page table, to form
// a virtual page table at virtual address UVPT.
// (For now, you don't have understand the greater purpose of the
// following line.)
// Permissions: kernel R, user R
kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
```

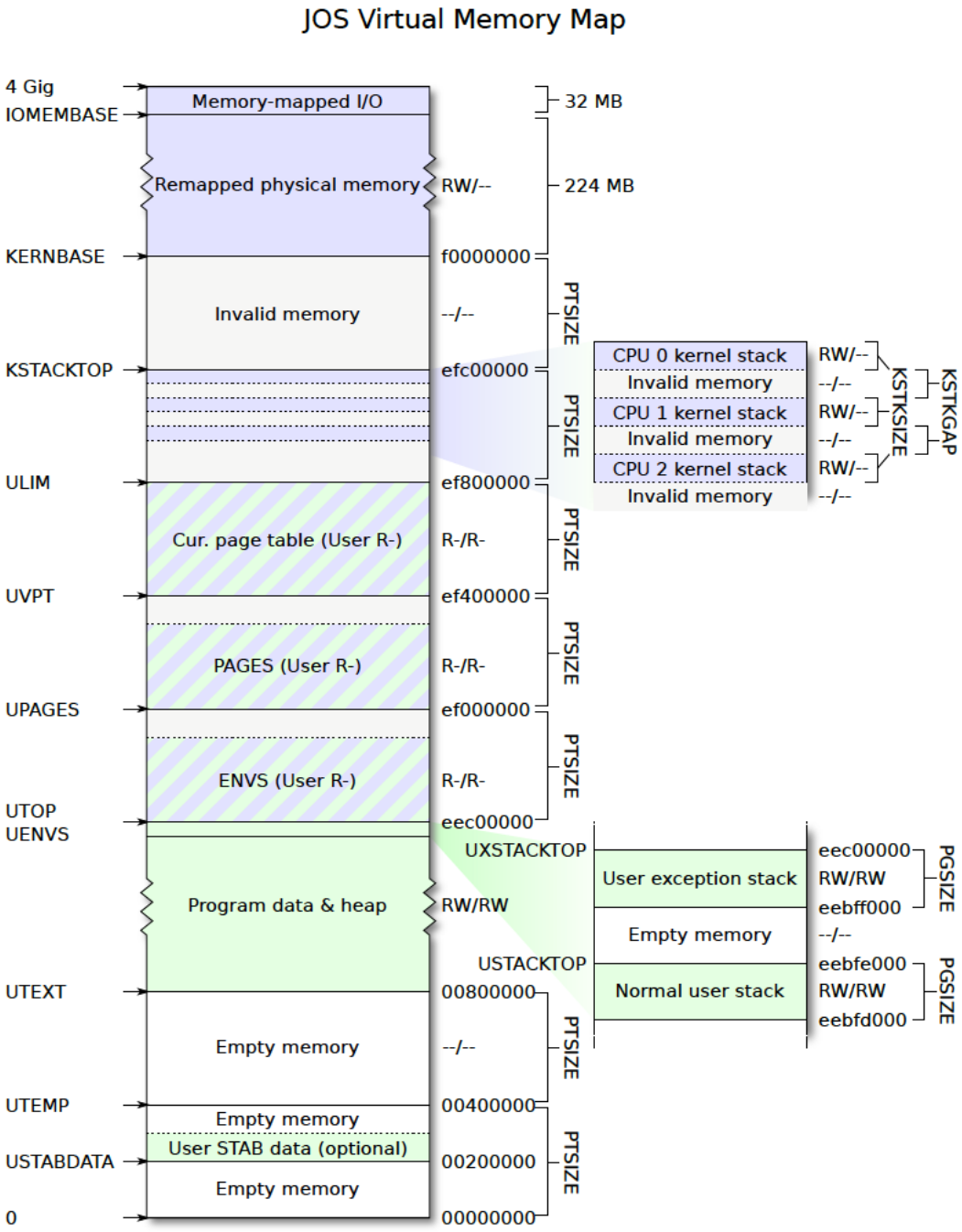

Figura 5.2: Layout di Memoria

```
//////////////////////////////////////////////////////////////////////
// Allocate an array of npages 'struct PageInfo's and store it in 'pages'.
// The kernel uses this array to keep track of physical pages: for
// each physical page, there is a corresponding struct PageInfo in this
// array.  'npages' is the number of physical pages in memory.
pages = boot_alloc (npages * sizeof(struct PageInfo));

//////////////////////////////////////////////////////////////////////
// Make 'envs' point to an array of size 'NENV' of 'struct Env'.
envs = boot_alloc (NENV * sizeof(struct Env));

//////////////////////////////////////////////////////////////////////
// Now that we've allocated the initial kernel data structures, we set
// up the list of free physical pages. Once we've done so, all further
// memory management will go through the page_* functions. In
// particular, we can now map memory using boot_map_region
// or page_insert
page_init();
...
```

L'istruzione `i386_detec_memory()` si occupa principalmente di rilevare la memoria fisica disponibile sul sistema e di calcolare i valori di: `npages` che contiene il numero di pagine fisiche disponibili sul sistema, e `npages_basemem` che contiene il numero di pagine fisiche che compongono la parte di base memory (memoria fisica entro il 1 MB). Dopodiché viene richiamato la procedura `bootalloc (uint_t n)` che riserva un blocco n byte di memoria restituendo nella variabile `nextfree` l'indirizzo d'inizio del blocco stesso. Vediamo più in dettaglio questa procedura:

```
boot_alloc(uint32\_t n)
{
static char *nextfree; // virtual address of next byte of free memory
char *result;

// Initialize nextfree if this is the first time.
// 'end' is a magic symbol automatically generated by the linker,
// which points to the end of the kernel's bss segment:
// the first virtual address that the linker did *not* assign
// to any kernel code or global variables.
```

```
if (!nextfree) {
extern char end[];
nextfree = ROUNDUP((char *) end, PGSIZE);
}

// Allocate a chunk large enough to hold 'n' bytes, then update
// nextfree.  Make sure nextfree is kept aligned
// to a multiple of PGSIZE.
//
if (n > 0) {
 // Round-up alloc_size promises round-up nextfree.
uint32_t alloc_size = ROUNDUP(n, PGSIZE);
result = nextfree;
nextfree += alloc_size;

// Because in the beginning phase of booting,
// only 4MB physical memory is mapped.
/ Memory allocation cannot exceeds the limit.
if ((uintptr_t)nextfree >= 0xf0400000)
   panic("boot_alloc: out of memory");
} else {
result = nextfree;
}

return result;
}
```

La prima volta che viene chiamata la procedura `boot_alloc` inizializza la variabile `nextfree` facednola puntare al primo byte disponibile dopo il segmento dati. Questo indirizzo viene calcolato dal linker e passato attraverso la variabile `end`. La procedura `boot_alloc` provvede anche ad allineare questo valore alla pagina. Successivamente la procedura `boot_alloc` provvede ad assegnare gli n byte richiesti provvedendo prima a verificare che gli stessi non superino la memoria disponibile. ricordiamo che in questa fase la memoria che la MMU è in grado di gestire sono solo i prima 4MB di memoria fisica, perché sono gli unici mappati da page directory/page table. Nell'ambito della procedura `boot_alloc`, `nextfree` contiene sempre l'indirizzo del primo byte di memoria disponibile e `result` l'indirizzo iniziale della zona di memoria che è stata allocata. Una volta spiegata la `boot_alloc()`,

possiamo ritornare alla `mem_init()`, che procede (attraverso la procedura `boot_alloc`) ad allocare:

1. una zona di memoria che conterrà la nuova page directory, il cui indirizzo d'inizio viene caricato nella variabile `kern_pgdir`;
2. una zona di memoria che conterrà la struttura dati `pages` per la gestione della memoria fisica, struttura che viene inizializzata dalla procedura `page_init` già studiata;
3. una zona di memoria che conterrà i PCB (ENV in JOS).

Svolte queste operazioni è necessario provvedere a mappare le diverse zone di memoria seguendo lo schema sopra riportato. In particolare, in questa prima fase noi lavoreremo sulle sole componenti che interessano il kernel a cui ricordiamo è riservata la parte superiore delle memoria in particolare la zona compresa tra $[3GB, 4GB)$, che dovranno essere mappati usando la nuova page directory indirizzata da `kern_pgdir`.

Adesso dobbiamo mappare all'interno di questa nuova Page Directory, quello che punta la variabile `pages`, dove `pages` è l'indirizzo d'inizio della struttura dati predisposta per la gestione delle pagine:

```
//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
pgdir = kern_pgdir;
dummy = page_alloc(0);
pgdir0 = &pgdir[PDX((uint32_t  *)UPAGES)]; // Mettere il suo indirizzo nella director
*pgdir0 = page2pa(dummy) | PTE_P;
//IN OGNI ELEMENTO DELLA TABELLA CARICA
pt1 = (uint32_t *)page2kva(dummy);
n = ROUNDUP(npages*sizeof(struct Page), PGSIZE);
for (i = 0; i < n; i += PGSIZE){
*pt1 = (PADDR(pages) + i)|PTE_W|PTE_P; //abilita pagina a lettura/scrittura
 pt1++;
}
```

In particolare dobbiamo mappare la struttura dati in questione (in Read Only per gli utenti, quindi accessibile in scrittura solo dal kernel) nella

zona di memoria logica [UPAGES, UVPT) in particolare all'indirizzo lineare UPAGES. Ricordiamo che $UPAGES \rightarrow 0xef000000$ e $UVPT \rightarrow 0xef400000$ e complessivamente la zona di memoria indicata occupa un numero di byte pari a PTSIZE.

In sostanza noi dobbiamo fare in modo che indirizzi che fanno riferimento all'indirizzo virtuale UPAGES, facciano fisicamente riferimento alla struttura dati per la gestione delle pagine puntata dalla variabile `pages` che noi sappiamo essere fisicamente poco sopra lo spazio occupato dal kernel. Per ottenere questo obiettivo dobbiamo fare in modo che quando all'MMU arriva un indirizzo compreso nell'intervallo [UPAGES, UVPT), l'MMU sia condotta ad assegnare a questo indirizzo l'indirizzo fisico voluto. Noi sappiamo che dato un indirizzo la prima operazione che la MMU compie è quella di usare primi 10 bit più significativi dell'indirizzo come indice per accedere alla Page Directory il cui indirizzo è contenuto nel registro CR3. All'interno di questo elemento si aspetta di trovare l'indirizzo di una Page Table. La Page Directory è già stata allocata ed inizializzata a 0, alla posizione indicata dai primi 10 bit a sinistra di UPAGES ($0xef000000$) dobbiamo inserire l'indirizzo fisico di una page table che conterrà gli indirizzi dei frame di memoria che contengono la struttura `pages`. Analizziamo ora il codice per fare questo mapping.

La prima cosa che dobbiamo fare è allocare una Page Table attraverso l'istruzione `dummy = page_alloc(0);` che ci restituisce in `dummy` l'indirizzo di una pagina messa a disposizione dal kernel che useremo per costruire la nostra tabella delle pagine di secondo livello. Questo indirizzo va scritto all'interno della Page Directory all'indirizzo corrispondente a UPAGES, quindi l'istruzione: `pgdir0 = &kern_pgdir[PDX((uint32_t *) UPAGES)];` ci restituisce l'indirizzo dell'elemento di page directory indirizzato dai primi 10 bit di UPAGES (cioè `PDX((uint32_t *) UPAGES)`) all'interno di questo elemento inseriamo con l'istruzione `*pgdir0 = page2pa(dummy) | PTE_W | PTE_P;` l'indirizzo della page table di secondo livello con le relative protezioni.

A questo punto l'elemento UPAGES-esimo della Page Directory punta all'indirizzo fisico della Page Table puntata da dummy. La Page Table puntata da `dummy` però è ancora vuota e dobbiamo riempirla facendo in modo che la Page Table contenga gli indirizzi dei frame di memoria che contengono la struttura dati puntata da `pages`, le seguenti istruzioni svolgono questo compito:

```
Ricordiamo prima le macro utilizzate:
#define PTSIZE (PGSIZE*NPTENTRIES)  // bytes mapped by a page directory entry
#define PGSIZE 4096  // bytes mapped by a page
```

```
#define NPTENTRIES 1024  // page table entries per page table

pt1 = (uint32_t *) page2kva(dummy);  // pt1  = indirizzo logico della Page Table.
for (i = 0; i < PTSIZE; i += PGSIZE) {
*pt1 = (PADDR(pages) + i) | PTE_U | PTE_P;
pt1++;
}
```

Con l'istruzione `pt1 = (uint32_t *) page2kva(dummy);` faccio puntare `pt1` al primo elemento della page table di secondo livello. Attraverso il ciclo for riempio l'intera tabella di secondo livello (che ricordiamo indirizza 4MB di memoria) con gli indirizzi dei frame fisici che compongono la strutturata indirizzata da `pages` e relativi bit di protezione. In questo modo ogni volta che avrò un indirizzo logico a partire da UPAGES per l'estensione di 4MB lsarà mappato sulla memoria fisica puntata da `pages`.

5.5.1 Kernel page table

In questa sezione ci poniamo il problema di mappare la zona di memoria virtuale riservata al kernel (gli ultimi 256MB del IV GB di memoria virtuale), sui primi 256 MB di memoria fisica, quindi implementeremo il seguente mapping:

$$[KERNBASE, 2^{32}] \rightarrow [0x00000000, 0xffffffff]$$

. Quest'area è condivisa da ogni processo, il che significa che ogni page directory relativa ad processo dovrà possedere esattamente questa porzione di page directory.

A questo proposito ricordiamo che:

1. ogni Page Table indirizza 1024 pagine;
2. ogni pagina è 4 KB;
3. ogni elemento di Page Directory indirizza 4 KB * 1024 = 4 MB di memoria fisica, quindi abbiamo un elemento di Page Directory ogni 4 MB di memoria fisica;
4. ogni blocco dei 4 MB di memoria fisica, è suddiviso in pagine di 4096 bytes ed ognuna di queste è indirizzata da un elemento di Page Table. Ogni Page Table indirizza 1024 pagine;

Vediamo il codice che iniziativa la porzione di page directory e page table in questione.

```
k=0;
for (i= KERNBASE; i<(2^32); i = i + PTSIZE)
{
     dummy = page_alloc(0); // alloca pagina per Page T.
pgdir0 = &pgdir[PDX((uint32_t *)i)];
*pgdir0 = page2pa(dummy)| PTE_P | PTE_W ;
pt1 = (uint32_t *)page2kva(dummy);
for (j= 0; j<NPTENTRIES; j = j+1)
{
pt1[j] = k | PTE_P |PTE_W; //abilita pagina  lettura/scrittura
k = k + PGSIZE;
}
}
```

$k = 0$; la variabile k, sarà usata nel ciclo interno per individuare gli indirizzi fisici dei frame in memoria centrale a partire dal frame 0.

`for (i = KERNBASE; i < (2^32); i = i + PTSIZE)` Inizializziamo l'indirizzo i a KERNBASE e lo incrementiamo ogni volta di PTSIZE, ovvero di 4MB. Quindi questo è un ciclo for che suddivide lo spazio da KERNBASE a 4 GB in blocchi da 4 MB.
1° blocco/giro → Qual è l'indirizzo logico di riferimento di questo blocco? Questo: `pgdir0 = &pgdir[PDX((uint32\_t *) i];` ovvero prendiamo i (che in questo giro vale KERNBASE) e lo passiamo a PDX() che si occupa di prendere i primi 10 bit di i, ovvero i primi 10 bit di KERNBASE, ovvero la posizione che l'indirizzo KERNBASE occupa nella Page Directory `pgdir` cioè la Page Directory che stiamo costruendo. Quindi `pgdir0` conterrà l'indirizzo dell'elemento di `pgdir` che verrà utilizzato dalla MMU per mappare gli indirizzi virtuali che ricadranno nel range [KERNBASE, KERNBASE+4MB). Dopo di che, cosa mettiamo in questo elemento di Page Directory? Dobbiamo mettere l'indirizzo di una page table che conterrà i 1024 frame fisici su cui è mappato il suddetto spazio virtuale. Page Table che però non abbiamo e che quindi dobbiamo costruire noi. Allora, ci facciamo dare una pagina dal sistema tramite l'allocatore `page_alloc` attraverso l'istruzione `dummy = page\_alloc(0);` ed inseriamo il corrispondente indirizzo fisico di `dummy` in `pgdir0`: `*pgdir0 = page2pa(dummy) | PTE_P | PTE_W;` In questo modo l'elemento di Page Directory corrispondente a KERNBASE punta all'indirizzo fisico di un frame in cui costruiremo la tabella delle

pagine relative allo spazio virtuale [KERNBASE, KERNBASE+4MB). Nella Page Table inseriremo gli indirizzi fisici delle pagine in memoria e più precisamente: 1° Page frame → indirizzo fisico 0 → da 0 a 4095 bytes 2° Page frame → indirizzo fisico 4096 → da 4096 a 8191 bytes 3° Page frame → indirizzo fisico 8192 → da 8192 a ... bytes e cosi via... Di conseguenza come elementi della Page Table che stiamo costruendo, dobbiamo mettere gli indirizzi fisici dei frame che che otteniamo sfruttando la variabile k, più precisamente

```
for (j = 0; j < NPTENTRIES; j = j + 1) {
pt1[j] = k | PTE_P | PTE_W;
k = k + PGSIZE;
}
```

e mettiamo all'interno della Page Table gli elementi che puntano agli indirizzi fisici delle pagine di memoria centrale a partire dalla Page Frame di indirizzi fisico 0.

Difatti:

```
1° giro
j = 0 /* indice della 1° Page Frame nella Page Table */
k = 0 /* indirizzo fisico della 1°  Frame nella Page Table */
/* assegniamo 1° Page Frame della Page Table = indirizzo fisico 0 */
pt1[0] = 0 | PTE\_P | PTE\_W
/* incrementiamo l'indirizzo fisico di 4096 per il prossimo Frame */
k = k + PGSIZE;
2° giro
j = 1 /* indice del 2° Page Frame nella Page Table */
k = 4096 /* indirizzo fisico della 2° Page Frame nella Page Table */
pt1[1] = 4096 | PTE\_P | PTE\_W
/* incrementiamo l'indirizzo fisico di 4096 per il prossimo Page Frame */
k = k + PGSIZE;
.
.
.
1024 ° giro
j = 1023 /* indice del 2° Page Frame nella Page Table */
k = (1024 * 4096) /* indirizzo fisico del 1024°  Frame in Page Table */
pt1[1] = (1024 * 4096) | PTE\_P | PTE\_W
/* incrementiamo l'indirizzo fisico di 4096 per la prossima Page Frame */
k = k + PGSIZE;
```

```
Dopo il 1024 giro, il ciclo for più interno  termina ed in questo modo
abbiamo la prima Page Table e quindi
abbiamo mappato i primi 4 MB di memoria fisica a partire da KERNBASE.
2° blocco/giro

i = i + PTSIZE = i + (PGSIZE * NPTENTRIES) = i + (4096 * 1024) = KERNBASE + 4MB
Costruiamo la seconda Page Table allo stesso modo a partire da KERNBASE + 4 MB, che s
occuperà di mappare le pagine fisiche da 4 MB a 8 MB di memoria centrale.
... e così via.
```

Quindi sostanzialmente abbiamo mappato parte del IV GB della memoria virtuale utilizzato unicamente dal kernel nel 1° primo GB della memoria fisica.

5.6 page_walk

in questa sezione mostriamo ed analizziamo la routine `page_walk()` che dati in input:

1. l'indirizzo di una page directory;
2. un indirizzo lineare `va`;
3. un intero che indica se creare o meno la Page Table in caso di sua assenza.

Verifica se l'indirizzo `va` è mappato dalla page directory ed in caso affermativo restituisce un puntatore all'indirizzo dell'elemento di Page Table che punta a `va`. In caso contrario, alloca, in funzione del valore del terzo parametro, una nuova Page Table, carica il suo indirizzo nella corretta posizione di `pgdir` e restituisce un puntatore all'indirizzo dell'elemento di Page Table appena allocata, che punta all'indirizzo virtuale `va`.

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
struct PageInfo *new_page;
if ((pgdir[PDX(va)] & PTE_P) != 0)
return &(((pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])))[PTX(va)]);
else
   {
if (create == 0)
```

```
return NULL;
else
{  new_page = page_alloc(ALLOC_ZERO);
if (new_page == NULL)
return NULL;
new_page->pp_ref  = 1;
pgdir[PDX(va)] = page2pa (new_page) |PTE_P | PTE_W | PTE_U;
return &(((pte_t *)KADDR(PTE_ADDR(pgdir[PDX(va)])))[PTX(va)]);
}
   }
}
```

Commentiamo la soluzione. Consideriamo i primi dieci bit di `va` (PDX(va)) che usiamo come indice all'interno della Page Directory data, e verifichiamo se in questa posizione di `pgdir` vi è l'indirizzo di una Page Table. `pgdir[PDX(va)]` indica l'elemento di Page Directory relativo all'indirizzo lineare `va`. Usiamo poi il bit di Presenza per verificare se esiste già una page table associata (`if ((pgdir[PDX(va)] & PTE_P) != 0)` Se la condizione è vera ci limitiamo a restituire il puntatore all'elemento di Page Table relativo all'indirizzo lineare `va` ed abbiamo terminato
(`return &(((pte_t *) KADDR(PTE_ADDR(pgdir[PDX(va)])))[PTX[va]);`).
Spieghiamo questa return: `PDX(va)` è l'indirizzo di Page Directory relativa all'indirizzo lineare `va` (primi 10 bit). `pgdir[PDX(va)]` è l'elemento di Page Directory relativo all'indirizzo lineare `va`, cioè l'indirizzo di un elemento di page directory. Con `PTE_ADDR(pgdir[PDX(va)])` dall'elemento di page directory in oggetto estraiamo la parte relativa all'indirizzo fisico della Page table associata. Adesso dobbiamo accedere alla tabella delle pagine perché dobbiamo restituire il puntatore alla tabella delle pagine associato a `va`. Per farlo però dobbiamo trasformare l'indirizzo fisico `PTE_ADDR(pgdir[PDX(va)])` in indirizzo virtuale attraverso la seguente operazione `(pte_t *) KADDR(PTE_ADDR(pgdir[PDX(va)]))`, che ci restituisce l'indirizzo virtuale della tabella delle pagine relativa dell'indirizzo lineare `va`. Però noi dobbiamo restituire all'interno di questa tabella delle pagine, l'indirizzo dell'elemento relativo all'indirizzo `va`, per cui dobbiamo usare gli altri 10 bit di `va`, quindi prendiamo l'indirizzo virtuale della tabella delle pagine `(pte_t *) KADDR(PTE_ADDR(pgdir[PDX(va)]))` e ci indirizziamo all'elemento di posizione `PTA(va)`.

Consideriamo ora il caso in cui l'elemento di page directory associato a `va` non sia presente, quindi caso in cui `va` non sia ancora stato mappato. Se il parametro create è uguale a 0, allora possiamo restituire un puntatore a

NULL perché non è stata richiesta la creazione della Page Table. Altrimenti se `create` è diverso da 0 dobbiamo costruire un mapping per `va`. Come facciamo a costruirlo?

1. Allochiamo una tabella delle pagine in cui inseriremo il nuovo mapping;
2. mettiamo nella Page Directory l'indirizzo di questa tabella delle pagine;
3. restituiamo il puntatore all'elemento di Page Table relativo all'indirizzo lineare `va`.

Allochiamo una nuova pagina con l'istruzione `new_page = page_alloc(ALLOC_ZERO);` e carichiamo il suo indirizzo fisico l'indirizzo fisico all'interno della Page Directory puntata da `pgdir` con settati il bit di Presenza , il bit di lettura e scrittura ed il bit di accesso per gli utenti (ll bit PTE_U è inserito perché serve nel LAB 3), l'istruzione che effettua questa operazione è `pgdir[PDX(va)] = page2pa(new\_page) | PTE\_P | PTE\_W | PTE\_U;`. A questo punto, una volta allocata la pagina della tabella delle pagine e collegata alla Page Directory, la procedura termina il valore di ritorno già precedentemente commentato.

5.7 boot_map_region

Scopo di questa procedura è mappare una regione di indirizzi logici [va, va + SIZE] in una regione di indirizzi fisici [pa, pa + SIZE], all'interno di una data page directory. Più precisamente questa procedura prende come parametri di input:

1. `pgdir`: l'indirizzo logico della page directory in cui costruire il mapping richiesto;
2. l'indirizzo logico di inizio della regione di memoria che vogliamo mappare `va`;
3. la dimensione, multiplo di PGSIZE, della regione di memoria che vogliamo mappare;
4. l'indirizzo fisico su cui vogliamo far mappare l'indirizzo logico `va`;
5. I bit di permesso `perm` alle pagine fisiche che fanno parte del mapping;

Vediamo il codice:

```
static void
boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
size_t index;
if (size & 0xFFF) panic ("boot_map_region: size is not a multiple of PGSIZE");
for (index = 0; index < size; index = index + PGSIZE)
*(pgdir_walk (pgdir, (void *)(va + index), 1))
=  (pa + index) | perm | PTE_P;
}
```

Per prima cosa, verifichiamo con l'istruzione `if (size & 0xFFF)` se la dimensione `size` dello spazio che vogliamo mappare è multiplo di PGSIZE in caso contrario ci fermiamo. Dopo di che, con il ciclo `for (index = 0; index < size; index = index + PGSIZE)` mappiamo lo spazio di memoria virtuale che parte dall'indirizzo virtuale `va`, e per la dimensione di `size` sull'indirizzo fisico che inizia dall'indirizzo `pa`. Per effettuare questo mapping ci avvaliamo della procedura `pgdir_walk()` precedentemente illustrata, che ricordiamo restituisce a partire da una page directory l'indirizzo dell'elemento di Page Table associato ad un determinato indirizzo logico `va`.

Quindi:

```
1 giro
index = 0
Mappiamo l'indirzzo logico  $va + index$ all'indirizzo fisico $pa + index$:
*(pgdir_walk(pgdir, (void *) (va + 0), 1)) = (pa + 0) | perm | PTE_P;
dopo di che incremento index di 4096 byte;
2 giro
index = 4096
Mappiamo l'indirizzo logico va + index all'indirizzo fisico pa + index:
*(pgdir_walk(pgdir, (void *) (va + 4096), 1)) = (pa + 4096) | perm | PTE_P;
dopo di che incremento index di altri 4096 byte;
E continuiamo ad eseguire il mapping fino ad esaurire l'intero spazio.
```

Usando la procedura `boot_map_region()` siamo ora in grande di allocare ed inizializzare gli spazi di indirizzamento necessari al Kernel di JOS. Terminata questa fase dobbiamo sostituire la page directory corrente con la nuova page directory che ospita il mapping di tutte le strutture dati di Kernel.

```
////////////////////////////////////////////////////////////
```

```
 Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory,
// but we just set up the mapping anyway.
// Permissions: kernel RW, user NONE

boot_map_region(kern_pgdir, KERNBASE,
                    (0xFFFFFFFF)-KERNBASE + 1, 0, PTE_W);

////////////////////////////////////////////////////////////
// Map the 'pages' data structure at linear address UPAGES in
// such a way that it is read-only by the user
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE

boot_map_region (kern_pgdir, UPAGES,
ROUNDUP(npages*sizeof(struct PageInfo), PGSIZE), PADDR(pages),
PTE_U);

////////////////////////////////////////////////////
// Map the 'envs' array read-only by the user at linear
// address UENVS
// (ie. perm = PTE_U | PTE_P).
// Permissions:
//    - the new image at UENVS  -- kernel R, user R
//    - envs itself -- kernel RW, user NONE
boot_map_region (kern_pgdir, UENVS,
ROUNDUP(NENV*sizeof(struct Env), PGSIZE), PADDR(envs),
PTE_U);

//////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to
// as the kernel stack.  The kernel stack grows down
// from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE,
// KSTACKTOP) to be the kernel stack, but break this
// into two pieces:
```

```
//  * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by
//  physical memory
//  * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not
// backed; so if the kernel overflows its stack, it
// will fault rather than overwrite memory.  Known as a "guard
// page".  Permissions: kernel RW, user NONE

boot_map_region (kern_pgdir,
KSTACKTOP-KSTKSIZE, KSTKSIZE, PADDR(bootstack), PTE_W);
```

Per modificare la Page Directory di riferimento è sufficiente caricare nel registro CR3 l'indirizzo fisico della nuova Page Directory il cui indirizzo è dato da `PADDR(kern_pgdir)` tramite l'istruzione: `lcr3(PADDR(kern_pgdir);`. A questo punto la Page Directory e le relative Page Table che abbiamo precedentemente costruito diventano operative e la MMU e quindi il sistema inizierà a lavorare con la nuova Page Directory.