

Contents

Sistemi Operativi Avanzati	4
Hardware Insights	4
Introduzione	4
Scheduling e parallelismo nell'architettura di Von Newman	5
Velocità di computazione	6
Pipeline	6
Pipeline vs Sviluppo	8
Pipeline superscalare	9
Algoritmo di Robert Tomasulo	10
Dipendenza RAW	10
Dipendenza WAW e WAR	10
Architettura di riferimento di Tomasulo	11
Esempi di schemi di esecuzione	12
Organizzazione architettonica dei processori x86 OOO	13
Processori hyper-threaded	13
Gestione degli interrupt	15
Gestione delle eccezioni	15
Attacco Meltdown	15
Insights sui processori x86 e x86-64	16
Codice assembly dell'attacco Meltdown	18
Contromisure per l'attacco Meltdown	18
Insights sui branch	20
Predittori per i salti condizionali	20
Tournament Predictor	22
Predittore per salti indiretti	22
Attacco Spectre	23
Spectre V1	23
Spectre V2	23
Contromisure per gli attacchi Spectre	24
Retpoline - Return trampoline	24
Esistono altre tecniche per Spectre V2?	25
IBRS (Indirect Branch Restricted Speculation)	25
IBPB (Indirect Branch Prediction Barrier)	25
Sanitizzazione	25
Introduzione	25
Come funziona	25
Come funziona - for dummies	26
Loop unrolling	26
Power wall	26
Symmetric Multiprocessors	27
Chip Multi Processor (CMP) o Multicore	27
Symmetric Multi Threading (SMT) o Hyperthreading	27
Non Uniform Memory Access (NUMA):	28
Cache Coherency	28
Protocolli CC Cache Coherency	29
Protocollo MSI (Modified-Shared-Invalid)	30
Protocollo MESI (Modified-Exclusive-Shared-Invalid):	30
Protocollo MOESI (Modified-Owned-Exclusive-Shared-Invalid):	31
Protocolli directory based:	32
Implementazioni x86	33
False cache sharing	33
Attacco Flush + Reload	33
ASM inline	35
Direttive di compilazione C per gli operandi	35
Possibili operandi per l'ASM	35
Esempio di utilizzo di ASM inline	35

Memory Consistency	37
Consistenza sequenziale	37
Definizione di Lamport, 1979	37
Total Store Order TSO	37
Memory Fencing	38
Linearizzabilità	39
Linearizzabilità vs operazioni RMW	40
Non-blocking coordination	40
Sincronizzazione lock-free	41
Sincronizzazione wait-free	43
Read Copy Update RCU	44
Funzionamento	44
Non-preemptable RCU vs preemptable RCU	45
Vettorizzazione	45
Vettorizzazione esplicita ed implicita	46
	47
Kernel Programming Basics	47
Indirizzamento della memoria – come vedo la memoria?	47
Supporti alla segmentazione	48
Segment selector	48
Accesso alla memoria nei sistemi x86	49
Real mode	49
80386 protected mode	49
Long mode	49
Tabelle di segmento	50
Global Descriptor Table	50
Local Descriptor Table	50
Segmentazione e paginazione	50
Modello di protezione basato su segmentazione	50
Composizione degli indirizzi x86 con la segmentazione	52
GDT Entries	52
80386 protected mode	53
Long mode	53
Accesso alle GDT entries	53
Schema di accesso alle GDT entries, livello hardware	54
Segmenti code/data in Linux	54
Regole di aggiornamento dei segment selector	55
Quali sono le GDT entries in un sistema Linux x86?	55
TSS Task State Segment	56
LDR Load Task Register	57
Replicazione della GDT	57
Esempio di utilizzo della per-CPU memory:	58
TLS Thread Local Storage	58
arch_prctl	58
Registri di controllo in x86_64	59
Interrupt e trap	59
Dispatching delle system call	60
Atomicità dell'esecuzione del dispatcher	61
Componenti sw per la gestione delle system call	61
Indicizzazione delle system call	62
Task di livello USER per accedere al GATE	62
Formato predeterminato per le system call	62
Convenzioni per l'invocazione delle syscall	64
Convenzione di UNISTD_32 per l'invocazione delle system call	64
Convenzione di UNISTD_32 per l'invocazione delle system call	65
Dettagli sui passaggi dei parametri	66
Esempi di creazione di system call in UNISTD_32	66

Esempio di override della fork	67
Implicazioni dell'uso di int 0x80	67
Fast system call path	67
Sysenter 32 bit	67
Syscall 64 bit	68
Sysexit 32 bit	68
Sysret 64 bit	68
Aggiornamento dei registri MSR	68
Costrutto syscall()	68
Esempio echo	69
Esempio stub system call	69
Virtual Dynamic Shared Object VDSO	69
Definizione della system call table	70
Costrutto asmlinkage	70
Dispatcher con INT0x80, kernel 2.4, UNISTD32	70
Dispatcher con SYSCALL, kernel 2.4, UNISTD64	71
Implementazione del dispatcher in kernel 4	72
Implementazione del dispatcher nelle versioni più recenti del kernel	73
Attacco swapgs	74
Contromisure	74
Compilazione del kernel	74
System map	75
Startup del kernel	75
Task dello startup	76
Bios - Basic I/O System	76
UEFI - Unified Extended Firmware Interface	76
Task BIOS/UEFI durante il kernel del SO	77
Startup del kernel in macchine multicore	77
Kernel Level Memory Management	77
Caricamento del kernel in memoria fisica	77
Caso base - indirizzi non randomizzati	78
Avvio della paginazione durante lo startup	79
Patch Meltdown hardware	79
Ram durante lo startup	79
Funzioni <u>__init</u>	80
Reachable Page	80
Organizzazione della RAM nelle macchine moderne	80
Numactl	81
Memblock	81
Strutture dati per la gestione della memoria	81
Kernel page table	82
Pagine di memoria directly mapped	82
Zone	82
Caso di Linux nei sistemi x86_64 long mode	83
Organizzazione kernel e struttura page table in i386	83
Organizzazione del kernel in i386	83
Struttura page table in i386	84
Paginazione nei sistemi i386 in Linux	85
Entry della PDE nei sistemi i386	86
Entry della PTE nei sistemi i386	87
Relazione tra page table ed eventi di trap/interrupt	87
Algoritmo di inizializzazione delle page table in i386 nel kernel 2.4	88
PAE Physical Address Extension	88
Architetture x86-64	89
Page table nelle architetture x86-64	89
Direct mapping vs non-direct mapping in x86-64	91
Huge Pages	91
Supporto hardware alla virtual memory	91

Attacco L1 Terminal Fault L1TF	92
Core map	94
Free list	94
Buddy Allocator	95
Schema associato ad uno specifico nodo NUMA	96
Caso di kernel NUMA-aware	96
Contesti di allocazione	97
Api del buddy system	97
Caso di allocazioni/deallocazioni frequenti	98
Quicklist	99
SLAB / SLUB allocator	99
Slab allocator virtuali	99
Slab coloring	100
Allocazioni di aree di memoria molto ampie	100
Effetti sul TLB, interazione con hardware	100
Operazioni implicite vs esplicite sul TLB	101
Tipi principali di eventi sul TLB	102
Costi del flush del TLB	102
Flush globale del TLB su Linux	102
Flush parziale del TLB su Linux	103
Cross Ring Data Move	103
Introduzione	103
Caso della segmentazione flessibile	104
Caso della segmentazione constrained	104
Esempio di utilizzo di <code>addr_limit</code>	105
API kernel per lo user/data move	105
Service redundancy	106
Constrained supervisor mode	106
Constrained supervisor mode in x86	106
Kernel masked SEGFAULT	106
Linux Modules p111	107

Sistemi Operativi Avanzati

- Versione originale: Matteo Fanfarillo
- Umile formattazione: Simone Festa

Hardware Insights

Introduzione

In generale, non è possibile dire che lo stato di un'applicazione sia semplicemente dato dal “puzzle” degli stati dei componenti software sviluppati. Di fatto, quando mandiamo in esercizio tali componenti software, stiamo generando dei cambi di stato al livello hardware che concorrono a determinare qual è lo stato effettivo del nostro sistema. Tra l'altro, alcuni dei cambi di stato al livello hardware possono non essere voluti o specificati dal programmatore. Il fatto che l'esecuzione di moduli software sviluppati a qualsiasi livello impatti sui moduli sottostanti (e quindi sull'hardware), come vedremo, può rappresentare un problema importante per quanto riguarda la sicurezza: talvolta, per ottenere un sistema più sicuro e performante a livello hardware, sarà necessario ristrutturare l'applicazione software.

Ma quali sono le entità che si frappongono tra ciò che viene specificato dal programmatore e ciò che realmente avviene nel sistema?

- Il **compilatore**: il programma compilato è un oggetto molto complesso che può interfacciarsi direttamente con l'hardware; il compilatore può decidere ad esempio di inserire particolari istruzioni macchina secondo uno specifico ordine in modo completamente trasparente al programmatore.

- Le **hardware run-time decisions**: una volta che è stato generato il flusso esatto delle istruzioni macchina da eseguire per mandare in esercizio l'applicazione, l'hardware in realtà può prendere delle decisioni a run-time su come gestire tale flusso. A parità di istruzioni macchina, processori di vendor diversi possono prendere delle decisioni a run-time differenti.
- La **disponibilità** (o l'assenza) di specifiche features dell'hardware.

In pratica, si ha una sorta di non-determinismo dell'hardware e, quindi, del software.

Esempio: Se implementiamo l'algoritmo del Panificio di Lamport senza l'ausilio di librerie di sistema (e quindi senza l'ausilio di spinlock o semafori), l'algoritmo a un certo punto della sua esecuzione si romperà. Infatti, le macchine moderne (a meno che non siano single core, dove non è assicurata consistenza globale) non garantiscono una visione consistente per tutti i thread di ciò che sta succedendo in memoria. Prima, ad ogni cpu veniva associato un flusso di esecuzione, cosa non vera per i sistemi moderni.

Scheduling e parallelismo nell'architettura di Von Newman

L'architettura di calcolatore più semplice a cui siamo stati abituati a pensare è quella di **Von Newman**, ed è caratterizzata da:

- Un'unica CPU.
- Un'unica memoria.
- Un unico flusso di controllo (*fetch – execute – store*).
- Transizioni nell'hardware time-separated: le istruzioni devono essere eseguite tutte una alla volta.
- Stato della memoria ben definito all'inizio di ciascuna istruzione, la quale quindi deve poter vedere la memoria in uno stato coerente con l'esecuzione delle istruzioni precedenti.

La maniera moderna di pensare le architetture non è basata sull'idea di seguire il flusso di *esecuzione esattamente così com'è* stato codificato nel programma, bensì è basata sul concetto di **scheduling** (e.g. dell'utilizzo dei componenti hardware) che, alla fine della fiera, porterà a eseguire un flusso di esecuzione equivalente al flusso codificato nel programma (gli stati intermedi possono essere diversi, possono cambiare di run in run, non sono deterministicici, ma influenzabili da fattori esterni come la temperatura). In particolare, lo scheduling dovrebbe consentire di eseguire più cose in **parallelo**, anche in contesti con un'unica CPU, un'unica memoria e un unico flusso di controllo. Per quanto riguarda lo scheduling, ne esistono diverse tipologie. A livello **hardware** abbiamo:

- Scheduling delle istruzioni all'interno di un singolo program flow.
- Scheduling delle istruzioni in program flow paralleli (**speculativi** = non so se l'esecuzione sia corretta o meno, ad esempio la branch condition non sempre lo è. Il processore sceglie come propagare l'update, non è imposto. Ho garantita l'equivalenza software, non ciò che avviene dentro).
- Propagazione dei valori tra i componenti hardware del sistema.

A livello software invece abbiamo:

- Scheduling dei thread da assegnare alle CPU / ai CPU-core.
- Scheduling delle attività da eseguire sull'hardware (e.g. gli interrupt).
- Supporti di sincronizzazione tra thread software-based.

Anche per quanto riguarda il **parallelismo**, ne esistono diverse tipologie:

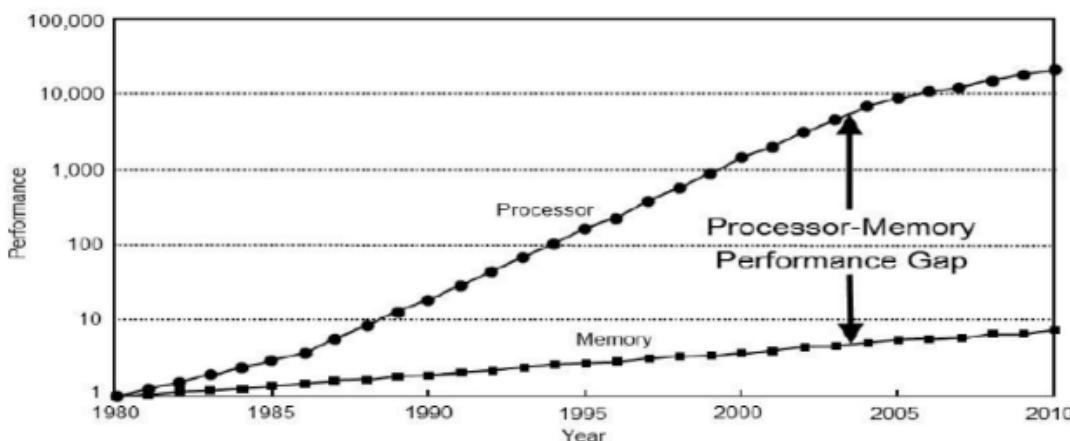
- A livello hardware si parla di **ILP (Instruction Level Parallelism)**, che consiste nell'impiegare le risorse hardware in modo tale da eseguire contemporaneamente istruzioni macchina diverse. (ad esempio posso eseguire al tempo 't' sia A sia B, anche in un unico flusso).
- A livello software, invece, abbiamo il **TLP (Thread Level Parallelism)**, secondo cui un programma può essere pensato come la combinazione di molteplici flussi di esecuzione concorrenti. (Ad esempio ho 3 flussi su 3 processori, singolarmente sarebbero ILP, che cambiano).

Velocità di computazione

È generalmente correlata alla velocità di un processore (espressa in GHz), anche se in realtà esistono istruzioni che possono richiedere un numero arbitrario di cicli di clock a causa di più possibili fattori:

- Possono essere istruzioni più onerose per loro natura.
- Possono dover richiedere a un certo punto una risorsa hardware tuttora occupata da un'altra istruzione, per cui devono rimanere in attesa.
- Possono esservi delle asimmetrie a livello hardware (e.g. un CPU-core può essere più veloce di un altro).
- I pattern per l'accesso ai dati influiscono a loro volta sulle prestazioni: ad esempio, se un dato viene memorizzato in cache, l'accesso a esso sarà più efficiente e viceversa.

Nel corso base di Sistemi Operativi abbiamo parlato di thread CPU-bound e di thread I/O-bound; introduciamo ora una terza categoria di thread (che, di fatto, è una sottocategoria dei CPU-bound): i **memory-bound**. Essi sono dei thread che utilizzano in maniera intensiva la CPU ma, mentre sono in esecuzione, utilizzano in maniera intensiva anche la memoria. I thread (o comunque i programmi) che presentano questa caratteristica possono rappresentare un problema dal punto di vista prestazionale: come riportato dal seguente grafico, il divario prestazionale tra processore e memoria aumenta sempre di più col tempo; questo fenomeno è detto **memory wall**.



Per evitare che i thread memory-bound sperimentino e causino ad altri thread un crollo delle prestazioni, sono necessari dei meccanismi avanzati ad-hoc al livello dell'hardware.

Pipeline

È una **tecnica di scheduling e parallelismo hardware-based**. Infatti:

- Non prevede una separazione temporale tra le finestre di esecuzione delle diverse istruzioni: più istruzioni possono essere in esecuzione contemporaneamente (questo è *parallelismo*).
- Le istruzioni che vengono sequenzializzate dal programmatore non sono necessariamente eseguite secondo la stessa sequenza nell'hardware, anche per conseguire la proprietà di parallelismo (questo è *scheduling*).

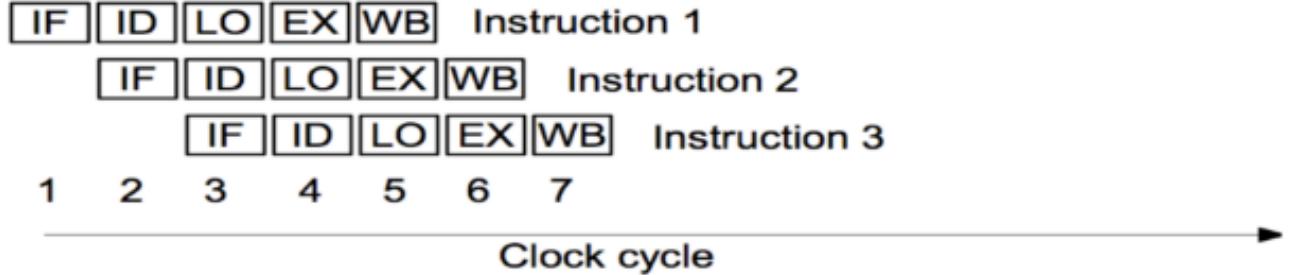
In ogni caso, ci sono spesso e volentieri coppie di istruzioni (i_1, i_2) in cui i_2 , per essere eseguita, necessita del risultato ottenuto da i_1 . In tal caso, la *causalità deve essere preservata*, per cui i_1 e i_2 non possono essere schedulate in modo del tutto arbitrario. Questo non è altro che un modello **data flow** per l'esecuzione dei programmi.

Ricordiamo che le fasi (stage) delle istruzioni sono:

- **IF (Instruction Fetch)**: caricamento dell'istruzione nel processore (richiede certamente un accesso in memoria).
- **ID (Instruction Decode)**: decodifica dell'istruzione, in cui viene stabilito ciò che deve effettivamente essere fatto per eseguire l'istruzione stessa.
- **LO (Load Operands)**: caricamento degli operandi richiesti per l'esecuzione dell'istruzione (potrebbe richiedere un accesso in memoria).
- **EX (Execute)**: esecuzione vera e propria dell'istruzione.

- **WB (Write Back)**: scrittura dell'output dell'istruzione su un registro o in memoria (potrebbe quindi richiedere un accesso in memoria).

Il fatto che un thread o un'applicazione sia memory-bound o meno dipende proprio dagli stage *LO* e *WB*. Inoltre, per permettere l'esecuzione di più istruzioni in parallelo, è necessario che ciascuno *stage coinvolga una componente hardware differente del processore*, in modo tale da non avere collisioni tra più istruzioni che vengono eseguite contemporaneamente. In particolare, un'astrazione di base della pipeline è quella riportata nella seguente figura.



In questo scenario, idealmente, sarebbe possibile completare un'istruzione per ogni ciclo di clock (anche se poi nella realtà non è esattamente così per i motivi esposti nel paragrafo “Velocità di computazione”). Supponiamo comunque di trovarci nello scenario ideale in cui ciascuno stage viene eseguito in un unico ciclo di clock, e supponiamo di voler fornire N risultati (1 per ogni istruzione), di avere L diversi stage per le istruzioni (gli stage sarebbero *IF*, *ID*, *LO*, *EX* e *WB* che devo attraversare) e di avere un ciclo di clock di durata pari a T .

- Senza pipeline si ha un ritardo pari a $N \cdot L \cdot T$
- Con la pipeline si ha un ritardo pari a $(N + L) \cdot T$

Lo speedup è dunque pari a $(N \cdot L)/(N + L)$, che tende a L per N tendente a infinito. Dal punto di vista delle prestazioni, sarebbe magnifico avere un L molto grande (ovvero molti stage diversi per le istruzioni, ovvero riesco a parallelizzare molto di più se aumento il numero di componenti parallelizzabili) ma ciò nella realtà non accade. Il caso estremo richiederebbe avere hardware infinito, visto che ogni stage è un pezzo hw! Nella realtà:

- I processori Pentium prevedevano 5 stage.
- I processori i3 / i5 / i7 prevedono 14 stage.
- I processori ARM-11 prevedono 8 stage.

Questa scelta è dovuta alla necessità di preservare la causalità tra le istruzioni. Più istruzioni si prendono in considerazione insieme, più è probabile che tra tali istruzioni ce ne siano alcune (e.g. i_1, i_2) legate da una relazione di causalità, e sappiamo che i_2 , per essere eseguita, deve attendere che il risultato di i_1 sia pronto; in tal caso, più L è grande, più la pipeline è lunga, più l'attesa di i_2 sarà lunga. Lo scenario di cui abbiamo appena parlato è una **data dependency**. Oltre a questa esiste anche la **control dependency**, che si può avere nel momento in cui c'è un salto condizionale il cui esito dipende dagli outcome delle istruzioni precedenti al salto. Per quanto concerne la data dependency tra le istruzioni i_1 e i_2 , abbiamo che lo stage **LO** di i_2 tipicamente non può precedere lo stage **WB** di i_1 . Analogamente, per quanto riguarda la control dependency tra le istruzioni i_1 e i_2 (dove i_1 è l'istruzione di salto condizionato), non si conosce l'esito del salto prima della fase **EX*** di i_1 , per cui la fetch di i_2 non dovrebbe precedere lo stage **EX*** di i_1 . Tutte queste condizioni possono comportare dei rallentamenti nell'esecuzione dell'applicazione rispetto al caso ideale di pipeline. Per gestire tali condizioni è possibile ricorrere a svariati meccanismi:

- **Stalli software** (compiler driven): possono essere aggiunti all'interno della pipeline con lo scopo di distanziare due istruzioni i_1, i_2 in modo tale che uno stage x (e.g. **LO**) di i_2 venga eseguito dopo uno stage y (e.g. **WB**) di i_1 .
- **Rischedulazione software** (compiler driven): se devono essere eseguite tre istruzioni i_1, i_2, i_3 tali per cui i_2 dipende da i_1 ma i_3 non dipende né da i_1 né da i_2 , il compilatore può frapporre i_3 tra i_1 e i_2 in modo tale da svolgere lavoro utile mentre i_2 è in attesa che si completi uno specifico stage di i_1 .
- **Propagazione hardware**: se l'istruzione i_2 dipende dall'istruzione i_1 e, ad esempio, lo stage **LO** di i_2 consiste nell'acquisire un valore prodotto dallo stage **EX** di i_1 , allora è possibile per i_1 propagare il valore a i_2 subito dopo la fase **EX** senza dover attendere il completamento della write-back.
- **Azzardi hardware supported**: contestualmente a un'istruzione di salto condizionale, il processore può provare a *indovinare* se il salto viene preso o meno per poi anticipare la fetch delle istruzioni successive di conseguenza. Dal punto di vista delle performance, una predizione errata equivale a un inserimento di stalli per attendere

passivamente l'esito dell'istruzione di salto; di conseguenza, la tecnica degli azzardi è statisticamente conveniente da adottare.

- **Rischedulazione hardware:** è un meccanismo noto anche come **out-of-order pipeline (OOO)**, e prevede che le istruzioni vengano completate non necessariamente nel medesimo ordine con cui sono entrate all'interno della pipeline. In particolare, un'istruzione i_k , per accedere allo stage x , non deve necessariamente attendere che tutte le istruzioni a lei precedenti abbiano completato lo stage x . Si può dunque avere un meccanismo di **superamento** delle istruzioni all'interno della pipeline. Tale tecnica è vantaggiosa poiché permette all'istruzione i_k di essere completata prima e, quindi, di liberare un posto all'interno della pipeline. Tuttavia, è una tecnica che *richiede la possibilità da parte dell'hardware di ospitare più istruzioni contemporaneamente all'interno dello stesso stage* (altrimenti non sarebbe possibile effettuare il sorpasso): i processori con questa caratteristica sono detti **superscalari**. Inoltre, ovviamente, affinché si possa avere una out-of-order pipeline, *l'istruzione che effettua il sorpasso non deve dipendere dalle istruzioni che vengono sorpassate*. Non si hanno effetti reali sull'**Instruction Set Architecture** finché non si deve "mostrare" l'output.

Esempio:

Se in pipeline su un thread ho $A \rightarrow B$, e nella pipeline B supera A, ma A è oggetto di *trap* (quindi non può fare quello che stava facendo, come dividere per 0, di conseguenza non potrà averla in ISA), cosa accade a B? Vedremo che in OOO si producono valori registrati in maniera "speculativa" che poi butterò, ma non posso eseguire una UNDO.

Nota: le istruzioni tra due thread sono indipendenti, dipendono solo se usano info condivise, ma ciò è di interesse al programmatore, non al processore. L'ordine della sorgente (programma) non è detto coincida con l'ordine del compilatore, tuttavia abbiamo la sicurezza che il data flow sia compatibile. A livello hardware, se ho operazione x, y, z può capitare quindi di avere z, x, y ; ma ciò è realizzato dinamicamente dall'hardware. Ho sorpasso se non ho dipendenza.

Pipeline vs Sviluppo

Come abbiamo visto, i programmatori non hanno il diretto controllo del comportamento di un processore (trattasi di microcodice) ma, comunque sia, il modo con cui viene scritto il software può impattare sulle prestazioni effettive della pipeline. A livello ISA vedo il set di istruzioni e risorse usabili, ma non vedo la pipeline. **Esempio:** Esempi di ISA sono *ADD*, *COMPARE*, *JUMP*. Consideriamo le seguenti istruzioni C:

- 1) $a = *++p$
- 2) $a = *p++$

L'istruzione 1 prevede che prima debba essere incrementato il puntatore p affinché referenzi la entry successiva e solo dopo si possa accedere alla entry appena referenziata; di conseguenza, l'accesso dipende dall'incremento del puntatore. Al contrario, l'istruzione 2 prevede che prima si debba accedere alla entry attualmente referenziata dal puntatore p e solo poi si debba incrementare p; stavolta, l'accesso non dipende dall'incremento del puntatore. Questo vuol dire che c'è dipendenza. Consideriamo due programmi, di cui il primo prevede l'istruzione 1 all'interno di un loop e il secondo prevede l'istruzione 2 all'interno di un loop. Nel secondo c'è indipendenza. La differenza prestazionale tra i due programmi è abbastanza significativa (può raggiungere tranquillamente il 20/25%).

Per giunta, esistono delle istruzioni macchina che, da sole, hanno degli effetti devastanti sulle prestazioni del programma. Un esempio è **cpuid**, che ha lo scopo di restituire l'id del processore (o del CPU-core o dell'hyperthread) che ha processato l'istruzione stessa. Ma, oltre a questo, effettua anche lo **squash** della pipeline: in particolare, nel momento in cui **cpuid** viene realmente eseguita, la pipeline viene svuotata e le altre istruzioni al suo interno vengono buttate. Questo non deve necessariamente rappresentare uno svantaggio: avere istruzioni pendenti all'interno della pipeline significa dire che tali istruzioni possono essere schedulate dinamicamente dall'hardware secondo regole non meglio identificate, e ciò può impattare non solo sulla correttezza, ma anche sulla sicurezza del sistema. Più precisamente, quello di flushare la pipeline è un concetto legato al termine "**serializzazione**". Di fatto, **cpuid** è detta **istruzione serializzante**, poiché riporta la pipeline a lavorare secondo uno schema sequenziale. Non solo: **cpuid**, come tutte le istruzioni serializzanti, garantisce che qualunque modifica apportata a flag, registri e memoria da parte delle istruzioni precedenti sia completata (finalizzata) **prima** che una qualsiasi istruzione a lei successiva venga fetchata ed eseguita. *Ciò implica anche che cpuid non può superare alcuna istruzione davanti a lei nella pipeline.* (e.g: Perchè le istruzioni successive devono ancora essere fetchate ed eseguite, quindi come potrei superarle? Ciò che viene dopo questa istruzione serializzante è come se andasse in stallo finché non completo questa istruzione serializzante, e garantisce anche che le istruzioni precedenti vengano viste da quelle successive.)

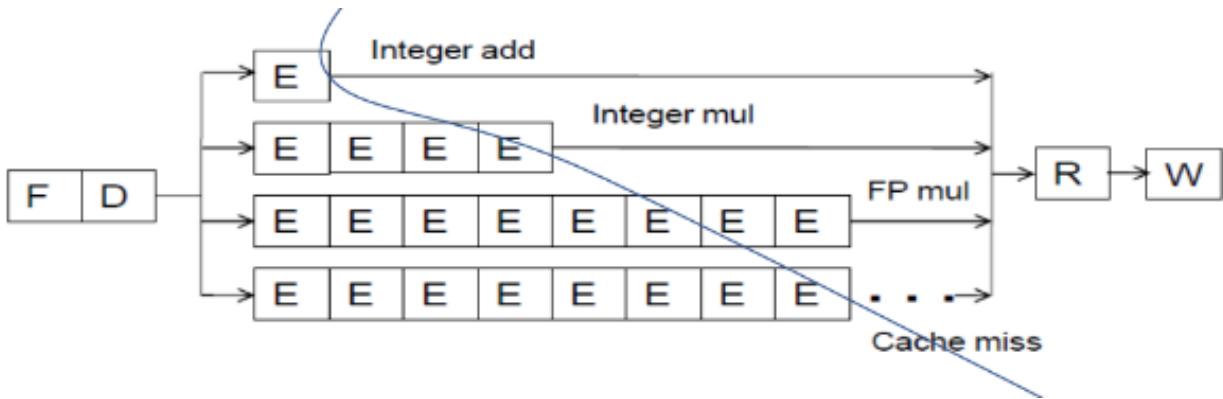
Se avessi SOLO istruzioni serializzanti, il sistema sarebbe molto più lento.

Pipeline superscalare

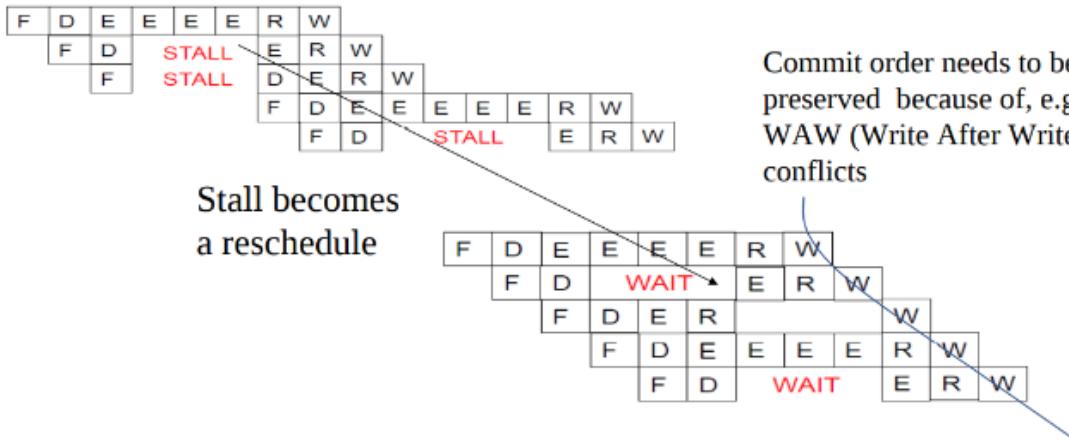
È un insieme di molteplici pipeline che operano simultaneamente all'interno del processore. La si può ottenere aggiungendo *ridondanza alle risorse hardware* in modo tale che più componenti distinti siano adibite a uno stesso stage della pipeline. Come detto in precedenza, questa possibilità permette anche di adottare il modello OOO, la cui idea di base consiste in:

- Effettuare il **commit** (o **retire** o **finalizzazione**) delle istruzioni esattamente nel medesimo ordine in cui sono entrate nella pipeline, indipendentemente dagli eventuali sorpassi avvenuti all'interno della pipeline. Ciò significa che le scritture dei risultati delle istruzioni in memoria, su un registro qualunque o su un registro di stato *devono avvenire esattamente nell'ordine prestabilito*.
- **Processare le istruzioni indipendenti** (sia sui dati che sulle risorse hardware) il **prima possibile**, dove un'istruzione indipendente sulle risorse hardware è un'istruzione che non ha bisogno di attendere che un qualche componente si liberi per processare un certo stage x .

L'immagine riportata di seguito mostra molto bene come lo stage EX delle istruzioni possa avere una durata variabile in termini di numero di cicli di clock, ed è a partire da tale presupposto che risulta utile avere un out-of-order pipeline.



La seguente altra figura mostra invece la differenza prestazionale che si può avere tra il modello non OOO e il modello OOO:



: Con una pipeline che ha ridondanza hardware (come i pc moderni), possiamo ad esempio parlare di “*Pipeline parallela a X canali*”, che è come avere una autostrada a X corsie. Con Out-Of-Order Pipeline si evitano “grosse latenze”, poiché se “libero una corsia perché sono veloce” anche gli altri andranno più veloci (liberando risorse utili ad altri). Lo stallo impatta su tutti invece. Nella O.O.O, se posso andare avanti, lo faccio, non mi importa che devo aspettare il commit, se posso eseguire qualcosa lo eseguo. Per questo parliamo di WAIT e non più di STALL. Nella WAIT, appena ho un risultato (outcome di una “corsia”) lo fornisco a chi lo necessita, non mi serve aspettare che si liberi tutta la corsia.

Ora, poiché tra l'**emission** (= iniezione all'interno della pipeline, normalmente di più di una istruzione) e il **retire** (= **commit**) delle istruzioni si ha una fase di esecuzione in cui le istruzioni stesse possono sorpassarsi a vicenda, si va

incontro al cosiddetto problema delle **eccezioni imprecise (OFFENDING)**: supponiamo di avere un'istruzione i_2 che ha superato un'istruzione i_1 , e assumiamo che i_1 , durante lo stage EX, generi un'eccezione (*e.g.* perché magari si tratta di una divisione per 0); a questo punto, anche se il risultato di i_2 non è stato ancora committato e, quindi, esposto a livello di ISA, i_2 può aver comunque toccato delle risorse non esposte a livello di ISA ed effettuato dunque dei cambi di stato (in particolare cambi di **stato micro-architetturale**). Quindi con offending instructions non esponiamo su ISA, ma potrei cambiare lo stato hardware, il che è usabile da malintenzionati. Questo perchè, lavorare in un contesto Out-Of-Order permette il superamento di istruzioni (**SPECULAZIONE**) che lascia delle tracce. Di conseguenza, l'eccezione sollevata da i_1 vede uno stato interno dell'hardware che ingloba anche delle attività relative a i_2 . Un problema analogo lo si ha anche a parti invertite: se l'istruzione che genera l'eccezione è i_2 , l'eccezione vedrà uno stato interno dell'hardware che non comprende il risultato prodotto da i_1 . Questo è un problema dello stato e delle informazioni all'interno del processore (*ma non esposte nell'ISA*). Peggio ancora: i_2 potrebbe sollevare un'eccezione che in realtà, secondo il program flow, non sarebbe dovuta mai esistere, magari perché i_1 è a sua volta un'istruzione che solleva un'eccezione o un'istruzione di salto. In realtà, si possono avere eccezioni imprecise anche in assenza di sorpassi: se l'istruzione i_2 viene dopo l'istruzione i_1 che genera un'eccezione, i_2 può aver comunque attraversato degli stage e, quindi, può aver modificato lo stato interno dell'hardware. Come vedremo, tutto questo rappresenta un grave problema per la sicurezza dei sistemi: **Meltdown** è solo il primo di una valanga di attacchi che hanno sfruttato tale vulnerabilità.

Algoritmo di Robert Tomasulo

Secondo Robert Tomasulo, in uno scenario di utilizzo di una pipeline speculativa out-of-order (dove speculativa = caratterizzata dall'esecuzione di istruzioni che potrebbero servire solo in un secondo momento), se consideriamo due istruzioni A, B tali che $A \rightarrow B$ nell'ordine di programma, dobbiamo stare attenti nell'evitare i seguenti tre tipi di azzardo:

- **RAW (Read After Write)**: B deve leggere un dato R necessariamente dopo che A lo ha aggiornato.
- **WAW (Write After Write)**: B deve scrivere su un dato R necessariamente dopo che A lo ha aggiornato.
- **WAR (Write After Read)**: B deve scrivere su un dato R necessariamente dopo che A ne ha letto il valore precedente (altrimenti vorrebbe dire che A è in grado di leggere dal futuro).

Dipendenza RAW

Qui è necessario bloccare l'istruzione B e tenere traccia di quando il dato che deve essere letto da B sarà disponibile (disponibile non vuole dire necessariamente committato).

Dipendenza WAW e WAR

Qui è necessario adottare una tecnica nota come **register renaming**, secondo cui al programmatore vengono esposti dei registri logici, e ciascuno di questi registri logici (che sono di fatto dei multi-registri) ingloba un insieme di registri fisici non visibili al livello dell'ISA. Quindi noi non scriviamo MAI sul registro esposto in ISA, ma su delle '*copie numerate*' o alias, cioè registri multivalore. Ci ritroviamo dunque nella seguente situazione:



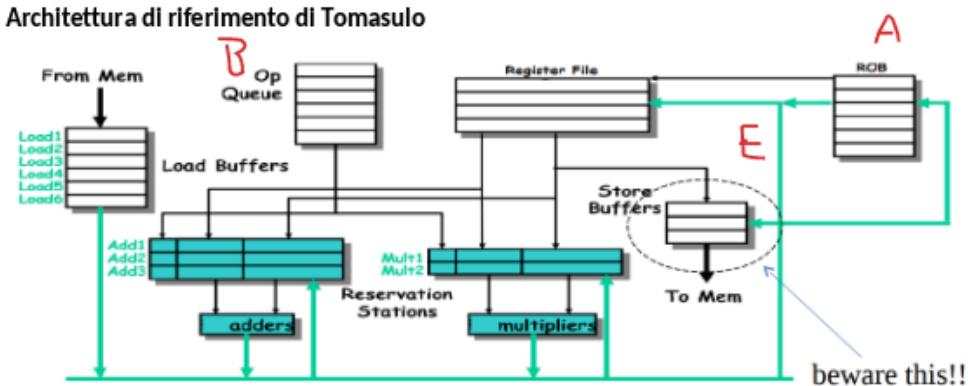
Qui i vari registri fisici rappresentano diverse versioni del medesimo registro logico R . Nel momento in cui all'interno della pipeline entra un'istruzione che vuole scrivere sul registro logico R , si considera il primo registro fisico R_k all'interno del quale viene effettivamente memorizzato il valore (quindi scrivo sempre sul primo tag libero, e leggo dall'ultimo tag in pipeline. Se A scrive in TAG 0, e B scrive in TAG 1, leggo da TAG1). Tipicamente, per la selezione del registro fisico, si segue un approccio round-robin; se però a un certo punto tutti i registri fisici di R sono stati sovrascritti e nessuna delle istruzioni che ha eseguito la scrittura è andata in commit, per un'eventuale altra scrittura su R bisognerà attendere.

Posso superare solo dopo essere entrato nella pipeline (nelle condizioni discusse nelle pagine precedenti), ma non posso superare prima di entrare in pipeline.

In pratica, un *renamed register* materializza il concetto di speculatività di una pipeline: quando vengono effettuate delle write, vengono scritti dei valori che non necessariamente verranno considerati come validi, ma che comunque potranno essere letti dalle istruzioni successive (che sono state introdotte nella pipeline speculativamente).

Relativamente al registro *R*, vengono memorizzati anche dei metadati di gestione di *R* che indicano qual è il tag contenente la versione committata; la versione committata è la versione del valore di *R* scritto dall'ultima istruzione andata in commit che ha toccato proprio *R*. In questi registri abbiamo il “futuro” che avverrà, perché non li ho ancora committati.

Architettura di riferimento di Tomasulo



- **A)**: I metadati associati alle istruzioni fetchate vengono memorizzati all'interno del **ROB** (Re-Order Buffer), che ci aiuta a scrivere le istruzioni nello store buffer, in quanto posso andare in memoria solo se sono committed, ma io non posso aspettare questa fase prima di utilizzare i dati. Tali metadati possono essere:

- *L'ordine con cui le istruzioni dovranno andare in commit.*
- Che cosa dovrebbero fare le istruzioni nella loro esecuzione speculativa.
- Qual è l'alias (l'istanza fisica) dei registri che dovranno essere usati da ciascuna istruzione; ad esempio, se un'istruzione A deve scrivere su un certo registro R e un'istruzione B successiva deve leggere dallo stesso registro R con una dipendenza RAW, è chiaro che A e B dovranno utilizzare il medesimo alias.
- **B)**: Le operazioni vere e proprie da eseguire (quindi gli OP code delle istruzioni) vengono memorizzate all'interno dell'**OP queue** e, in base alla loro tipologia, verranno date in input a un particolare componente di processamento (add, mult e così via). Nel momento in cui un'istruzione è pronta per essere processata, devono essere recuperati i metadati associati a essa e, in particolare, gli alias dei registri da usare. A tale scopo, c'è un bus comune (detto **Common Data Bus – CDB**) che mette in comunicazione i componenti di processamento col ROB. Se un alias non è pronto per essere utilizzato (perché magari bisogna attendere che venga sovrascritto da un'istruzione precedente), l'istruzione viene posta in attesa all'interno del relativo componente di processamento. I componenti di processamento sono detti anche **reservation station**. *Cosa è Reservation Station?* Per ogni componente in grado di eseguire uno specifico calcolo, si ha una coda/buffer/corsia dove posso mettere varie istruzioni, e le operazioni identificate dai registri usati (sorgenti). Non si ha sorpasso per utilizzare tali componenti in queste code, al massimo se ho due istruzioni dipendenti cerco di metterle nella stessa reservation station.
- Si fa uso di una **cache** per leggere in modo più efficiente i dati provenienti dalla memoria.
- Nel momento in cui viene **committata** un'istruzione, l'eventuale alias aggiornato da tale istruzione viene installato all'interno del **register file** come valore valido, ovvero come **valore esposto all'interno dell'ISA**.
- **E)**: Se l'istruzione committata prevede una scrittura in memoria, il valore di output, prima di essere riportato in memoria, viene inserito nello **store buffer** in modo tale da velocizzare il completamento dell'istruzione. Tuttavia, ciò implica che il valore non sarà in memoria per un po' di tempo, il che può rappresentare un problema dal punto di vista della consistenza.

Nota: Supponiamo che un'istruzione sia in fase di ritiro, quindi è totalmente conclusa. In questo istante, il dato dovrebbe passare da Store Buffer alla memoria, ma in realtà non è istantaneo, passa un piccolo lasso di tempo. Ciò crea problemi con > 1 thread, perché se pescò un dato in memoria potrei non vedere alcuni dati. Per questo l'algoritmo della “pasticceria” non funziona nei processori moderni, suppone che tale tempo sia nullo. Esistono però delle istruzioni per controllare lo stato dello Store Buffer. Questo approccio è l'unico per lavorare con O.O.O, ovvero non esistono altre tecniche per affrontare le O.O.O pipeline.

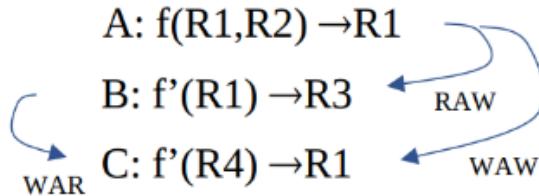
Esempi di schemi di esecuzione

Esempio 1 Supponiamo di avere tre istruzioni A , B , C tra cui B e C hanno una stessa latenza d , mentre A ha una latenza $d' > d$.

Assumiamo inoltre che:

- A esegua l'operazione $f(R1, R2)$ e riporti l'output sul registro $R1$. (in particolare, scrive su un alias di $R1$).
- B esegua l'operazione $f'(R1)$ e riporti l'output sul registro $R3$. (legge da STESSO alias di $R1$)
- C esegua l'operazione $f'(R4)$ e riporti l'output sul registro $R1$. (scrive su UN ALTRO alias di $R1$).

Ci ritroviamo dunque nel seguente scenario:



È abbastanza evidente che B debba leggere **lo stesso alias** scritto da A , mentre C potrà riportare il valore di output su un **alias differente**. In tal modo è possibile **processare C parallelamente ad A** :

In fondo, anche se C genera il suo output prima di A , le dipendenze che legano le tre istruzioni non vengono violate. Infatti, B sarà comunque in grado di leggere dall'alias sovrascritto da A e, ovviamente, rimarrà comunque possibile mandare in commit C solo dopo il completamento di A e B . Il vantaggio che porta questo approccio è la riduzione del tempo necessario per il completamento di C .

Esempio 2 Vediamo con un secondo esempio riportato qui di seguito come viene associata una entry del ROB a ciascun alias differente:

Before:	add r3,r3,4	after	add rob6,r3,4
	add r4,r7,r3		add rob7,r7,rob6
	add r3, r2, r7		add rob8,r2,r7

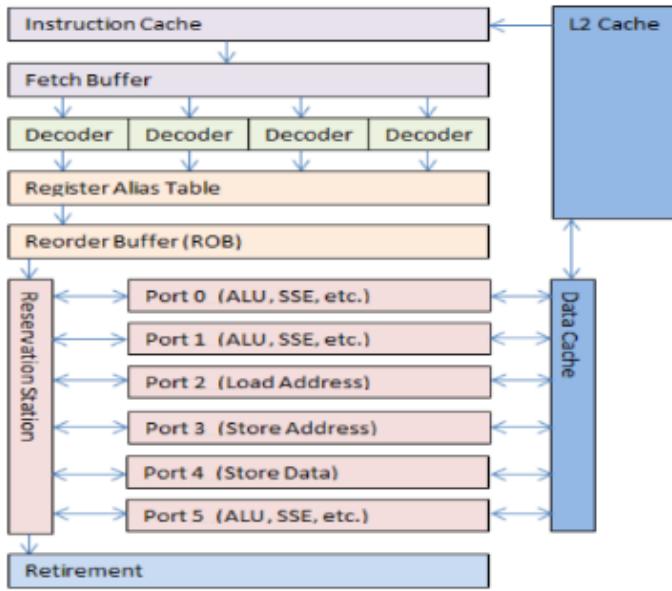
E' importante ricordare che, anche se è possibile eseguire in parallelo, non posso pensare di aumentare all'infinito le prestazioni, perchè ad un certo punto eccedo il numero di istruzioni che posso gestire.

Questo porta ad un sistema scarico, ovvero non occupo corsie perchè ci sono caricamenti di elementi dalla memoria, oppure eseguo una predizione/azzardo sbagliata che mi porta a svuotare etc...

E' come fare un'autostrada a 20 corsie: è molto difficile saturarla il "giusto".

Che soluzione è possibile adottare? Introduciamo i **Processori HYPERTHREAD**.

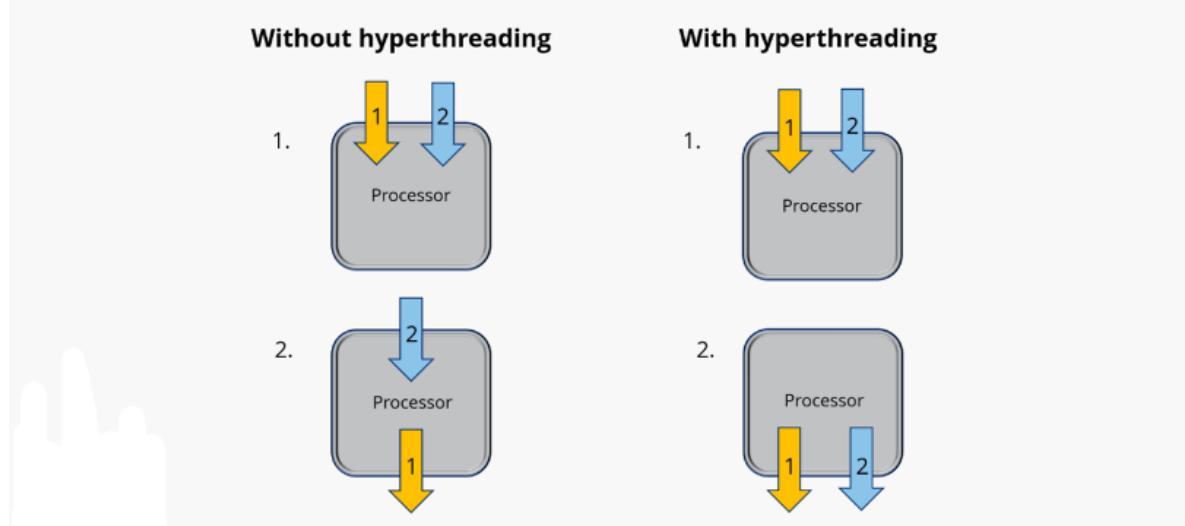
Organizzazione architetturale dei processori x86 OOO



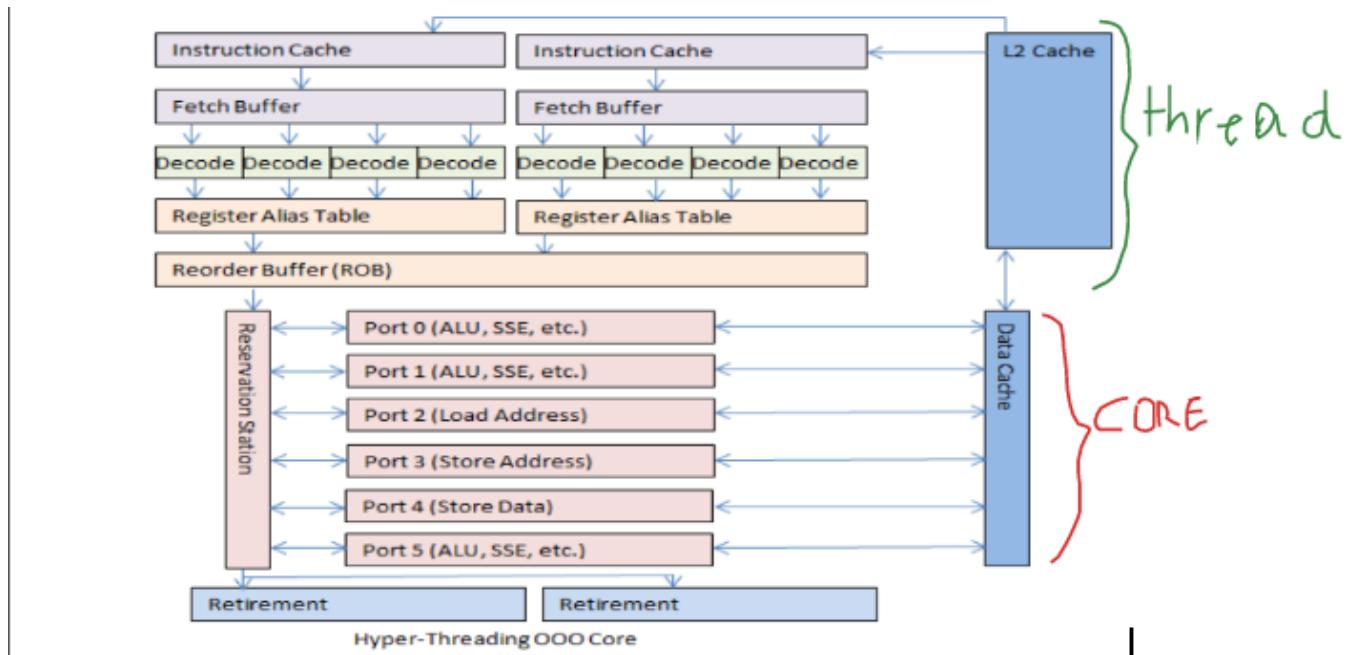
Processori hyper-threaded

Come abbiamo osservato all'inizio della trattazione, le architetture moderne sono soggette al **memory wall**. In particolare, si ha una latenza di esecuzione non solo quando si devono *recuperare i dati dalla memoria*, ma anche quando devono essere estratte le istruzioni stesse. Dunque, si può anche avere una pipeline molto efficiente con un **ILP** elevato, ma il processore non arriva mai a processare in parallelo tutte le istruzioni possibili a *causa dei ritardi causati dalla memoria*, per cui la potenza e le risorse del processore risultano sprecate. Per ovviare all'inconveniente, si può assegnare una *stessa reservation station a più program flow diversi*. Da qui nascono i processori **hyper-threaded**, che hanno un *unico core fisico* (*i.e.* *un'unica information station*) che permette di eseguire le reali operazioni dettate dalle istruzioni; d'altra parte, la porzione di processore che memorizza le istruzioni da fetchare, decodifica le istruzioni per uno specifico flusso (Decode) e tiene traccia dei registri logici dell'ISA (mediante Register Alias Table) viene replicata e viene definita **hyperthread**. Una CPU può contenere uno o più core. Un core è un'unità all'interno della CPU che realizza l'effettiva esecuzione. E' un "oggetto engine". Con l'Hyper-Threading, un microprocessore fisico si comporta come se avesse due core logici, ma virtuali. In questo modo si consente a un unico processore l'esecuzione di più thread contemporaneamente. Questo procedimento aumenta le prestazioni della CPU e migliora l'utilizzo del computer.

Esempio: ho una bocca (core) e due mani con cui prendo il cibo (thread). Per migliorare la prestazione devo puntare ad aumentare i thread (più mani = più cibo in entrata). Se aumentassi il numero di bocche, non migliorerei, perché sempre quel cibo ingerisco, anche se finisco prima devo aspettare la prossima forchettata.



In tal modo, è possibile eseguire più **workflow in parallelo su un unico core fisico**, e l'architettura risultante è riportata nella pagina seguente:



Tale architettura risulta molto vantaggiosa per la maggior parte dei workload, mentre risulta un po' più avversa nel caso in cui tutti i thread che girano sullo stesso core sono **CPU-bound** ma non memory-bound. Infatti, in quest'ultimo caso è più probabile che si verifichi un conflitto tra i thread in esecuzione nell'utilizzo delle risorse della reservation station. Un processore hyper-threaded viene tipicamente marcato con l'indicazione su quanti *core* (= "motori") e quanti *hyperthread* (= "thread fisici") possiede.

Esempio:

Un processore a due core e quattro thread fisici viene marcato come **2C/4T**.

In un contesto senza Hyperthread avremmo 2 core e 2 thread (1 per ogni core).

In un contesto con Hyperthread possiamo avere 2 core e 4 thread (2 thread per ogni core).

Se ho un caso 2C/4T, vuol dire che il mio programma non può generare più di 4 thread?

No, vuol dire che ogni core può gestire al massimo due thread con un proprio set di registri aventi alias (quindi due workflow in parallelo su unico core), quindi in totale posso gestire al massimo quattro thread/workflow in contemporanea. Ci sarà lo scheduling che realizzerà il parallelismo, ma comunque ne verranno gestiti quattro insieme. I flussi sui thread possono essere speculativi.

Gestione degli interrupt

Ne sono esempi: muovere mouse, premere sulla tastiera.

L'interrupt è un segnale *proveniente da un certo dispositivo hardware* che indica la necessità di **cambiare il flusso di esecuzione** (e.g. andando a eseguire un handler). Poiché è buona norma processare gli interrupt il prima possibile, quando ne occorre uno, si attende solo che una delle istruzioni correntemente in pipeline vada in commit; dopodiché si effettua lo squash (ovvero si svuota) la pipeline e si iniziano a fetchare le istruzioni dell'handler dell'interrupt. Ciò può portare a un rallentamento dell'esecuzione, ma non vale la pena memorizzare da qualche parte lo stato di esecuzione delle istruzioni che vengono buttate (richiederebbe hardware aggiuntivo, inoltre non ho certezza che dopo l'handler, ritornando al flusso di esecuzione, io parta dall'istruzione subito dopo); tra l'altro, anche mentre viene eseguito il gestore di un certo interrupt può subentrare un ulteriore interrupt, e questo può avvenire iterativamente un numero arbitrario di volte.

Attenzione:

Buttare delle istruzioni dalla pipeline implica prevenire i loro effetti sull'ISA ma, se esse hanno sporcato lo stato micro-architetturale, quest'ultimo non può essere ripristinato: rimangono comunque gli effetti dell'esecuzione speculativa delle istruzioni che sono state buttate. A livello hardware non ho meccanismi di gestione priorità o trap, operazioni offending come una divisione per 0 non vengono svolte dal processore, ma passano ad un handler.

Gestione delle eccezioni

Esse avvengono durante esecuzione di un programma, a livello software. Le eccezioni risultano più complesse da gestire rispetto agli interrupt poiché vengono sollevate dall'esecuzione di particolari istruzioni. Di fatto, un'istruzione A (cosiddetta **offending**) che solleva un'eccezione e_A può essere eseguita speculativamente all'interno della pipeline e, di conseguenza, potrebbe non esistere nel flusso di esecuzione definito dal programmatore. Ad esempio, poco prima dell'istruzione A potrebbe esserci un'istruzione B che solleva a sua volta un'eccezione e_B : in tal caso sarà e_B a esistere realmente e non e_A . Una Istruzione è offending se vuole usare qualcosa che non è usabile. Non genera una trap, perché non so se arrivo in retire. Se l'istruzione successiva in fondo non viene committata, cancella tutto. A valle di queste considerazioni, un'eccezione viene presa in carico solo nel momento in cui l'istruzione che l'ha generata va in retire, anche se ci si può accorgere prima che l'istruzione sia offending. (Vedi sotto, etichettamento offending).

Ma quali sono gli stage in cui un'istruzione può rivelarsi offending?

- **Instruction Fetch / Memory stages**

(MEM stage = stage in cui avviene l'accesso agli operandi): qui si può avere un page fault (ovvero un tentato accesso a un indirizzo logico di memoria che attualmente non ha un corrispettivo fisico), un accesso in memoria disallineato o una violazione delle protezioni applicate a una pagina di memoria (e.g. si tenta di accedere in scrittura a una locazione read-only).

- **Instruction Decode stage:** qui può emergere che l'istruzione in esercizio sia illegale.

- **Execution stage:** qui può essere sollevata un'eccezione aritmetica (e.g. divisione per zero).

- **Write-Back stage:** qui non possono essere sollevate eccezioni.

Per etichettare un'istruzione come offending, si imposta a 1 un apposito bit dei metadati. Quando tale istruzione va in *retire*, se il bit vale 1 allora il commit non viene effettuato, bensì viene attivato il gestore di eccezioni opportuno. A tal punto tutte le istruzioni successive a quella offending diventano *phantom* (fantasma).

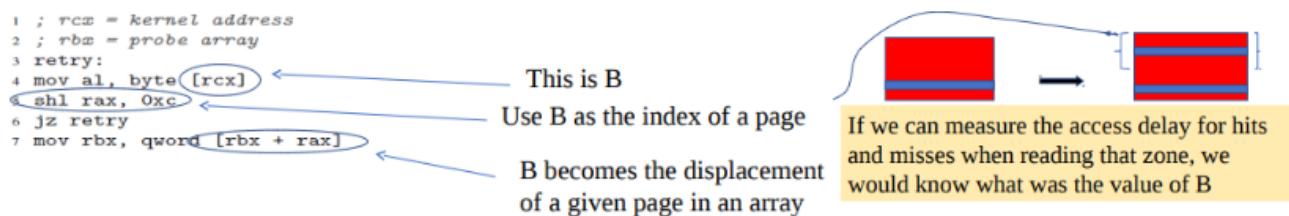
Attenzione: Con le eccezioni si verifica lo stesso problema riscontrato con gli interrupt. In particolare, quando un'istruzione offending va in retire, si hanno altre istruzioni all'interno della pipeline che hanno già modificato lo stato micro-architetturale e, dunque, hanno lasciato tracce di informazione. Questa problematica lascia spazio al cosiddetto attacco **Meltdown**. Sto propagando “attività illecite” nella pipeline.

Attacco Meltdown

Sappiamo bene che *ciascun processo ha il proprio address space suddiviso in zone di memoria dedicate all'esecuzione user mode e zone di memoria dedicate all'esecuzione kernel mode*. L'attacco ha come scopo ultimo quello di accedere a un'area di memoria situata nel kernel anche se non si hanno i permessi, magari per andare a leggere delle informazioni sensibili di un utente differente del sistema. Andando nel dettaglio, l'attacco viene eseguito nella maniera spiegata qui di seguito. Anzitutto si definisce un array A nella memoria user space e si svuota la cache tramite l'istruzione `cflush`. Dopodiché, mediante una `mov`, si preleva un **particolare byte** x (e.g: address) situato nel kernel e si memorizza il byte in un registro; naturalmente questa `mov` è un'istruzione offending. Si accede alla entry con **spiazzamento** x rispetto all'indirizzo base dell'array A (e.g. caricandone il contenuto in un registro) in modo da farla salire in cache:

tal aggiornamento della cache rappresenta proprio il *side effect* lasciato sullo stato micro-architetturale da parte dell'istruzione successiva a quella offending che, chiaramente, viene eseguita *solo speculativamente*. A tal punto, poiché l'array A si trova in user space, è possibile accedere a tutte le sue entry e, per ogni entry, cronometrarne il tempo necessario per l'accesso: la entry relativa al tempo di accesso minore sarà chiaramente quella di spiazzamento x , perché si tratta dell'unica entry il cui contenuto era stato memorizzato in cache. Ed ecco qui: **ora è noto lo spiazzamento x** , che era proprio il byte di memoria prelevato inizialmente dal kernel space. Ricapitolando, il valore x è stato ottenuto in maniera indiretta misurando i tempi di accesso alle informazioni presenti nell'array A. Di conseguenza, abbiamo a che fare con un **side-channel (o covert-channel) attack**, ovvero con un attacco che fa uso di un canale laterale per andare a rubare delle informazioni. Io parto da un byte B presente nella zona kernel, lo salvo in un registro (non potrei), accedo a **array[B]** che va in cache. Successivamente accedo a tutte le entry di array (in un loop continuo, indipendente da B). L'accesso più veloce sarà quello ad **array[B]**, e quindi tra tutti gli accessi effettuati, riesco a capire cosa c'è in **array[B]**, perché più veloce. Quindi prendendo un byte nella zona kernel, riesco da user a leggere qualcosa a livello kernel.

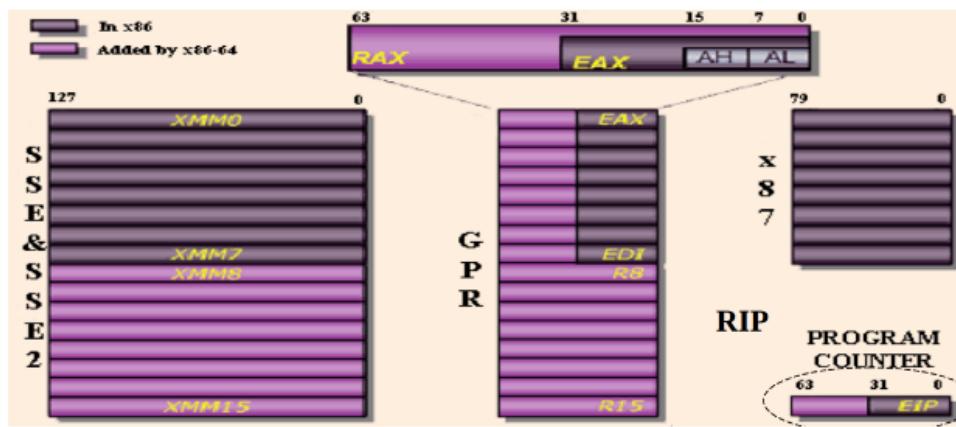
Meltdown può essere esteso anche al caso in cui si vuole scoprire un'intera stringa all'interno del kernel space, che può essere una password, una chiave segreta e così via. Negli esempi



Insights sui processori x86 e x86-64

X_86 ha 8 registri general purpose, e Program Counter 32 bit.

X_86_64 è backward compatibile, ha 15 registri general purpose a 64 bit e Program Counter a 64 bit.



- **GPR:** registri general purpose.
- **SSE & SSE2:** registri vettoriali, che consentono a singole istruzioni di fare più cose in parallelo.
- **x87:** floating-point stack registers.
- **RIP:** program counter.

Vediamo alcune istruzioni fondamentali per il trasferimento di dati nell'architettura x86-64:

Istruz. (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione	Esempio*
mov	movl	movw	movb	Trasferisce un valore su un registro destinazione.	movw \$5, %ax
push	pushl	pushw	\	Aggiunge un elemento sullo stack.	pushl %ebx
pop	popl	popw	\	Elimina un elemento dallo stack e lo salva su un registro.	popl %ebx

*Negli esempi specificati in tabella si è adottata la sintassi AT&T che, a differenza di quella Intel, prevede che la sorgente venga posta prima della destinazione.

La parte più complessa però sta nello specificare la sorgente e/o la destinazione **in memoria logica** (e non su un registro). La figura riportata in seguito mostra il meccanismo utilizzato. In ISA non ho i nomi delle variabili, solo quelli dei registri, per arrivare ad una variabile di memoria devo usare l'indirizzo di tale variabile in memoria.

Nota: Le componenti *base*, *index*, *scale* sono sempre contenute nelle parentesi, ad esempio *movl* (..., ..., ...)

- **Displacement**: può ad esempio essere un indirizzo assoluto noto a tempo di compilazione. Possiamo vederlo come il punto dell'address space che specifica la sorgente (es: *foo* è *displacement diretto*)
- **Base**: può essere relativo al valore di un pointer.
- **Index*scale**: può rappresentare uno spiazzamento rispetto all'indirizzo base; ad esempio, quando si itera su un array di interi, *scale* vale sempre 4 mentre *index* viene incrementato di 1 a ogni iterazione.

(nb: in %ebc è dove carico il tutto).

$$\begin{aligned}
 \text{Offset} = & \left[\begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right] + \left[\begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right] * \left[\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \right] + \left[\begin{array}{c} \text{None} \\ 8\text{-bit} \\ 16\text{-bit} \\ 32\text{-bit} \end{array} \right] \\
 & \text{Base} \quad \text{Index} \quad \text{scale} \quad \text{displacement} \\
 \text{Displacement} & \xrightarrow{\quad} \text{movl } \underline{\text{foo}}, \%eax \\
 \text{Base} & \xrightarrow{\quad} \text{movl } \underline{(\%eax)}, \%ebx \\
 \text{Base + displacement} & \xrightarrow{\quad} \text{movl } \text{foo}(\underline{\%eax}), \%ebx \\
 & \xrightarrow{\quad} \text{movl } 1(\underline{\%eax}), \%ebx \\
 (\text{Index} * \text{scale}) & \xrightarrow{\quad} \text{movl } (\%, \underline{\text{eax}}, 4), \%ebx \\
 \underline{\text{Base}} + \underline{(\text{index} * \text{scale})} + \underline{\text{displacement}} & \xrightarrow{\quad} \text{movl } \underline{\text{foo}}(\underline{\%ecx}, \underline{\%eax}, 4), \%ebx
 \end{aligned}$$

Vediamo ora le istruzioni logiche e aritmetiche:

Istruzione (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione
and	andl	andw	andb	dest = source && dest
or	orl	orw	orb	dest = source dest
xor	xorl	xorw	xorb	dest = source ^ dest
not	notl	notw	notb	dest = ^dest
sal	sall	salw	salb	dest = dest << source
sar	sarl	sarw	sarb	dest = dest >> source
add	addl	addw	addb	dest = source + dest
sub	subl	subw	subb	dest = dest - source
inc	incl	incw	incb	dest = dest + 1
dec	decl	decw	decb	dest = dest - 1
neg	negl	negw	negb	dest = ^dest
cmp	cmpl	cmpw	cmpb	Compara due valori; se essi sono uguali, imposta un determinato registro di stato.

Codice assembly dell'attacco Meltdown

Nella pagina seguente viene mostrato il codice assembly che permette di effettuare l'attacco Meltdown. La sintassi utilizzata è quella di Intel. Chiaramente il codice può essere incapsulato all'interno di un programma C.

```
; rcx = kernel address
; rbx = probe array (array A)
retry:
mov al, byte [rcx] //istr. offending. Prendo byte[rcx] e lo metto in 'al', ultimo byte del reg. 'eax'
shl rax, 0xc          //shiftto rax di 0xc = 12, cioè 12 bit a 0 (verso sx shiftto la parte meno significativa) l
jz retry             //se il valore letto nel kernel è nullo, cioè rax=0 riprovare.
mov rbx, qword [rbx + rax] //qui si effettua l'accesso al probe array con spiazzamento pari a rax
```

Vale la pena fare alcune osservazioni:

- 4096 byte è esattamente la dimensione di una pagina di memoria. Ma perché gli spiazzamenti che si prendono in considerazione all'interno dell'array A sono esclusivamente la prima entry di ciascuna pagina di memoria? Per evitare problemi con la cache: di fatto, quando si accede a un certo valore in memoria, non è solo lui a essere caricato in cache, ma come minimo una **linea di cache**, che è lunga 64 byte. Leggiamo sempre lo 0-esimo byte, perché la cache lavora a blocchi e potrei trovare '64 byte' rapidi, e quindi non essere in grado di differenziare gli accessi. Per giunta, i processori tipicamente applicano il cosiddetto **pre-fetching**: nel momento in cui viene caricata in cache una linea, vengono conseguentemente caricate anche alcune linee adiacenti per il principio di località. Perciò, per evitare che i valori relativi a più di uno spiazzamento in A vengano caricati in cache a seguito di un unico accesso, si prendono gli spiazzamenti in modo tale che siano distanti tra loro, e una distanza di 4096 byte risulta essere sufficiente.
- Perché se nel kernel space viene letto il byte 0 si fa un nuovo tentativo? Perché un'area di memoria inizializzata a zero o è memoria non valida o è relativa a un terminatore di stringa, per cui si tratta di un'area di memoria non significativa. Resta comunque possibile ritentare ad accedere al medesimo byte all'interno del kernel space perché non è escluso che, in concorrenza, il kernel possa cambiare il valore di quel byte (da zero a un valore significativo e di interesse).

Contromisure per l'attacco Meltdown

- KASLR (Kernel Address Space Randomization): Quando si fa il setup del kernel del sistema operativo, le strutture dati di livello kernel possono cadere ovunque all'interno del kernel space: in particolare, si stabilisce randomicamente un *offset F* rispetto all'indirizzo base del kernel a partire da cui sono definite le varie strutture dati del kernel. In questo modo, si complica la vita all'attaccante poiché lui non conosce a priori l'indirizzo del

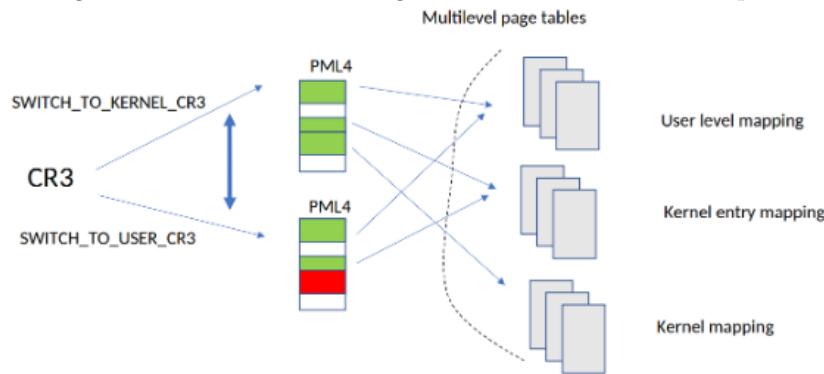
kernel space in cui andare a effettuare l'attacco; tuttavia, con un approccio brute-force, può comunque fare in modo che, prima o poi, l'attacco vada a buon fine. Per questa ragione, KASLR non è la soluzione definitiva contro l'attacco Meltdown.

- Cash flush:

Si effettua semplicemente un flush della cache ogni volta che il kernel prende il controllo od ogni volta che un thread viene rischedulato. Tuttavia, così facendo, si avrebbe un calo inaccettabile delle prestazioni: le cache sono condivise tra i vari hyperthread, per cui ciascun cache flush impatterebbe su tutti i thread in esecuzione all'interno dell'host.

- KAISER (Kernel Isolation in Linux):

Quando un processo gira in modalità *user*, utilizza una page table PT_U all'interno della quale il mapping della maggior parte degli indirizzi del kernel space non è presente: ci sono esclusivamente quegli indirizzi della porzione del kernel che vengono utilizzati ad esempio per gestire gli interrupt (se non ci fossero almeno loro, non saremmo neanche in grado di gestire gli interrupt o comunque di trasferire il controllo al kernel). Dall'altro lato, quando un processo gira in modalità *kernel*, utilizza un'altra page table PT_K che contiene il mapping degli indirizzi di memoria mancanti in PT_U . Di fatto, gli indirizzi del kernel space il cui mapping è presente nella page table PT_U costituiscono la cosiddetta **entry zone** del kernel, che contiene quasi esclusivamente il codice che permette di effettuare lo switch delle page table (PT_U a PT_K). Perché quest'ultima soluzione risulta funzionante? Nel momento in cui un processo che esegue in user mode incontra un'istruzione offending che vuole accedere a un indirizzo di memoria del kernel space, tipicamente non troverà tale indirizzo nella page table che ha a disposizione. In altre parole, si solleva un'eccezione di tipo **page fault**, che non consente al processo di continuare ad eseguire le istruzioni anche speculativamente. Inoltre, si tratta di una soluzione che, sì, ha dei costi prestazionali, ma mai quanto il cash flush; in particolare, il calo delle performance viene osservato dalla **TLB - Translation Lookaside Buffer** che, a ogni switch da user mode a kernel mode e viceversa, dovrà cachare le informazioni sulla nuova memory view su cui ci siamo spostati (che sia essa relativa alla PT_U o alla PT_K). Entrando un po' più nel dettaglio sulle page table, se ne hanno di livelli differenti (page table di livello 1, page table di livello 2, e così via). Solo le **page table di livello 1** sono replicate tra l'esecuzione user mode e l'esecuzione kernel mode, mentre le altre sono a **istanza unica**; in particolare, al livello 1, la page table PT_K è in grado di raggiungere tutte le informazioni delle page table ai livelli inferiori, mentre la page table PT_U punta solo a un sottoinsieme di informazioni delle page table ai livelli inferiori. Da user possiamo comunque accedere a qualcosa di livello kernel, ma solo per i moduli kernel utili/indispensabili (compile time) per avere il minimo supporto necessario. KAISER sfrutta quindi questa tecnica di **PTI-Page Table Isolation**, disattivabile da GRUB. Il selettore CR3, ad ogni cambio, azzera la TLB e genera cache miss. Però è un'operazione automatica ed abbastanza semplice.



Insights sui branch

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Per i salti **condizionali** e **indiretti**, l'indirizzo da cui riprenderà l'esecuzione è noto soltanto a partire da un certo stage S della pipeline. Chiaramente, però, vogliamo che venga comunque eseguito del lavoro all'interno della pipeline prima che l'istruzione di salto raggiunga lo stage S, altrimenti il calo delle prestazioni sarebbe certo e significativo. A tal proposito, si introducono dei **dynamic predictor** (o **branch predictor**), che hanno lo scopo di predire quale sarà la destinazione verso cui si dovrebbe saltare con la più alta probabilità. Ciò permette di comporre in maniera speculativa un program flow all'interno della pipeline in funzione della predizione che è stata attuata dal predittore. La reale implementazione dei branch predictor è basata sui **Branch History Table (BHT)**, detti anche **Branch-Prediction Buffer (BPB)**, che contengono metadati che associano un'istruzione di salto all'ipotesi su dove tale istruzione salterà. È una zona di cache hardware contenente [indirizzo istruzione, target da usare se jump], e l'indirizzo istruzione può anche essere il target stesso. È un suggeritore, non so nulla dell'affidabilità. L'implementazione minimale (andando avanti verranno aggiunte altre componenti) per una BHT consiste in una cache indicizzata tramite i *bit meno significativi* di ogni istruzione di salto. Ogni qual volta viene fetchata un'istruzione di salto, si può avere una cache hit o una cache miss, dove la *cache hit* la si ha nel caso in cui sono disponibili sufficienti informazioni che permettano di effettuare una predizione sulla destinazione del salto. Queste informazioni consistono in particolari bit di stato: ciò che succede nel passato è rappresentativo di ciò che si prevede che accadrà in futuro. Nei salti **condizionali** ho due destinazioni, sappò quella corretta in fase di esecuzione, e quindi è register dependent, perché dipende a runtime cosa ci sarà nel registro). I salti **unconditional** e le **call**, dove salteremo è già noto nello stage di decode, salterò sempre in quel punto.

Anche le **return** vengono sempre eseguite, ma ho più destinazioni note nello stage di esecuzione.

Per i salti **indiretti** salterò sempre (come per le call), ma con più destinazioni. Il target potrebbe essere in un registro.

Predittori per i salti condizionali

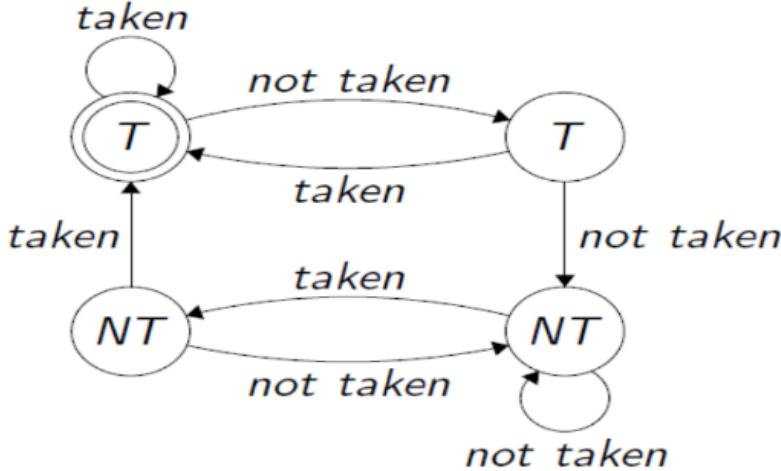
Predittori coi bit di stato Il modo più facile per implementare un predittore per i salti condizionali è sfruttando un **unico bit di stato**: se l'ultima istruzione di salto ha avuto come esito "taken" (ovvero ha realmente portato a effettuare il salto), allora il predittore prevederà che anche il prossimo salto avrà come esito "taken", e viceversa. Qui la storia passata è rappresentata esclusivamente dall'*ultima istruzione di salto*, per cui non si tratterebbe neanche di un vero e proprio storico.

Esempio Caso semplice:

ciclo for, sbaglio la predizione quando uscirò. Per questo motivo, sono stati introdotti anche i predittori a *due bit* di stato. Di fatto, questi predittori si comportano meglio negli scenari in cui si hanno molte istruzioni di salto con esito "taken" e poche istruzioni di salto con esito "not taken" (e viceversa). Lo scenario classico è quello dei *nested loop*. In particolare, la predizione cambia da "taken" a "not taken" (o viceversa) non più a seguito di un solo errore, ma a seguito di **due errori consecutivi del predittore**. La macchina a stati che rappresenta i predittori a due bit è la seguente, dove:

- T = "predico che il salto sarà taken"
- NT = "predico che il salto sarà not taken"

Esempio doppio ciclo: Nel ciclo interno iterò, e la predizione mi dice che salterò. Quando finisco le iterazioni nel ciclo interno, la predizione sbaglia e dice che continuerò nel ciclo. Invece aumento l'iterazione sul ciclo esterno (1 errore). Però poi rientro effettivamente nel ciclo interno, quindi in realtà non devo cambiare predizione anche se ho fatto un errore, perché stavolta ci rientro davvero dentro. *Solo se sbaglio 2 volte cambio previsione.*



Ricordiamo che, in caso di previsione sbagliata, in pipeline vengono caricate istruzioni che alla fine subiranno uno squash, che porterà al refill della cache. Quindi è importante effettuare la predizione con un buon tradeoff affidabilità/supporto hw.

Esistono anche predittori più sofisticati come il **Two-Level Correlated Predictor** e l'**Hybrid Local/Global Predictor** (noto anche come **Tournament Predictor**).

Two-Level Correlated Predictor Consideriamo il seguente codice:

```

if (aa == VAL) aa = 0;
if (bb == VAL) bb = 0;
if (aa != bb) {
    //do the work
}

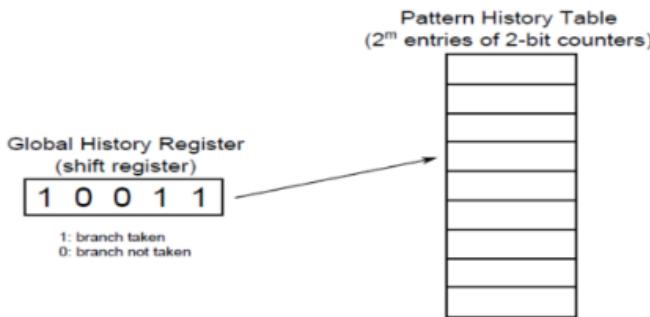
```

Ci piacerebbe avere un predittore tale per cui, se i branch relativi ai primi due if vengono presi, predica che il terzo branch (che dipende dai due precedenti) non venga preso. È quello che fa il **(m,n) Two-Level Correlated Predictor**, dove:

- m = numero di salti precedenti, il cui esito determina la previsione che verrà effettuata.
- n = numero di bit del valore che indica quanti **errori** deve commettere il predittore prima di **cambiare** la sua **previsione** (esattamente come nei predittori con bit di stato).

Esempio:

Consideriamo il caso con m=5, n=2. Si ha il seguente scenario:



Il **global history register** è composto da m bit, ciascuno dei quali indica l'esito di uno delle ultime m istruzioni di salto condizionale. Da destra ho i salti più recenti. Tale registro può assumere 2^m valori diversi, e a ciascuno di essi è associato un predittore con bit di stato (= una macchina a stati) differente. Il predittore per la previsione corrente viene scelto sulla base dei risultati degli ultimi m branches, come è codificato nella 2^m bitmask. Sostanzialmente usiamo il Global History Register come indice.

Un esempio semplice con ($m = 2$, $n = 2$).

Nel global register posso avere 4 casi: 00, 01, 10, 11, dove 0 vuol dire "not taken" e 1 "taken". Per ogni casistica

associa una entry nel Branch History Table.

00 mappato nell'entry 0, 01 mappato nell'entry 1, 10 mappato nell'entry 2, 11 mappato nell'entry 3.

Il fatto che $n = 2$ vuol dire che ogni entry mantiene 2 bit nell'array, ovvero include l'automa a stati visto precedentemente.

Se ho tre statement if, e salto sempre al terzo in modo condizionato, la mia branch sequence è 001001001001...

Se ($m = \text{indifferent}$, $n = 2$) mantengo 4 entry con due bit ciascuno (00, 01, 10, 11).

- Entry 00: ho sbagliato due volte, allora il terzo salto sono sicuro.
- Entry 01: ho sbagliato una volta, resto su 0.
- Entry 10: ho sbagliato una volta, resto su 0.
- Entry 11: non capita perchè abbiamo detto di non saltare due volte consecutivamente.

Se avessi ($m = 0$, $n = 2$), avrei 0 elementi per identificare l'entry. Sarebbe un predittore basico.

Tournament Predictor

In realtà non è sempre vero che i salti condizionali siano correlati tra loro. Per questo motivo si introduce l'**Hybrid Local/Global Predictor**, che consiste in una "sfida" tra un **predittore locale** (come quelli con bit di stato) e un **predittore globale** (che si basa sulla storia passata e assume che i salti condizionali siano correlati). In pratica, si hanno entrambi questi predittori e in più una macchina a 4 stati che indica se correntemente conviene utilizzare il predittore locale oppure quello globale: se stiamo sfruttando il predittore locale, *switchamo a quello globale dopo due errori consecutivi, e viceversa*. Ciò è realizzato mediante *Return Stack Buffer RSB*, tipo una semplice cache, con 32 entries (quindi 32 livelli di annidamento), cioè indirizzi associati al return point dell'istruzione call, in cui salvo nella prima entry libera tale indirizzo di ritorno.

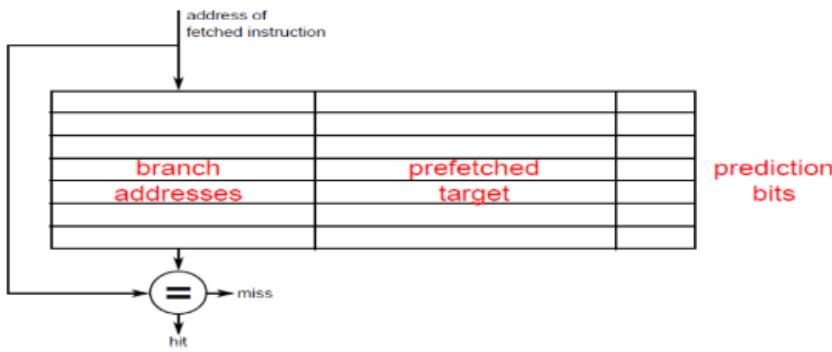
Osservazione su RSB Faccio call di una funzione, salvo indirizzo successivo alla call in RSB, quando esco dalla funzione il return punta all'indirizzo salvato nell'RSB precedentemente. RSB è una cache di tipo LIFO, contenuta nel processore ed è modificabile solo tramite *call* (inserimento valore) o *return* (prelievo valore). Non è esposta nell'ISA.

Questi switch e letture avvengono in elementi mantenuti in cache, allora non ho perdite di prestazioni.

Predittore per salti indiretti

I salti indiretti sono tali per cui l'indirizzo di destinazione viene caricato in un registro. Se l'istruzione deve saltare, questo è funzione del risultato delle istruzioni precedenti, ovvero di cosa queste hanno scritto nei registri, che verranno usati per saltare.

Quindi, il valore di tali registri può *variare sempre nel tempo*, il che rende la predizione più complessa e con più side effects generati. Per questo motivo, le destinazioni possibili possono essere molteplici, e la predizione può risultare più farraginosa. I predittori per i salti indiretti sono dotati della seguente struttura dati:



Ciascuna entry della tabella è relativa all'indirizzo di una particolare istruzione di salto (**branch address**). Ogni branch address è associato a un **prefetched target** (che è il *presunto* indirizzo destinazione del salto, se c'è cache miss non posso prelevarlo, oppure prendo l'istruzione successiva) e a dei **prediction bit** (che, come al solito, determinano numero di errori consecutivi da commettere prima di cambiare la predizione, ovvero prima di cambiare il prefetched target). In particolare, la prima volta che si incontra una certa istruzione di salto indiretto, si effettua il training, in cui il prefetched target verrà inizializzato all'effettivo indirizzo destinazione del salto. In queste tabelle riporto solo i bit meno significativi, per questioni di scalabilità. Tutto questo sta nel core, e se ho hyperthreading, la predizione viene fatta da una comune componente hardware per più thread. Ovvero se ho due thread (supponiamo facciano stesse cose)

su due hyperthread e 1 core, questo core comune può essere sfruttato dal thread 2 per avere informazioni sui salti ereditate da thread 1, per ottenere un boost delle performance. A livello di sicurezza è un po' problematico: se thread1 seguisse un comportamento tale da suggerire al thread 2 di fare salti (anche speculativi) che in realtà non dovrebbe fare?

Attacco Spectre

Chiaramente anche la predizione dei target dei salti condizionali può portare il processore a riempire la pipeline con istruzioni eseguite in modo speculativo. In particolare, se la predizione è scorretta, le istruzioni successive al salto dovranno poi essere buttate. Ma anche qui, come nel caso delle eccezioni, le istruzioni considerate in maniera speculativa che poi vengono scartate lasciano dei side effect nello stato micro-architetturale. (Solito discorso: un salto speculativo non installa nulla nell'ISA, ma lascia tracce a livello micro architetturale). Da qui nasce la possibilità di compiere una famiglia di attacchi denominati **Spectre** che sfruttano un covert-channel.

Questi attacchi sono anche più gravi di Meltdown poiché:

- Non richiedono l'esecuzione di un'istruzione offending.
- È possibile effettuare un **training scorretto** del branch predictor inserendo nel codice di livello user degli appositi branch / delle istruzioni di salto condizionale.

L'unica soluzione per evitare questi attacchi sarebbe quella di non fare predizioni, ma i sistemi diventerebbero molto più lenti (con o senza hyperthread), a livello di marketing sarebbe una mossa controproducente!

Spectre V1

È un attacco che inizialmente effettua il flush della cache (come Meltdown) per poi sfruttare i salti condizionali. In particolare, prevede l'utilizzo del seguente blocco di codice:

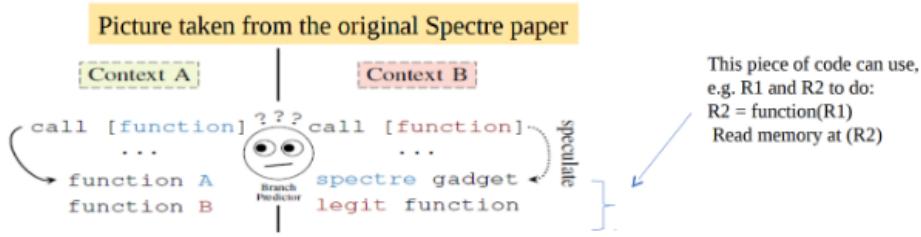
```
if (x < array1_size)
    y = array2[array1[x] * 4096]
//il prodotto per 4096 corrisponde a uno shift a sx di 12 posizioni
```

- *array1* è un array che può trovarsi in qualunque punto dell'address space, e *array1[x]* è un particolare valore che verrà moltiplicato per 4096 per ottenere un indice di pagina, che verrà utilizzato per accedere ad *array2*.
- In ogni caso, il valore *x* viene selezionato in modo tale che sia molto elevato (maggiore di *array1_size*), cosicché il branch non venga realmente preso, mentre magari il predittore prevedeva il contrario; inoltre, *x* può essere tale che l'accesso ad *array2* porti a reperire (speculativamente) un'informazione segreta (*y*), ad esempio all'interno del kernel space.
- Quel che si ottiene è che il valore *y* viene caricato all'interno della cache in modo speculativo. A questo punto, come in Meltdown, si effettuano gli accessi alla prima entry di ogni pagina di *array2* finché non si giunge a quella caricata in cache (di cui si osserva un tempo di accesso ridotto).
- Quindi ciò che carico è in funzione sia dell'array sia di *x*, se salto molto lontano potrei andare in una zona kernel. Lo scopo è far sì che questa condizione venga eseguita speculativamente, quindi il predittore deve essere indotto a credere che tale flusso di esecuzione sarà quello realmente eseguito. **Non devo eseguirlo realmente!**
- No trap perchè sono sempre in user mode e tutto dipende dall'esito dell'IF. Aver *patchato* Meltdown non implica risolvere anche Spectre. La soluzione di Meltdown risiedeva nel fatto che tale attacco creasse una trap poichè si passava da user a kernel, e c'era il cambio del puntatore alla page table. Qui sono sempre in user mode.

Spectre V2

Con più thread passo alla versione v2! È un attacco che sfrutta i salti multi-target (indiretti) ed è di tipo **cross-context**. Ciò vuol dire che l'attaccante è in grado di inferire (aggiungere) delle informazioni sensibili da un contesto di esecuzione diverso dal suo: supponiamo di avere due thread A, B che girano sul medesimo hyperthread in modalità processor sharing o su due hyperthread relativi allo stesso core. Il thread A può portare avanti delle attività che vanno a cambiare lo stato del branch predictor all'interno del CPU-core. Di conseguenza, B può osservare il nuovo stato del branch predictor all'interno del medesimo CPU-core e, quindi, le attività che lui eseguirà in modo speculativo (in funzione dello stato del branch predictor) vengono *praticamente decise, e poi osservate mediante side effect, dal thread A*. Tale side effect, come al solito, può consistere nel caricamento di un'informazione sensibile all'interno della cache attraverso il suo accesso in memoria. L'efficacia dell'attacco diventa particolarmente evidente quando l'attaccante si basa su un contesto che utilizza una libreria condivisa (shared library) col contesto del thread B. Qui, infatti,

le pagine di memoria della shared library vista dal thread B e le pagine di memorie della shared library vista dal thread A mappano esattamente sugli stessi indirizzi fisici. Perciò è ovvio che qualunque side effect la vittima lasci speculativamente su tali locazioni fisiche di memoria sia direttamente visibile all'attaccante.



Con riferimento alla figura, il **gadget** è un blocco di codice che è definito nell'address space della vittima e viene sfruttato dall'attaccante affinché la vittima lo esegua in modo speculativo a seguito di un errore del branch predictor; in tal modo, nello stato micro-architetturale, la vittima lascia i side effect desiderati dall'attaccante. Nell'esempio mostrato nella figura vengono utilizzati due registri ($R1$ e $R2$), di cui $R1$ viene utilizzato per calcolare $R2$ con una particolare funzione, mentre $R2$ viene usato per memorizzare l'indirizzo verso cui accedere in memoria. In realtà, sarebbe sufficiente l'utilizzo di un solo registro $R1$.

NB: l'utilizzo del medesimo CPU-core tra **thread A e thread B** è richiesto solo per effettuare un training errato sul **branch predictor**. Dopodiché, poiché la cache è condivisa tra più CPU-core, gli accessi in memoria effettuati per stabilire quali dati sono saliti in cache possono essere effettuati anche in un momento in cui il thread vittima (B) gira in un CPU-core differente. Tra l'altro, per portare a termine l'attacco, è possibile anche utilizzare una macchina virtuale su cui possono girare delle applicazioni che, in qualche modo, vanno a cambiare lo stato del branch predictor: in fondo l'hardware sottostante viene utilizzato anche per eseguire le macchine virtuali!

Contromisure per gli attacchi Spectre

Retpoline - Return trampoline

Al posto di invocare l'istruzione di salto indiretto (che sia essa una jump o una call), si esegue un blocco di istruzioni funzionalmente equivalente che non consente di effettuare una mis-prediction (ovvero un training scorretto del branch predictor) per portare a compimento l'attacco Spectre. Una versione semplificata del blocco di istruzioni è riportata di seguito. Non faccio più salti indiretti, solo diretti!

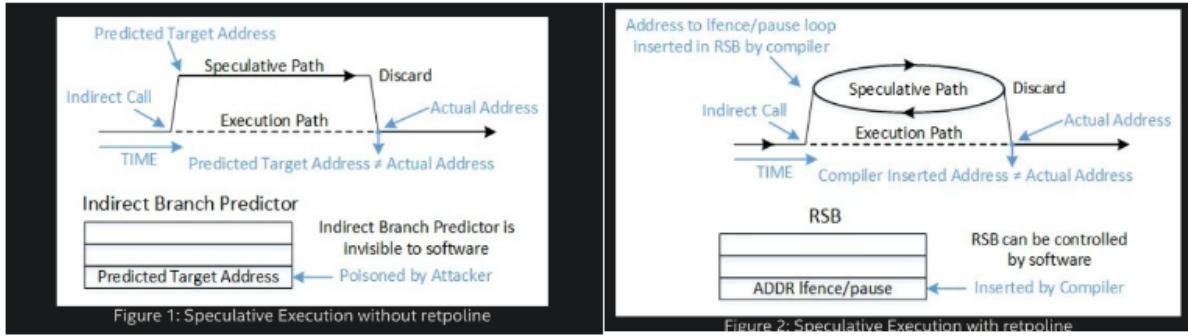
Invece di *Jump su R*, eseguo *Return su R*, non c'è più predizione. Il *RET* usa il *Return Stack Buffer* per la predizione (di tipo *Lifo*), controllabile via software, a differenza del predittore di Branch Indiretto.

```
push target_address
1: call retpoline_target
    //put here whatever you would like (with no side effects)
    jmp 1b
retpoline_target:
    lea 8(%rsp), %rsp //we do not simply add 8 to RSP since FLAGS registry should not be modified
    ret   //this will jump to target_address
```

Vediamo nel dettaglio cosa fa questo blocco di istruzioni:

- Carica l'indirizzo destinazione del salto (`target_address`) sullo **stack**. (ora è in cima allo stack)
- Effettua una call di `retpoline_target` (per cui viene caricato sullo **stack anche l'indirizzo dell'istruzione successiva alla call**). (Graficamente, nell'address space avremo `target_address` e sopra di lui, l'indirizzo successivo alla call. Questo perché chiamata `retpoline_target`, prevediamo che al suo *return* ripartiamo dall'istruzione successiva ad essa). `rsp` a 64 bit, quindi 8 byte, per questo faccio quello shift.
- In `retpoline_target` si **elimina** l'indirizzo dell'istruzione successiva alla call e, tramite la return, si salta verso l'indirizzo che si trova in cima allo **stack** (ovvero `target_address`). Facendo lo shift di 8 byte, cancelliamo il punto di ritorno della chiamata `retpoline_target`. Dopo ciò, in cima allo stack abbiamo proprio `target_address`.
- Essendoci una **call** (non è register dependent, mi manda verso la funzione chiamata), il predittore non deve essere soggetto a training: prevede che, a seguito della return, si salti verso l'istruzione successiva alla call. Per questo motivo, dopo la call devono essere eseguite delle istruzioni innocue, come una jump verso l'istruzione stessa di call. La *return* vede qualcosa nell' `rsb` che non può essere bypassata. La *ret* esegue speculativamente ciò che gli dice `rsb`. Tuttavia, se la CPU specula, l'RSB la porta ad eseguire `jmp 1b`, intrappolandolo in un loop.

Successivamente, la CPU realizza che il valore in RSB è diverso da quello nello stack (*target_address*), e stoppa la speculazione. **O vado in target_address, o resto in un loop.** RSB ha senso per singolo processo, mentre il predittore si muove tra più flussi.



Esistono altre tecniche per Spectre V2?

IBRS (Indirect Branch Restricted Speculation)

È possibile aggiornare il registro **MSR** (Model Specific Register, è un registro di controllo nella CPU) per creare un'enclave di esecuzione (= uno spazio di esecuzione chiuso entro determinati confini) dove la storia non è influenzata da cosa avviene al di fuori dell'enclave stessa. In pratica, a livello hardware siamo in grado di utilizzare la branch prediction in maniera differenziata a seconda se stiamo lavorando a livello user (MSR=0) o a livello kernel (MSR=1): nel primo caso il branch predictor dei salti indiretti si basa sulla storia passata della sola esecuzione a livello user, mentre nel secondo caso si basa sulla storia passata della sola esecuzione a livello kernel. Sostanzialmente, a tempo t decido che lo stato del predittore non dovrà più essere usato per un certo tempo. E' un guscio che mi protegge dall'esterno. Utile se ho app che lavorano a livelli diversi (user e kernel), se ad esempio volessi che le scelte kernel non venissero influenzate dall'user.

IBPB (Indirect Branch Prediction Barrier)

Anche qui ci si basa sull'utilizzo del MSR. In particolare, nell'istante in cui si va a scrivere all'interno di tale registro, stiamo cancellando tutto ciò che il branch predictor dei salti indiretti ha imparato finora, creando così una sorta di barriera. Al tempo t resettiamo il branch predictor.

IBRS e **IBPB** sono operazioni software, spesso si usa la syscall `prcr()` per lavorare col Processor Control. Se lavoriamo con `exec()` (programma che chiama un altro programma) funzionano uguali, perchè facenti parte dello stesso programma iniziale.

Sanitizzazione

Introduzione

Per Spectre V1 abbiamo visto la possibilità di riusare le patch di Meltdown. Spectre V1 mi permetterebbe di leggere dati kernel? Sì, perchè la predizione a livello kernel potrebbe essere basata su predizioni livello user.

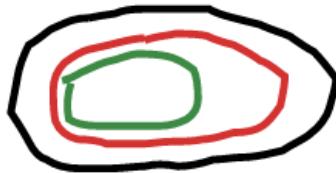
A livello kernel abbiamo una tabella, che chiamiamo **driver**, in cui per ogni indice ho un servizio che richiama una *system call*. L'indice è passato dall'user, ma è compatibile con la taglia della tabella? Dipende, con l'indice facciamo un salto verso una entry, ma se eccedo la dimensione? **Speculativamente** starei usando un indice errato come pointer per una zona codice Kernel, perchè se le precedenti `call` erano corrette, il predittore si sarà settato in un certo modo. (Se ha sempre saltato, speculativamente lo rifarà). Soluzione? Sanitizzazione.

Come funziona

È una contromisura per i salti condizionali. In particolare, ciascuna condizione `if` che coinvolge una certa variabile `V` prevede dei valori per `V` ammissibili e dei valori per `V` non ammissibili. Quello che si fa è ridurre all'osso l'insieme dei valori non ammissibili, facendo sì che tutti gli altri valori non ammissibili non possano in alcun modo essere assunti da `V`. Questa è una tecnica estremamente utile per gli indici delle tabelle. Infatti, supponendo che per una tabella gli indici ammissibili vadano da 0 a $j - 1$, se per qualche motivo l'indice dovesse assumere un qualunque valore maggiore di $j - 1$, viene forzato ad assumere il valore j . In pratica, abbiamo imposto che j sia l'unico valore non ammissibile che può essere materializzato. A questo punto, se l'indice vale j , viene imposto ad esempio un accesso in memoria innocuo (i.e. alla prima locazione di memoria subito dopo la tabella vengono inserite delle informazioni non sensibili in modo

tale che, anche speculativamente, viene acceduta o una entry della tabella o l'unica locazione di memoria ammissibile e innocua al di fuori della tabella).

Come funziona - for dummies



Normalmente avremmo “indici OK” (verde) ed “indici NON OK” (rosso). Però se usassi questi indici NON OK, potrei andare in zone delicate. Introduco la “zona nera”, più ampia. Tutto quello che cade li dentro lo butto nella zona rossa, che sarà la più piccola possibile, per limitare i danni.

Come lo faccio? Applicando una maschera di bit.

Perchè devo differenziare zona rossa e nera?

Perchè se lascio solo zona rossa posso andare in zone brutte. Se questa è piccola (ad esempio un indirizzo non critico) e faccio si che tutti gli altri indici *NON OK* vadano lì dentro, limito i danni.

Perchè non voglio che applicando questa maschera ad un indice NON OK, questa mi porti ad una zona verde. Quindi prima applico la maschera, e poi faccio il controllo.

Loop unrolling

Ricordiamo che i salti sono azzardi, sbagliare salto compromette performance (squash pipelin) e sicurezza. E se riducessimo i salti?

```
int s=0;
for (int i=0; i<16; i++)
{ s+=i;}
```

Questo ciclo si traduce nella seguente sequenza di istruzioni assembly (dove la sintassi adottata è AT&T):

400545:	8b 45 fc	mov	-0x4(%rbp), %eax
400548:	01 45 f8	add	%eax, -0x8(%rbp)
40054b:	83 45 fc 01	addl	\$0x1, -0x4(%rbp)
40054f:	83 7d fc 0f	cmpb	\$0xf, -0x4(%rbp)
400553:	7e f0	jle	400545 <main+0x18>

Come si può notare, a ogni iterazione del ciclo vengono eseguite cinque istruzioni, di cui tre di controllo (in blu scuro) e solo due di lavoro effettivo, in cui viene incrementata la variabile **s** (in nero): in pratica, in questo particolare esempio, abbiamo un overhead di computazione pari ai 3/5 delle istruzioni totali, il che porta inevitabilmente a un degrado delle prestazioni. Per questo motivo, corre in aiuto il **loop unrolling**, che è una tecnica per “srotolare” i cicli: da una parte si riduce il numero di iterazioni che devono essere eseguite, dall'altra, all'interno di ciascuna iterazione, si esegue il lavoro utile più volte. In tal modo, si riduce il numero di volte in cui devono essere eseguite le istruzioni di controllo. Per effettuare l'unroll in modo automatico, è possibile ricorrere alle seguenti direttive di C:

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")
//region to unroll
#pragma GCC pop_options
```

È anche possibile specificare esplicitamente il fattore di unroll mediante **#pragma unroll (N)**. Ma affinché queste direttive siano effettivamente attive, è necessario compilare il file C col flag **-O**.

Attenzione: il loop unrolling porta alla necessità di utilizzare un maggior numero di registri per eseguire tutte le operazioni della medesima iterazione. Inoltre, porta ad avere le istruzioni macchina del ciclo su un range di indirizzi più ampio, per cui si ha una minore località. Per questi motivi, non è una tecnica che può essere sfruttata in modo spregiudicato. Per sfruttarlo bene, bisogna capire architettura e obiettivi!

Power wall

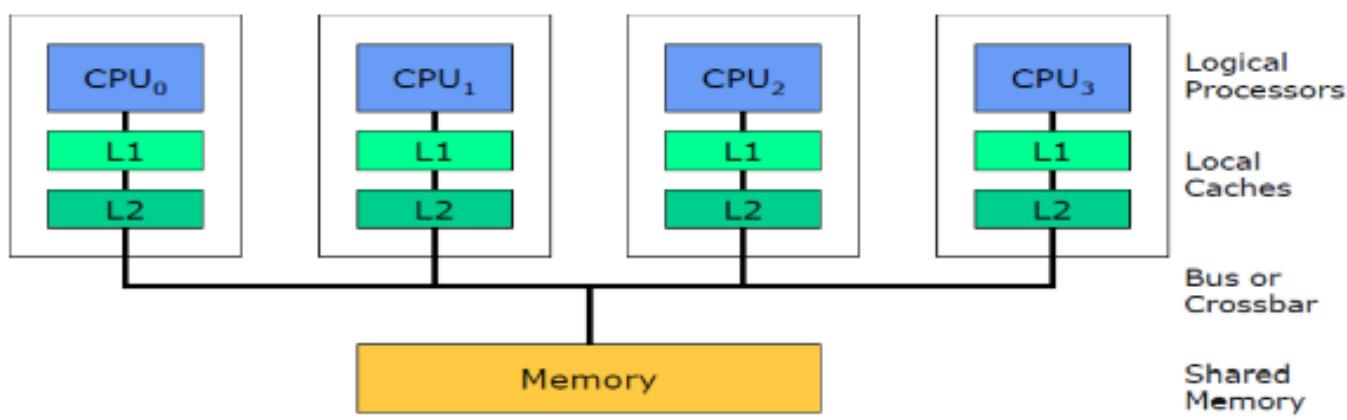
Per aumentare le prestazioni dei processori, è possibile seguire due approcci:

1. Aumentare la frequenza del clock.
2. Incrementare i componenti hardware a supporto del processore.

Per quanto riguarda la strategia 1, c'è una limitazione: la dissipazione massima che un processore può tollerare senza che si bruci è 130 W. Ma la dissipazione è pari a $V \cdot V \cdot F$, dove V è il voltaggio ed F è la frequenza del clock. V non può essere al di sotto di una certa soglia minima, altrimenti i componenti hardware non funzionerebbero proprio. Di conseguenza, esiste un upper-bound per F che è stato già raggiunto. Questa limitazione è detta **power wall**. A causa del power wall, ad oggi siamo obbligati a seguire la strategia 2 per rendere più efficienti i processori. In particolare, nel tempo, sono state adottate le soluzioni architetturali descritte di seguito.

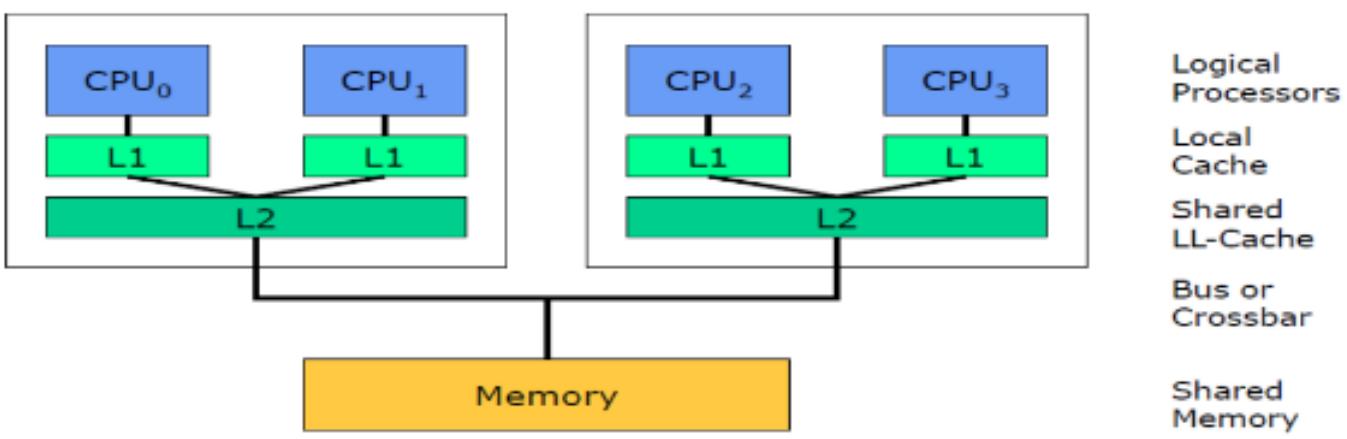
Symmetric Multiprocessors

Si hanno semplicemente molteplici processori, ciascuno dei quali, per accedere alla memoria, impiega la stessa quantità di tempo. Abbiamo per ogni core un thread.



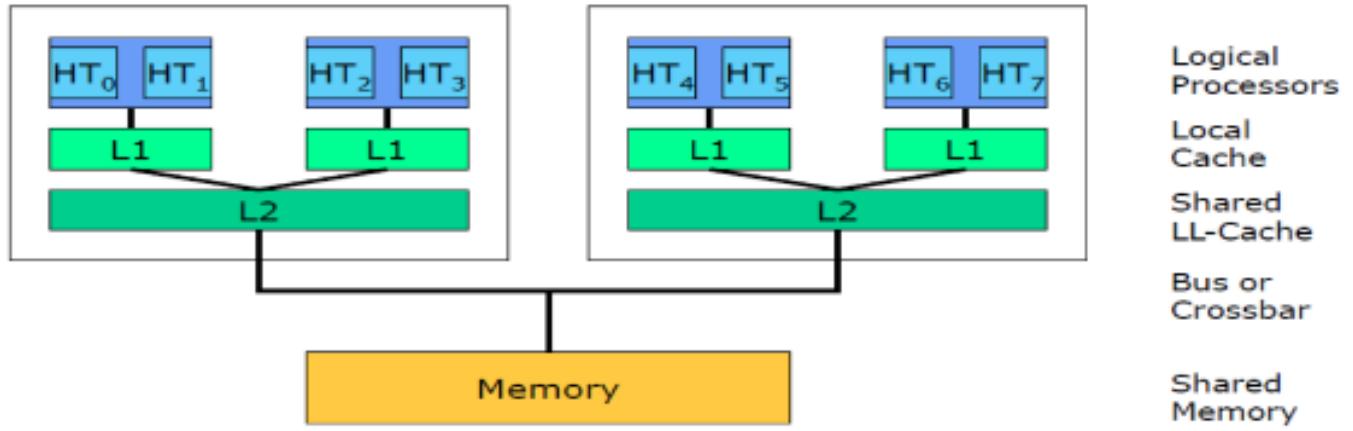
Chip Multi Processor (CMP) o Multicore

All'interno di ciascun processore si hanno più motori (più core). Un hypethread per core.



Symmetric Multi Threading (SMT) o Hyperthreading

All'interno di ciascun core possono esserci **più hyperthread distinti**. Dal punto di vista del sistema operativo, è esattamente come se ci fosse un numero di processori pari al numero totale di hyperthread. Il problema qui è che la memoria rappresenta un collo di bottiglia importante, anche perché viene acceduta in modo concorrente da tutti gli hyperthread. Dobbiamo vederla come architettura distribuita. La memoria è esposta in ISA (ma non le componenti cache $L_1, L_2 \dots$).



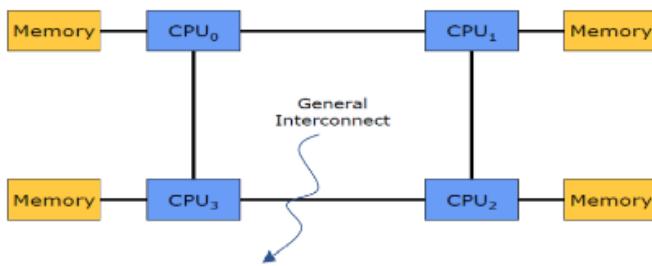
Successivamente si è passati all'**UMA**: memoria unica ad accesso uniforme, cioè stesse componenti hardware accedute in parallelo. Poi **NUMA**.

Non Uniform Memory Access (NUMA):

Qui la memoria è divisa in banchi, ciascuno dei quali è direttamente collegato a uno specifico core (o a un insieme di core). Se un thread che gira in CPU-core i accede al banco di memoria a esso vicino, l'interazione con la memoria risulta molto efficiente, mentre se deve accedere a un altro banco di memoria, impiegherà più tempo. Tale soluzione risulta vantaggiosa nel momento in cui si riesce a fare in modo che ciascun CPU-core acceda prevalentemente al *proprio* banco di memoria (ovvero effettui prevalentemente degli accessi locali). Ciascuna **coppia** (CPU-core i , banco di memoria i) o (insieme i di CPU-core, banco di memoria i) costituisce un **nodo NUMA**. I thread possono leggere da indirizzi diversi (quello che abbiamo detto prima), tuttavia l'interconnect è time shared, si genera traffico, che deve essere ben gestito.

Osservazione:

- Un accesso locale (CPU verso memoria adiacente) richiede un tempo pari a $50(hit) + 200(miss)$ cicli, quindi è anche facile capire, osservando il tempo, se si ha hit o miss.
- Se l'accesso è NON locale e SENZA TRAFFICO, si passa a $200(hit) + 300(miss)$. Tempi totalmente diversi, che possono creare **problemi di coerenza nella cache**.



NB: ISA definisce come il processore interpreta e esegue le istruzioni, ma non specifica i dettagli sulla gerarchia di memoria o la presenza delle cache.

Cache Coherency

Com'è possibile osservare, nelle architetture di memoria elencate precedentemente la cache è **replicata**: da qui sorge il problema della **cache coherency**, secondo cui bisogna stabilire qual è il *valore corretto da restituire a un thread che effettua una determinata lettura in memoria*. Disponiamo infatti di un processore e di una zona di memorizzazione (cache e RAM). Osservare solo l'interfaccia memoria-processore ci fornisce una visione limitata. Se una istruzione scrive una **mov**, questa passa prima per lo *store buffer*, non viene subito esposto nell'ISA. Se un altro programma dovesse leggere, non è detto che il valore sia già stato scritto. La coerenza viene **definita** in tre punti fondamentali:

- **Causal consistency**: se un processore p_1 scrive un valore v nella locazione di memoria X e poi effettua una lettura da X , dovrà leggere il valore v , a meno che non ci siano altri processori che nel frattempo hanno finalizzato

delle scritture concorrenti su X (coerenza di tipo RAW – Read After Write).

- **Avoidance of staleness:** se un processore p_1 scrive un valore v nella locazione di memoria X (memoria qui intesa come RAM, visibile su ISA) e *dopo una quantità sufficiente di tempo* un altro processore p_2 effettua una lettura da X , p_2 dovrà leggere il valore v , a meno che non ci siano altri processori che nel frattempo hanno finalizzato delle scritture concorrenti su X .
- **Avoidance of inversion of memory updates:** le varie scritture su uno stesso dato X devono essere viste da tutti i processori nello stesso ordine. Se ho una sequenza di scrittura dalla cache verso la RAM, l'ordine di scrittura in cache deve essere riproposto anche nella RAM. Queste proprietà valgono sulla singola locazione.

Le due principali tecniche utilizzate per mantenere la consistenza, ricordando che la copia master è in memoria, e le copie in cache, sono:

- **Write through cache:** Prima aggiorniamo in cache e poi in memoria. Questo porta al seguente problema: una CPU potrebbe leggere due volte lo stesso dato dalla cache, e tra le due letture un'altra CPU può aver toccato il dato e aggiornato in memoria. Quindi per la CPU in esame, abbiamo una incoerenza tra quello che c'è in cache e quello che c'è in memoria.
- **Write back cache:** Aggiornare la cache e procedere alla memorizzazione in un secondo momento che decidiamo noi. Il problema è che lasciar decidere a noi potrebbe portare a scritture in memoria non nello stesso ordine di come sono state effettivamente eseguite.

Protocolli CC Cache Coherency

Permettono di conseguire la cache coherency e sono il risultato delle scelte di:

- Un insieme di transazioni (e.g. aggiornamento di una replica della cache) supportate dal cache system distribuito.
- Un insieme di stati per i cache block (= unità minime della memoria che possono essere portate all'interno della cache).
- Un insieme di eventi che capitano all'interno del cache system distribuito.
- Un insieme di transizioni di stato.

Il design dei protocolli CC dipende da svariati fattori come:

- La topologia dei componenti (e.g. single bus, gerarchica, ring-based).
- Le primitive di comunicazione (i.e. unicast, multicast, broadcast).
- Le cache policy (e.g. write-back, write-through).
- Le feature della gerarchia di memoria (e.g. numero di livelli della cache, inclusiveness).

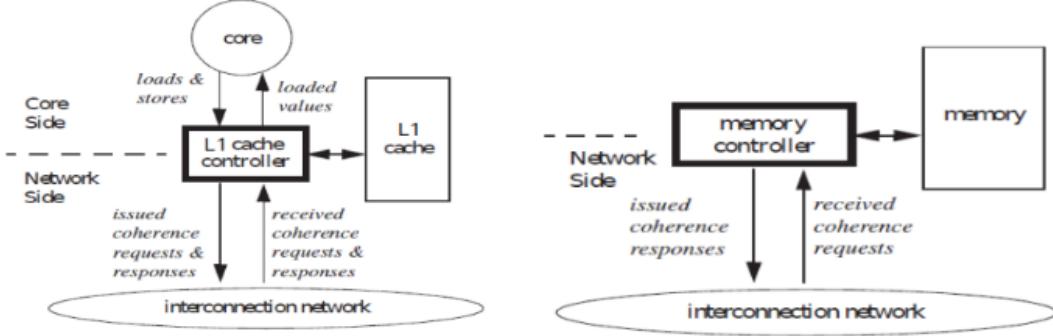
Implementazioni di cache coherency differenti possono presentare prestazioni diverse in termini di:

- **Latenza:** tempo per completare una singola transazione.
- **Throughput:** numero di transazioni completate per unità di tempo; aumenta se si fa in modo che scritture su aree di memoria diverse (e sufficientemente distanti) da parte di thread differenti avvengano in modo concorrente.
- **Overhead spaziale:** numero di bit richiesti per mantenere lo stato di un cache block.

Esistono due famiglie principali di protocolli CC:

- **Invalidation protocols:** quando un CPU-core scrive su una copia di un blocco, qualsiasi altra copia del medesimo blocco (i.e. della medesima informazione) viene *invalidata*: soltanto chi ha scritto l'ultima informazione ha la versione aggiornata del blocco. In tal caso, si utilizza poca banda ma si ha una latenza elevata per gli accessi in lettura: infatti, un CPU-core che si trova vicino a una replica invalidata, per andare a leggere il dato, ha la necessità di sfruttare l'unica replica valida, che è lontana. Sostanzialmente se aggiorno una linea di cache, spariscono tutte le altre repliche. Leggeranno tutti da me, perchè ho l'unica copia valida.
- **Update protocols:** quando un CPU-core scrive su una copia di un blocco, la nuova informazione viene propagata su tutte le altre copie del medesimo blocco. In tal caso, si ha una bassa latenza per gli accessi in lettura ma utilizza molta più banda.

Per poter invalidare / modificare una replica di un blocco che è stato aggiornato, si ricorre al cosiddetto **Snooping cache**: le componenti della cache e della memoria sono agganciati a un **broadcast medium** (= interconnection network, è un canale di comunicazione broadcast) tramite un controller, che ha la responsabilità di osservare le transizioni di stato degli altri componenti e di far reagire il componente locale di conseguenza (appunto invalidando o modificando la replica locale del blocco aggiornato). Naturalmente, il controller utilizza il broadcast medium anche per trasmettere gli aggiornamenti che avvengono in un blocco di cache locale. Mediante il broadcast medium siamo in grado di serializzare le transizioni di stato. *Inoltre, una transizione di stato non può occorrere finché il broadcast medium non viene acquisito in uso dal controller.*



Nel mondo reale viene adottato lo *Snooping cache* accoppiato con un protocollo basato sull'invalidazione. Vediamo dunque nel dettaglio alcuni di questi protocolli basati sull'invalidazione. Ad esempio, una componente di cache L_1 ha una replica r , vuole eseguire operazione su tale replica mediante il richiamo di broadcast medium, in cui annuncio alle altre componenti che, chi ha una copia della replica r , deve aggiornare. Nel mentre, nessuno può eseguire altri aggiornamenti se ho io il broadcast medium. Ciò crea un problema di scalabilità, se lavoro con tanti dati e repliche.

Osservazione: devo per forza aggiornare le repliche? E se non mi servissero? Ad esempio, un thread su CPU_0 con cache L_1 vi esegue un aggiornamento su linea di memoria e comunica in maniera distribuita di cancellare quella linea di cache. Ma se quella linea di cache fosse già invalida per altri?

Si mantiene traccia di queste operazioni mediante:

Protocollo MSI (Modified-Shared-Invalid)

È un protocollo in cui qualsiasi transazione di scrittura su una **copia di un cache block** invalida tutte le altre copie del cache block. Quando dobbiamo servire delle transazioni di **lettura**:

- Se la policy della cache è **write-through**, recuperiamo semplicemente l'ultima copia aggiornata dalla *memoria*.
- Se la policy della cache è **write-back**, recuperiamo l'ultima copia aggiornata o dalla memoria o da un altro componente di *caching*.

In questo protocollo è necessario tenere traccia dello stato della copia di un blocco:

- Si trova nello stato *modified* se è appena stata sovrascritta.
- Si trova nello stato *shared* se esistono dei CPU-core che stanno leggendo altre copie del medesimo blocco (per cui esistono più copie *valide* di tale blocco).
- Si trova nello stato *invalid* se un'altra copia del medesimo blocco è appena stata sovrascritta.

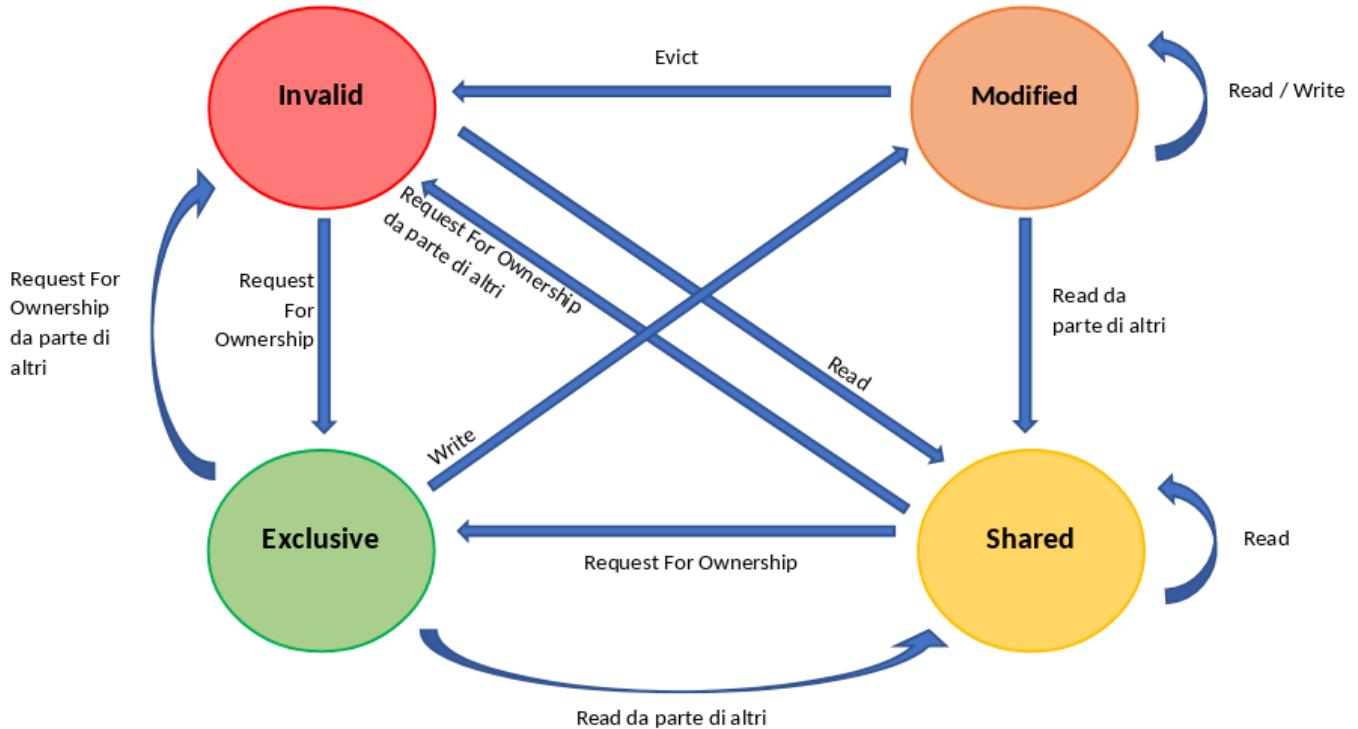
Il **problema** di MSI risiede nel fatto che richiede l'invio di un messaggio di invalidazione in broadcast (tramite il broadcast medium) ogni volta che una copia di un blocco viene modificata (e questo anche in caso di scritture successive sulla medesima copia). Questo può portare a impegnare massivamente il broadcast medium anche quando non ce n'è bisogno, perché magari tutte le altre copie erano nello stato *invalid* già da prima. Per ovviare a tale inconveniente, si ricorre ai protocolli descritti successivamente.

Protocollo MESI (Modified-Exclusive-Shared-Invalid):

Rispetto a MSI, prevede uno stato in più, che è **exclusive**. In pratica, un CPU-core i che vuole apportare delle modifiche alla propria copia di un blocco deve richiedere l'**ownership** (*l'esclusività*) di quel blocco; questa è l'unica occasione in cui CPU-core i utilizza il broadcast medium per comunicare agli altri nodi che devono *invalidare* la propria copia del blocco. Una volta che la copia è nello stato *exclusive*, CPU-core i può modificarla tutte le volte che vuole *senza dover comunicare più nulla* a nessuno, finché un altro CPU-core non richiede di effettuare una lettura da un'altra

copia dello stesso cache block (momento in cui la copia passa allo stato *shared*). L'automa raffigurato di seguito descrive in modo completo il funzionamento del protocollo MESI. Schematizzando:

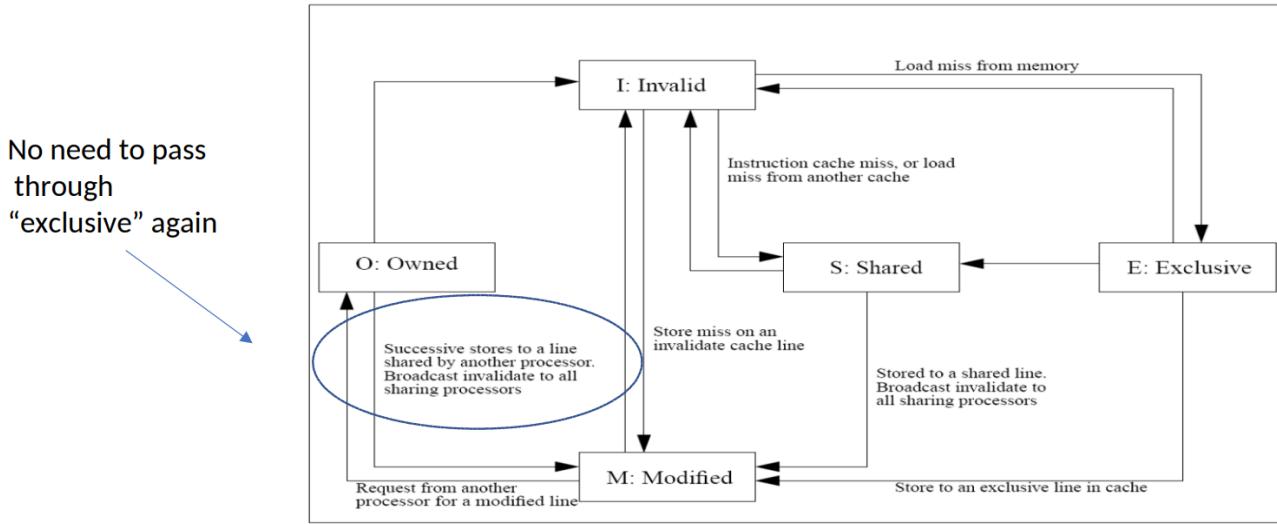
- **Modified:** Ho modificato un dato shared.
- **Exclusive:** Sono il solo proprietario del dato, posso modificarlo liberamente, senza bus message (passando a Modified).
- **Shared:** Ho una copia del dato che un altro processo possiede.
- **Invalid:** La mia copia del dato non è aggiornata.



Ogni volta che si esce dallo stato modified, si effettua il *write back* in memoria delle nuove informazioni che sono state scritte. Se volessi solamente leggere? Chi ha l'informazione si trova in *exclusive* e poi passa a *shared*. Potrei introdurre un quinto stato per evitare queste due transizioni. La linea exclusive è a mio uso esclusivo. Solo lo stato Invalid crea interazioni distribuite. Lo stato Modified è importante per cache write back, ma non devo informare tutti.

Protocollo MOESI (Modified-Owned-Exclusive-Shared-Invalid):

Rispetto a MESI, prevede un ulteriore stato in più, che è **owned**. In particolare, una copia di un cache block passa dallo stato modified allo stato *owned* (anziché shared) nel momento in cui stava per essere sovrascritta ma un'altra copia dello stesso blocco viene letta. *Owned* significa che la copia è tuttora in fase di aggiornamento ma, nel frattempo, altre copie dello stesso blocco vengono lette: se devono occorrere nuovi aggiornamenti quando si è nello stato *owned*, non ci si deve preoccupare di chiedere nuovamente l'esclusività del blocco, bensì si passa direttamente allo stato modified, invalidando tutte le altre copie. Di seguito è rappresentato l'automa completo del protocollo MOESI.



Nello stato **Owned** io ho l'uso esclusivo di un dato. Se mi chiedono di condividerlo, gli altri **non** possono scriverci. Tale stato ci permette di transitare tra più stati, in quanto non devo riacquisire l'esclusività.

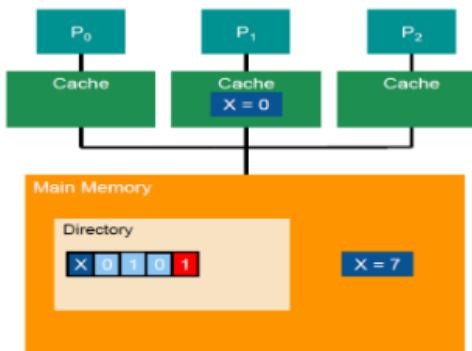
Protocolli directory based:

In realtà, i protocolli basati su snooping cache non scalano poiché le transazioni portano a una comunicazione broadcast. Per questo motivo sono stati introdotti i protocolli directory based, in cui i vari aggiornamenti possono essere point-to-point tra i vari CPU-core / processori. In particolare, supponiamo di avere un sistema con N processori P_0, P_1, \dots, P_{N-1} . Per ciascun blocco di memoria si mantiene una directory entry, composta da:

- N bit di presenza (l'i-esimo bit è impostato a 1 se il blocco si trova nella cache di P_i).
- 1 dirty bit, che indica se il blocco è stato **modificato** senza che gli aggiornamenti siano stati riportati in memoria.

Se dirty bit = 0 e ho più possessori, linea cache condivisa.

Dirty bit = 1 se qualcuno lo scrive, però per passare a condiviso deve ritornare a 0. E' una maschera che mi dice chi può fare cosa.



- **Caso 1:** P_0 vuole *leggere* il blocco X, ha un cache miss e il dirty bit è pari a 0. X viene letto dalla memoria, il bit **presence[0]** viene settato a 1 e i dati vengono consegnati al lettore.
- **Caso 2:** P_0 vuole *leggere* il blocco X, ha un cache miss e il dirty bit è pari a 1. X viene richiamato dal processore P_j tale che **presence[j]=1**. Dopo che il dirty bit viene settato a 0, il blocco viene condiviso e il bit **presence[0]** viene settato a 1. Infine, i dati vengono consegnati al lettore e propagati in memoria.
- **Caso 3:** P_0 vuole *scrivere* sul blocco X, ha un cache miss e il dirty bit è pari a 0. P_0 invia un messaggio di invalidazione non a tutti gli altri processori, bensì solo a quelli col bit **presence[j]** pari a 1. Dopo che tali **presence[j]** vengono settati a 0, e **presence[0]** e il dirty bit vengono impostati a 1.
- **Caso 4:** P_0 vuole *scrivere* sul blocco X, ha un cache miss e il dirty bit è pari a 1. X viene richiamato dal processore P_j tale che **presence[j]=1**. Dopo che **presence[j]** viene settato a 0 e **presence[0]** viene impostato a 1.

Implementazioni x86

Intel:

- Principalmente **MESI**.
- **Cache inclusive** (tutte le informazioni che si trovano in un particolare livello della memoria sono riportate anche in tutti i livelli *inferiori* della memoria).
- **Write back** (gli aggiornamenti effettuati su un blocco di cache B verranno riportati in memoria solo quando B dovrà essere rimosso dalla cache - “evicted”).
- Cache L1 composta da **linee (blocchi) di 64 byte**.

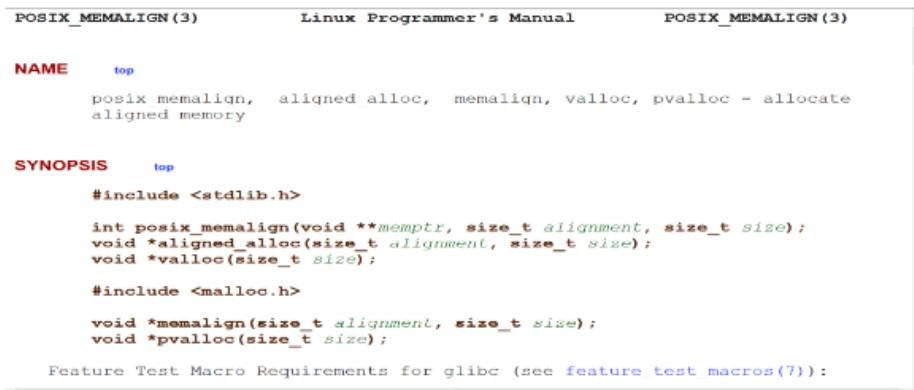
AMD:

- Principalmente **MOESI**.
- **Cache non inclusive** (o esclusive), in particolare al livello L3 (non è detto che la cache L3 contenga tutti i blocchi presenti in cache L1 e in cache L2, magari dato sale in L1 ma non è detto che sia in L3); qui la cache L3 è utile per ospitare i blocchi che devono essere rimossi dai livelli superiori della cache. Sicuramente avere cache non inclusive rende il sistema meno vulnerabile ad attacchi di tipo side-channel che si basano sull'utilizzo della cache.
- **Write back**.
- Cache L1 composta da **linee (blocchi) di 64 byte**.

False cache sharing

Supponiamo di avere un thread A che deve eseguire delle operazioni di scrittura su un dato X , e supponiamo di avere un thread B che deve eseguire delle operazioni di scrittura su un dato Y , dove X e Y sono *independent* tra loro. Idealmente, i due thread devono poter effettuare le loro scritture in modo concorrente. Se però X e Y si trovano sulla **stessa linea di cache**, abbiamo che A, per scrivere su X , deve richiedere in uso esclusivo tutti i 64 byte che costituiscono il blocco e, quindi, anche Y ; d'altra parte, B, per scrivere su Y , deve richiedere in uso esclusivo l'intero blocco e, quindi, anche X . Questo scenario, noto come **false cache sharing**, porta all'inondazione di transazioni distribuite dovuta a un ping-pong tra A e B di *Request For Ownership* per la medesima linea di cache: da qui consegue un crollo delle performance. Tale effetto deleterio lo avremmo avuto anche nel caso in cui A avesse dovuto eseguire delle operazioni di scrittura su X mentre B debba eseguire delle operazioni di lettura su Y .

Comunque sia, dobbiamo essere attenti anche allo scenario duale: se il thread A deve eseguire delle operazioni correlate tra loro sia sul dato X che sul dato Y , stavolta è opportuno avere X e Y sulla stessa linea di cache: in tal modo, diminuiamo la probabilità di ritrovarci dei dati scorrelati all'interno del blocco in cui si trovano X e Y .



The screenshot shows the Linux Programmer's Manual page for `POSIX_memalign(3)`. The page is titled "POSIX_memalign (3)" and includes sections for NAME, SYNOPSIS, and DESCRIPTION. The NAME section lists `posix_memalign`, `aligned_alloc`, `memalign`, `valloc`, and `pvalloc` as functions that allocate aligned memory. The SYNOPSIS section provides the header files and function prototypes:

```
#include <stdlib.h>
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
void *valloc(size_t size);

#include <malloc.h>
void *memalign(size_t alignment, size_t size);
void *pvalloc(size_t size);
```

At the bottom, it notes: "Feature Test Macro Requirements for glibc (see `feature_test_macros(7)`):".

Se $\text{ALIGNMENT} = 64$, sto allineando esattamente a una linea di cache. Parto sicuramente da li, poi in base alla size potrei eccedere.

Attacco Flush + Reload

È l'antenato degli attacchi Meltdown e Spectre visti precedentemente ed è stato documentato per la prima volta nel 2013. In quest'attacco si hanno due thread: A (la vittima) e B (l'attaccante); che hanno la possibilità di accedere alle medesime informazioni in memoria. B può eseguire il **flush** della cache per poi eseguire degli accessi cronometrati in memoria (**reload**): se per leggere un dato impiega poco tempo, vuol dire che, nel frattempo, quello stesso dato è stato

ricaricato in cache da parte di A (e quindi viene usato da A). Questo meccanismo può essere applicato anche sulle istruzioni macchina: anch'esse, quando vengono accedute per essere eseguite, vengono caricate in cache. Ciò vuol dire che B può essere in grado di capire anche quali sono le attività che A sta svolgendo. **Nell'ISA, l'unica operazione esposta per la cache è il FLUSH. Non posso fare altro!**

L'implementazione di Flush+Reload su x86 è basata su due building block:

- Un timer ad alta risoluzione. (**RDTSC**, *32 bit edx, 32 bit eax* (tot 64)per ogni processore).
- Un'istruzione non privilegiata che effettui il flush della cache. (quindi togliere contenuto cache con cflush, dando l'indirizzo della memoria).

Le immagini riportate di seguito mostrano i dettagli di queste due istruzioni.

RDTSC

Read Time-Stamp Counter

Opcode	Mnemonic	Description
0F 31	RDTSC	Read time-stamp counter into EDX:EAX.
Description		
Loads the current value of the processor's time-stamp counter into the EDX:EAX registers. The time-stamp counter is contained in a 64-bit MSR. The high-order 32 bits of the MSR are loaded into the EDX register, and the low-order 32 bits are loaded into the EAX register. The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 15 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3 for specific details of the time stamp counter behavior.		
When in protected or virtual 8086 mode, the time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the TSD flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDTSC instruction is always enabled.) The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.		
The RDTSC instruction is not a serializing instruction. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.		
This instruction was introduced into the IA-32 Architecture in the Pentium processor.		
Operation		
<code>if(CR4.TSD == 0 CPL == 0 CR0.PE == 0) EDX:EAX = TimeStampCounter; else Exception(GP(0)); //CR4.TSD is 1 and CPL is 1, 2, or 3 and CR0.PE is 1</code>		
Flags affected		
None.		

CLFLUSH

Flush Cache Line

Opcode	Mnemonic	Description
0F AE /7	CLFLUSH m8	Flushes cache line containing m8.
Description		
Invalidates the cache line that contains the linear address specified with the source operand from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. The source operand is a byte memory location.		
The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (bit 19 of the EDX register, see Section , CPUID-CPU Identification). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).		
The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, <i>data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line</i>).		
CLFLUSH is only ordered by the MFENCE instruction. It is not guaranteed to be ordered by any other fencing or serializing instructions or by another CLFLUSH instruction. For example, software can use an MFENCE instruction to insure that previous stores are included in the writeback.		
The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and, in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.		
The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.		

- L'istruzione **clflush** accetta come unico parametro una locazione di memoria, che identifica il blocco di memoria da rimuovere dalla cache.
- È bene utilizzare **clflush** subito dopo l'invocazione all'istruzione **mfence** (che approfondiremo più avanti): per ora basti pensare che **mfence** fa da **barriera** tra tutte le istruzioni che la precedono e tutte le istruzioni che la seguono. In tal modo, siamo sicuri che **clflush** mandi in write back e *rimuova davvero le informazioni che erano state portate in cache*. Infatti, anche se **committo** l'istruzione, il dato non va in cache, ma nello store buffer. **“Forzo” con mfence ASM, che porta il dato dallo store buffer in cache**. In caso contrario, anche se siamo sicuri che **clflush** venga committata dopo l'istruzione i di accesso alla memoria, **clflush** potrebbe

intervenire prima che la memoria cambi realmente stato per effetto dell’istruzione *i*. (Quindi svuotiamo una cache già vuota, perchè dentro non è stato ancora messo nulla!)

Questo è un problema di **memory consistency**, che analizzeremo successivamente.

ASM inline

È un modo per utilizzare la tecnologia assembly all’interno di un programma scritto in C. È molto utile se si vogliono incapsulare alcune istruzioni machine-dependent all’interno del programma. La sintassi è la seguente:

```
__asm__ [volatile][goto] (AssemblerTemplate
    [ : OutputOperands ]
    [ : InputOperands ]
    [ : Clobbers ]
    [ : GotoLabels ]);
```

- **AssemblerTemplate** = blocco di istruzioni assembly che si vuole inserire all’interno del flusso di esecuzione di una funzione C. Scritto in ISA, lavoro con memoria e registri, quindi niente comandi C, come *int c*.
- **OutputOperands** = blocco dell’ASM inline in cui è possibile specificare in quali variabili devono essere caricati i valori di determinati registri dopo l’esecuzione dell’AssemblerTemplate (post-movimento dati).
- **InputOperands** = blocco dell’ASM inline in cui è possibile specificare in quali registri devono essere caricati i valori di determinate variabili prima dell’esecuzione dell’AssemblerTemplate (pre-movimento dati). Entrambi servono per collegare il codice C all’assembly.
- **Clobbers** = registri di CPU che si vogliono memorizzare sullo stack prima dell’esecuzione dell’AssemblerTemplate e che si vogliono **ripristinare** dopo l’esecuzione dell’AssemblerTemplate. Mi basta specificare quali, senza preoccuparmi di altro.
- **GotoLabels** = istruzioni di jump presenti all’interno dell’AssemblerTemplate e che hanno come destinazione altri punti del programma (fuori dall’AssemblerTemplate). Questo blocco dell’ASM va specificato sempre assieme al prefisso **goto** (che viene posto subito prima delle parentesi tonde). Sarebbero i costrutti del tipo **fun_x: ...** Che troviamo in assembly.
- **Volatile** = prefisso opzionale che indica che il compilatore non dovrà attuare alcuna *ottimizzazione* sul codice assembly specificato nell’AssemblerTemplate (quindi non deve esserci il re-ordering delle istruzioni e così via).

Direttive di compilazione C per gli operandi

Il simbolo = all’interno dell’ASM inline significa che l’operando deve essere utilizzato come output. Se invece un operando non è associato al simbolo =, allora verrà utilizzato come input.

Possibili operandi per l’ASM

- **r** = registro generico che verrà scelto dal compilatore. Ha un indice associato, quindi è “recuperabile”.
- **m** = locazione di memoria generica che verrà scelta dal compilatore.
- **i/I** = operando a 64/32 bit immediato.
- **a** = eax (o una sua variante come rax, ax, al dipendentemente dalla dimensione degli operandi).
- **b** = ebx (o una sua variante come rbx, bx, bl dipendentemente dalla dimensione degli operandi).
- **c** = ecx (o una sua variante come rcx, cx, cl dipendentemente dalla dimensione degli operandi).
- **d** = edx (o una sua variante come rdx, dx, dl dipendentemente dalla dimensione degli operandi).
- **S** = esi.
- **D** = edi.
- **0-9** = indici degli operandi (e.g. se a è il primo operando a essere utilizzato all’interno dell’ASM inline, può essere identificato con l’indice 0).
- **q** = registro che può essere utilizzato per indirizzare un singolo byte (e.g. **eax** che incapsula **a1**).

Esempio di utilizzo di ASM inline

Vediamo un esempio di funzione che può essere invocata all’interno di un loop e ha lo scopo di misurare il tempo impiegato per effettuare un accesso in memoria in caso di cache miss.

```

unsigned long probe (char* adrs) {
    volatile unsigned long cycles;
    asm ( "mfence \n" //barriera per qualunque accesso in memoria
          "lfence \n" //barriera per gli accessi in memoria di tipo load
          "rdtsc \n" //preleva il timer da un registro specifico e
                      //carica i 32 bit meno significativi in eax e
                      //i 32 bit più significativi in edx
          "lfence \n" "movl %%eax, %%esi \n" //carica i 32 bit meno
                                              //significativi del timer in esi
          "movl (%1), %%eax \n" //effettua l'accesso in memoria
                                  //in particolare alla variabile adrs,
                                  //salvata in %1=ecx)
          "lfence \n"
          "rdtsc \n" //preleva il nuovo timer
          "subl %%esi, %%eax \n" //sottrae i 32 bit meno significativi
                                  //dei due timer, che corrisponde al
                                  //tempo impiegato per l'accesso in memoria
          "clflush 0 (%1) \n" //effettua il flush di adrs dalla cache
          : "=a" (cycles) //dopo l'esecuzione dell'ASM inline, il contenuto
                           //di eax (registro 0) andrà nella variabile cycles
          : "c" (adrs) //prima dell'esecuzione dell'ASM inline, il contenuto
                           //di adrs va nel registro ecx (registro 1)
          : "%esi", "%edx"); //esi, edx sono i clobbers
    return cycles;
}

```

Eliminando la sola istruzione `clflush`, è possibile misurare il tempo impiegato per effettuare gli accessi in memoria in caso di cache hit.

*NB*₁ : gli attacchi Meltdown e Spectre utilizzano proprio questo costrutto qui.

*NB*₂: esistono delle API di C che implementano l'invocazione di `rdtsc` e `clflush`, e sono rispettivamente `_rdtscp()` e `_mm_clflush()`.

*NB*₃ : i registri `eax`, `ecx` non sono stati inseriti tra i clobbers perché sono **caller save**: sono registri che, a ogni chiamata di funzione, vengono salvati dalla funzione chiamante, cosicché la funzione chiamata possa utilizzarli a piacimento.

Altri dettagli su Flush + Reload È possibile rendere l'istruzione `rdtsc` privilegiata, ovvero si può fare in modo che venga invocata esclusivamente a livello kernel. Tuttavia, l'attacco **Flush+Reload**, per andare a buon fine, non ha la stretta necessità di sfruttare `rdtsc` per tenere traccia del trascorrere del tempo: esiste un modo altresì efficace per emulare un timer a grana sufficientemente fine. In pratica è sufficiente creare un nuovo thread che dovrà eseguire la seguente funzione:

```

void *time_keeper (void *dummy) {
    while(1){
        timer = (timer+1); //timer è una variabile globale
    }
}

```

NB: una variabile volatile è toccabile da più entità, non è nel registro del singolo thread. Evitiamo quindi ottimizzazioni, prendo e scrivo in memoria.

Un'altra cosa che vale la pena osservare è che questo tipo di attacco ha una probabilità di gran lunga minore di avere successo nel caso in cui si abbiano le *cache non inclusive*. Supponiamo che il thread T_A della vittima giri sul CPU-core_A e il thread T_B dell'attaccante giri sul CPU-core_B. Se le cache sono *non inclusive*, qualunque blocco di memoria acceduto da parte di T_A viene caricato sulla cache di livello 1 propria di CPU-core_A ma magari non sulle cache di livelli inferiori. A tal punto, T_B non può vedere in alcun modo sulla cache il valore utilizzato dal thread T_A : al livello L1 non è presente perché ciascun CPU-core ha la sua copia privata della cache di livello L1; ai livelli sottostanti, invece, *il dato non è proprio presente*. Se lavoro su una cache in uso *esclusivo*, cioè ho ad esempio due core sulla stessa cpu che condividono risorse, le tempistiche sono più veloci in quanto non mi sposto tra stati diversi.

Memory Consistency

Con riferimento al *memory wall*, se imponessimo che tutte le istruzioni che prevedono un'interazione con la memoria effettuino l'accesso vero e proprio alla memoria immediatamente, avremmo un degrado delle prestazioni, dovuto al fatto che l'*accesso alla memoria richiede molti cicli di clock*;

Per esempio, se abbiamo un'istruzione A che scrive un dato sulla memoria e un'istruzione B che legge il medesimo dato dalla memoria, B deve attendere che il dato sia disponibile in cache o in RAM prima di essere finalizzata, allora le ottimizzazioni effettuate sulla pipeline vanno vanificate. Di conseguenza, è necessario un meccanismo che disaccoppi le visioni di:

- Come (e in quale ordine) le istruzioni di accesso alla memoria vengono viste dal **processore che le esegue (program order)**
- Come (e in quale ordine) le istruzioni di accesso alla memoria vengono viste dagli **altri processori** / program flow mediante l'utilizzo delle risorse condivise nell'architettura di memoria, come la cache o la RAM (**visibility order**).

Tale meccanismo prevede l'utilizzo degli **store buffer**, che sono delle piccole aree di **memoria privata per una CPU-core**, che mantengono temporaneamente i dati prodotti dalle istruzioni finalizzate ma ancora non riportate in cache o in RAM. In particolare, l'ordine con cui saranno viste le operazioni in memoria da parte degli altri program flow non corrisponde necessariamente con l'ordine in cui le istruzioni sono uscite dalla pipeline, e tale ordine dipende dal modello di consistenza della memoria utilizzato. *Non stiamo parlando di Cache coherency, perchè qui ancora non sappiamo se la scrittura in cache sia stata effettuata.*

Consistenza sequenziale

Definizione di Lamport, 1979

Un sistema multiprocessore è sequenzialmente consistente se il risultato di una qualunque esecuzione è lo stesso rispetto a se le operazioni di tutti i processori fossero eseguite in qualche ordine sequenziale e le operazioni di ciascun processore preso singolarmente apparissero in tale sequenza esattamente nell'ordine specificato dal suo programma.

In altre parole, per avere consistenza sequenziale, *non è possibile avere un'inversione delle istruzioni di uno specifico program order all'interno della sequenza globale delle operazioni.*

Esempio Supponiamo che CPU-core₁ debba eseguire le operazioni a_1, b_1 (dove a_1 viene prima di b_1 nel *program order*), e supponiamo che CPU-core₂ debba eseguire le operazioni a_2, b_2 (dove a_2 viene prima di b_2 nel *program order*). Allora, una sequenza che rispetta la consistenza sequenziale è data da:

a_1, a_2, b_1, b_2

Invece, una sequenza che non rispetta la consistenza sequenziale è data da: b_1, a_2, b_2, a_1 .

Infatti, tale sequenza vede b_1 prima di a_1 , il che rappresenta un'inversione rispetto all'ordine di programma proprio di CPU-core₁.

Nel mondo reale, la stragrande maggioranza delle macchine hanno dei chipset che non sono realizzati per soddisfare la consistenza sequenziale. Ma come mai questa scelta se la consistenza sequenziale è fondamentale per la correttezza di molte applicazioni concorrenti? Per *motivi di scalabilità* dell'architettura di memoria e di *costi* legati alla sequential consistency; tra l'altro, la consistenza sequenziale impone un ordinamento fisso per tutte le operazioni, anche per quelle completamente scorrelate tra loro. Algoritmi come *Bakery* o *Dekker* non vanno bene per le architetture odierne perché *non lavorano in questo modo*. Come si comportano quindi i computer moderni?

Total Store Order TSO

È il modello di consistenza preferito nelle architetture reali, come x86 e SPARC. È basato sull'idea per cui *effettuare lo store dei dati non è equivalente al riportare effettivamente tali dati nell'architettura di memoria*. Quest'ultima operazione viene tipicamente **ritardata** rispetto al commit dell'istruzione di store; ad esempio, si potrebbe dover attendere che la linea di cache su cui si dovrà scrivere il nuovo valore passi allo stato exclusive.

È proprio questo il modello di consistenza che prevede lo **store buffer**. In particolare, nel momento in cui un'istruzione di **store** viene committata, il contenuto da scrivere poi in memoria viene nel frattempo inserito all'interno dello **store buffer**, che risulta essere un componente intermedio tra il CPU-core e l'architettura di memoria (cache + RAM). Ricordiamo che, dopo aver committato un'istruzione, questa *non passa subito da Store Buffer a Cache*, bensì passa un certo tempo. Solo dopo che va in cache, il risultato di tale istruzione risulta *visibile*.

Non sarebbe meglio andare direttamente in cache, in modo che i dati prodotti siano subito disponibili?

Il problema di questa idea risiede nell'automa **Mesi**. Come sappiamo, la linea di cache su cui scriviamo deve essere esclusiva, e questo richiede step intermedi come l'uso del *broadcast medium*, la richiesta di esclusività etc... Solo quando ho effettivamente queste condizioni passo a copiare in cache.

Capiamo subito che operare sulla cache è un'operazione abbastanza delicata. Lo Store Buffer, invece, è a uso dedicato del flusso in corso, non ho conflitti con altri thread. L'unica limitazione è che altri non vedono ciò che produco. Come vedremo dopo, gli Store Buffer, in x86 hanno una gestione FIFO, e possiamo usare alcune funzioni per forzare la scrittura in memoria.

Il TSO è un modello di consistenza che offre anche dei grossi vantaggi:

- Se nel program order di un CPU-core si hanno due istruzioni A, B, di cui A effettua una store di un dato X e B va a rileggere quel dato X , allora B ha la possibilità di recuperare il dato aggiornato andando semplicemente a leggere nello store buffer. In tal modo si risparmiano diversi cicli di clock per finalizzare le istruzioni perché chiaramente un **accesso allo store buffer è molto più veloce di un accesso alla cache**.
- È possibile effettuare il packaging delle informazioni da riportare nell'architettura di memoria. Se si hanno molteplici operazioni di scrittura che insistono sul medesimo blocco di memoria, si hanno altrettanti accessi allo store buffer ma poi lo store buffer dovrà *interagire con l'architettura di cache solo una volta*. Questo chiaramente riduce i costi di accesso alla memoria.

Tuttavia, il TSO non offre consistenza sequenziale perché, se abbiamo due istruzioni A, B in cui A viene prima di B nel program order, è possibile riportare nell'architettura di memoria prima gli effetti di B e poi gli effetti di A. In particolare, la presenza dello store buffer causa il **load-store bypass**, che è un fenomeno che si verifica quando si hanno due istruzioni A, B di cui A è una **store** di un dato X eseguita da un certo thread T_A , mentre B è una **load** del medesimo dato X eseguita da un altro thread T_B in un'istante successivo rispetto ad A. Di fatto, in tal caso, quando l'istruzione A viene finalizzata, il dato da essa prodotto potrebbe trovarsi soltanto nello store buffer. Di conseguenza, se il thread T_B gira su un CPU-core *diverso* rispetto a T_A , nel momento in cui deve andare ad accedere al dato X , *può farlo solo mediante la cache / RAM, per cui esegue la load di una versione vecchia di X*.

L'effetto finale è che l'istruzione B viene resa visibile prima dell'istruzione A.

Un'altra considerazione da fare è che *non necessariamente lo store buffer seguia la disciplina FIFO*. (in x86 si, in altre architetture no). Nel caso in cui non lo si segue, si ha lo **store-store bypass**, secondo cui due istruzioni di **store** successive possono essere invertite nel visibility order.

Memory Fencing

Il programmatore ovviamente deve avere la possibilità di prevenire i fenomeni di **load-store bypass** e **store-store bypass** per rendere corretto il software che sta sviluppando. Corrono così in aiuto tre categorie di istruzioni di **memory fencing** (= sincronizzazione delle attività che vengono eseguite sulla memoria), che sono:

- **SFENCE** (Store Fence): serializza le operazioni di **store** verso la memoria; tutte le store *precedenti* a lei vengono riversate in memoria prima dell'esecuzione di qualunque altra istruzione di store successiva a lei.
- **LFENCE** (Load Fence): serializza le operazioni di **load** dalla memoria; tutte le load precedenti a lei vengono completate prima dell'esecuzione di qualunque altra istruzione di load successiva a lei.
- **MFENCE** (Memory Fence): serializza tutte le operazioni in memoria; è un tipo di istruzione che unisce le capability di **sfence** e di **lfence**.

NON devo abusare di queste istruzioni, ma usarle con criterio per contesti delicati, altrimenti è come se stessi flushando la pipeline in ogni operazione.

Queste istruzioni sono garantite essere ordinate rispetto a qualsiasi altra istruzione serializzante (serializing instruction), come **cpuid** (che, se ricordiamo, effettua, tra le varie cose, lo squash della pipeline).

Inoltre, esistono delle istruzioni che, se precedute dal prefisso **lock**, diventano istruzioni serializzanti. Con **lock** si ha un approccio del tipo *blocco, lavoro, rilascio*. Quando lavoro è linearizzabile, grazie al lock vedo l'effetto delle altre attività. Queste sono: **add**, **adc**, **and**, **btc**, **btr**, **bts**, **cmpxchg**, **dec**, **inc**, **neg**, **not**, **or**, **sbb**, **sub**, **xor**, **xand**.

Esempio cmpxchg Significa *compare then exchange*, ha due operandi (o_1, o_2 , dove o_1 può essere una locazione di memoria) e compara il contenuto di o_1 col registro **rax** / **eax** / **ax** / **al**.

- Se i due valori sono **uguali**, allora il contenuto di o_2 viene riversato in o_1 ;
- Altrimenti, il contenuto di o_2 viene copiato in **rax** / **eax** / **ax** / **al**.

Si tratta di un'istruzione che può effettuare un primo accesso in memoria, una comparazione e poi un secondo accesso in memoria, per cui **non è atomica**. Se la si vuole eseguire in modo atomico, si utilizza appunto il prefisso **lock**, quindi ho una linea di cache esclusiva per me. Eseguire **cmpxchg** in modo **atomico** vuol dire riversare gli aggiornamenti in memoria già al completamento dell'istruzione stessa, per cui *non ci si può appoggiare allo store buffer*. Di conseguenza, è richiesto che, prima della **lock cmpxchg**, il contenuto dello *store buffer* venga riportato all'interno dell'architettura di memoria. È tale caratteristica che rende l'istruzione *serializzante* (ma, a differenza di **cpuid**, non effettua lo squash della pipeline).

La **lock cmpxchg** viene utilizzata anche per implementare i *lock* (e in particolare gli *spinlock*) per gli accessi in sezione critica. Per fare un esempio, di seguito è riportata una funzione C contenente un ASM inline, che implementa un trylock mediante **cmpxchg**

NB: trylock = se non riesco a ottenere il lock, non rimango in attesa e lascio perdere).

```
int try_lock (void *uadr) { //uadr = indirizzo dove si trova il lock
    unsigned long r = 0;
    asm volatile (
        "xor %%rax, %%rax\n" //rax = 0, sto azzerando il contenuto.
        "mov $1, %%rbx\n" //rbx = 1
        "lock cmpxchg %%rbx, ($1)\n" //if (uadr == 0) uadr = 1
        "sete (%0)\n" //if ("cmpxchg had success") r = 1
        : : "r" (&r), "r" (uadr) //prima dell'esecuzione dell'ASM inline,
                                //due registri qualsiasi vengono popolati con &r, uadr
        : "%rax", "%rbx" //rax, rbx sono i clobbers
    );
    return (r) ? 1 : 0; //if ("cmpxchg had success") return 1; else return 0
}
```

Le istruzioni come la **cmpxchg** che eseguono la *load* di una locazione *L* di memoria in un registro, modificano il valore di tale registro ed eseguono la *store* del nuovo contenuto del registro all'interno della locazione *L* di memoria sono le cosiddette istruzioni **Read-Modify-Write (RMW)**.

Ci permettono di rendere le operazioni globalmente visibili.

Anche qui esistono delle API di C che implementano l'invocazione di sfence, lfence, mfence e cmpxchg, e sono rispettivamente:

`_mm_sfence(), _mm_lfence(), _mm_mfence() e sync_bool_compare_and_swap()`

In ogni caso, utilizzare le istruzioni **RMW** per implementare i lock non è una soluzione particolarmente scalabile per realizzare degli schemi di coordinazione. Infatti, con lo spinlock, si ha un thread in sezione critica e tanti thread in *busy waiting*, per cui se disgraziatamente il thread in sezione critica viene *deschedulato* (ciò avviene comunemente se giro su una VM), si ha non solo un ritardo ma anche uno spreco di risorse e di energia da parte di chi è in *busy waiting*. Con lock, la linea di cache è a mio uso esclusivo.

Per questo motivo, si preferisce proporre degli algoritmi che prevedono un utilizzo alternativo delle istruzioni RMW e, a tal proposito, si hanno due possibilità principali che **non richiedono Lock**:

- **Non-blocking coordination** (*lock-free / wait-free synchronization*). Nel caso *lock-free*, posso ottenere degli abort, e quindi ritento (alg. Linearizzabile). Nel caso *wait-free*, non ho mai abort, ed è consistente.
- **Read Copy Update (RCU)**. Include *Read intensive + aggiornamento*. Tecnica che *non è bloccante solo per chi legge* la struttura dati, mentre per chi aggiorna deve serializzare. Qui stiamo totalmente cambiando approccio, è un diverso modo di fare.

Cosa usare è influenzato dal “con cosa sto lavorando”!

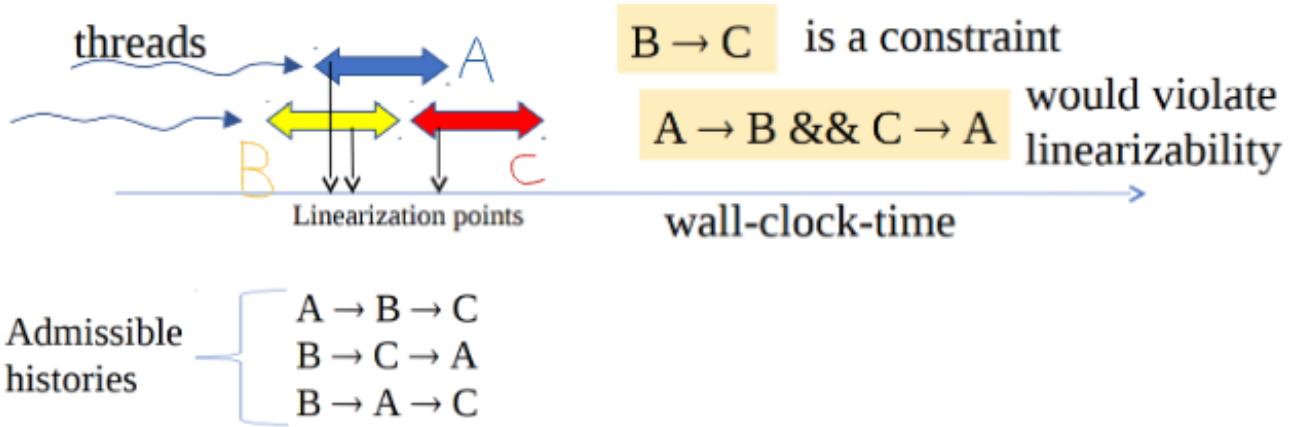
Linearizzabilità

Supponiamo di avere una struttura dati *S* e delle funzioni che accedono a *S*; diciamo che un'esecuzione concorrente di tali funzioni è corretta se è **linearizzabile**, ovvero *se è come se le operazioni fossero eseguite in modo sequenziale* (dove quella successiva viene eseguita solo dopo il completamento della precedente). Questo è vero se:

- Gli accessi concorrenti a *S*, nonostante possano durare diversi cicli di clock, possono essere visti come se i loro effetti si materializzassero in un unico punto del tempo.
- Tutte le operazioni che si sovrappongono nel tempo possono essere ordinate in base al loro istante di materializzazione selezionato.

Esempio Supponiamo di avere tre funzioni A, B, C che accedono a una stessa struttura dati condivisa, e supponiamo di avere due thread T_1, T_2 , di cui T_1 invoca la funzione A mentre T_2 invoca la funzione B e successivamente la funzione C.

Allora vale ciò che è riportato nella figura seguente:



Siamo solamente sicuri che B venga prima di C, non possiamo dire altro.

Tuttavia, lo studio di algoritmi concorrenti che utilizzano la materializzazione istantanea delle operazioni non è così banale. Ad esempio, in presenza di un salto condizionale, una determinata operazione può essere materializzata in istanti differenti a seconda se il salto viene preso oppure no.

Osservazione: Lavorare con istruzioni *atomiche* vuol dire *bloccare una linea di cache*, quindi *non ho concorrenza su un qualcosa che è solo mio*, al massimo posso averla se opero su linee diverse.

Prendiamo l'esempio sopra:

Siamo sicuri che il thread T_2 , quando esegue l'operazione C, riesca a vedere tutto ciò che è stato fatto in B? (sopra è quasi stato dato per scontato!).

In un caso di *Sequential Consistency*, sì (anche se noi non operiamo in questo contesto), altrimenti non è scontato. Ad esempio, il thread potrebbe essere stato spostato su una CPU diversa, e prima di andarci lo Store Buffer nella vecchia CPU non è stato flushato. Quindi nella nuova CPU non posso recuperare il “vecchio Store Buffer”.

Linearizzabilità vs operazioni RMW

Le operazioni RMW (Read-Modify-Write), nonostante implementino accessi in memoria non banali, appaiono in modo atomico sull'intera architettura hardware. Di conseguenza, possono essere sfruttate per definire dei punti di linearizzazione delle operazioni, in modo tale da ordinare le operazioni in una storia linearizzabile. Inoltre, le operazioni RMW possono fallire; di conseguenza il loro esito, come per i salti condizionali, può influenzare l'esecuzione o la non-esecuzione di una particolare istruzione e l'istante in cui essa viene eventualmente materializzata.

Sappiamo che con le operazioni RMW è possibile implementare un meccanismo di locking. I lock basati su RMW incentivano ancor più la linearizzazione, poiché portano le operazioni a essere eseguite in modo sequenziale.

Ma, come dicevamo in precedenza, preferiamo delle tecniche alternative all'utilizzo dei lock per sincronizzare i thread mediante operazioni RMW. Analizziamole ora. Con *lock* si ha un approccio del tipo *blocco, lavoro, rilascio* come è già stato detto. Utile per operazioni in tempo reale su strutture condivise, ma poco scalabile. Poi, con *fence, le rendo effettivamente visibili*. Vediamo delle alternative:

Non-blocking coordination

È un tipo di sincronizzazione **non bloccante**, che prevede due possibili approcci:

- 1) **Lock-freedom:** almeno un'istanza di una chiamata a funzione termina con successo in tempo finito AND tutte le istanze di chiamata a funzione terminano in tempo finito (o con successo o no). *Se fallisco, posso ricominciare da zero.*
- 2) **Wait-freedom** tutte le istanze di una chiamata a funzione terminano con successo in tempo finito. *Tutto quello che faccio va “sempre bene”.* Dipende però con che struttura lavoro.

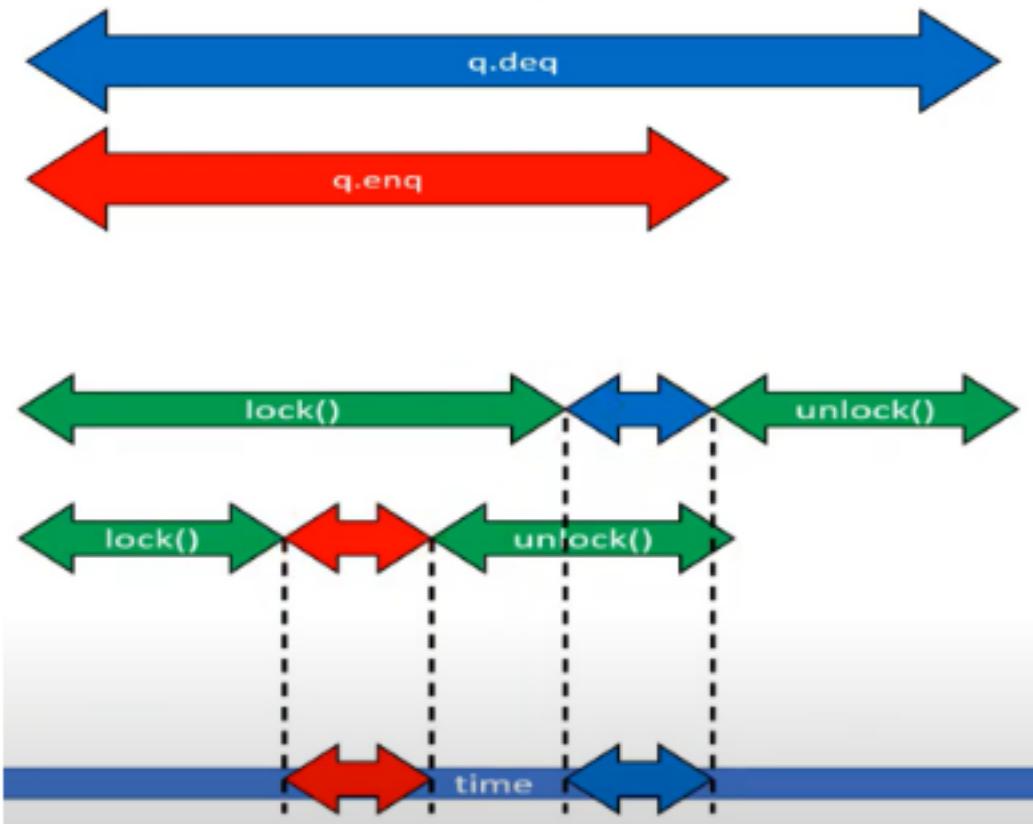


Figure 1: 43.png

Sincronizzazione lock-free

Prevede la seguente logica: se due operazioni ordinate sono incompatibili (ovvero portano a fallire una qualche operazione RMW), allora una di loro può essere accettata ma l'altra deve essere rifiutata ed eventualmente rieseguita con una nuova try. Perciò, sono algoritmi basati sulla logica **abort / retry**: se una qualche operazione non va a buon fine, viene semplicemente abortita e in caso si effettua un nuovo tentativo. Chiaramente, per essere un approccio efficiente, deve essere caratterizzato da un basso numero di abort, in modo tale da non sprecare troppo lavoro eseguito in CPU e nell'hardware in generale.

Un grosso vantaggio della sincronizzazione *lock-free* è che non dà problemi nel caso in cui un thread dovesse crashare. Infatti, qui il comportamento di ciascun thread è *indipendente* da quello che succede a tutti gli altri thread; nel caso in cui si utilizzi un lock, invece, il crash del thread che detiene correntemente il lock porta all'impossibilità per tutti gli altri thread di acquisire successivamente il lock. (Vedi descheduling di un thread su Macchina Virtuale).

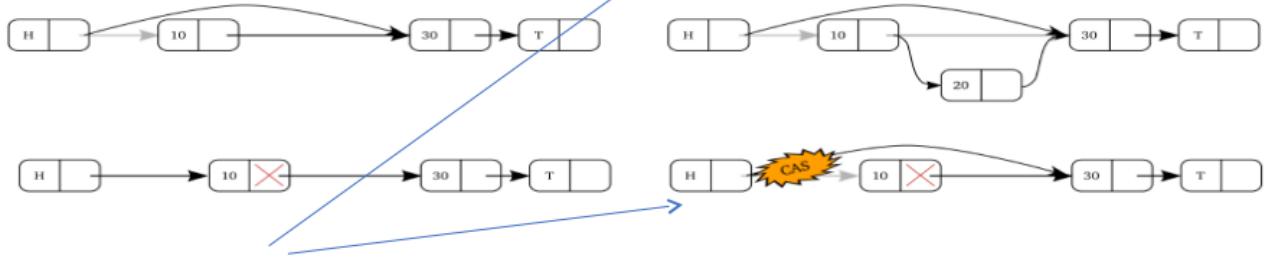
Esempio In questo schema, supponiamo di avere in memoria condivisa un certo *val*. Quando lo leggo, e poi lo voglio usare per una **compare and swap**, il confronto lo faccio tra il valore in una certa locazione e quel *val*. Se sono uguali eseguo lo swap. Se qualcuno lo ha cambiato, ho un fallimento.

On the lock-free linked list example

- Insert via CAS on pointers (based on failure retries)



- Remove via CAS on node-state prior to node linkage



These CAS can fail but likely will not depending on the access pattern

- **Insert:** supponendo di inserire il nodo 20 tra i nodi 10 e 30 all'interno della lista collegata, si procede nel seguente modo: si ispeziona la lista fino al punto di aggiunta, si imposta 30 come successore di 20 e poi, tramite un'operazione di Compare And Swap (CAS, che è un'istruzione atomica. Sarebbe come il compare and exchange: o aggiorno io, o aggiorna un altro. Vince chi lo fa prima, il secondo farebbe un confronto errato. Se fallisco, ripeto!), si imposta 20 come successore di 10.

Con l'approccio **lock-free**, tale operazione non dà problemi; l'unico caso in cui si ha un fallimento (e quindi un abort) è quello in cui c'è un altro inserimento concorrente nello stesso punto della lista collegata. Ad esempio, se concorrentemente a 20, un altro thread tenta di inserire un nodo 21 tra i nodi 10 e 30, solo uno dei due inserimenti avrà successo, mentre l'altro fallirà; viceversa, se si hanno inserimenti concorrenti in posizioni differenti della lista, avranno tutti successo nonostante non ci siano lock / attese.

- **Remove:** supponiamo di voler rimuovere il nodo 10 dalla lista collegata. Non possiamo farlo semplicemente deallocandolo ed eseguendo una Compare And Swap sul puntatore al successore del nodo testa (come nella figura in alto a sinistra). Infatti, può succedere che concorrentemente venga tentata l'esecuzione di un inserimento di un nodo 20 proprio subito dopo il nodo 10; se l'operazione di rimozione nel frattempo dealloca il nodo 10, poi l'inserimento è impossibilitato a cambiare il puntatore al successore di 10 perché si ritrova a lavorare su un *path* non più valido (vedere figura in alto a destra). Quello che si fa, dunque, è marcire il nodo 10 come da eliminare, per poi deallocaffarlo effettivamente in un secondo momento (vedere figura in basso a sinistra). Di conseguenza, vengono utilizzati dei bit all'interno di ciascun nodo che indicano lo stato del nodo stesso; questo a discapito dei puntatori, che si ritroveranno quindi con meno bit a disposizione. Ne consegue che i puntatori non possono più essere utilizzati con la massima granularità. Dopo aver marcato il nodo come da eliminare, si può procedere con una Compare And Swap per correggere il puntatore del nodo precedente (come nella figura in basso a destra).

Attenzione

In tale contesto è molto pericoloso eliminare un determinato nodo per poi riutilizzarlo per inserirvi delle informazioni differenti.

In breve: se un thread stava lavorando sul nodo 10, viene deallocated, e nel mentre il nodo 10 viene eliminato per far posto ad altro, quel thread, quando ritorna, avrebbe un indirizzo a qualcosa che non fa più parte della lista, come ad esempio ar ee sensibili! Il thread esegue **traversing**, si muove solo sulla lista, non conosce altro!.

In lungo: Supponiamo che un thread A debba effettuare una lettura sul nodo n , e supponiamo che venga deschedulato subito dopo aver recuperato l'indirizzo di memoria di quel nodo ma prima di leggere il valore contenuto; supponiamo anche che un thread B deallochi proprio il nodo n e riutilizzi l'indirizzo di memoria di n per scrivere una nuova informazione da inserire nella lista collegata (o in una qualsiasi struttura dati). Allora, il thread A, quando verrà

rischedulato, leggerà la nuova informazione anche se non era previsto dal flusso di esecuzione. Questo può anche rappresentare un problema di sicurezza nel momento in cui nei nodi sono riportati dei dati sensibili oppure, ad esempio, dei dati relativi agli utenti che stanno effettuando l'accesso a un determinato sistema.

Sincronizzazione wait-free

Qui non abbiamo né una logica di abort né una logica di retry: tutti i thread svolgono sempre lavoro utile, indipendentemente da quello che stanno facendo altri thread in concorrenza. Chiaramente si tratta di un approccio molto più “challenging” del precedente. Come risolviamo il problema del non poter riusare quegli spazi di memoria nella linked list?

Un esempio di struttura dati wait-free è il **registro atomico (1,N) wait-free**, che prevede un *solo scrittore* e N lettori ed è un caso particolare del registro atomico (M,N) wait-free. È un registro arbitrariamente grande in cui le operazioni di scrittura e lettura devono essere effettuate in modo atomico e potrebbero richiedere un alto numero di cicli di clock per essere eseguite. In una situazione del genere non ci piace l'utilizzo dei lock, perché farebbe crollare vertiginosamente le prestazioni; piuttosto, si procede nel seguente modo:

- Si hanno molteplici istanze del registro atomico e un puntatore che referenzia l'istanza più aggiornata.
- Lo scrittore, quando deve aggiornare il registro, ne alloca una nuova istanza; nel momento in cui ha terminato la scrittura, aggiorna con una **Compare And Swap** il puntatore per farlo referenziare verso la nuova istanza.
- I lettori sfruttano il puntatore per accedere all'area di memoria dov'è allocata l'istanza correntemente *più aggiornata* del registro atomico. Una volta che la lettura è iniziata, il lettore è sicuro che non ci saranno mai interferenze con l'istanza da parte di scrittori (per cui l'operazione andrà certamente a buon fine).

Questa soluzione presenta però una difficoltà in particolare: non è banale stabilire quando è possibile effettuare la garbage collection delle varie istanze del registro atomico. Comunque sia, la letteratura stabilisce che servono almeno $N + 2$ buffer (istanze) per far funzionare correttamente il meccanismo:

- N buffer sono (al limite) per gli N lettori.
- 1 buffer è adibito per contenere l'ultimo eventuale valore che non è stato ancora acceduto da alcun lettore.
- L'ultimo buffer serve allo scrittore per inserire un eventuale nuovo valore.

Quindi sostanzialmente, ad ogni modifica creo un nuovo buffer, dico che quello è il più aggiornato, e chi lavora col vecchio lo usa finché non ha terminato. Un limite da gestire è che non posso mantenere infiniti buffer, però qualcuno potrebbe puntarci ancora!

Scendendo in qualche ulteriore dettaglio dell'implementazione di tale meccanismo, si ha un'unica variabile di sincronizzazione composta da due campi, 32 per identificare l'istanza e 32 per il contatore:

- nel primo è indicato qual è lo slot (tra gli $N+2$ totali) che è stato aggiornato più recentemente dallo scrittore.
- Nel secondo è riportato il numero di lettori che si sono attestati esattamente a quello slot; questo secondo campo viene aggiornato dai lettori mediante una chiamata a **fetch_and_add atomica**, un'istruzione che esegue la lettura, e l'incremento di una determinata variabile in modo atomico. Conto chi ha preso il riferimento sostanzialmente. Ovviamente le operazioni devono essere finite, sennò non posso contarle. Incremento solo se mi sposto tra aree di memoria, altrimenti non aumento se sto sempre sulla stessa.
- Il reader decrementa il counter se ha finito di leggere.
- Quando lo scrittore deve effettuare una scrittura, esegue una **atomic_exchange** sulla variabile di sincronizzazione, ovvero legge e mette da parte il contenuto attuale della variabile di sincronizzazione e riscrive tale variabile con:
- L'indirizzo di memoria del nuovo buffer (nel primo campo).
- Il valore 0 per indicare che inizialmente non ci sono lettori che hanno acceduto al nuovo buffer (nel secondo campo).

Chiaramente ciascun lettore dovrà essere in grado di visualizzare sempre la versione della variabile di sincronizzazione associata allo slot in cui l'operazione di lettura era iniziata. Nel momento in cui in uno slot non ci sono più letture pendenti (per cui $\#lettura\ iniziata = \#lettura\ terminata$), tale slot può essere sfruttato dallo scrittore per inserirvi dei nuovi dati aggiornati; tra l'altro, con $N + 2$ slot totali, esiste sempre almeno uno slot che soddisfa questa proprietà, per cui lo scrittore non rimarrà mai bloccato per eseguire le scritture.

Esistono anche delle *ottimizzazioni* del protocollo: ad esempio, è possibile fare in modo che ciascun lettore esegua un'unica `fetch_and_add` sul contatore dei lettori per ogni diversa istanza di registro che va a leggere. In tal modo, si riduce il numero totale di operazioni atomiche eseguite e questo è un beneficio anche per la *cache coherency*: sappiamo che le istruzioni atomiche di tipo RMW portano un particolare CPU-core a prendere un blocco di cache nello stato exclusive, lasciando così gli altri CPU-core bloccati nel caso in cui vogliono accedere al medesimo blocco di cache.

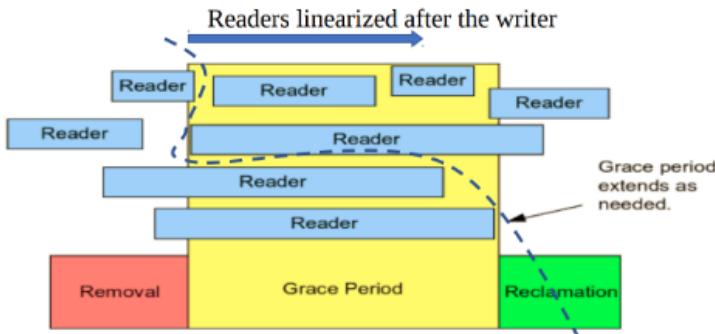
Read Copy Update RCU

È un tipo di sincronizzazione che fa da trade-off tra l'utilizzo dei lock e la lock-freedom. Infatti, non offre le stesse garanzie della lock-freedom (in cui tutte le istanze di chiamate a funzione terminano in tempo finito) ma, di contro, semplifica il meccanismo di garbage collection; questo rende RCU più scalabile. Qui possiamo ammettere, su una struttura dati concorrente, *un solo writer* e *n* di reader per volta:

- I reader, per eseguire le *lettura*, non devono attendersi né a vicenda né col writer.
- Il writer, per eseguire una *scrittura*, deve attendere solo eventuali altri writer.

Si tratta dunque di un approccio vincente per le strutture dati **read-intensive**, che nella pratica sono tantissime; infatti, Linux implementa RCU per molte strutture dati usate a livello kernel. (Ad esempio, la lettura di una *hash table* per vedere il thread control block (*TCB*) di un thread!)

RCU prevede che un buffer o un nodo di una struttura dati non può essere deallocated finché non siamo sicuri che sia diventato inutilizzato. In particolare, tra l'istante in cui viene invocata la deallocation (e il buffer/nodo viene marcato come "da deallocare") e l'istante in cui il buffer/nodo viene effettivamente deallocated, si ha il cosiddetto *grace period*.



Nella figura qui sopra si hanno tre *reader* (escludiamo quello che conclude prima di *grade period*) che eseguono una lettura concorrente alla removal e che quindi devono essere attesi prima della rimozione vera e propria (*reclamation*): il grace period dura fin tanto che tutti e tre questi reader non hanno concluso la loro lettura. Il Grace period è un periodo di grazia in cui, anche se sono pronto a cambiare indirizzo, lascio "attivo" il vecchio indirizzo per far concludere le letture. In particolare, è necessario attendere tutti e tre i reader indistintamente perché lo scrittore non sa qual è l'istante esatto in cui avviene la linearizzazione della removal, per cui deve assumere che tutte le letture concorrenti siano iniziate antecedentemente a tale istante. Non posso modificare *removal* finché qualcuno la usa. Quindi, se reader concorrenti e su *cpu diverse*, non vedono che un "pezzo" è stato sganciato, allora devo aspettarli. Questo perché chi linearizza prima di me, non potrebbe rientrare se togliessi la struttura.

Funzionamento

Il lettore

- 1) Segnala la sua presenza.
- 2) Legge la struttura dati.
- 3) Segnala che se ne sta andando.

Lo scrittore

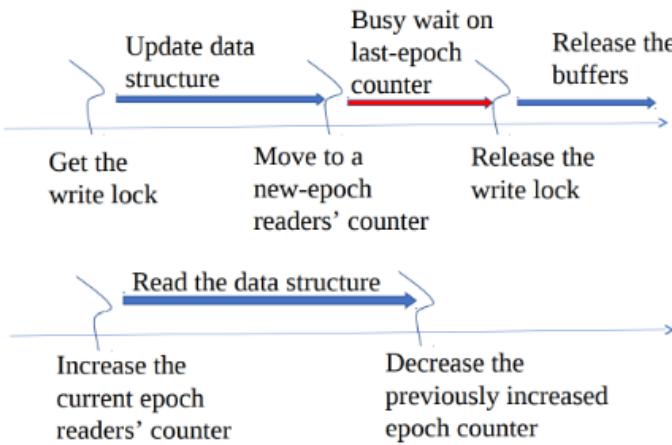
- 1) Acquisisce il lock di scrittura.
- 2) Aggiorna la struttura dati.
- 3) Attende che i lettori "*standing*" terminino le loro operazioni; notiamo che i lettori che operano sull'istanza della struttura dati già modificata sono dei *don't care reader*.

- 4) Dealloca la vecchia istanza della struttura dati.
- 5) Rilascia il lock di scrittura.

Non-preemptable RCU vs preemptable RCU

A questo punto è necessario risolvere il seguente problema: come fa lo scrittore a distinguere un lettore standing da un lettore che opera sull'istanza della struttura dati già modificata? Di seguito vengono proposte due possibili soluzioni.

- **Non-preemptable RCU:** qui è necessario che i reader disattivino la possibilità di essere interrotti (“prelazionati” dalla CPU, cioè thread su cpu non interrompibili, non rilasciano la cpu su richiesta) durante la loro esecuzione. In tal caso, lo scrittore, subito dopo aver aggiornato la struttura dati, invoca: `for_each_online_cpu(cpu), run_on(cpu)`. In pratica, lo scrittore si fa un giro su tutte le CPU della macchina e ne richiede l'utilizzo. Appena riesce a essere schedulato sulla CPU_i , significa che sulla CPU_i non è più in esecuzione (o non lo è mai stato) un lettore che era standing al completamento dell'aggiornamento. Di conseguenza, quando lo scrittore verrà schedulato su tutte quante le CPU, sarà possibile concludere il grace period senza problemi. Il grosso svantaggio di questo approccio sta nell'impossibilità di interrompere l'esecuzione dei lettori: se a un certo punto dovesse subentrare un thread con priorità arbitrariamente elevata, dovrà in ogni caso aspettare che un qualche lettore termini prima di essere schedulato.
- **Preemptable RCU:** Qui potrei subire un *interrupt*, ma stando al “ragionamento” di prima, lasciare la CPU vuol dire aver finito. Si usa il concetto di **epoca**: Si ricorre all'utilizzo di un **presence-counter atomico** che va a indicare quanti lettori stanno correntemente insistendo su una determinata versione (**epoca**) della struttura dati; è atomico perché, come al solito, vengono acceduti in lettura e in scrittura con un'unica istruzione atomica (i.e. `fetch_and_add`). Nel momento in cui lo scrittore aggiorna la struttura dati, redireziona il puntatore al presence-counter verso una nuova istanza di presence-counter (*quello relativo alla nuova epoca*), in modo tale che i lettori successivi aggiornino quest'ultimo; d'altra parte, i lettori standing, quando completano le loro operazioni, **decrementano** il presence-counter della vecchia epoca (last-epoch), in modo tale che sia tutto consistente. Chiaramente, il grace period termina quando il last-epoch counter diviene uguale a zero. Nella pagina seguente è riportato uno schema riassuntivo sul funzionamento del preemptable RCU.



Vettorizzazione

È un meccanismo in cui le singole istruzioni hanno un vettore di sorgenti e un vettore di destinazioni; in altre parole, vengono coinvolti molteplici registri contemporaneamente. Di conseguenza, si ha un miglioramento delle performance. La vettorizzazione è una forma di **SIMD (Single Instruction Multiple Data)** alla base delle GPU; in alternativa al SIMD esistono:

- **MIMD (Multiple Instruction Multiple Data):** è la forma di calcolo realmente utilizzata nelle macchine reali; rispetto al SIMD prevede l'utilizzo di molteplici processori/core. Chip con hyperthread, ad esempio un hyperthread (MI) e speculazione (MD).
- **MISD (Multiple Instruction Single Data):** è una forma di calcolo più rara, anche se qualcuno sostiene che un processore speculativo sia MISD; di fatto, i processori speculativi introducono la possibilità di eseguire più istruzioni in parallelo su uno stesso dato, sfruttando anche molteplici istanze del medesimo registro logico.
Nb: i renamed register non sono esposti in ISA.

- **SISD (Single Instruction Single Data)**: è la forma di calcolo più banale, dove è previsto un unico CPU-core non speculativo. Sarebbe l'architettura di Von Neumann.

Per poter implementare la vettorizzazione, la macchina chiaramente deve fornire appositi registri, apposite istruzioni macchina e appositi meccanismi dell'hardware, come ad esempio:

- L'**hardware data gather**, per raccolta di dati dalla memoria verso un registro vettoriale.
- L'**hardware data scatter**, per il riversamento di dati dal registro vettoriale verso la memoria.

Tramite flag `-O` posso implementare calcoli vettoriali senza API, esistono anche `-O2` o `-O3`, più cresce più ottimizza, ma deve essere opportunamente gestito, cioè sezioni critiche vanno regolate con `mfence` per evitare problemi ad accessi in memoria. Alcune ottimizzazioni abilitabili sono il *vectorize* e *loop-unrolling*. Il *volatile* è invece una non-ottimizzazione, che però può essere utile per cose più delicate.

In x86, la vettorizzazione è detta **SSE (Streaming SIMD Extension)**, mentre in x86-64 è detta **SSE2**.

In SSE si hanno 8 registri vettoriali a 128 bit, che sono $XMM_0, XMM_1, \dots, XMM_7$. Tali registri sono in grado di ospitare:

- ✓ 2 64-bit double precision floats
- ✓ 4 32-bit single precision floats
- ✓ 1 128-bit integer (unsigned)
- ✓ 2 64-bit integers (signed or unsigned)
- ✓ 4 32-bit integers (signed or unsigned)
- ✓ 8 16-bit integers (signed or unsigned)
- ✓ 16 8-bit integers (signed or unsigned)



In SSE2, invece, si hanno 16 registri vettoriali a 128 bit, che sono $YMM_0, YMM_1, \dots, YMM_{15}$.

Le istruzioni di `mov` che coinvolgono i registri XMM e YMM (caricamento di dati nei registri vettORIZZati / store dei dati dai registri vettORIZZati in memoria) potrebbero richiedere che le informazioni siano allineate in memoria (e.g. agli 8 byte, ai 16 byte). L'utilizzo delle istruzioni che richiedono l'allineamento sui dati non allineati causa un **general protection error**.

Esistono più modi per allineare le informazioni in memoria:

- `__attribute__((aligned(16)))`
- `mmap()`, che alloca delle pagine di memoria allineate ai 4 KB.

Vettorizzazione esplicita ed implicita

- **Vettorizzazione esplicita**: il programmatore utilizza esplicitamente le istruzioni che coinvolgono i registri vettORIZZati.
- **Vettorizzazione implicita**: il programma viene compilato col flag `-O` in modo tale che il compilatore `gcc` scandisca il codice sorgente e lo mappi, ove possibile, su istruzioni vettoriali.

Per quanto riguarda la vettorizzazione esplicita, in realtà in C sono offerti degli intrinsics (delle funzioni ad hoc) per sfruttare la vettorizzazione:

- Vectorized addition - 8/16/32/64-bit integers

MMX(64-bit)	<code>d = _mm_add_pi8(a, b)</code>	<code>_m64 a, b;</code>	<code>_m64 d;</code>
SSE2(128-bit)	<code>d = _mm_add_epi8(a, b)</code>	<code>_m128i a, b;</code>	<code>_m128i d;</code>
MMX(64-bit)	<code>d = _mm_add_pi16(a, b)</code>	<code>_m64 a, b;</code>	<code>_m64 d;</code>
SSE2(128-bit)	<code>d = _mm_add_epi16(a, b)</code>	<code>_m128i a, b;</code>	<code>_m128i d;</code>
MMX(64-bit)	<code>d = _mm_add_pi32(a, b)</code>	<code>_m64 a, b;</code>	<code>_m64 d;</code>
SSE2(128-bit)	<code>d = _mm_add_epi32(a, b)</code>	<code>_m128i a, b;</code>	<code>_m128i d;</code>
MMX(64-bit)	<code>d = _mm_add_si64(a, b)</code>	<code>_m64 a, b;</code>	<code>_m64 d;</code>
SSE2(128-bit)	<code>d = _mm_add_epi64(a, b)</code>	<code>_m128i a, b;</code>	<code>_m128i d;</code>

- Vectorized addition - 32-bit floats

SSE(64-bit)	<code>d = _mm_add_ss(a, b)</code>	<code>_m128 a, b;</code>	<code>_m128 d;</code>
SSE(128-bit)	<code>d = _mm_add_ps(a, b)</code>	<code>_m128 a, b;</code>	<code>_m128 d;</code>

- Vectorized addition - 64-bit doubles

SSE2(64-bit)	<code>d = _mm_add_sd(a, b)</code>	<code>_m128d a, b;</code>	<code>_m128d d;</code>
SSE2(128-bit)	<code>d = _mm_add_pd(a, b)</code>	<code>_m128d a, b;</code>	<code>_m128d d;</code>

Kernel Programming Basics

Abbiamo visto che il comando *cpuid* manda la pipeline in squash, scopriremo che è possibile risolverlo!

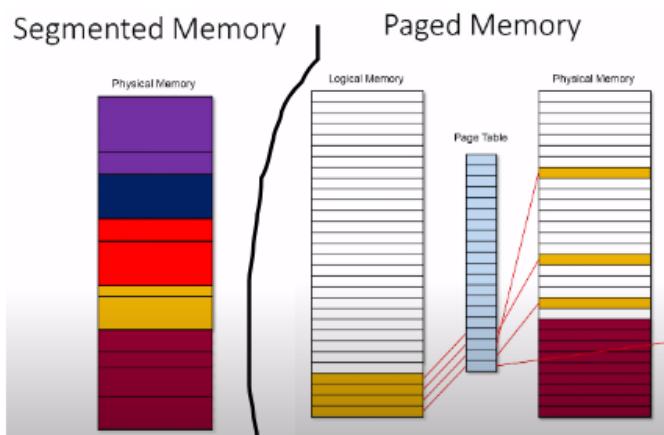
Indirizzamento della memoria – come vedo la memoria?

La memoria può essere indirizzata secondo più modalità possibili. Una di queste è data dall'**indirizzamento lineare**, dove si utilizzano appunto gli indirizzi lineari che sono caratterizzati esclusivamente da un *offset*, indipendentemente da se stiamo operando in memoria fisica (i.e. ram) o in memoria logica (i.e. address space del processo). Un thread, ad esempio, ha associato un contenitore di memoria, in cui si sposta mediante *offset*, e.g. : byte 44 del contenitore.

Un meccanismo più interessante per indirizzare la memoria è la **segmentazione**. Qui vediamo l'address space come suddiviso in più segmenti, e ciascun indirizzo deve essere espresso con l'id del segmento e l'offset da applicare a quel segmento.

Nelle architetture moderne, l'indirizzamento lineare e la segmentazione vengono *combine* per avere una gestione molto potente dell'indirizzamento della memoria. In particolare, con la presenza dei segmenti all'interno dell'address space, è sempre possibile specificare l' *id* del segmento e l'offset *D* da applicare a quel segmento; da qui si può ottenere il corrispettivo **indirizzo lineare**, che è dato dalla somma tra la base del segmento e *D*.

Quando abbiamo raggiunto l'idea di esprimere in maniera lineare un indirizzo, non è detto che sia l'**indirizzo definitivo dello storage effettivo**. Di fatto, come abbiamo già accennato, *sia la memoria logica sia quella fisica possono essere espresse in termini di indirizzi lineari*. Se abbiamo a che fare con una memoria logica, è necessario mappare tale rappresentazione logica nello storage effettivo utilizzando la **paginazione** e la **memoria virtuale**: in particolare, si dispone di una *page table* che mappa ciascuna pagina logica dell'address space in una pagina di memoria fisica all'interno della RAM; ricordiamo inoltre che la memoria virtuale è data dalle pagine logiche di memoria che **ancora non sono state materializzate in RAM** (per cui non esiste ancora un mapping tra indirizzi logici e indirizzi fisici).



Osservazione:

- Con la **sola segmentazione** divido in segmenti (testo, stack,...), e questi segmenti me li ritroverei così in memoria se non usassi altre tecniche. Se ho un intero segmento associato ad uno scopo, non posso usarlo per altro (può portare ad uno spreco dello spazio), ma permette al programmatore di organizzare le aree di lavoro.
- Con la **sola paginazione** ragioniamo su pagine (più piccole dei segmenti), le quali possono essere non contigue in memoria. Però fare accessi non contigui è un po' più lento.

Normalmente si possono **usare insieme**: prima organizzo in aree di lavoro (segmenti), e poi mappo sulla memoria fisica.

Comunque sia, se i numeri di segmento non sono specificati all'interno delle istruzioni macchina, viene utilizzato un qualche segmento di default per raggiungere un certo dato in memoria. Questo offre trasparenza alla segmentazione anche ai programmatori assembly.

I processori moderni sono equipaggiati in un modo tale da supportare la segmentazione in modo efficiente, in combinazione col meccanismo di paginazione accennato poc'anzi. (Stanno supportando il Kernel). Di fatto, quello che avviene in pratica è che ciascun indirizzo lineare viene prima tradotto in una pagina logica col relativo offset di pagina (page address) e poi, con l'aiuto della page table, viene convertito in una pagina fisica all'interno dello storage col medesimo offset di pagina. In definitiva, lo schema di mapping completo è il seguente:

segmented addr \Rightarrow linear addr \Rightarrow paged addr \Rightarrow physical addr

In realtà, si possono avere anche più livelli di paginazione. Nella paginazione a due livelli, ad esempio, l'address space è diviso in sezioni, ciascuna delle quali è suddivisa in pagine, per cui gli indirizzi sono composti da una parte alta indicante la sezione, una parte intermedia indicante la pagina all'interno della sezione e una parte bassa indicante l'offset all'interno della pagina.

Esempio:

```
mov (%rax),%rbx  
push %rbx
```

Qui stiamo accedendo al valore puntato dal registro **rax** (che in questo caso specifico è un puntatore) e lo stiamo salvando in **rbx**; dopodiché memorizziamo tale valore sullo stack. (facciamo infatti una *push*).

Ecco, stiamo utilizzando ben tre segmenti di default:

- uno relativo all'accesso in memoria tramite la **mov**.
- uno legato allo stack
- uno legato alla sezione **text** dell'address space dove si muove il program counter (facciamo fetch delle istruzioni infatti!)

Supporti alla segmentazione

I processori *di sistema*, per lavorare in maniera efficiente, sfruttano dei particolari componenti hardware che consentono un accesso veloce e trasparente alle informazioni sui segmenti di memoria. Tali componenti sono degli appositi **registri di CPU** e delle **tabelle di memoria** direttamente puntate dai registri di CPU. Badiamo che i puntatori alle tabelle di memoria possono puntare sia alla memoria fisica sia alla memoria logica; in questo secondo caso, il meccanismo di paginazione è richiesto per mappare anche le pagine logiche delle tabelle di memoria sulla RAM.

Segment selector

Detto anche **registro di segmento**, contiene l'*identificatore* del segmento target a cui vogliamo accedere. (quindi non il nome del registro bensì il nome del segmento). Perciò, gli indirizzi di memoria sono in realtà composti dall'identificatore del registro di segmento usato e da un offset. Il segment selector viene utilizzato per motivi di modularità e flessibilità: se usiamo il medesimo registro di segmento, a seconda di come lo popoliamo, andiamo su un certo segmento target oppure su un altro. Tipicamente è il kernel del sistema operativo a manipolare tale registro. Un esempio è nel caso dei thread: non devo riscrivere nulla, il *blocco di codice* è lo stesso, bensì *riprogrammo* il segmento target, in modo che ogni thread vada dove deve andare.

Accesso alla memoria nei sistemi x86

Real mode

È stata una modalità primitiva di accesso alla memoria, in cui l'**indirizzo lineare e quello fisico coincidevano** e non si aveva la paginazione. La segmentazione c'era, e tipicamente veniva utilizzata direttamente dai programmati assembly. Non c'era il concetto di indirizzo logico né supporto alla memoria virtuale. Qui avevamo:

- Quattro registri di segmento a 16 bit che mantenevano l'ID dei corrispettivi quattro segmenti target.
- Registri general-purpose a 16 bit che mantenevano l'offset dei segmenti.
- Un meccanismo statico per calcolare l'indirizzo base di ciascun segmento, che era uguale al suo ID moltiplicato per 16. L'indirizzo fisico a cui si accedeva era così calcolato: $SegmentID \cdot 16 + Offset$
- Un limite massimo di 1 MB (ovvero 2^{20} byte) di memoria consentita: di fatto, se i registri di segmento erano a 16 bit potevano esistere al più 2^{16} segmenti diversi, noi moltiplichiamo per 16, allora stiamo esprimendo un valore con 20 bit ($2^{16} \cdot 2^4$), quindi abbiamo $2^{16} \cdot 2^4$ byte totali a disposizione.
- **Nessuna** informazione di **protezione dei segmenti**: non si specificava chi, come e quando poteva raggiungere ciascun segmento. In definitiva, la real mode non è adeguata per i sistemi moderni.

80386 protected mode

È stata una modalità di accesso alla memoria in cui è possibile lavorare con o senza paginazione:

- Se la paginazione viene sfruttata, allora gli indirizzi lineari = indirizzi logici
- Altrimenti gli indirizzi lineari = indirizzi fisici.

Cioè noi abbiamo un indirizzo lineare (*segmento + offset*, dato da segmentazione e indirizzamento lineare). Ora, questo indirizzo è esattamente l'indirizzo di memoria fisica?

Se **non ho paginazione**, deve per forza esserlo, perchè non faccio altre trasformazioni di indirizzo.

Se **ho paginazione**, posso passare per un indirizzo intermedio, che è quello logico.

Qui avevamo:

- Registri di segmento a 16 bit, di cui 13 bit erano utilizzati per mantenere l'ID del segmento target, mentre gli altri 3 erano bit di controllo (di protezione); questi tre bit di protezione mettevano dei paletti per l'accesso ai segmenti, per cui non era più vero che qualunque processo potesse utilizzare a piacimento qualunque segmento. Potevo creare viste sulla memoria!
- Registri general-purpose a 32 bit che mantenevano l'offset dei segmenti.
- Una **tabella di segmento** che teneva traccia dell'*indirizzo base di ciascun segmento*. Di conseguenza, gli indirizzi lineari (indipendentemente dal fatto che fossero logici o fisici) venivano calcolati nel seguente modo:
 $address = TABLE [segment].base + offset$.
- Un limite massimo di 4 GB (ovvero 2^{32} byte) di memoria lineare consentita.

Long mode

È la modalità di accesso alla memoria utilizzata dai processori *moderni* (anche se è tuttora possibile impostare la protected mode per motivi di retrocompatibilità). Qui abbiamo:

- Registri di segmento a 16 bit, di cui 13 bit sono utilizzati per mantenere l'ID del segmento target, mentre gli altri 3 sono bit di protezione.
- Registri general-purpose a 64 bit che mantengono l'offset dei segmenti. In realtà, per questo specifico utilizzo, sono permessi solo 48 di questi 64 bit: di conseguenza, è possibile esprimere 2^{48} locazioni di memoria differenti.
- Una tabella di segmento che tiene traccia dell'indirizzo base di ciascun segmento. Anche qui, gli indirizzi lineari (che, da capo, possono essere paginati o meno) vengono calcolati nel seguente modo:
 $address = TABLE [segment].base + offset$
- Un limite massimo di 256 TB (ovvero 2^{48} byte) di memoria lineare consentita, visibile da un thread che esegue.

Tabelle di segmento

Come accennato poc’anzi, sono tabelle che tengono traccia dell’*indirizzo base* di ciascun *segmento*, per cui consentono di tradurre gli indirizzi segmentati in indirizzi lineari. In realtà ce ne sono > 1 in memoria fisica, e si hanno due registri di processore che puntano a una di loro. Di conseguenza, sono esattamente due le tabelle puntabili in ogni istante di tempo dal processore/hyperthread; esse sono la **Global Descriptor Table (GDT)** e la **Local Descriptor Table (LDT)**.

Global Descriptor Table

Determina dove, all’interno dello spazio di indirizzamento lineare, sono collocati (almeno) i segmenti che afferiscono alle locazioni di livello *kernel* (e.g. segmento testo di livello kernel, segmento dati di livello kernel).

Local Descriptor Table

Determina dove, all’interno dello spazio di indirizzamento lineare, sono collati i segmenti relativi alla parte *user* dell’applicazione che non appaiono nella GDT. Deprecata.

Se nella CPU ho un registro che punta a una di due LDT, allora su tale CPU posso passare da un flusso di esecuzione all’altro, cambiando la vista della memoria, cambiando questo pointer. Magari thread_1 usa LDT_1 , e thread_2 usa LDT_2 . Le LDT ci dicono **segmento codice** e **segmento data** a cui il thread può accedere, e dove sono nell’indirizzamento lineare.

In x86, quando abbiamo un **segment selector**, abbiamo un ID e le tabelle considerate possono essere 2 per ogni flusso di esecuzione, cioè GDT o LDT. Specifichiamo quale delle due con un apposito *bit* di controllo. In sistemi moderni, si considera solo la GDT, che ad ogni istante di tempo dice, per un thread in esercizio sulla CPU, esattamente ogni segmento toccabile dai selettori dei registri usati dal thread. La GDT è un vettore, in cui per ogni entry abbiamo la *base* di un certo segmento ed alcuni flag sull’utilizzo.

Quando usata, sale nel processore, e coi flag aggiorniamo i registri nel processore usati nella gestione.

Ho una GDT per CPU, e a seconda del thread posso cambiare un certo valore della entry, portando ad un cambio di contesto.

Segmentazione e paginazione

- La **segmentazione** è nata per *regolamentare i permessi di accesso* al codice e ai dati delle applicazioni (vedi i tre bit di protezione). Quindi posso fare differenziazioni in base ai thread. Se due thread eseguo stesso blocco di codice, ma volessero andare in aree di memoria diverse, NON potrei farlo solo con la paginazione, mentre con la segmentazione basta aggiungere un selettore del segmento.
Quindi se parlo di contesti di esecuzione parlo di segmentazione, non di paginazione.
- La **paginazione** è nata per migliorare l’*utilizzo della memoria fisica* andando a risolvere le problematiche legate alla frammentazione esterna. Di conseguenza, la paginazione lavora a grana molto più fine rispetto alla segmentazione (i.e. una pagina è molto più piccola di un segmento). Basti pensare che, nel momento in cui bisogna accedere a una qualche informazione in memoria, tutta la pagina che la contiene deve essere materializzata in RAM; se la pagina è troppo grande, la sua gestione può risultare complicata (ad esempio, è più agevole effettuare swap-in / swap-out di pagine di memoria piccole). Non differenzio in base ai thread.

In realtà, all’interno delle page table, non si hanno soltanto le informazioni sul mapping delle pagine logiche sui frame fisici in memoria, bensì esistono anche dei bit di controllo che indicano delle regole di utilizzo di quei frame.

Ma a questo punto a cosa serve mantenere la segmentazione che era nata proprio per motivi di protezione?

Tramite la segmentazione riusciamo a implementare tecniche efficienti di gestione dell’accesso alla memoria in architetture multicore e multi-thread: di fatto, possiamo fornire delle viste di segmenti differenti a thread diversi (vedi il **Thread Local Storage – TLS**) o anche a CPU-core diversi. Nella segmentazione, la grana di protezione è più “grossa” in quanto con pochi bit si possono definire comportamenti su un’intera area di memoria, ma è utile soprattutto quando si lavora con TLP, quindi in maniera concorrente, inoltre la segmentazione è stata portata avanti quando la paginazione era già supportata dai sistemi.

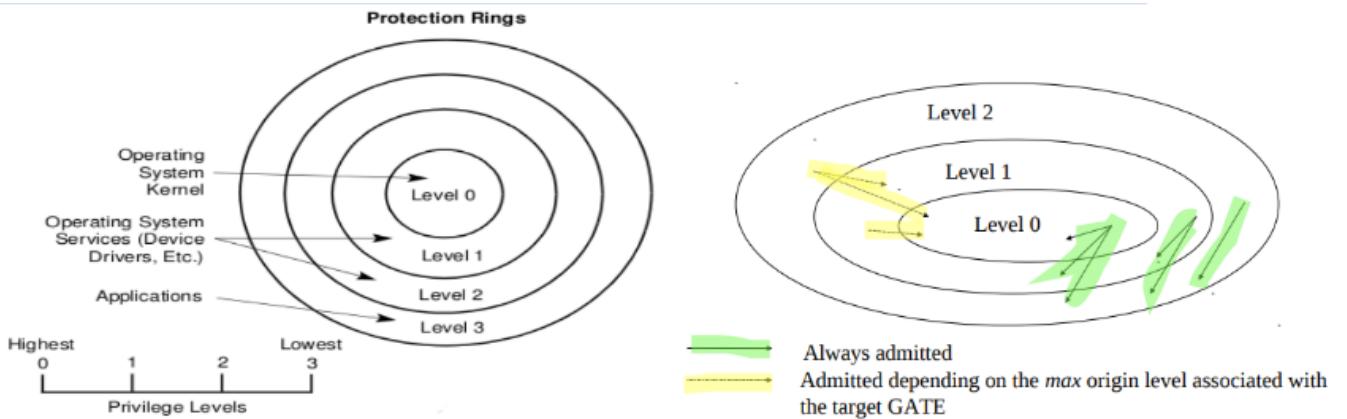
Modello di protezione basato su segmentazione

A ciascun segmento è assegnato un numero intero h che indica il suo livello di protezione (di privilegio).

0 è il livello di protezione massimo e, via via che h aumenta, il livello di protezione decresce. Ciascuna routine (e

ciascuna istruzione) assume il livello di protezione del segmento a cui appartiene. Ciascuna routine r_1 avente un livello di protezione pari ad h può invocare una qualsiasi altra routine r_2 col medesimo livello di protezione h , sia se r_1 e r_2 appartengono allo stesso segmento (per cui parleremmo di **intra-segment jump**), sia se non vi appartengono (per cui parleremmo di **cross-segment jump**). Invece, per saltare da una routine r_1 con livello di protezione pari ad h a una routine r_3 con livello di protezione diverso da h , è richiesto un **cross-segment jump**. In particolare, è sempre ammesso saltare dal livello di privilegio h a un livello di privilegio $h + i$ (con $i > 0$) poiché porta ad *abbassare* il grado di protezione. Per migliorare i propri privilegi e, quindi, per passare da un livello di protezione pari ad h a un livello di protezione pari ad $h - i$ (con $i > 0$), è necessario sfruttare dei particolari access point detti **GATE**. Ciascun GATE è identificato dalla coppia $\langle \text{seg.id}, \text{offset} \rangle$, dove seg.id è l'identificatore del segmento e offset è lo spiazzamento all'interno di quel segmento dove si trova il GATE. A ciascun GATE (e.g. appartenente a un segmento con livello di protezione h) è associato un massimo livello di privilegio $h + j$ a partire dal quale è possibile passare al livello h passando per quello stesso GATE (perciò, se si proviene dal livello $h + j + i$, con $i > 0$, quel GATE non può essere utilizzato).

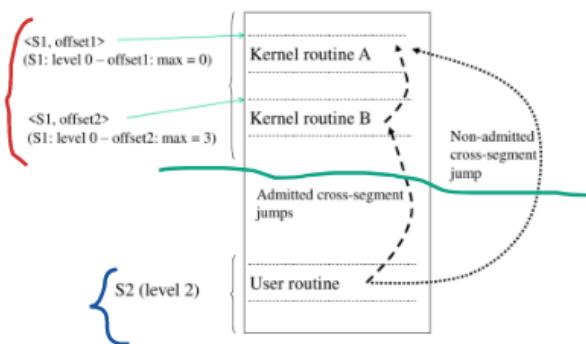
Esempio Ho segmento livello 3, ho un GATE (porta) che mi dice che posso diventare livello 3 solo se sono livello 4. Se fossi livello 5, non rispetto i requisiti minimi per attraversarlo. Senza i gate non posso migliorare la protezione. A seconda della configurazione di un GATE, posso fare o non fare determinate cose. Io definisco queste *porte*. Posso sempre peggiorare il mio livello, non servono GATE, ma poi posso migliorare il mio livello solo se c'è un GATE *che me lo permette*. Il meccanismo appena descritto può essere schematizzato col **ring model**. Nei sistemi x86 moderni si hanno esattamente 4 livelli di protezione, che sono schematizzati nella seguente figura:



Nei sistemi operativi convenzionali, i GATE sono tipicamente associati a:

- **Gestori degli interrupt** (invocazioni **asincrone**). Ad esempio da livello user a liv. Kernel uso un gate.
- **Trap software** (invocazioni **sincrone**), che siano esse delle eccezioni (e.g. page fault) o delle system call.

Tutto dipende da ciò che c'è nella GDT comunque. Inoltre per codificare questi livelli ci servono due bit, che sono proprio i due bit dei tre bit di protezione visti nelle pagine prima. Infatti se vediamo:



I due Kernel routine sono nello stesso segmento S_1 . La user routine è invece in un altro segmento S_2 .

- Il GATE kernel routine **A** ha livello 0 (del segmento), un certo offset rispetto S_1 , e il “max” livello di privilegio per usare questo GATE è 0. Quindi User non può andarci, essendo livello 2 e non 0.
- Il GATE kernel routine **B** è anch'esso di livello 0, ha un certo offset₂ e il “max” livello di privilegio per usare tale GATE è 3, quindi l'User può usarlo, essendo livello 3.

User entra in Kernel routine **B** mediante **cross-segmento** (perchè passo da S_2 a S_1) e poi da Kernel Routine **B** a Kernel Routine **A** mediante **intra-segmento** (perchè parto da S_1 e rimango in S_1).

Composizione degli indirizzi x86 con la segmentazione

Come analizzato precedentemente, all'interno delle architetture x86 gli indirizzi vengono indicati nel seguente modo:
`<segment_selector_register, displacement>`.

I registri/selettori di segmento sono 6:

- **CS (code segment register)**: indica qual è il segmento di memoria che contiene il codice che stiamo correntemente utilizzando in CPU.
- **SS (stack segment register)**: indica qual è il segmento di memoria che ospita lo stack su cui stiamo correntemente lavorando.
Esempio: la push prende ptr stack come offset del segmento SS.
- **DS (data segment register)**: indica qual è il segmento dati corrente, ovvero il segmento che bisogna utilizzare all'occorrenza di un'istruzione macchina che prevede un accesso alla memoria. Di default.
- **ES (data segment register)**: ha la stessa funzionalità di DS, ma è riservato ad alcune specifiche istruzioni macchina (i.e. le istruzioni che coinvolgono le stringhe, come stos e movs).
- **FS (data segment register)**: è stato aggiunto nei sistemi 80386, e viene utilizzato solo nel momento in cui è previsto dal programmatore o dal compilatore. In particolare, può essere sfruttato in maniera esplicita a livello di programmazione. Usato di solito con *Thread Local Storage*, per avere viste diverse per ogni thread.
- **GS (data segment register)**: è stato aggiunto nei sistemi 80386, ed è perfettamente analogo a FS. Usato in contesti *Per CPU – memory, cioè varia per ogni CPU*.

Abbiamo già accennato la struttura dei **segment selector**. Analizziamola nel dettaglio:

- I primi 13 bit rappresentano l'indice della entry della GDT / LDT relativo al segmento d'interesse.
- Il bit successivo (**Table Indicator – TI**) indica se correntemente stiamo utilizzando la GDT oppure la LDT.
- Gli ultimi due bit (**Requestor Privilege Level – RPL**) indicano il livello di protezione a cui stiamo correntemente lavorando.

Per il registro di segmento CS, il RPL è chiamato anche **CPL (Current Privilege Level)**, che rappresenta il livello di protezione corrente del thread in esecuzione.

NB: Queste cose appena viste hanno un ruolo *paragonabile* alla segmentazione, quindi Linux usa la segmentazione per il minimo necessario. Per questo si usa principalmente una **tabella**, ed il selettore di segmento ci porta da una parte all'altra sulle possibili due tabelle (ma alla fine se ne usa una e ci si spiazza su quella). Dipende dalla struttura software.

Torniamo ora al primissimo esempio che abbiamo fatto in questo capitolo:

```
mov (%rax),%rbx  
push %rbx
```

- L'accesso in memoria effettuato dall'istruzione **mov** coinvolge in modo implicito il segment selector **DS**.
- L'accesso in memoria effettuato dall'istruzione **push** coinvolge in modo implicito il segment selector **SS**.
- L'accesso in memoria effettuato per eseguire il **fetch** delle istruzioni coinvolge in modo implicito il segment selector **CS**.

GDT Entries

Ogni entry della GDT ci dà informazioni sul segmento associato a un dato indice. La tabella è necessaria, perchè il registro può essere cambiato e quindi devo mantenere le informazioni.

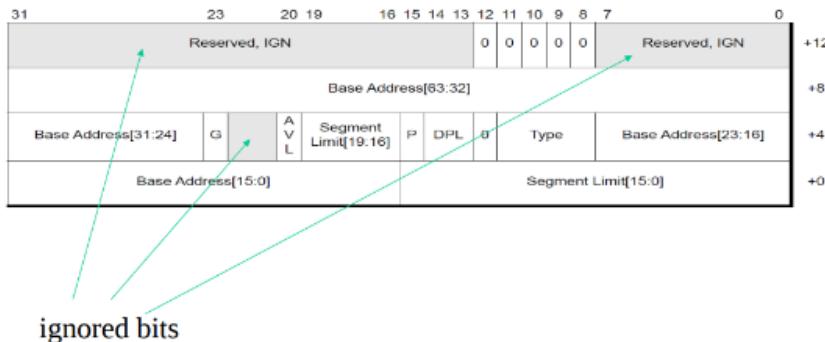
80386 protected mode

31	16	15	0
Base 0:15		Limit 0:15	
63 56 55 52 51 48 47 40 39 32			32
Base 24:31	Flags	Limit 16:19	Access Byte
			Base 16:23

- **Base:** indica l'indirizzo base del segmento puntato dalla entry. La GDT ci porta alla base del segmento. Per esprimere un segmento utilizziamo *base 0:15*, *base 16:23*, e *base 24:31*.
- **Limit:** indica la dimensione del segmento.
- **Flags:** tra i flag rientra il **granularity bit**:
 - se vale 0, il campo **limit** viene espresso in byte.
 - se vale 1, il campo **limit** viene espresso in pagine da 4 KB. Se lavoro con blocchi 4KB, ho *limit 0:15 e 16:19* cioè 20 bit, che combinati con 4KB = 2^{12} mi danno fino a $2^{32} = 4$ GB, la taglia dei segmenti.
- **Access byte:** qui si hanno i bit che descrivono la **protezione** del segmento. Tra questi rientrano i due **privilege bit** (che infatti vengono copiati all'interno del registro di segmento) e l'**executable bit**, che indica se all'interno del segmento c'è del codice che può essere eseguito.

Long mode

Sarebbero 64 bit, anche se effettivi sono 48.



Accesso alle GDT entries

È possibile accedere direttamente alla GDT (che è in memoria lineare, in quanto noi con essa gestiamo la memoria segmentata) via software sfruttando il **gdtr register**, che è un registro di tipo *packed* composto da due campi:

- La parte *bassa* tiene traccia dell'indirizzo lineare in cui si trova la tabella (32 bit per la 80386 protected mode, 64 bit per la long mode).
- La parte *alta* indica la taglia della tabella (16 bit). Cioè il numero di elementi.

L'istruzione (non privilegiata) che permette di *leggere* il **gdtr register** è la **Store Global Descriptor Table Register (sgdt)**, i cui dettagli sono riportati di seguito:

Opcode	Mnemonic	Description
0F 01 /0	SGDT m	Store GDTR to m.

Description
Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the low 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s. SGDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3 for information on loading the GDTR and IDTR.

Operation
<pre> if(OperandSize == 16) { Destination[0..15] = GDTR.Limit; Destination[16..39] = GDTR.Base; //24 bits of base address loaded Destination[40..47] = 0; } else { //32-bit Operand Size Destination[0..15] = GDTR.Limit; Destination[16..47] = GDTR.Base; //full 32-bit base address loaded } </pre>

IA-32 Architecture Compatibility
The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; the Pentium 4, Intel Xeon, P6 family, Pentium, Intel486, and Intel386 processors fill these bits with 0s.

Su x86_64 utilizzo 8 byte per l'indirizzo, e 2 byte per la size, in totale 10 byte. D'altra parte, esiste anche la **Load Global Descriptor Table Register**, che è un'istruzione **privilegiata** (ovvero invocabile solo se ci troviamo nel ring 0) che modifica il contenuto del gdtr register.

NB: non esiste un'unica GDT, bensì *una GDT per ogni hyperthread* presente all'interno della macchina. Questo fa sì che ciascun thread possa avere una visione diversa della memoria a seconda di in quale hyperthread gira.

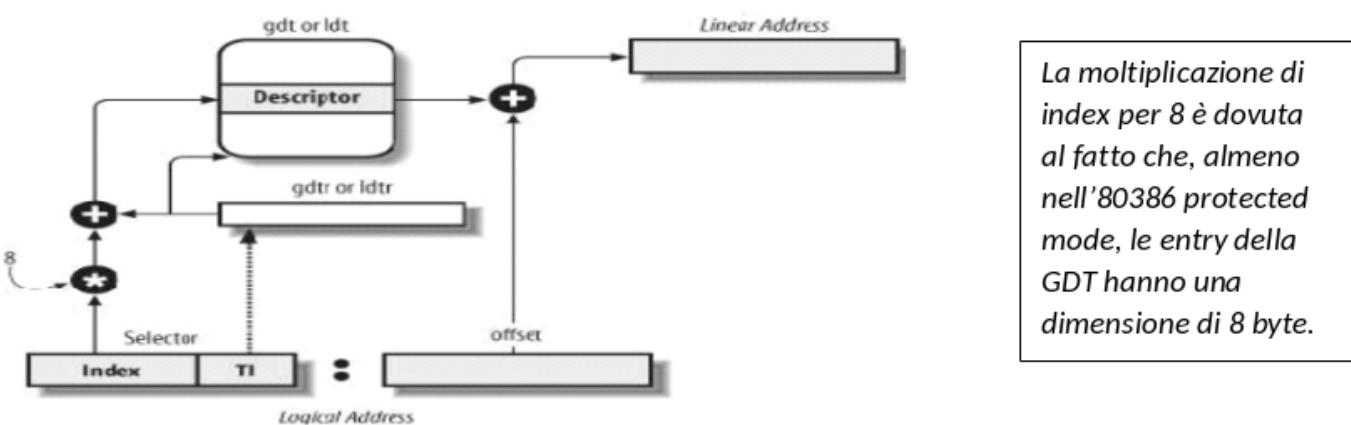
Se ho 2C/4T → posseggo 4 GDT. Per memorizzare le informazioni del registro **sgdt**, necessito di una struttura di tipo *packed*.

Se vediamo l'indirizzo in cui si colloca la GDT, troviamo un indirizzo *alto*, cioè in fondo all'address space.

Come già detto, ho una GDT per ogni hyperthread, che mi permette, mediante una struttura comune, di cambiare alcune entry, creando **viste diverse**. Nei Sistemi operativi moderni, non posso saltare da una GDT all'altra.

Schema di accesso alle GDT entries, livello hardware

A questo punto della trattazione, possiamo pensare che, per accedere a un qualsiasi dato in memoria, è necessario un accesso preliminare alla **GDT che si trova sempre in memoria**, proprio come rappresentato nella seguente figura:



Ciò però non ci piace perché introduce parecchio overhead di esecuzione. Quello che sia fa, dunque, è *portare nel processore le informazioni di alcune entry della GDT, in una sorta di meccanismo di caching*. Chiaramente, se una entry della GDT viene aggiornata, l'aggiornamento viene riportato all'interno della cache, in modo tale che quest'ultima sia sempre consistente.

Segmenti code/data in Linux

Come riportato nella figura nella pagina seguente, **in Linux si hanno diversi segmenti (ma non tutti) con un indirizzo base pari a 0**, il che porta a una *sovraposizione*. Questa scelta è dovuta al fatto che in tal modo è possibile

programmare software *molto portabile*, poiché adatto sia alle macchine che prevedono l'utilizzo della segmentazione, sia a quelle che non lo prevedono.

Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0ffff	1	10	3	1	1
user data	0x00000000	1	0ffff	1	2	3	1	1
kernel code	0x00000000	1	0ffff	1	10	0	1	1
kernel data	0x00000000	1	0ffff	1	2	0	1	1

x86-64 directly forces base to 0x0 for the corresponding segment registers

Is the segment present?

Can we read/write/execute?

La tabella vale in particolar modo per i sistemi *x86* a 32 bit. Per quanto concerne l'*x86-64*, il fatto che alcuni segmenti abbiano una base pari a 0 viene imposto non più dal software mediante la GDT, bensì direttamente dall'hardware. Questo vale in particolar modo per i registri di segmento CS, SS, DS ed ES, che puntano necessariamente a un segmento con base 0.

Se abbiamo un offset per identificare un dato o una istruzione, in realtà questo è un offset assoluto (in quanto, assumendo che le basi siano tutte 0, non sono discriminanti per capire la posizione), *si identifica in maniera univoca una posizione nell'address space*. D'altra parte, i registri di segmento FS e GS possono puntare a una entry della GDT relativa a un segmento con *indirizzo base arbitrario*: ciò permette, a parità di istruzione macchina, di accedere a locazioni di memoria differenti semplicemente andando ad aggiornare FS/ GS. Da qui si può avere la **per-thread memory**: FS (così come GS) può puntare a una entry della GDT relativa a un segmento TLS, e questa entry può essere modificata a piacere in modo tale che possa puntare a un certo Thread Local Storage all'interno dell'address space piuttosto che a un altro. In particolare, quando è in esecuzione un thread A, quella entry della GDT può contenere come indirizzo base TLS_A, mentre quando è in esecuzione un thread B, quella entry della GDT può contenere come indirizzo base TLS_B, cosicché thread differenti, a parità di istruzione macchina, possano accedere a punti della memoria diversi (i.e. a sezioni TLS diverse), ove richiesto. Quindi, FS/GS, associate al contesto del thread nella CPU, vengono aggiornate a seconda del thread in esecuzione, e un thread può vedere in GS una certa base, mentre un secondo thread vede in GS un'altra base. Inoltre, sia per spiazzamenti *kernel* sia *user*, si parte sempre da 0 per gli altri segmenti. La particolarità è che in questi altri segmenti carico anche il livello di protezione, quindi **posso arrivare ovunque, ma non posso modificare tutto!** La protezione normalmente è statica, ma posso cambiarla.

Regole di aggiornamento dei segment selector

- Poiché CS mantiene il **CPL (Current Privilege Level)**, può essere aggiornato soltanto mediante dei “**control flow variations**” (i.e. dei salti che permettono di *spostarci da un segmento a un altro* all'interno del flusso di esecuzione), ma non può essere aggiornato in modo esplicito dal programmatore.
- Tutti gli altri segment selector possono essere aggiornati esplicitamente, a patto che il nuovo **RPL (Requestor Privilege Level)** non sia relativo a un livello di protezione migliore rispetto a quello indicato dal CPL corrente. Chiaramente, con *CPL = 0* è concesso qualunque aggiornamento degli altri segment selector.

Quali sono le GDT entries in un sistema Linux x86?

Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)
	0xF8 double fault TSS

- Mentre viene eseguito codice di livello *user*, il registro CS referenzia la entry *user code* della GDT; viceversa, mentre viene eseguito codice di livello kernel, CS referenzia la entry *kernel code* della GDT.
- Quando vengono acceduti dati di livello user, il registro DS referenzia la entry *user data* della GDT; viceversa, mentre vengono acceduti dati di livello kernel, DS referenzia la entry *kernel data* della GDT.

Sono presenti **tre** TLS, uno associato al segmento FS (TLS vero e proprio), uno al segmento GS (usato dal kernel in contesti per-CPU-memory), ed il terzo perchè è sempre possibile dover lavorare in kernel mode.

Ricapitolando:

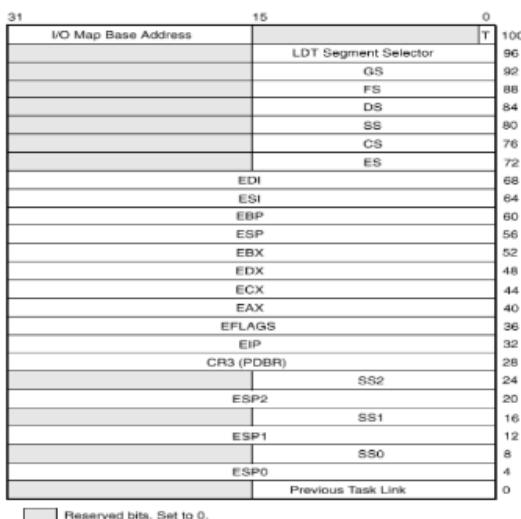
GDT ha entry che descrive, nell'AS osservato dalla CPU, il segmento in questione. Noi vediamo solamente gli offset, perchè i vari segmenti sono posti “*uno sopra l'altro*”, (quindi in uno stesso rettangolo non ho solamente CS, ma CS/DS etc...). Questo perchè non si usa molto la segmentazione.

Se prendo CS, ho anche le info di controllo (rilevanti), non c'è solo l'indirizzo. Questo perchè, come abbiamo visto, CS e DS sono *sia user sia kernel* (e le differenzio mediante i bit). Le tre TLS implementano Thread Local Storage (cioè *vista della memoria per un thread*). Esse vengono aggiornate a seconda del thread, portando alla visione di zone diverse; stiamo lavorando con il segmento FS, e quindi una entry è usata per l'offset rispetto tale segmento.

TSS Task State Segment

È il segmento in cui viene descritto lo stato di un **task**, dove il task è **il thread correntemente in CPU**. Qui sono riportati gli **snapshot** del CPU-core in cui il task viene eseguito e le informazioni necessarie per supportare correttamente il modello ring. Più precisamente, esistono in tutto *più aree TSS*, ciascuna delle quali è relativa a un CPU-core. Di fatto, ogni GDT mantiene all'interno della entry associata al TSS un indirizzo differente. Ho un array di TSS (non per forza contigui, anche sparsi va bene), *uno per ogni CPU (e puntato da una CPU)*. Salvo **tutto** ciò che c'è in una CPU (stack, registri sia general sia specific purpose) in un'area puntata da TSS. Ciò mi permette, quindi, di salvare un'immagine e caricarne un'altra, in un'operazione simile al context switch. Oggi viene usato solo per avere informazioni sul thread corrente, non per context switch. Il TSS ha un **DPL (Descriptor Privilege Level)** pari a 0 poiché viene usato per l'operatività a livello sistema; di conseguenza, racchiude delle informazioni che possono essere sfruttate esclusivamente a livello kernel. Di seguito viene raffigurata un'immagine che riporta in maniera più dettagliata il contenuto del TSS nei sistemi x86 a 32 bit. Questo segmento TSS non viene fissato a base 0 (come altri segmenti), in quanto non sarebbe identificabile se tutti partissero da base 0. Se due thread eseguono stessa operazione, con stessi parametri, tra cui l'offset, *cadrebbero* sulla stessa area. Allora discriminano con la *base*. Solo per SS,DS,ES,CS si parte da 0.

segmento FS, e quindi una entry è usata per l'offset rispetto tale segmento.



- Le zone di memoria riservate ai registri EDI, ESI, ..., EIP concorrono a mantenere lo snapshot dello stato del processore.
- Le zone di memoria riservate ai segmenti GS, FS, ..., ES mantengono gli indici della GDT in cui sono riportati i segmenti a supporto dell'esecuzione del thread (del task).
- Le zone di memoria più in basso tengono traccia della posizione in **memoria degli stack** utilizzati dal thread corrente. Di fatto, per ciascun thread, devono esistere per lo meno quattro stack, uno per ciascun livello di privilegio. Questo perchè le informazioni riportate sullo stack quando il thread esegue al livello di **protezione 3**

non devono mischiarsi con le informazioni riportate sullo stack quando il thread esegue a un livello di protezione superiore (e così via); in altre parole, quando si cambia il livello di privilegio, è necessario cambiare anche lo stack. In particolare, si tiene traccia di tre stack pointer:

- ESP0 (= stack pointer usato quando si accede alla modalità ring 0),
- ESP1 (= stack pointer usato quando si accede alla modalità ring 1),
- ESP2 (= stack pointer usato quando si accede alla modalità ring 2).

Per quanto riguarda lo stack pointer relativo alla modalità ring 3, viene salvato da qualche parte in memoria quando si passa a un livello di protezione superiore. Ho anche supporto ai RING, perché da liv.3 a salire, mi servono i GATE.

Per quanto invece riguarda i sistemi x86-64, le zone di memoria contenenti gli stack pointer diventano a 64 bit. Vengono sacrificati tutti i registri general purpose (EDI, ESI, ..., EIP), per far spazio a info stack area. Ciascun TSS viene **aggiornato** ogni qual volta si ha context switch, ovvero ogni qual volta nella relativa CPU un thread viene deschedulato a vantaggio di un altro thread. Inoltre, quando un thread in CPU necessita una risposta per proseguire, la CPU legge la risposta proprio da TSS.

offset	31-16	15-0
0x00	reserved	
0x04	RSP0 (low)	
0x08	RSP0 (high)	
0x0C	RSP1 (low)	
0x10	RSP1 (high)	
0x14	RSP2 (low)	
0x18	RSP2 (high)	



LDR Load Task Register

È un'istruzione che permette di aggiornare il **registro di TSS**, che è un registro che serve a mantenere informazioni sul TSS; in particolare, il TSS register è un registro **packed** (= con più di un'informazione) che tiene traccia di:

- L'indice della entry della GDT usata per descrivere dove si trova il segmento TSS in memoria RAM (o eventualmente cache). Quindi posso raggiungere informazioni più o meno velocemente.
- *L'indirizzo lineare dove si trova il TSS.* Questo registro permette dunque di **non passare per la GDT** quando si vuole ottenere la posizione in memoria del TSS. Ciò è importante nel momento in cui un thread cambia molte volte il livello di privilegio durante la sua esecuzione e ha necessità di accedere al TSS per recuperare lo stack pointer relativo al livello di protezione in cui è entrato; se c'è bisogno di accedere ogni volta alla GDT, si ha chiaramente un alto overhead di esecuzione.

Replicazione della GDT

Sappiamo che **ciascuna CPU ha la sua GDT**, e sappiamo che questo permette di avere un *TSS differente per ciascun thread in esecuzione*, cosicché ogni thread sappia dove andare a pescare le informazioni necessarie (come lo stack pointer opportuno) per passare dall'esecuzione in modalità user all'esecuzione in modalità kernel e viceversa. In realtà, si hanno altri motivi rilevanti per cui la GDT è replicata:

- **Performance:** nelle architetture NUMA, se ci fosse un'unica GDT, questa sarebbe vicina ai CPU-core di un solo nodo NUMA, mentre rispetto agli altri CPU-core sarebbe molto lontana. Avere una GDT per ogni CPU-core porta tutti i CPU-core a poter accedere alla propria GDT in modo efficiente.
- **Trasparenza alla separazione degli accessi ai dati:** se due thread eseguono la stessa identica istruzione macchina di un programma utilizzando gli stessi parametri, è possibile, **avendo GDT diverse**, che essi vadano ad accedere a due locazioni di memoria lineare differenti. Ciò è possibile grazie al segmento GS: supponiamo che un thread t_1 in esecuzione sul CPU-core₁ abbia un segmento GS con base B e che un thread t_2 in esecuzione sul CPU-core₂ abbia un segmento GS con base B' ; supponiamo inoltre che i due thread debbano eseguire una stessa istruzione che prevede un accesso in memoria nel segmento GS. Allora, t_1 e t_2 faranno riferimento al **medesimo**

offset della propria GDT e, all'interno di tali offset, sono indicati indirizzi lineari differenti, che corrisponderanno a indirizzi fisici differenti (che sono proprio le locazioni di memoria a cui i thread accederanno).

Alla base del meccanismo appena descritto si ha la **per-CPU memory**, secondo cui ogni CPU / CPU-core deve poter avere a disposizione e accedere direttamente ai suoi metadati e alle sue informazioni. Se non avessimo la per-CPU memory, all'interno del kernel bisognerebbe avere un array *A* in cui ogni entry mantiene i metadati relativi a una singola CPU; dopodiché, per accedere alle sue informazioni private, ciascuna CPU dovrebbe invocare l'istruzione `cpuid` per accedere alla propria entry di *A*. Ma sappiamo che ‘cpuid’ è un'istruzione devastante dal punto di vista delle prestazioni, poiché è serializzante e causa lo squash della pipeline.

Esempio di utilizzo della per-CPU memory:

```
DEFINE_PER_CPU(int, x); //definizione di una var x di tipo int che è per-CPU  
int z = this_cpu_read(x); //load del valore di x all'interno di una variabile z
```

Lo statement appena descritto è equivalente alla seguente istruzione macchina: `mov ax, gs:[x]`

Comunque sia, per operare senza particolari *define*, è anche possibile recuperare l'indirizzo lineare in cui è posta la variabile per-CPU *x*:

```
y = this_cpu_ptr(&x);
```

Ogni CPU ha la sua variabile, ovvero ho associato un'area per ogni zona usata da per-CPU-memory. GS è “assicurato”, so che c'è perchè anche il kernel lo usa, ciò che faccio io è aggiungere altri pezzi all'area.

Posso scrivere funzioni in cui definisco variabili per una CPU, le modifco, e poi le faccio leggere a quella CPU?

No, concettualmente è sbagliato, perchè in qualsiasi istante il thread in esecuzione potrebbe essere deschedulato ed essere riassegnato ad un'altra cpu. Soffrirei quindi la *preemption* e le *interrupt*. Dovrei, per risolvere, lavorare SOLO su una specifica CPU, mediante `get_cpu` e `leave_cpu`.

TLS Thread Local Storage

È una zona di memoria all'interno dell'address space dell'applicazione che contiene le variabili locali riservate a uno specifico thread. Esiste dunque un TLS per ogni thread. Questa zona di memoria può essere utilizzata con l'aiuto del segmento FS. In particolare, nel momento in cui un thread *t* viene creato, viene allocata un'area di memoria all'interno dell'address space (un TLS, appunto) e viene comunicata al sistema operativo la presenza di tale TLS.

Ogni qual volta che *t* viene schedulato in CPU, una particolare entry della GDT (TLS#1 / TLS#2 / TLS#3) viene aggiornata in modo tale da puntare alla base del TLS di *t*, e il nuovo contenuto della entry viene caricato all'interno del segmento FS. Sarà proprio FS a essere utilizzato direttamente dal codice macchina per accedere al TLS. Il meccanismo appena descritto è alla base della **per-thread memory**.

arch_prctl

Abbiamo visto quindi che GS è importante per il kernel (*per-CPU memory*), mentre FS è di supporto al TLS (*per-thread memory*). Ecco le system call che servono per la gestione dei segmenti FS e GS. Variano in base all'operazione:

- `int arch_prctl (int code, unsigned long addr)`
- `int arch_prctl (int code, unsigned long addr)`

Ad esempio:

- Se si vuole eseguire una **SET**, il contenuto del registro **GS** (= indirizzo base del segmento GS), bisogna settare il parametro **code** con un valore che indica “voglio eseguire una store su GS” e il parametro **addr** con l'indirizzo base che si vuole assegnare al segmento GS (dove **addr** è un `unsigned long`, che è un valore a 64 bit).
- Se si vuole eseguire una **GET** dell'indirizzo base del segmento GS o FS del thread corrente, il parametro **addr** è invece un puntatore a `unsigned long`, in quanto risulterà essere un parametro di output anziché un parametro di input.

In definitiva, il parametro **code** può assumere uno dei seguenti quattro valori:

- `ARCH_SET_FS`
- `ARCH_GET_FS`
- `ARCH_SET_GS`

- ARCH_GET_FS

Quindi con le GET ottengo l'indirizzo di memoria in cui c'è la base di FS o GS, mentre con la SET posso dire dove devono puntare FS o GS nell'address space (non modifco il contenuto, solo a dove puntano).

Vengono anche aggiornati i relativi registri. In entrambe le system call ho a che fare con indirizzi (forniti o ricevuti). Posso usare queste chiamate per *superare* il TLS, in particolare:

Prendo l'indirizzo TLS di un thread *A* e lo salvo in una variabile (ARCH_GET_FS). Successivamente, posso far puntare un thread *B* a questo address (ARCH_SET_FS), se poi questo ultimo thread *B* scrive, lo farà dove ha lavorato il thread *A*.

La system call restituisce 0 in caso di successo, -1 altrimenti.

Registri di controllo in x86_64

Originariamente i processori x86 disponevano di 4 **registri di controllo** (CR0, CR1, CR2, CR3) ma, in un secondo momento, è stato introdotto anche un quinto registro **CR4**.

- **CR0**: baseline; è il registro che determina qual è l'**operatività** del processore. Il suo 0-esimo bit indica se stiamo operando in *real mode* (la modalità primitiva di accesso alla memoria senza paginazione) o in *protected mode* (posso usare o meno la paginazione). Per discriminare se stiamo operando in long mode (il che comunque è possibile solo se CR0 ci dice che siamo in protected mode), in realtà, bisogna ricorrere a un bit addizionale posto nel registro **EFER** (**Extended Feature Enable Register**), che appartiene alla classe **MSR** (**Model Specific Register**). Inoltre, il trentunesimo bit di CR0 indica se stiamo utilizzando la **paginazione**.
- **CR1**: registro riservato per l'operatività interna del processore.
- **CR2**: registro che tiene traccia dell'indirizzo lineare che ha causato un eventuale fault di memoria: tale informazione è utile per il gestore del page fault. Usato dal kernel (es: uso `mmap` ma la pagina non è ancora allocata). Poichè possono incorrere diversi page fault, non posso aspettare "troppo tempo" per esaminarlo.
- **CR3**: Se CR0 supporta *memoria virtuale* e paginazione, allora ho puntatore che referenzia la page table in memoria fisica. (quindi indirizzo fisico di Page table, perchè supporta la risoluzione di *indirizzo virtuale*).

La tabella illustrata di seguito riporta in modo più dettagliato le informazioni incapsulate nel registro CR0:

Bit	Name	Full Name	Description
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
1	MP	Monitor coprocessor	Controls interaction of WAIT/FWAIT instructions with T3 flag in CR0
2	EM	Emulation	If set, no x87 FPU is present, if clear, x87 FPU is present
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
29	NW	Not-write through	Globally enables/disable write-through caching
30	CD	Cache disable	Globally enables/disable the memory cache
31	PG	Paging	If 1, enable paging and use the CR3 register, else disable paging

Interrupt e trap

Sono i supporti che abbiamo per accedere al software di livello kernel. In particolare:

- Gli **interrupt** sono eventi **asincroni**, che consistono in una richiesta da parte di *dispositivi esterni alla CPU* di passare da un flusso di esecuzione in modalità *user* a un flusso di esecuzione in modalità *kernel*. Possiamo chiamare una **interrupt** in modo *sincrono* tramite la macro INT (interrupt), la quale sfrutta i GATE invece che le JMP. Essa permette, tramite i GATE, di passare in kernel mode.
- Le **trap** (o eccezioni) sono eventi **sincroni**, che possono essere causati dalle istruzioni *eseguite in CPU* e possono a loro volta portare a un passaggio da un flusso di esecuzione in modalità *user* a un flusso di esecuzione in modalità *kernel*. Molteplici esecuzioni del medesimo programma, a parità di input, potrebbero (ma non necessariamente) sollevare le stesse eccezioni; questo può dipendere anche da ciò che avviene dal punto di vista della concorrenza: se per esempio nel programma abbiamo una divisione per una variabile *v* condivisa tra più thread, si può avere o non avere un'eccezione in prossimità di tale divisione a seconda se nel frattempo un qualche thread abbia impostato la variabile *v* a zero o meno. **Storicamente, le trap sono state utilizzate come un meccanismo esplicito e on-demand per passare all'esecuzione in modalità kernel.**

In conclusione, gli interrupt e le trap portano a *migliorare* il livello di privilegio con cui il thread gira. Sappiamo che, per far ciò, bisogna ricorrere al meccanismo dei **GATE**.

Il kernel mantiene una **trap / interrupt table**, che nei sistemi x86 è chiamata **Interrupt Descriptor Table (IDT)**. In generale, si chiama **Trap Interrupt Table (TIT)**. Ciascuna entry di questa tabella contiene un **GATE descriptor**, che fornisce le informazioni sull'indirizzo destinazione associato al GATE (i.e. `<segment.id, offset>`) e il livello di protezione massimo (ovvero peggiore) necessario per poter accedere al GATE. Di conseguenza, il contenuto della trap/interrupt table viene sfruttata per determinare se ciascun accesso a un qualche GATE può essere abilitato o meno, e questo viene fatto con un confronto con i registri di segmento (in particolare **CS**) che specificano appunto il *current privilege level* (CPL). Le varie entry della IDT, in realtà, possono avere anche altri metadati, come ad esempio un'informazione che indica se, a seguito del salto verso il segmento destinazione, il thread è interrompibile o meno. È possibile avere all'interno della trap/interrupt table più entry associate al medesimo GATE, di cui una può prevedere ad esempio un livello di protezione massimo differente rispetto a un'altra. Se uso un GATE non permesso, genero una TRAP. Uno stesso GATE può avere più entry uguali nella IDT. Non ho limitazioni!

Ricapitolando, le variazioni del flusso di esecuzione possono avvenire in tre modi possibili:

- **Intra-segmento**: avviene con un'istruzione standard di **jump** (e.g. `JMP <displacement>`). Qui il firmware si limita a verificare se il displacement cade all'interno del segmento in cui stiamo correntemente effettuando il fetch delle istruzioni.
- **Cross-segmento**: avviene con un'istruzione di **long jump** (e.g. `LJMP <segment.id>, <displacement>`). Qui il firmware *verifica se non stiamo migliorando il livello di privilegio* e se il displacement cade all'interno del segmento destinazione. Quindi NO GATE.
- **Cross-segmento via GATE**: avviene con un'istruzione di **trap** (e.g. `INT <table displacement>`). Qui il firmware controlla se il salto è permesso andando a comparare il livello di privilegio attuale col massimo previsto dalla entry della trap/interrupt table specificata come parametro dell'istruzione. Se il salto è concesso, viene acceduta la GDT per capire qual è il nuovo livello di privilegio da registrare all'interno di CS.

Nelle architetture x86 la IDT è accessibile mediante il registro **idtr**, che è un altro registro **packed** contenente l'indirizzo lineare in cui è ubicata la IDT e il suo numero di entry. Questo registro può essere modificato tramite l'istruzione macchina **LIDT**, in modo tale da cambiare la IDT a cui far riferimento.

Per quanto riguarda i sistemi Linux e Windows, il **GATE** per l'accesso al livello di protezione 0 on-demand (i.e. mediante l'istruzione **INT**) è **unico**. In particolare:

- È la **entry 0x80** della IDT per i sistemi Linux (per cui l'istruzione ammissibile è `INT 0x80`).
- È la entry 0x2e della IDT per i sistemi Windows (per cui l'istruzione ammissibile è `INT 0x2E`).
- Qualunque altro GATE è riservato esclusivamente alla gestione delle trap e degli interrupt. Con INT, sto puntando al registro di tipo **packed idtr**.

Puntiamo ad una struttura contenente indirizzo logico (non fisico) e la size. Tramite istruzione macchina **LIDT**, posso eseguire il load del contenuto.

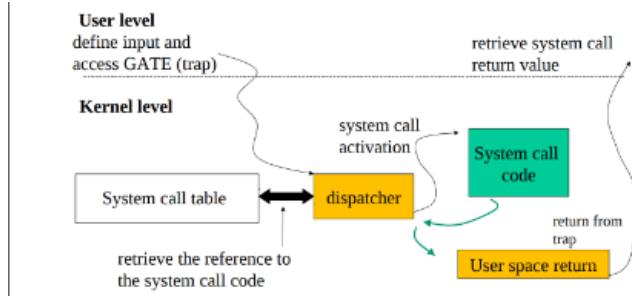
Osservazione: Se caricassi un modulo per cambiare la gestione delle interrupt, questo avverrebbe per la CPU in opera, e non per le altre CPU, che punterebbero alla vecchia gestione. Se facessi puntare tutte le CPU a questa nuova gestione delle interrupt **romperei tutto**, perchè, per via della mitigazione PTI dovuta a Meltdown, starei facendo questi puntamenti solo a livello kernel, mentre a livello user non so dove è stata memorizzata la mia nuova gestione delle interrupt.

Il blocco di codice del kernel raggiungibile a partire dall'istruzione di INT è detto **dispatcher**, che è un modulo software in grado di triggerare l'attivazione delle system call. Il dispatcher identifica quale *entry* usare. Questa organizzazione è importante per motivi di scalabilità: di fatto, viene impegnata un' **unica entry della IDT per tutte le system call che il kernel mette a disposizione**.

Dispatching delle system call

In base a quanto detto prima, per gestire le singole system call, si ricorre alla **system call table**, che è una struttura dati software che, per ogni entry, mantiene l'indirizzo di memoria di una specifica system call. Perciò, quando un processo deve invocare una system call, si rivolge al dispatcher, e lo fa passandogli l'indice numerico indicante la entry della system call table associata alla system call da invocare ed eventuali altri parametri. Dopodiché il dispatcher invocherà la system call mediante l'istruzione **call** che accetta come parametro l'offset del segmento corrente (che è

quello relativo al codice di livello kernel) verso cui bisogna saltare (i.e. in cui inizia il codice della system call). La system call, a seguito della sua esecuzione, fornisce il suo valore di output all'interno di un registro di processore. Più precisamente, restituisce il controllo al dispatcher che si occuperà di attivare l'esecuzione del blocco di codice che ha lo scopo di ritornare alla modalità user: è qui che si ha l'istruzione **RTI** (return from interrupt) e si fornisce l'output della system call al chiamante. Di seguito è mostrato uno schema che fornisce un quadro d'insieme di quel che succede.



Nelle architetture moderne, in realtà, il dispatcher è più complesso di così: ad esempio implementa anche delle facility di sicurezza.

Atomicità dell'esecuzione del dispatcher

Il dispatcher di default è **interrompibile**. Per proteggersi da eventuali interrupt, è possibile fare uso di due istruzioni:

- **CLI** (clear interrupt bit), la quale elimina la possibilità per gli interrupt di interrompere l'esecuzione del dispatcher,
- **STI** (set interrupt bit), la quale ripristina l'opzione precedente.

Ciò potrebbe non essere sufficiente per l'atomicità dell'esecuzione del dispatcher: nelle architetture multiprocessore, si è soggetti a un accesso concorrente ai dati. Per ovviare a questo problema, si può ricorrere alle soluzioni classiche di **spinlock** o **Compare And Swap**.

Componenti sw per la gestione delle system call

- Lato **user**: Si ha un modulo software che fornisce i parametri di input al GATE (e, quindi, al dispatcher), attivi il GATE e recuperi il valore di ritorno della system call. Il gate è INT 0x80, gli altri gate sono in IDT e si usano per runtime errors e interrupt. Quindi c'è una trap (appartenente alla IDT) che permette all'utente di usare il dispatcher. Il suo handler è interrompibile (ma abbiamo visto prima come settarlo).
In x86_64 non si usa più il meccanismo di trap.
- Lato **kernel**: Si hanno il dispatcher, la system call table e il codice vero e proprio della system call (il quale, a sua volta, potrebbe anche invocare altre funzioni e così via). Non ho problemi in questa modalità, visto che sono elementi usati anche dallo user. Con una politica *per-cpu-memory ancora meglio*.

Per aggiungere una nuova system call, si possono seguire più possibili approcci:

1) Definire un nuovo dispatcher:

Questo lo si fa nel caso in cui si voglia dispatchare la nuova **system call con regole diverse da quelle previste dal dispatcher già esistente**. Tale approccio richiede che esista una entry della IDT libera e che la si impegni per l'utilizzo del nuovo dispatcher. Non è consigliato soprattutto per le system call che devono essere montate in modo *definitivo* all'interno del kernel poiché è giusto avere tutti i servizi allineati nel dispatching e nell'attivazione.

2) Ampliare la system call table:

Il problema grosso di quest'approccio è che porta alla necessità di effettuare enormi modifiche all'interno del Makefile del kernel, per cui non è agevole nella pratica. Ad esempio, ampliare la tabella senza altre modifiche nel kernel potrebbe portare a un overlap delle strutture dati in memoria.

3) Sfruttare le entry libere della system call table:

Come approfondiremo meglio tra breve, abbiamo la fortuna di avere delle entry libere all'interno della system call table, che è possibile utilizzare senza dover apportare grosse modifiche al kernel e al suo Makefile. Di conseguenza, è *questo l'approccio che si adotta nella pratica*. Chiaramente ciò è possibile nel momento in cui il formato della nuova system call è **compatibile** con quello predefinito per il sistema operativo su cui stiamo lavorando.

Indicizzazione delle system call

È un problema che consiste nell'associare ciascuna system call (*più precisamente il puntatore a ciascuna system call*) a un particolare codice numerico. Originariamente i sistemi Linux avevano lo schema di indicizzazione UNISTD_32. Oggi invece hanno UNISTD_64 ma ancora con una **retrocompatibilità** con UNISTD_32. A livello user troviamo le informazioni sull'indicizzazione dei servizi del kernel all'interno di appositi header di programmazione (`unistd_32.h`, `unistd_64.h`).

- Per lo schema di indicizzazione UNISTD_32, viene utilizzato proprio secondo le modalità che abbiamo descritto: si accede alla entry 0x80 della IDT per passare il controller al dispatcher che poi si occuperà di passare il controllo alla system call.
- Per lo schema di indicizzazione UNISTD_64, si fa uso di *un'altra system call table* e di *un altro dispatcher*, il quale però **non è accessibile tramite la IDT** e, quindi, tramite l'istruzione INT (ne approfondiremo più avanti il meccanismo). Il formato UNISTD_64.h è *retrocompatibile*, ovvero posso usare il dispatcher mediante TRAP, mentre nell'applicazione senza retrocompatibilità non devo usare la TRAP. Questo mi permette, ad esempio, di accedere ad una syscall indicizzata da k mediante il dispatcher D , e da k' mediante il dispatcher D' .

UNISTD_32 listing

```
#ifndef __ASM_X86_UNISTD_32_H
#define __ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
#define __NR_time 13
#define __NR_mknod 14
#define __NR_chmod 15
#define __NR_lchown 16
```

UNISTD_64 listing

```
#ifndef __ASM_X86_UNISTD_64_H
#define __ASM_X86_UNISTD_64_H 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
```

Task di livello USER per accedere al GATE

Sono i seguenti:

- 1) Specificare i parametri di input tramite dei registri di CPU, di cui uno è per il dispatcher (che sarebbe il codice numerico della system call da invocare) e i restanti, se esistono, sono da passare alla system call vera e propria.
- 2) Utilizzare istruzioni assembly (e.g. le trap) per triggerare i GATE: l'uso dell'assembly porta questo task ad essere machine dependent (chiaramente le istruzioni assembly dipendono dalla specifica architettura che stiamo utilizzando).
- 3) Recuperare il valore di ritorno della system call da appositi registri di CPU: anche quest'altra operazione è machine dependent.

Formato predeterminato per le system call

È una regola che stabilisce come devono essere scritti i moduli che realizzano i tre task di livello user di cui sopra. Avere un formato predeterminato per il modulo user delle chiamate delle system call permette al modulo user stesso di eseguire delle attività compatibili col funzionamento del dispatcher: questo porta ad avere uno stato del processore che il dispatcher sia in grado di leggere e fornire al dispatcher i parametri in modo tale che lui abbia la possibilità di interpretarli e di manipolarli correttamente. Per ottenere questa compatibilità, si ricorre a degli **appositi file header**, che contengono delle macro associate a particolari funzioni. Queste funzioni implementano proprio i tre task di livello user per accedere al GATE in modo da seguire il formato predeterminato. Non solo: nei file header è anche possibile sfruttare le macro per definire delle nostre nuove funzioni, che possono essere composte da delle parti scritte in C e parti scritte in **assembly** (ASM inline). Queste funzioni sono anche chiamate **stub di system call**.

Un vincolo che abbiamo è che al software di livello kernel, quando si vuole invocare una system call, è possibile passare al più 7 parametri: uno è appunto il codice numerico della system call che serve al dispatcher, mentre i restanti (al più 6) sono i parametri da fornire alla system call.

Vediamo un esempio di invocazione a una system call che non accetta parametri in input (e.g. `fork()`):

```

#define __syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name)); \
__syscall_return(type,__res); \
}

```

Assembler instructions

Tasks to be done after the execution of the assembler code block

Tasks preceding the assembler code block

- `__syscall0(type, name)` è la **macro** associata alla funzione `name()` scritta successivamente, il cui tipo di ritorno è specificato proprio da `type`. Nell'esempio, `syscall0`, passo 0 parametri, oltre tipo di ritorno e nome della funzione di ritorno.
- In `"0" (__NR_##name)`, il valore numerico (**l'indice**), viene associato al nome della system call `__NR_##name`. (*i.e.* nel caso della `fork()` il nome sarà `__NR_fork`). Tale valore deve essere posizionato all'interno del registro `eax/rax` affinché lo stato del processore sia coerente rispetto a quello che il dispatcher si aspetta. `name` viene usato per richiamare il simbolo `__NR_##name`, e prenderne il codice numerico.
- In `"=a" (__res)` si impone che il valore di `eax/rax`, a seguito dell'invocazione dell'istruzione `int $0x80`, venga registrato all'interno della variabile `__res`. In pratica, in `__res` viene inserito il valore di ritorno della system call.
- L'ultima riga del blocco di codice, infine, prevede che il valore di `__res` venga processato in qualche modo dalla funzione associata alla macro `__syscall_return(type, __res)`.

Approfondiamo quest'ultima riga:

```

/* user-visible error numbers are in the range -1 - -124:
see <casm-i386/errno.h> */

#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-125)) { \
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)

```

Case of `res` within the interval $[-1, -124]$

- Tipicamente, se l'esito della system call è positivo, il suo valore di ritorno è ≥ 0 ; altrimenti il valore di ritorno è nel range $(-124, -1)$. In questo secondo caso, viene *restituito* -1 al chiamante e viene impostato `errno` col codice di errore *specifico* (che è dato dal valore assoluto dell'output della system call).
- Il blocco di codice è racchiuso all'interno di un `do-while(0)`, che prevede dunque un'*unica iterazione* (non è di fatto un ciclo). È “*one-shot*” perchè ritorna subito qualcosa, in quanto passiamo forzatamente per il `return`. Questa strutturazione permette di inserire la macro `__syscall_return(type, __res)` in qualunque punto del codice: ad esempio in tal modo è possibile aggiungere un “punto e virgola” dopo la macro (i.e. alla fine del blocco di codice) o incapsulare il blocco di codice all'interno di un costrutto `if`. Vengono anche chiamati blocchi “*all-or-nothing*”.

Vediamo ora un esempio di invocazione a una system call che accetta un unico parametro in input “`arg1`” (e.g. `close()`):

```

#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long)(arg1))); \
__syscall_return(type,__res); \
}

```

2 registers used for the input

- `__syscall1 (type, name, type1, arg1)`, come è possibile notare, prevede 4 parametri:
 - I primi due sono in comune con `__syscall0`.
 - `type1` è il tipo di dato che la system call accetta in input

- `arg1` è il valore di input vero e proprio da passare alla system call.
- All'interno del blocco di codice della funzione, l'unica differenza rispetto al caso di `_syscall10`, sta nel pre-caricamento dei registri: in particolare, oltre a inserire in `eax/rax` il codice numerico associato alla system call da invocare, il blocco di codice carica in `ebx/rbx` il valore di input (*castato a long*) da passare alla system call stessa.

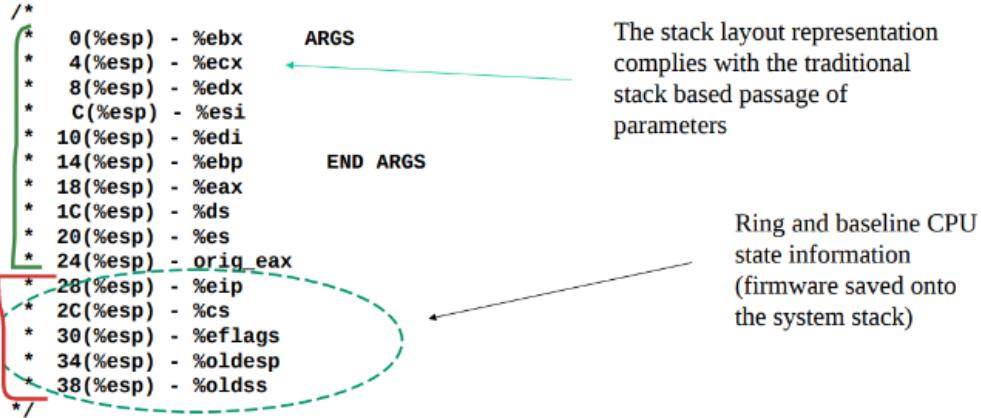
Per quanto invece riguarda le system call che accettano 6 parametri in input:

```
#define _syscall16(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5,type6 arg6) \
{ \
long __res; \
__asm__ volatile ("push %%ebp ; movl %%eax,%%ebp ; movl %1,%%eax ; int $0x80 ; pop %%ebp" \
: "=a" (__res) \
: "i" (__NR_##name), "b" ((long)(arg1)), "c" ((long)(arg2)), \
"d" ((long)(arg3)), "S" ((long)(arg4)), "D" ((long)(arg5)) \
"0" ((long)(arg6))); \
__syscall_return(type,__res); \
}
```

Qui è interessante osservare come, in realtà, le architetture x86 non dispongano di 7 registri general purpose, ma solo 6 (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, o comunque i corrispettivi dell'x86-64). Di conseguenza, quello che si fa è sfruttare anche il *registro ebp*. In particolare, in fase di pre-caricamento dati (prima di `INT 0x80`), in `eax` non viene inserito più il codice numerico della system call da invocare (che invece viene caricato in una locazione di memoria intera), bensì il **sesto parametro di input della system call** (quindi settimo parametro totale). In tal modo, subito prima dell'istruzione `int $0x80`, viene salvato il vecchio valore di `ebp` sullo **stack**, viene aggiornato il valore di `ebp` col sesto parametro della system call e viene aggiornato `eax` in modo da contenere il codice numerico della system call stessa. Chiaramente, a valle dell'esecuzione dell'istruzione `int 0x80`, il registro `ebp` dovrà essere ripristinato mediante una `pop`. Con `movl %1,%%eax` il valore 1 coincide con primo elemento notazione, cioè dove ho `__NR_##name`, ovvero il **numero della syscall** da attivare. Non potevo caricarlo direttamente in `eax`, in quanto usato come registro tampone per scriverci `ebp` (`movl %%eax, %%ebp`).

Convenzioni per l'invocazione delle syscall

Convenzione di UNISTD_32 per l'invocazione delle system call



Nell'immagine qui sopra è descritto come si presenta lo stack del kernel nel momento in cui viene invocata una system call.

- Ebx, ecx, edx, esi, edi, ebp sono i registri contenenti *i parametri da passare alla system call*.
- **Orig_eax** (che corrisponde al vecchio valore di `eax`) contiene il codice numerico della system call da invocare.
- Il registro `eax`, dopo che il suo vecchio valore è stato trasferito in `ebp`, dovrà contenere il valore di ritorno della system call.
- Si hanno anche:
 - I *registri di segmento* (ds, es, cs, oldss),

- L'*instruction pointer* da cui bisogna ripartire dopo l'invocazione della system call (eip),
- Lo *stato dei flag del processore* (eflags)
- Il *vecchio valore dello stack pointer* (oldesp), poiché si tratta di informazioni che devono essere memorizzate affinché vengano ripristinate a seguito dell'esecuzione della system call.

Tale organizzazione dello stack è **details**, in modo tale che il dispatcher sia estremamente semplice, le operazione compiute sono:

- Posizionare sullo stack tutto ciò che va da *ebx* a *orig_eax* nell'ordine giusto, i registri restanti vengono salvati sullo stack dal firmware.
- Far sapere alla system call *dove accedere sullo stack* per reperire le informazioni che le servono. Quindi c'è un intermediario, detto **wrapper**, che vede tutta la stack area, ma quando chiama l'oggetto mette solo i registri necessari (definiti dallo standard **ABI**, non c'entra nulla *asmlinkage*) ed aggiunge entropia.

Lo stack alignment 32 che abbiamo appena descritto è definito all'interno di una struct chiamata **pt_regs**, che specifica appunto l'ordine con cui devono essere collocate sullo stack le informazioni rappresentative per lo snapshot di CPU. Tra l'altro, il kernel ha la possibilità di accedere alle informazioni relative allo snapshot semplicemente tramite un puntatore a una struttura di tipo **pt_regs**.

Convenzione di UNISTD_32 per l'invocazione delle system call

```
/*
 * Register setup:
 * rax system call number
 * rdi arg0
 * rcx return address for syscall/sysret, C arg3
 * rsi arg1
 * rdx arg2
 * r10 arg3 (--> moved to rcx for C)
 * r8 arg4
 * r9 arg5
 * r11 eflags for syscall/sysret, temporary for C
 * r12-r15,rbp,rbx saved by C code, not touched.
 *
 * Interrupts are off on entry.
 * Only called from user space.
 */
```

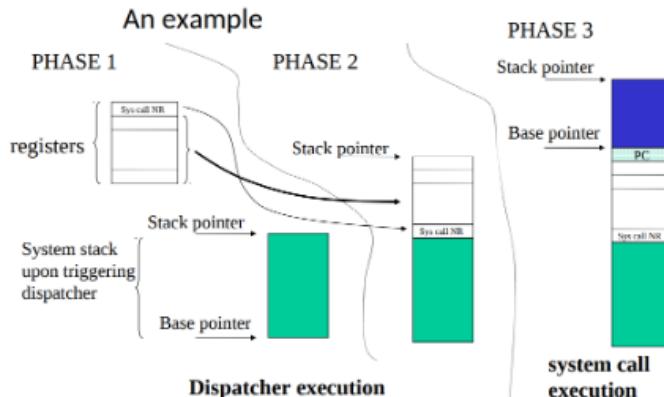
Qui vengono definite soltanto lo stato dei flag del processore (tramite il registro *eflags*) e le informazioni relative ai registri general purpose. Infatti, in **x86-64** lavoriamo primariamente con un'**architettura di dispatching completamente diversa che non utilizza i GATE** (tant'è vero che non si utilizza l'istruzione **INT** bensì qualcos'altro come **syscall/sysret**) e non ha la necessità di memorizzare le informazioni associate ai registri di sistema (eccetto *eflags*).

- Il registro **rax** contiene il codice numerico della system call da invocare.
- I registri *rdi*, *rsi*, *rdx*, *r10*, *r8*, *r9* contengono, nell'ordine, i parametri da passare alla system call. Il quarto argomento (**arg3**) può essere inserito nel registro **rcx** anziché in **r10 solo** nel caso in cui stiamo lavorando con del **codice puramente C** (senza ricorrere a costrutti o operazioni machine dependent); altrimenti **rcx** verrà popolato dal firmware con l'indirizzo di ritorno dal quale bisogna riprendere l'esecuzione una volta che si è tornati in modalità user.
- I registri *r12*, *r13*, *r14*, *r15*, *rbp*, *rbx* sono gestiti al livello del codice C ma non vengono toccati quando si passa il controllo al kernel. In particolare, se la funzione *callee* ha la necessità di utilizzare questi registri, sarà lei a dover preoccuparsi di salvarli all'inizio e ripristinarli prima di restituire il controllo al chiamante; il salvataggio e il ripristino degli altri registri, invece, sono a carico del chiamante. Questa regola, data dall'**ABI (Application Binary Interface) System V AMD64**, vale in generale e non solo per le system call, ed è molto utile nel momento in cui il caller* e il callee si spartiscono il lavoro di salvare i registri sullo stack.

Il fatto che diverse informazioni, come l'indirizzo di ritorno, non vengano più salvate sullo stack per l'invocazione delle system call rappresenta un grosso vantaggio dal punto di vista prestazionale perché evita parecchi accessi in memoria. Un altro effetto che si ha è che il software è l'unico componente a registrare le informazioni sullo stack, anche quelle direttamente gestite dal firmware (come *eflags*).

Dettagli sui passaggi dei parametri

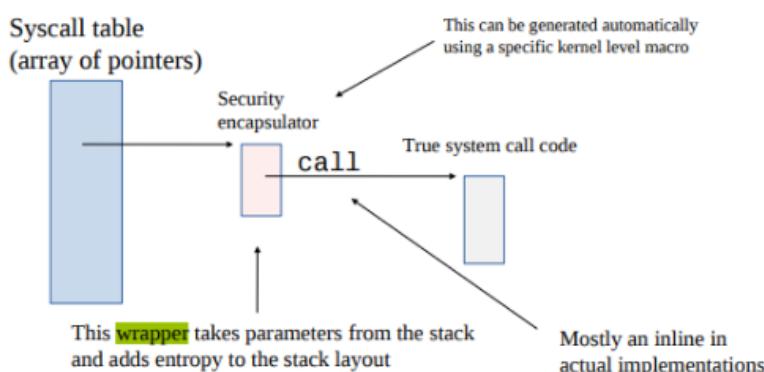
Come abbiamo anche accennato in precedenza, una volta ottenuto il controllo, il dispatcher prende uno snapshot di registri di CPU e lo memorizza sullo stack di livello di sistema. Dopo di che invoca la system call allo stesso modo di come si effettuano le chiamate a *subroutine* (i.e. mediante l'istruzione di **CALL**). La system call vera e propria recupererà i parametri secondo l'ABI appropriata (non per forza lo farà mediante lo stack, ma potrebbe farlo leggendo i registri, perché magari il dispatcher ha dovuto eseguire delle attività complesse, ad esempio orientate alla sicurezza, che possono aver toccato i valori posti sullo stack). Lo snapshot dei registri di CPU può essere modificato contestualmente alla return della system call, ad esempio per lasciare al chiamante il valore di output.



- La porzione colorata in verde è relativa ai registri che vengono salvati nello stack per *opera del firmware*.
- La porzione bianca è relativa ai registri che vengono salvati nello stack da *parte del software* (qui in particolare si hanno il codice numerico della system call e i parametri da passare alla system call stessa).
- La porzione colorata in blu è relativa alle variabili locali *utilizzate dalla system call*.

NB: Nello standard **UNISTD_32**, anche quando vengono invocate system call con 0 parametri, viene comunque salvato sullo stack l'intero snapshot dei registri di CPU. Ciò implica che il **kernel è in grado di visualizzare anche le informazioni che non gli competono**, il che rappresenta un'importante problematica per la sicurezza. Il sottosistema kernel prende i parametri sempre nello stesso modo, non posso decidere arbitrariamente di prenderli una volta dallo stack, e poi da altre parte. Dalla versione 4.17 del kernel, **una entry della system call table non punta più direttamente al codice della system call**, bensì punta ad un **wrapper**, che sarà colui che la chiamerà effettivamente, passando anche i parametri dallo stack.

Il wrapper si trova nel kernel, e svolge anche il ruolo di *allineatore di parametri*, ovvero adegua la taglia, poichè, causa retrocompatibilità, potrei avere un *elf 32 bit* e un *kernel 64 bit*, e lui ha il compito di risolvere queste incompatibilità. Inoltre, maschera dallo stack i valori non utilizzati, risolvendo il problema di sicurezza precedentemente esposto.



Esempi di creazione di system call in UNISTD_32

Esempio di aggiunta di nuove syscall

```
#include <unistd.h>
#define _NR_my_first_sys_call 254
#define _NR_my_second_sys_call 255
```

```
_syscall0(int,my_first_sys_call);
_syscall1(int,my_second_sys_call,int,arg);
```

Esempio di override della fork

In *unistd.h* posso usare le macro, viene esposta la nuova syscall. Prima si prende il valore `__NR_##name` e dopo si chiama INT 0x80.

```
#include <unistd.h>
#define __NR_my_fork 2 //same numerical code as the original
#define _new_syscall0(name)
int name(void)
{
asm("int $0x80" : : "a" (__NR_##name) );
return 0;
}
_new_syscall0(my_fork)
int main(int a, char** b){
my_fork();
pause(); // there will be two processes pausing!!
}
```

Implicazioni dell'uso di int 0x80

Questa istruzione di `trap` porta in primo luogo a effettuare un accesso alla IDT (sulla porta 0x80) per accedere all'unico GATE che permette il passaggio dal livello di protezione 3 al livello di protezione 0 on-demand. Il passaggio per tale GATE comporta poi la necessità di accedere a un *segmento differente* dell'address space, per cui bisogna ricorrere alla GDT per recuperarlo. Non solo: tipicamente serve anche un secondo accesso alla GDT per reperire il *segmento TSS relativo al thread corrente*, perché è qui che si trovano le informazioni utili per recuperare lo stack di livello **kernel**. (infatti nel TSS mantengo l'indirizzo stack sia a livello 0 sia a livello 3). Quando creo un thread, automaticamente alloco stack. In generale, per gli accessi in memoria conseguenti all'istruzione `int 0x80`, possono trascorrere numerosi cicli di clock, e la loro varianza può essere molto elevata nel momento in cui abbiamo a che fare con hardware asimmetrici (vedi l'architettura **NUMA**). Questo è inaccettabile quando abbiamo a che fare con system call che richiedono un'alta precisione con il tempo, come `gettimeofday()`, che ha il compito di leggere il registro di CPU `rdtsc` per restituire il valore corrente del tempo. Infatti, in queste condizioni, tale system call fornisce un risultato inaffidabile.

Fast system call path

A causa della problematica appena esposta, è stata fatta una rivoluzione nei sistemi x86 che prevede la possibilità di passare il controllo al kernel **senza effettuare alcun accesso in memoria** e, quindi, senza accedere mai alla *IDT* e alla *GDT*. Affinché questo sia possibile, è necessario mantenere a livello di processore *le informazioni che ci permettono di transitare al livello kernel* e, in particolare, quali devono essere il *code segment* e l'*instruction pointer* nel momento in cui inizia l'esecuzione in modalità kernel. L'accesso alle informazioni senza ricorrere alla memoria è possibile grazie alla presenza di registri particolari **MSR (Model Specific Register)**. Essi, infatti, mantengono:

- Il segmento **CS** per il codice di livello kernel.
- L'indirizzo dell'entry point del segmento **CS** relativo all'esecuzione in modalità kernel.
- Il segmento **DS** per i dati di livello kernel.
- Lo stack di livello kernel.

Questo meccanismo è chiamato **fast system call path** e, a differenza dell'invocazione a `int 0x80`, non crea più dei side effect sulla memoria, bensì soltanto all'interno dell'architettura di processore. Per fare uso del fast system call path, si ricorre all'istruzione **sysenter** nelle architetture a 32 bit e all'istruzione **syscall** nelle architetture a 64 bit (long).

Sysenter 32 bit

- Imposta il registro **CS** al valore contenuto in **SYSENTER_CS_MSR**.
- Imposta il registro **EIP** al valore contenuto in **SYSENTER_EIP_MSR**. (per i riferimenti ai registri **MSR**).
- Imposta il registro **SS** a “8+ valore contenuto in **SYSENTER_CS_MSR**”.

- Imposta il registro ESP al valore contenuto in `SYSENTER_ESP_MSR`, cambia la stack area.

Syscall 64 bit

- Imposta il registro CS al valore dato da una *bitmask* presa da `IA32_STAR_MSR`.
- Imposta il registro EIP al valore contenuto in `IA32_LSTAR_MSR`.
- Imposta il registro SS al valore dato da una *bitmask* presa da `IA32_STAR_MSR`.
- **Non** imposta il registro ESP.
Di fatto, quando viene invocata l'istruzione syscall, non viene effettuato uno switch automatico dello stack, bensì sarà il kernel a implementare il suo stack switch funzionale alle sue regole.

Sysexit 32 bit

È un'istruzione usata nelle architetture a 32 bit per *tornare all'esecuzione in modalità user dopo l'esecuzione di una system call*. Più precisamente:

- Imposta il registro CS a “16+ valore contenuto in `SYSENTER_CS_MSR`.
- Imposta il registro EIP al valore contenuto in EDX.
- Imposta il registro SS a 24+ valore contenuto in `SYSENTER_CS_MSR`.
- Imposta il registro ESP al valore contenuto in ECX.

Sysret 64 bit

È un'istruzione usata nelle architetture a 64 bit per *tornare all'esecuzione in modalità user dopo l'esecuzione di una system call*. Più precisamente:

- Imposta il registro CS al valore dato da una *bitmask* presa da `IA32_STAR`.
- Imposta il registro EIP al valore contenuto in RCX. Serve per uscire dal kernel e tornare allo user.
- Imposta il registro SS al valore dato da una *bitmask* presa da `IA32_STAR`.
- **Non** imposta il registro ESP. Esso viene utilizzato solo se serve salvare qualche cosa fatta!

Lo slow path esiste ancora oggi, usato da trap e interrupt. Ovviamente tra `unistd32` e `unistd64` si usano GATE e tabelle delle syscall diverse.

Aggiornamento dei registri MSR

I registri MSR hanno delle macro associate a dei codici numerici in modo tale che queste macro possano essere sfruttate per effettuare delle operazioni di scrittura o di lettura sui registri stessi. Per le scritture si utilizza l'istruzione `wrmsr`, mentre per le letture si utilizza l'istruzione `rdmsr`.

```
/arch/x86/include/asm/msr-index.h (kernel 5)
#define MSR_IA32_SYSENTER_CS 0x174
#define MSR_IA32_SYSENTER_ESP 0x175
#define MSR_IA32_SYSENTER_EIP 0x176

/arch/x86/kernel/cpu/common.c (kernel 5)
void enable_sep_cpu(void) {
    wrmsr(MSR_IA32_SYSENTER_CS, tss->x86_tss.ss1, 0);
    wrmsr(MSR_IA32_SYSENTER_ESP, (unsigned long)
        (cpu_entry_stack(cpu) + 1), 0);
    wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long)entry_SYSENTER_32, 0);
}
```

Costrutto syscall()

È un'API universale di `stdlib`, che permette al software di essere agnostico rispetto alla modalità in cui viene chiamata una system call (che può essere l'istruzione `int 0x80` o l'istruzione `sysenter/syscall1`). Infatti, al suo interno può invocare una o l'altra istruzione in base a qual è disponibile nell'architettura corrente. Tale API accetta un numero di

parametri variabile, che sono il *codice numerico* del servizio che si vuole invocare e gli eventuali *parametri da passare* tale servizio. Ciò che ne consegue è una trasformazione in blocchi assembly che chiamano **syscall 64bit fast** oppure **int 0x80 32 bit slow**.

Esempio echo

funzione echo: `time ./a.out > /dev/null`, compilo con `gcc -nostartfiles`, e poi testo con flag `-m32` (che risulterà essere poco più lento).

Esempio stub system call

Di seguito è mostrato un esempio di utilizzo di `syscall()`, in cui vengono definiti due nuovi stub di system call:

```
#include <stdlib.h>
#define __NR_my_first_sys_call 333
#define __NR_my_second_sys_call 334

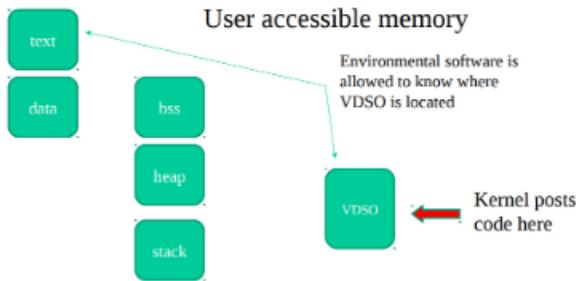
int my_first_sys_call(){
    return syscall(__NR_my_first_sys_call);
}

int my_second_sys_call(int arg1){
    return syscall(__NR_my_second_sys_call, arg1);
}

int main(){
    int x;
    my_first_sys_call();
    my_second_sys_call(x);
}
```

Virtual Dynamic Shared Object VDSO

È una piccola libreria **condivisa** (che occupa una o poche pagine di memoria) contenente l'effettiva implementazione dei meccanismi dati da *sysenter* e *sysexit*. Il kernel mappa tale libreria automaticamente a **run-time** nella zona **user** dell'address space di tutte le applicazioni. Lo scopo ultimo del VDSO è offrire un'interazione più efficiente e migliore possibile tra le applicazioni e il kernel, in particolar modo quando devono essere invocati i servizi del kernel. Quando si scrivono le applicazioni, non è necessario preoccuparsi esplicitamente di questi dettagli dato che tipicamente il VDSO viene chiamato dalla libreria C. Questo meccanismo ad oggi è molto utilizzato per implementare le invocazioni alle system call poiché, rispetto all'uso dell'istruzione `int 0x80`, riduce di molto il numero di accessi in memoria necessari e abbassa fino al 75% il numero di cicli di clock richiesti per avviare la system call. Per giunta, il VDSO è randomizzato all'interno dell'address space in modo tale da avere un maggiore grado di sicurezza. Ne abbiamo uno per ogni **processo** (ad esempio chiamata `exec()`).



Il VDSO può anche essere utilizzato direttamente dal programmatore mediante la seguente API:

```
void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR)
```

Tale API ha lo scopo di prelevare dei valori ausiliari sull'operatività del VDSO, come l'indirizzo di memoria in cui quest'oggetto è posizionato.

Ulteriori dettagli sulla system call table Come abbiamo accennato in precedenza, il modo più opportuno per aggiungere nuovi servizi nel kernel consiste nel riutilizzare le entry della system call table attualmente libere. Non possiamo infatti incrementare l'address space, poiché potrei avere sovrapposizione tra strutture allineate in memoria,

ciò è dato dal fatto che per qualche struttura abbiamo una mappatura 1 – 1 tra indirizzo logico ed indirizzo fisico. Ciò richiederebbe la riconfigurazione del kernel, che è delicata. In particolare, nelle versioni del kernel più datate, la system call table era “oversized”, per cui presentava una vera e propria zona composta da tante entry del tutto inutilizzate. Nelle versioni più moderne, invece, non si ha più questo fenomeno (anche per motivi di sicurezza: meno memoria libera mettiamo a disposizione e minore è la probabilità di poter fare danni) ma esistono comunque delle entry sparse della system call table che sono fatte in un modo tale da puntare a un particolare modulo kernel che, quando viene invocato, l'unica cosa che fa è restituire il controllo all'applicazione utente. Tale modulo è detto `sys_ni_syscall`.

Queste entry a syscall non sviluppate puntano a “nessun servizio”, non a “null”, altrimenti avrei segmentation fault.

Definizione della system call table

- Per la versione 2.4 del kernel e le macchine i386 la system call table è definita nel seguente file: `arch/i386/kernel/entry.S`.
- Per le versioni 2.6 del kernel, la system call table è definita in: `arch/x86/kernel/syscall_table32.S`.
- Per le versioni 4.15 del kernel e UNISTD_64, la tabella è definita in: `arch/x86/entry/syscall_64.c`.

Attualmente, la table è prodotta a **compile time**, ed è un array di `size syscall_max+1`, quindi si ha una entry in più non usata. Tale entry serve per la normalizzazione, implementata per salti speculativi, ovvero salti ad indirizzi scorretti vengono mappati su questa entry che non fa nulla.

Possiamo notare che inizialmente questi file avevano un formato `.S`, erano cioè scritti con delle direttive ASM pre-processore. Nelle versioni più recenti del kernel, invece, ha iniziato a essere possibile definire la system call table direttamente in C (mediante un **array**).

Inoltre, questi file contengono anche i GATE che vengono utilizzati per accedere alla modalità kernel. (Gate se slow, o l'equivalente dei gate per le fast syscall).

Costrutto asmlinkage

Supponiamo di aver inserito all'interno della system call table dei puntatori a delle nuove system call in prossimità delle entry libere. Come possiamo fare per far sì che tali system call siano compliant (conformi), ad esempio, al passaggio dei parametri tramite lo stack di livello kernel? Basta utilizzare il costrutto **asmlinkage**. Per esempio, per il modulo `sys_ni_syscall` abbiamo:

```
asmlinkage long sys_ni_syscall (void) {
    return -ENOSYS;
}
```

Ciò dipende dallo standard **ABI** adoperato, che definisce quali sono le istruzioni in linguaggio macchina da usare per fare le chiamate (system call) al kernel, il modo in cui devono essere passati i parametri per tali chiamate e come ottenere i valori di ritorno. Con **asmlinkage** il dispatcher effettua l'allineamento, ovvero genera codice che sa dove trovare i suoi parametri. (in realtà bisognerebbe citare anche la presenza del wrapper, qui per semplicità ancora non viene introdotto).

Dispatcher con INT0x80, kernel 2.4, UNISTD32

```
ENTRY(system_call)
    pushl %eax          # save orig_eax
    SAVE_ALL
    GET_CURRENT(%ebx)
    testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
    jne tracesys
    cmpl $(NR_syscalls),%eax
    jae badsys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl %eax,EAX(%esp)   # save the return value
ENTRY(ret_from_sys_call)
    cli                 # need_resched and signals atomic test
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
    restore_all:
    RESTORE_ALL
```

Manipulating
the CPU
snapshot in
the stack

Beware this!!!

- L'istruzione `cmp1 $(NR_syscalls)`, %eax si occupa di confrontare il valore del registro `eax` (che contiene il codice numerico della system call che si vuole invocare) con `NR_syscalls`, che è il numero totale di system call che il dispatcher può gestire (comprese le varie `sys_ni_syscall`); in altre parole, controlla se eax cade all'interno dell'indirizzamento delle system call supportate dal kernel. Se la risposta è no, allora viene eseguita una jump verso una zona del codice etichettata con `badsys`.
- Nel caso in cui il registro `eax` contenga un valore lecito, viene invocata la system call che, all'interno della system call table, si trova a spiazzamento `eax*4`, proprio perché ogni entry è di 4 byte.
- restiamo molta attenzione alla porzione di codice evidenziata in giallo: qui si ha la possibilità di passare come parametro al dispatcher un indice numerico arbitrariamente grande (che eccede le dimensioni della system call table) e, attraverso un branch mis-prediction, anche se in maniera speculativa, di andare a invocare una call a un indirizzo di memoria che va oltre la system call table. Questo può lasciare degli effetti micro-architetturali importanti all'interno della macchina. Per ovviare a tale inconveniente, nelle versioni successive del kernel si è ricorsi alla tecnica della **sanitizzazione**.

Dispatcher con SYSCALL, kernel 2.4, UNISTD64

```

ENTRY(system_call)
    swapgs
    movq %rsp,PDAREF(pda_olderp)
    movq PDAREF(pda_kernelstack),%rsp
    sti
    SAVE_ARGS 8,1
    movq %rax,ORIG_RAX-ARGOFFSET(%rsp)
    movq %rcx,RIP-ARGOFFSET(%rsp)
    GET_CURRENT(%rcx)
    test SPT_TRACESYS,tsk_ptrace(%rcx)
    jne tracesys
    cmpq $__NR_syscall_max,%rax
    ja badsys
    movq %r10,%rcx
    call *sys_call_table(%rax,8) # XXX: rip relative
    movq %rax,RAX-ARGOFFSET(%rsp)
    .globl ret_from_sys_call
    .....
ret_from_sys_call:
sysret_with_reschedule:
    GET_CURRENT(%rcx)
    cli
    cmpq $0,tsk_need_resched(%rcx)
    jne sysret_reschedule
    cmpl $0,tsk_sigpending(%rcx)
    .....

```

#define PDAREF(field) %gs:field

Part of the stack switch work originally done via firmware is moved to software

Beware this!!!

Ricordiamo che qui non ci sono gate, c'è altro, ma si usa sempre il dispatcher.

- La seconda e la terza istruzione (le `mov`) si occupano di eseguire lo switch dello stack dato che, come sappiamo, contestualmente al fast system call path, non viene effettuato alcun cambio automatico dello stack. (Qui ci serve, ma visto che la fast system call non lo fa in automatico, lo "forziamo" noi).
- L'istruzione `swapgs` porta ad aggiornare il registro di segmento GS. Swapgs switcha due registri MSR, uno contenente la GS area corrente e l'altro la GS area alternativa. Si lavora sempre col *current*. All'inizio parto da user, poi con swapgs passo a kernel. Questo è importante perché, quando si entra in *modalità kernel*, si inizia ad avere a che fare anche con la *per-CPU memory*, il che richiede di avere **GS** *settato* in modo tale da portarci in un'area di memoria **contenente le informazioni relative alla CPU su cui il thread corrente sta girando**, oltre a dove è posta **la stack area a livello sistema**.
- La zona di codice evidenziata in giallo è del tutto analoga a quella del caso precedente e, in particolare, senza sanitizzazione è vulnerabile agli attacchi basati sul mis-prediction per i salti condizionali.

Implementazione del dispatcher in kernel 4

```

ENTRY(entry_SYSCALL_64)
UNWIND_HINT_EMPTY
/*
 * Interrupts are off on entry.
 * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
 * it is too small to ever cause noticeable irq latency.
 */
swaps
/*
 * This path is only taken when PAGE_TABLE_ISOLATION is disabled so it
 * is not required to switch CR3.
 */
movq    %trap, PER_CPU_VMR(rsp_scratch)
movq    PER_CPU_VMR(cpu_current_top_of_stack), %trap
/* Construct struct pt_regs on stack */
pushq    $USER_DS           /* * pt_regs->ss */
pushq    PER_CPU_VMR(rsp_scratch)   /* * pt_regs->sp */
pushq    %r11              /* * pt_regs->flags */
pushq    %USER_CS          /* * pt_regs->cr2 */
pushq    %rcx              /* * pt_regs->ip */
GLOBAL(entry_SYSCALL_64_after_hwframe)
pushq    %rax              /* * pt_regs->orig_ax */
PUSH_AND_CLEAR_REGS rax=$ENOSYS
TRACE_IRQS_OFF
/* IRQs are off. */
movq    %rax, %rdi
call    do_syscall_64        /* returns with IRQs disabled */
TRACE_IRQS_IRETQ
/*
 * Try to use SYSRET instead of IRET if we're returning to
 * a completely clean 64-bit userspace context. If we're not,
 * go to the slow exit path.
 */

```

Here we pass control to a C-stub, not to the actual system call

- Questo modulo, a differenza dei precedenti, non va a invocare direttamente il front-end che implementa il servizio richiesto, bensì un oggetto intermedio (uno **stub** o **dispatcher secondo livello**), che poi a sua volta passerà il controllo alla system call effettiva. Tale stub (che è mostrato qui di seguito) serve ad innalzare il livello di sicurezza.

```

271 #ifdef CONFIG_X86_64
272 __visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
273 {
274     struct thread_info *ti;
275
276     enter_from_user_mode();
277     local_irq_enable();
278     ti = current_thread_info();
279     if (READ_ONCE(ti->flags) & _TIF_WORK_SYSCALL_ENTRY)
280         nr = syscall_trace_enter(regs);
281
282     /*
283      * NB: Native and x32 syscalls are dispatched from the same
284      * table. The only functional difference is the x32 bit in
285      * regs->orig_ax, which changes the behavior of some syscalls.
286      */
287     nr &= SYSCALL_MASK;
288     if (likely(nr < NR_syscalls)) {
289         nr = array_index_nospec(nr, NR_syscalls);
290         regs->ax = sys_call_table[nr](regs);
291     }
292     syscall_return_slowpath(regs);
293 }
294#endif

```

Wrong-speculation
cannot rely on arbitrary
sys-call indexes!!!!

Also, from kernel 4.17
the system call table entry
no longer points to the
actual system call code,
rather to another wrapper
that masks from the stack
non-useful values

- Al codice che identifica la system call viene applicata una maschera di bit che fa sì che il valore risultante non superi il numero di system call supportate dal kernel. Tale mascheramento viene effettuato prima del controllo su se il codice è compatibile con gli indici della system call table. Se la risposta fosse no, si accederebbe all'unica entry al di fuori della tabella che porterà a svolgere lavoro dummy. Ed ecco qui come abbiamo introdotto la sanitizzazione.
- Consideriamo lo statement `regs->ax = sys_call_table[nr](regs)`. Regs è il puntatore allo snapshot di CPU che viene salvato sullo stack. Inoltre, `sys_call_table[nr]` è proprio il puntatore alla funzione che si vuole invocare e che accetta come parametro proprio lo snapshot di CPU regs. Di conseguenza, l'istruzione mira a modificare il campo `ax` di `regs` inserendovi il valore di ritorno della system call. Non solo: stiamo anche facendo vedere in maniera semplicissima l'intero snapshot di CPU (con tutte le informazioni di livello user) alla system call stessa. Ovviamente questo non ha senso quando la system call invocata non accetta parametri in input. Per risolvere il problema, a partire dalla versione 4.17 del kernel, dentro la system call table non sono più registrati gli indirizzi di memoria dei front-end dei servizi (i.e. gli indirizzi di memoria delle system call vere e proprie): piuttosto si hanno i puntatori a delle funzioni intermedie che, prima di invocare a loro volta le system call, offuscano lo stack di livello kernel (**wrapper**). Quindi non devo fare il disaccoppiamento user-kernel.

Una tecnica per farlo è aggiungere del padding sullo stack in modo tale che la distanza tra lo stack pointer e lo snapshot di CPU con le informazioni relative alla precedente esecuzione in modalità user sia indeterminata. Per creare in modo automatico questo doppio livello di funzioni, si ricorre alle macro SYSCALL_DEFINE0, SYSCALL_DEFINE1,..., SYSCALL_DEFINE6 (una per ciascun numero di parametri che possono essere passati alle system call). Quindi mediante queste macro si genera sia il wrapper sia la vera syscall (con il nostro codice). Decido io che nome dargli, e servono per essere identificate.

Esempio

```
SYSCALL_DEFINE2 (name, param1type, param1name, param2type, param2name) {
    //actual body implementing the kernel side system call
}
```

La macro crea una funzione dal nome `sys_name` oppure `__x86_sys_name` (questo è il nome che affido io al wrapper) la quale passa all'effettiva system call soltanto i valori strettamente necessari (i.e. `param1name` e `param2name`), offuscando tutte le altre informazioni sullo stack.

La system call prende il nome di `__se_sys_name` (dove “se” sta per secure), oppure `__do_sys_name`, dove “*name*” è lo stesso della macro vista sopra. È proprio quest'ultima a implementare il body di `SYSCALL_DEFINE2`.

Implementazione del dispatcher nelle versioni più recenti del kernel

```
ENTRY(entry_SYSCALL_64)
UNWIND_HINT_EMPTY
/*
 * Interrupts are off on entry.
 * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
 * it is too small to ever cause noticeable irq latency.
 */

swaps
/* tss.sp2 is scratch space. */
movq    PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

/* Construct struct pt_regs on stack */
pushq  $__USER_DS           /* pt_regs->ss */
pushq  PER_CPU_VAR(cpu_tss_rw + TSS_sp2)   /* pt_regs->sp */
pushq  %r11                 /* pt_regs->flags */
pushq  $__USER_CS           /* pt_regs->cs */
pushq  %rcx                 /* pt_regs->ip */
GLOBAL(entry_SYSCALL_64_after_hwframe)
pushq  %rax                 /* pt_regs->orig_ax */

PUSH_AND_CLEAR_REGS rax=$-ENOSYS

TRACE_IRQS_OFF

/* IRQs are off. */
movq    %rax, %rdi
movq    %rsp, %rsi
call    do_syscall_64        /* returns with IRQs disabled */

TRACE_IRQS_IRETQ           /* we're about to change IF */

/*
 * Try to use SYSRET instead of IRET if we're returning to
 * a completely clean 64-bit userspace context. If we're not,

```

Switch to the kernel view of memory

- Poco dopo l'istruzione `swaps` si ha uno `SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp`, che permette di conseguire la **Page Table Isolation (PTI)** mediante uno switch della page table (dove si passa dalla page table relativa alla modalità user alla page table relativa alla modalità kernel). Quindi col corretto `GS` posso utilizzare la per-cpu-memory. Essa fa alterazione dei registri, ma in maniera *speculativa*. Abbiamo due MSR come già detto: `GS` corrente e alternativo.

Notiamo che, sebbene si tratti dell'implementazione della funzione `syscall()`, qui stiamo pagando dei costi computazionali notevoli, soprattutto in termini di sicurezza: rispetto alla versione 2.4 del kernel, vengono eseguite molte più attività prima dell'invocazione vera e propria della system call, e questo pesa in particolar modo sulle applicazioni system call intensive.

Attacco swapgs

È basato sull'esecuzione speculativa di un pezzo di codice del kernel mediante la branch miss-prediction e sullo sfruttamento del cache side channel per determinare il valore acceduto speculativamente. Vediamo un esempio:

```
if (coming from user space)
    swapgs
mov %gs: <percpu_offset>, %reg
mov (%reg), %reg1
```

Di fatto, qui è possibile dare luogo a una misprediction sulla condizione data dall'if e far sì che il processore, durante l'esecuzione del kernel, preveda che "coming from user space" sia *true* quando invece non si proviene dall'esecuzione in modalità user; di conseguenza, speculativamente, si cambia il segmento GS di riferimento (passando da quello utilizzato in modalità kernel a quello utilizzato in modalità user). I page fault sicuramente capitano a livello user, ma a livello kernel? Capita se a livello kernel devo specificare una pagina user (kernel scrive su una page su cui già opera l'user). Caotico! Le istruzioni successive consistono in un accesso al segmento GS; nel nostro caso, vengono lette in modo speculativo delle informazioni relative all'esecuzione user mode che, appunto, possono essere recuperate mediante un cache side channel. Devo fare l'operazione *swapgs* dentro l'if, sennò avrei problemi a non cambiare!

Contromisure

- Effettuare l'**override di qualunque swapgs** mentre si è già in esecuzione in modalità kernel. Questo, però, richiede un'operazione di patching piuttosto onerosa lato kernel. Cioè sovrascrivo in ogni caso!
- Utilizzo istruzioni serializzanti, così se opero speculativamente e sbaglio, tutto viene squashato.
- Sfruttare la **SMAP (Supervisor Mode Access Prevention)** dall'hardware. Questo evita l'accesso a una qualunque pagina di livello user mentre si è in esecuzione in modalità kernel. Vedremo successivamente com'è possibile adottare tale contromisura. Usato in *memory management*.

Compilazione del kernel

Qui *configuro* cose che posso usare durante lo startup Linux, non stiamo facendo lo startup di queste cose! Stiamo solo creando contenuto file system.

Avviene secondo i seguenti step:

1. **make config**: per ogni sottosistema software, chiede all'utente se il kernel deve includere quel sottosistema software oppure no. È buona norma seguire questo step impostando un apposito file di configurazione; in alternativa, si potrebbe effettuare la configurazione a mano, come:
 - *allyesconfig*, ma può dare problemi di conflitti.
 - *allnorconfig*, che invece non da mai errore in quanto è un *set minimale*. Con questo secondo approccio, nel core del kernel non c'è supporto al *virtual file system*. Non viene incluso alcun sottosistema software, ottenendo verosimilmente un kernel con non offre abbastanza servizi.
 - Scegliere io cosa mettere e non mettere, ma è lungo e non banale.
 - Prendere un file di configurazione dal web
 - Se ricompilo la versione del kernel che già possiedo, posso usare il file di config già esistente, nella directory `ls /boot`. Qui ci sono info su compilazione kernel, risultati (cioè immagini dei kernel) e mappe (danno metadati).
2. **make**: esegue la compilazione vera e propria del core del kernel, basandosi su quanto indicato dal file di configurazione. Alla fine della compilazione viene generata un'immagine I del kernel. Utilizza i moduli.
3. **make modules**: esegue la compilazione dei moduli, che sono oggetti che non fanno parte dell'immagine I generata col comando make. I moduli vengono agganciati a I per dare luogo a una nuova immagine I' del kernel. Chiaramente i moduli possono sfruttare le funzionalità già presenti nel kernel (i.e. nell'immagine I).
4. **make modules_Install (ROOT)**: effettua l'installazione dei moduli all'interno del sistema, andandoli a inserire nella directory `/lib/modules`. Poiché va a scrivere su delle porzioni del file system che non possono essere modificate da chiunque, è uno step che può essere seguito solo nei panni dell'utente root.

5. **make Install (ROOT)**: effettua l'installazione dell'immagine del kernel, della system map e del file di configurazione, andandoli a inserire nella directory /boot. Anche questo step può essere seguito solo nei panni dell'utente root.
6. **mkinitrd -o initrd.img-<vers> <vers>**: crea un RAM disk, che è un file system *montato temporaneamente* dal kernel *durante la fase di boot*, e contiene alcuni moduli compilati. Quando il RAM disk è montato, i relativi moduli vengono agganciati a run-time al kernel; dopodiché il file system viene smontato dal sistema. Tale oggetto deve essere generato! **Initrd** è un file system usabile come tale solo su finestra temporale ben precisa, perchè lo monta, estrae e monta i moduli, e poi smonta tale file system.

Infine, affinché il kernel (coi relativi moduli che sono stati compilati, installati ed eventualmente riportati su un RAM disk) sia avviabile a seguito della compilazione e dell'installazione, è necessario aggiornare il **boot loader** attraverso il comando **update-grub** (approfondiremo i dettagli sul boot loader più avanti).

Esempio: `grep HZ /boot/config-[version]` , mi da info sugli switch da interrupt.

NB: oggi è anche possibile agganciare una directory al Makefile per la compilazione del kernel; chiaramente tale directory deve contenere a sua volta un altro Makefile per eseguire correttamente gli step di compilazione aggiuntivi.

System map

Sono dei makefile o anatomiche (o tabella o mappa), compilate. È una serie di informazioni che indica qual è la mappa del kernel all'interno dello spazio di indirizzamento lineare. Mi dice cosa posso trovare ad un certo indirizzo logico. Più precisamente, ci dice qual è l'indirizzo lineare in cui si trova ciascuna routine e ciascuna struttura dati del kernel definita a tempo di compilazione. Tuttavia, non tiene conto della **randomizzazione**: perciò, per ottenere l'indirizzo lineare effettivo in cui si trova un oggetto kernel, bisogna sommare un offset randomico all'indirizzo indicato dalla system map.

All'interno della system map, ciascun simbolo (i.e. ciascun oggetto) è associato a un tag che indica di che tipo è quel simbolo:

- **T** = funzione globale.
- **t** = funzione locale all'unità di compilazione.
- **D** = dati globali.
- **d** = dati locali all'unità di compilazione.
- **R** = dati read-only globali. (qui neanche il root può scriverci!).
- **r** = dati read-only locali all'unità di compilazione.

La system map viene utilizzata per il debugging e per l'hacking a run-time del kernel. È inoltre riportata (anche se in modo parziale) all'interno dello pseudo-file `/proc/kallsym`, il quale soprattutto in passato è stato sfruttato per il montaggio di nuovi moduli del kernel: di fatto, se un modulo fa uso di una funzione o una struttura dati già presente nel kernel, è necessario averne un riferimento che, appunto, viene fornito direttamente da `/proc/kallsym`. Questo file è leggibile sia con root (con effettivi riferimenti), se lo leggo a livello user dà indirizzi tutti posti a 0.

Startup del kernel

I componenti che entrano in gioco durante lo startup del kernel (i.e. mentre il software del kernel viene caricato in memoria) sono:

- **Firmware**: è un programma codificato sulla ROM (Read-Only Memory). Presente sull'hw.
- **Bootsector**: è un settore predeterminato di un dispositivo che mantiene il codice dello startup del sistema. Ne sono esempi disco o ssd, è la "prima" zona del dispositivo. Tipicamente prelevo il bootloader, che caricherà il kernel del sistema operativo.
- **Bootloader**: è il codice eseguibile effettivo che viene caricato e lanciato prima di passare il controllo al sistema operativo. Viene mantenuto in parte all'interno del bootsector, in parte in altri settori.
Può essere utilizzato anche per parametrizzare il boot effettivo del sistema operativo.

Task dello startup

- 1) Il firmware va in esecuzione, e carica in memoria e lancia il contenuto del *bootsector*.
- 2) Il codice del bootsector viene dunque eseguito e carica a sua volta le altre porzioni del bootloader.
- 3) Il bootloader, in ultima istanza, carica in memoria il kernel del sistema operativo e gli passa il controllo.
- 4) **Il kernel esegue le sue azioni di startup**, che possono comprendere l'attivazione del codice software, delle strutture dati e dei processi. Tra i processi attivati figura sempre il cosiddetto **idle process**, che serve essenzialmente a far sì che all'interno del sistema *esista sempre almeno un processo in esecuzione*.

Il bootloader fa il boot, il kernel esegue lo startup del sistema operativo.

Bios - Basic I/O System

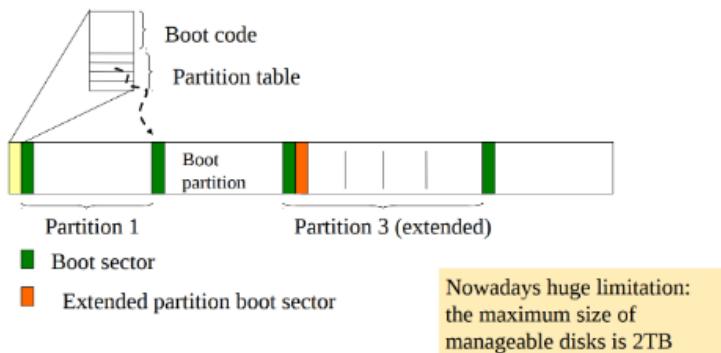
È il firmware tradizionale dei sistemi x86. È possibile colloquiare con lui lanciando particolari interrupt (e.g. tramite i tasti F1, F2,...). L'interazione col BIOS serve ad esempio per parametrizzare l'esecuzione del firmware all'avvio del sistema, dove la parametrizzazione può determinare l'ordine con cui i bootsector vengono cercati nei diversi dispositivi.

Comunque sia, il primo bootsector contiene il **master boot record (MBR)**, che mantiene del codice eseguibile e una tabella (la **partition table**) con *quattro entry*; ciascuna di queste entry identifica una *partizione* del dispositivo, e il *primo settore di ogni partizione può operare come un bootsector*.

È possibile anche che una partizione sia suddivisa a sua volta in quattro sotto-partizioni, ciascuna delle quali può mantenere il proprio bootsector (in tal caso parliamo di **partizione estesa**). L'evoluzione è UEFI. Nella pagina seguente sono riportati il MBR e un esempio di schema del dispositivo con il BIOS:

Offset	Size (bytes)	Description
0	436 (to 446, if you need a little extra)	MBR Bootstrap (flat binary executable code)
0x1b4	10	Optional "unique" disk ID ¹
0x1be	64	MBR Partition Table , with 4 entries (below)
0x1be	16	First partition table entry
0x1ce	16	Second partition table entry
0x1de	16	Third partition table entry
0x1ee	16	Fourth partition table entry
0x1fe	2	(0x55, 0xAA) "Valid bootsector" signature bytes

Grub (if you use it) or others



Boot partition = partizione dove si trova il software del sistema operativo.

Il limite di 2 TB per la quantità di memoria a disposizione per i dispositivi sui quali è possibile effettuare il boot è stato superato dalla **UEFI (Unified Extended Firmware Interface)**, con cui si possono raggiungere anche i 9 zettabyte (= $9 \cdot 10^{21}$ byte).

UEFI - Unified Extended Firmware Interface

È il nuovo standard per il supporto di base dei sistemi (e.g. gestione del boot). È in grado di eseguire gli **eseguibili EFI** (Extended Firmware Interface) piuttosto che caricare e lanciare semplicemente il codice del MBR. È più flessibile,

perchè posso impostare delle condizioni che devono verificarsi o meno. Inoltre, per essere configurato, offre delle interfacce al sistema operativo anziché essere raggiungibile esclusivamente tramite l'invio di interrupt.

In UEFI, il partizionamento del dispositivo è molto più complesso rispetto al BIOS ed è basato sulla **GPT (GUID Partition Table**, dove il GUID è il Globally Unique Identifier). Questa tabella è più complessa rispetto alla partition table introdotta precedentemente (taglia variabile) ed è replicata: quest'ultima caratteristica fa sì che, se una copia della GPT dovesse essere corrotta, continuerebbe ad essere possibile raggiungere e utilizzare le partizioni del dispositivo. Con essa identifico le partizioni, poichè fornisce degli *id*, il numero di caratteri è ampia, e permette un numero di collisioni molto limitato. Con GPT possiamo accedere ad una *partizione EFI SYSTEM PARTITION (STARTUP)* ed identifica la zona in cui ci sono gli eseguibili (*/boot/efi*).

Il type che evidenzia questa zona è “*vfat*”.

Task BIOS/UEFI durante il kernel del SO

1. Il bootloader / EFI-loader carica in memoria l'immagine iniziale del kernel del sistema operativo. L'immagine include un **machine setup code** (= codice che effettua il setup dell'architettura), che deve essere eseguito prima che il codice del kernel vero e proprio prenda il controllo.
2. Il machine setup code passa il controllo all'immagine iniziale del kernel, la quale inizia la propria esecuzione a partire dalla funzione `start_kernel()` che si trova in `init/main.c`. Richiama lo startup di altre cose. È altamente configurabile, posso dirgli di gestire un tot di memoria rispetto a quella totale che ho, ad esempio. All'inizio **non c'è** per-cpu-memory, però con `smp_processor_id()`, che richiama `cpuid`, posso riconoscere il primo processore che esegue lo startup. Non posso fare tutto a compile time, e alcune cose vengono generate ed inizializzate dal processore.

NB: Tale immagine del kernel è differente sia nelle dimensioni che nella struttura da quella che prenderà il controllo in steady state (i.e. dopo che il sistema è stato avviato del tutto).

Quindi prima *boot* del bootloader, poi *startup* del kernel, e poi kernel diventa *steady state*. Per il kernel, possiamo anche parlare di *boot* del kernel, sinonimo di *startup*.

Startup del kernel in macchine multicore

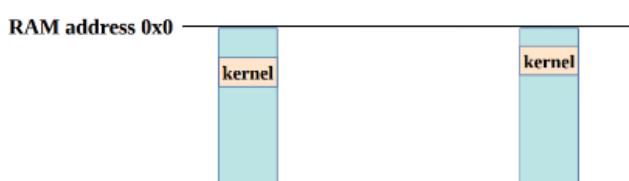
Nei sistemi multi-core o multi-hyperthread, per effettuare il boot di un sistema Linux si possono adottare diverse soluzioni. La più importante consiste nel far eseguire lo startup soltanto a una CPU (detta **master**), mentre le altre (dette **slave**) attendono mediante un busy waiting che venga caricata in memoria l'immagine del kernel in steady state. In questa soluzione, la prima cosa che fa ciascuna CPU è invocare l'istruzione `cpuid`: se risulta essere la `CPU0`, allora è il master e procede con lo startup del kernel; altrimenti, è uno slave e si limita a fare busy waiting. Il codice che decodifica queste (e altre) attività si trova all'interno del file `head.S` (o una sua variante), il quale viene eseguito da tutti i processori.

Kernel Level Memory Management

Allo startup, eseguiamo attività inizialmente esterne allo stack kernel, poi faccio setup stato processore e memoria. Il software usa indirizzi logici, e devo settare il supporto agli indirizzi fisici, senza considerare la randomizzazione. Tutto ciò che succede ci porta allo *steady-state*. Inoltre, molte cose servono solo allo startup, quindi potrei liberare memoria fisica, soprattutto in termini di sicurezza lascio dei **gadget** usabili per attacco.

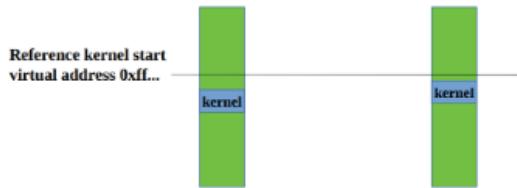
Caricamento del kernel in memoria fisica

La regione della memoria fisica in cui viene caricata l'effettiva immagine del kernel è determinata dal *bootloader* sfruttando la randomizzazione. Di conseguenza, nella zona iniziale della memoria RAM può esserci un buco non utilizzato:



Ricordiamo che il software del kernel, per effettuare gli accessi in memoria, utilizza tipicamente gli indirizzi virtuali. Perciò, in fase di startup è necessario *organizzare una page table corretta* per essere in grado di raggiungere la zona della memoria fisica desiderata. La tabella è insieme di entry da cui partono da indirizzo logico diretto verso indirizzi fisici, ma a quali indirizzi logici/entry li associamo?

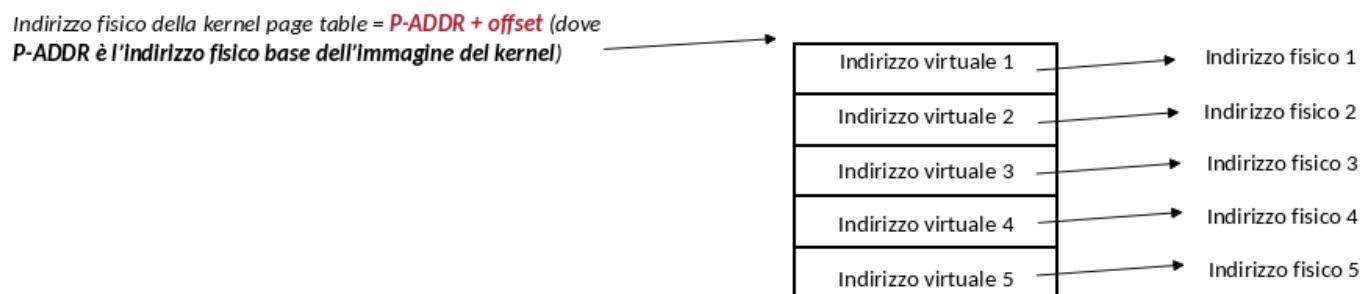
Tutto quello che abbiamo detto per la memoria fisica (RAM), vale anche per la memoria logica (address space): la zona dell'address space delle applicazioni in cui viene posta l'immagine del kernel è, di nuovo, determinata dal bootloader sfruttando la randomizzazione:



Dunque, nel momento in cui si definisce la page table, è necessario porre attenzione anche nella definizione degli indirizzi logici (da associare poi a quelli fisici). Possiamo avere PT diverse, e quindi collocazioni diverse.

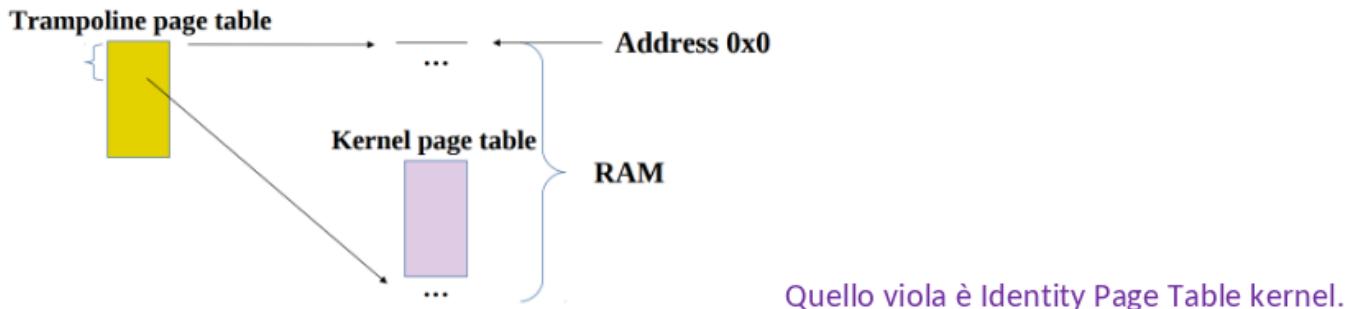
Come possiamo definire una page table (raggiungibile dal kernel) che associa correttamente tutti gli indirizzi logici dell'immagine del kernel ai corrispettivi indirizzi fisici in memoria tenendo conto che si ha la randomizzazione del posizionamento del kernel sia a livello logico che a livello fisico?

È chiaramente necessario conoscere la posizione (l'offset) della *page table del kernel all'interno dell'immagine del kernel*; questo impone che la compilazione del kernel abbia dei limiti, ad esempio per quanto riguarda la possibilità di espandere le strutture dati.



La kernel page table (raffigurata qui sopra) è nota come **identity page table**. Durante la fase di startup, prima che la identity page table sia finalizzata, il bootloader sfrutta un'altra page table *preliminare* che serve per ritrovare in memoria l'immagine del “booting kernel” (che ricordiamo essere diversa dall'immagine del kernel in steady state), inclusa l'identity page table;

questa page table preliminare è detta **trampoline page table**.



Caso base - indirizzi non randomizzati

- L'indirizzo fisico dell'identity page table è noto a compile time. So dove viene montata, a partire da `addr 0`.
- Anche l'indirizzo logico dell'identity page table è noto a compile time (così come qualunque altra porzione del kernel). E' un offset a partire dalla sua base.
- Il codice di startup per la traduzione degli indirizzi virtuali in indirizzi fisici può semplicemente impostare l'identity page table in modo tale che, a ogni indirizzo logico, sia associato il relativo indirizzo fisico già noto a compile time.

- Già a questo punto la paginazione può essere avviata sul processore.

Avvio della paginazione durante lo startup

Tornando allo startup di un sistema Linux multi-core / multi-hyperthread, prima dell'invocazione dell'istruzione `cpuid`, quello che si fa è appunto aprire la paginazione. A tal proposito riprendiamo il codice di `head.S` relativo alle architetture a 32 bit, che è stato menzionato precedentemente.

```
/* Enable paging */ 3:
movl $swapper_pg_dir - __PAGE_OFFSET, %eax /* eax = indirizzo della memoria logica in cui è locata
la page table - offset(dove l'offset indica la differenza
tra l'indirizzo della memoria logica e l'indirizzo della memoria fisica dove si trova il kernel)
In tal modo, scrivo su eax = indirizzo della memoria fisica in cui è locata la page table;
ci serve l'indirizzo fisico e non logico perché il registro cr3 può puntare solo a indirizzi fisici.
L'offset è di 3 Gb.*/
movl %eax, %cr3 /* set the page table pointer, scrivo eax su cr3 */
movl %cr0, %eax
orl $0x80000000, %eax /* or logico bit a bit, il numero a sinistra è un 1 (a sx) e 31 zeri (a dx) */
movl %eax, %cr0 /* set paging bit (= indicazione del fatto che si vuole paginare) */
```

Per quanto riguarda le architetture a 64 bit che supportano la versione 5 del kernel Linux, si ha il file `head_64.S`, la cui logica è del tutto analoga:

```
addq $(early_top_pgt - __START_KERNEL_map), %rax /* __START_KERNEL_map == __PAGE_OFFSET */
... /* here in the middle we account for other stuff like randomization.
* Qui ci riferiamo al trampolino per lo schema di randomizzazione,
early_top_pgt NON è la page table a livello kernel.
L'early è a compile time.*/
movq %rax, %cr3
... /* in the end, paging will be activated */
```

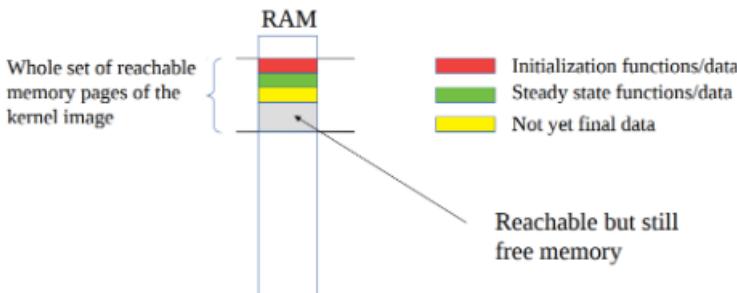
Patch Meltdown hardware

Abbiamo visto la Page Table Isolation come soluzione *software*, in cui facciamo operazioni non proprio banali, come resettare i TLB. Queste cose *rompono* la randomizzazione, perchè la mappatura in memoria fisica del kernel inizia in un certo punto, quello prima non è mappato. Se Page Table non ci dà info, ci mette in stallo, il che comporta non aver nulla in cache.

Se non vado in stallo, non leggo le info per via della patch, ma ho annullato la randomizzazione perchè ho tempi di risposta diversi. Posso migliorare la patch, mappando comunque le pagine non ancora mappate sullo stesso indirizzo di memoria fisica, così non ho più il **delay di tempo**.

Ram durante lo startup

Come sappiamo, durante lo startup del sistema abbiamo in memoria l'**immagine iniziale del kernel****. Più precisamente, l'immagine iniziale è organizzata in RAM secondo lo schema mostrato nella pagina seguente. (sia nel caso randomizzato che non).



La roba relativa all'inizializzazione, che corrisponde alla zona in rosso della figura, diventa completamente inutile una volta che il kernel ha raggiunto un comportamento steady state. Tale zona della RAM, dunque, dopo lo startup viene eliminata, e questo in generale lo si fa ogni volta che si hanno delle informazioni divenute inutili in modo tale che:

- Venga ottimizzata la gestione delle risorse (i.e. si ha più RAM libera a disposizione).

- Si abbia un maggior adattamento con l'aumentare della complessità dello startup.
- Si abbia una maggiore sicurezza (e.g. vengono eliminati dei potenziali gadget).

La page table è *non yet final data*.

La parte *reachable* consente *lettura* e *scrittura*, ma non può andare oltre le zone colorate.

Funzioni `__init`

Sono funzioni che devono essere in memoria soltanto durante la fase di boot del kernel.

Esempio `__init *start_kernel* (void)`

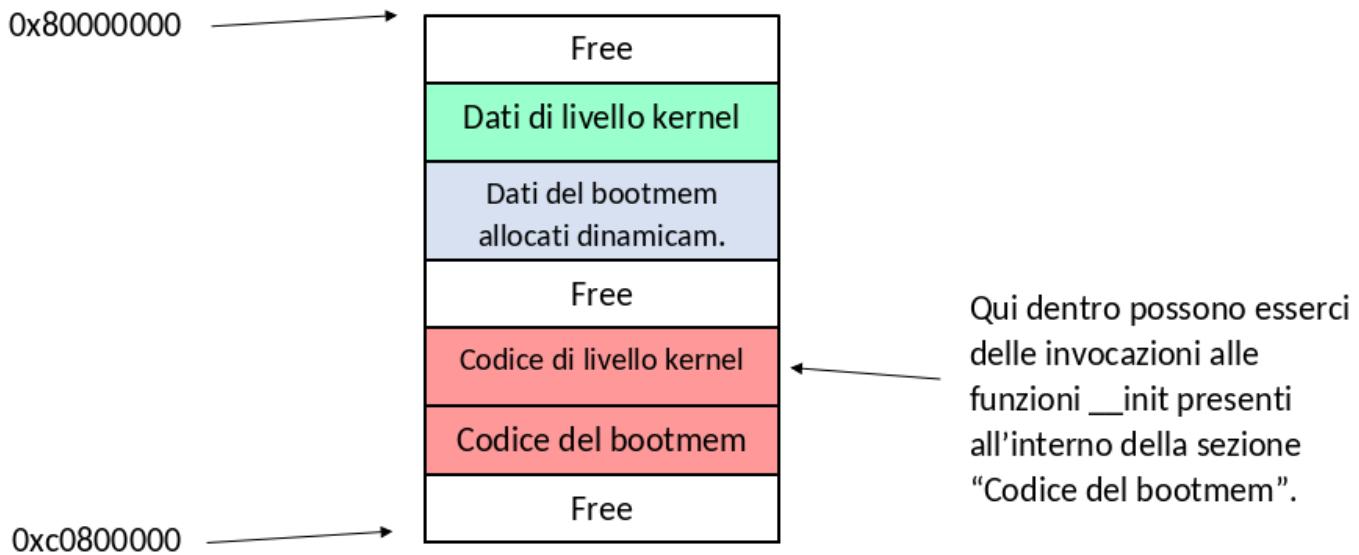
Tale funzione vivrà solo durante il kernel boot/startup.

La fase di linking del kernel si occupa di posizionare queste funzioni su specifiche pagine logiche. Esse sono identificate all'interno del sottosistema del kernel chiamato **bootmem** (memoria di boot, è un set di metadati con dei bit settati per essere identificati), che viene usato per gestire la memoria quando il kernel non si trova ancora a steady state. Al completamento del boot, queste pagine logiche vengono rilasciate come riutilizzabili e sovrascrivibili.

In bootmem non abbiamo esclusivamente le funzioni marcate come `__init`, bensì è anche possibile allocarvi dinamicamente della memoria.

Oggi si usano i **memblock** invece che **bootmem**, qui abbiamo dati che cascano all'interno di specifiche zone NUMA differenti, mentre con bootmem si aveva una visione lineare. Ovviamente cambiano anche le API. Sono tutti allocator di memoria, il concetto è simile alla mmap, ma con meccanismi diversi!

In definitiva, l'immagine iniziale del kernel (quella relativa alla **fase di boot**) appare così:



Reachable Page

Il software del kernel può accedere all'effettivo contenuto di una pagina in RAM semplicemente esprimendo un indirizzo *virtuale* che caschi all'interno della pagina. Le immagini iniziali sono compatte allo startup, aumentano nella fase di boot del kernel. L'unico modo che abbiamo per convertire l'indirizzo virtuale in un indirizzo fisico e, quindi, accedere alla corrispondente locazione fisica in memoria, è disporre di un'apposita page table. Dunque, le pagine che sono rappresentate all'interno della page table sono dette **pagine raggiungibili** (reachable pages).

Organizzazione della RAM nelle macchine moderne

Come già detto, le macchine moderne (multi-processore / parallele) hanno un'architettura NUMA (Non Uniform Memory Access) della RAM, in cui la latenza per l'accesso delle informazioni in memoria non è uniforme per tutti i CPU-core: si hanno alcuni banchi di RAM limitrofi a un CPU-core, altri banchi di RAM vicini a un altro CPU-core, e così via (dove ciascun processore risiede in uno specifico nodo NUMA). Di fatto, non siamo attualmente in grado di rendere questa latenza uniforme (ovvero di avere un'architettura UMA – Uniform Memory Access) in maniera efficiente.

UMA: passo per stesse componenti, poca concorrenza.

NUMA: CPU arrivano in memoria su componenti diverse, vero parallelismo. Oggi è *main-line*.

Numactl

È un comando (`numactl --hardware`) che permette di scoprire:

- Quanti nodi NUMA sono presenti in una determinata macchina.
- Quali sono i nodi NUMA vicini / lontani da ciascun CPU-core.
- Qual è la distanza effettiva dei nodi NUMA da ciascun CPU-core.
- La taglia di ogni nodo NUMA.

Memblock

È l'evoluzione della bootmem che implementa una logica aggiuntiva che consiste nel tenere traccia dei frame di memoria liberi (“free”) e occupati **per ciascun nodo NUMA**. Nonostante le API per gestire la memoria nella bootmem e nel memblock siano leggermente differenti, l'essenza di tali operazioni è rimasta la stessa. In particolare, per quanto riguarda l'allocazione di nuove pagine (“low pages”), abbiamo:

- La possibilità di ottenere l'indirizzo virtuale delle pagine allocate nella *bootmem* (mediante l'API `**alloc_bootmem_lowpages()`). Kernel ancora in setup! Stiamo facendo ancora startup.
- La possibilità di ottenere sia l'indirizzo *virtuale* (mediante le API `memblock_alloc*`()) sia l'indirizzo *fisico* (mediante le API `memblock_phys_alloc()`) delle pagine allocate nel memblock.

Quando ho concluso lo startup, queste non servono più!

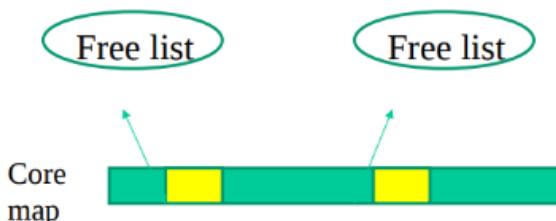
Strutture dati per la gestione della memoria

Si hanno tre principali strutture dati del kernel che supportano la gestione della memoria:

- **Kernel page table**: è la identity page table, ed è quindi una sorta di tabella delle pagine “ancestrale”, da cui derivano tutte le altre tabelle delle pagine. Serve a mantenere un mapping tra indirizzi logici e indirizzi fisici del codice e dei dati di livello kernel.
- **Core map**: è un array che tiene traccia dello stato dei frame di memoria fisica. Possiamo vederla come costituita da più parti, ognuna associata a un singolo nodo NUMA.
- **Free list**: è una struttura dati che ci riporta ai frame liberi di memoria fisica. Anch'essa è costituita da più parti, ognuna associata a un singolo nodo NUMA.

Nessuna di queste strutture dati è già finalizzata nel momento in cui il kernel del sistema operativo viene mandato in startup. In particolare, la core map e la free list non esistono proprio (le informazioni sui frame di memoria fisica sono date dalla bootmem o dal memblock), mentre la *kernel page table* non si trova ancora in uno stato definitivo. Di conseguenza, alla fine dello startup, le strutture dati devono essere istanziate o modificate.

Nella pagina seguente è riportato uno schema molto semplice che mostra la relazione tra la core map e le free list (dove si può vedere che le *free list* sono collegate alla core map dove si hanno frame di memoria fisica liberi). Free list viene chiamata, interroga Core map per farsi tornare zona.



Le free list sono accessibili in concorrenza da tutte le CPU per prendere memoria, dobbiamo sincronizzarle. Esistono meccanismi di Caching, in particolare facendo pre-reserving delle locazioni. La kernel page table prende indirizzo lineare (non segmentato), **la segmentazione c'è prima**.

Indirizzi logici e poi fisici, si alloca nello spazio lineare con indirizzi assoluti.

NB: quando programmiamo a livello kernel, la core map non è direttamente esposta; piuttosto, le API che abbiamo a disposizione (i.e. le API di buddy allocation, che analizzeremo successivamente) utilizzano le free list e, inoltre, sono in grado di gestire correttamente la concorrenza di molteplici thread. Esistono anche *quick list*, perchè sono per-cpu, quindi niente sync.

Kernel page table

Ci dice come vediamo lo spazio! Gli obiettivi del setup della kernel page table sono:

- 1) Permettere al kernel di utilizzare gli indirizzi virtuali per andare a raggiungere le informazioni poste in memoria fisica.
- 2) Permettere al kernel di raggiungere la quantità massima di memoria RAM disponibile (che dipende dalla specifica macchina e dallo specifico chipset).

In effetti, uno dei motivi per cui la forma finale della page table non è data all'interno dell'immagine del kernel in memoria è proprio che la quantità di RAM da pilotare deve essere *parametrizzata*. Ma di fatto come si esplica il secondo obiettivo? Durante la fase di startup, in realtà, **il kernel è contenuto in una porzione molto limitata della RAM**, e si espande solo successivamente. Perciò, col tempo, il kernel deve avere la possibilità di accedere a un insieme di indirizzi fisici sempre più ampio, ma questo implica che ci si deve poter espandere su una maggiore quantità di indirizzi logici. E, per aumentare la quantità di indirizzi logici da sfruttare, è necessario cambiare la struttura della kernel page table. Utilizzare questo meccanismo anziché caricare subito l'intera immagine del kernel in RAM presenta un ulteriore vantaggio in termini di efficienza nella fase di setup del kernel: infatti, per caricare subito l'intera immagine in RAM, è necessario prelevare informazioni da un dispositivo di I/O (e sappiamo bene che la comunicazione coi dispositivi di I/O è molto onerosa), mentre, dall'altra parte, è direttamente il software a scrivere delle informazioni in RAM in un secondo momento.

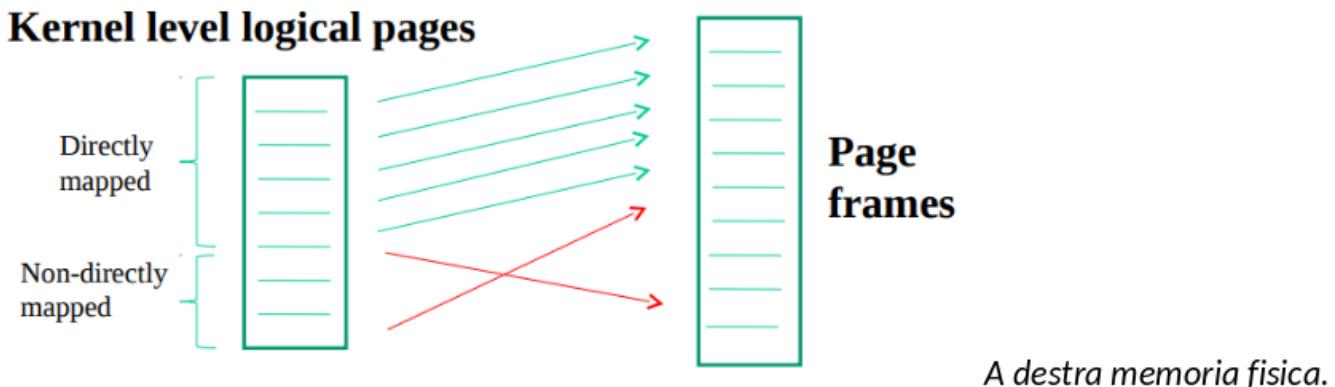
Pagine di memoria directly mapped

Sono pagine di livello kernel il cui mapping sui frame fisici è basato su un semplice shift (costante) tra gli indirizzi virtuali e quelli fisici.

Dati PA = Physical Address, VA = Virtual Address, abbiamo:

$PA = y(VA)$, dove $y()$ è una funzione che sottrae un **valore costante** predeterminato al parametro di input (VA). È possibile fare l'inverso. Funziona anche con randomizzazione, aggiustando la funzione $y()$.

Non tutte le pagine di livello kernel sono **directly mapped**, per cuiabbiamo questa situazione:



Tipicamente, **diverse pagine logiche directly mapped sono mappate su frame fisici differenti**.

Tuttavia, non è escluso che una qualche pagina logica non directly mapped venga mappata su un frame fisico che corrispondeva già a una pagina logica directly mapped. Per quanto riguarda la contiguità: - Le pagine *directly mapped* risultano contigue sia in memoria logica che in memoria fisica. - Le pagine *non directly mapped* possono essere contigue in memoria logica ma non contigue in memoria fisica. Posso navigare Identity Page Table per vedere dove è mappata fisicamente, a patto che non ci siano update concorrenti sulla table.

Zone

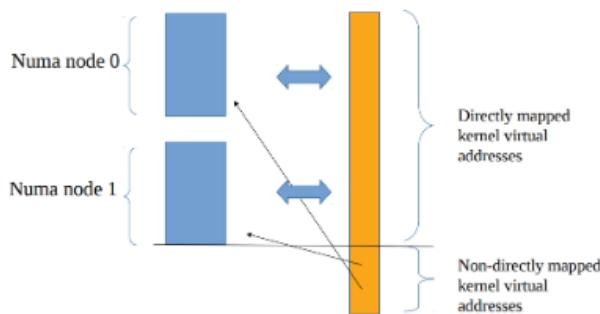
È tipico per il kernel del sistema operativo organizzare la memoria fisica in ZONE, ciascuna delle quali determina il tipo di utilizzo delle relative pagine. Le ZONE più note sono:

- **DMA**: è utilizzata per riservare la memoria a specifiche operazioni di dispositivo; comprende i primi 16 MB di memoria fisica. Legata alla randomizzazione del kernel, da 16 MB in su!
- **NORMAL**: è utilizzata per le pagine del kernel *directly mapped*; comprende la sezione della memoria fisica che va dal megabyte 16 al megabyte 896.
- **HIGH**: è utilizzata per le pagine del kernel *non directly mapped* e le *pages user*; comprende la sezione della memoria fisica successiva al megabyte 896. Elementi lato user non sono MAI directly mapped! Il suo scopo era aumentare la flessibilità. Posso rimapparla in maniera arbitraria per raggiungere zone.

Caso di Linux nei sistemi x86_64 long mode

Abbiamo 2^{48} indirizzi logici. Lo spazio è mappato direttamente, ma la stessa memoria può essere anche presa non directly-mapped, che uso quando ho frammentazione. (quindi ho un doppio mapping, inoltre lo scheAma directly mapped aveva il limite di 1 GB). Prendo pagine logiche contigue (zona arancione) che mappa su memoria fisica, anche appartenente a Numa code diversi. Ottimo anche per la sicurezza, con un mapping sempre $1 \rightarrow 1$ sarebbe più facile! Qui ci mettiamo cose che vogliamo mascherare, come le stack areas dei thread! Qui il kernel prende tutta la RAM per il direct mapping. In realtà, può anche prendere l'intera memoria RAM per le pagine non directly mapped.

Questa funzionalità deriva semplicemente dalla possibilità di indirizzare tantissima memoria logica e fisica nei sistemi x86-64: è possibile, infatti, usare 2^{48} byte nell'address space logico. *Comunque sia, qui le ZONE non sono più rilevanti, ma rimangono nell'architettura.*

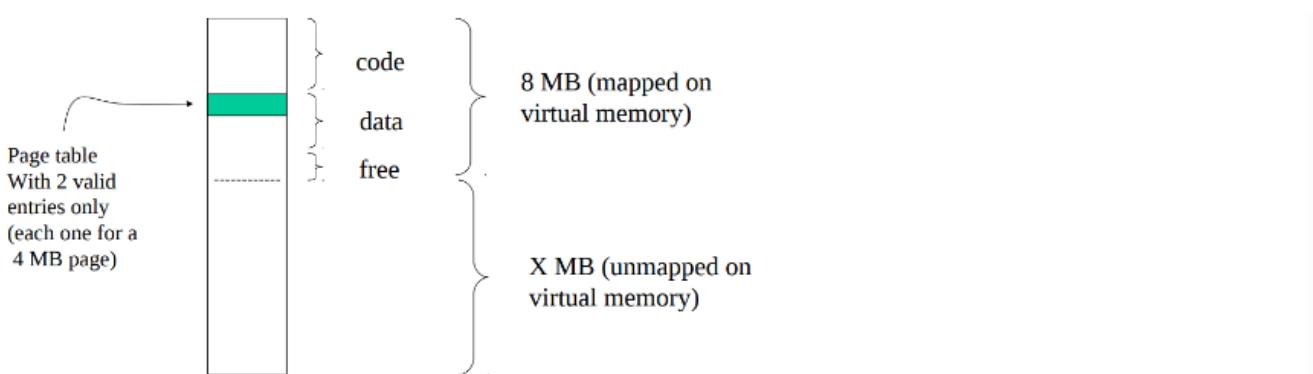


Organizzazione kernel e struttura page table in i386

Organizzazione del kernel in i386

Nei sistemi i386 (*x86 protected mode*) avevamo la versione 2.4 del kernel. Qui, allo startup del kernel, l'indirizzamento è basato solo su due pagine grandi 4 MB ciascuna, per cui in questa fase abbiamo un kernel di soli 8 MB e una page table iniziale con due sole entry utilizzabili. La regola di paginazione utilizzata è identificata da appositi bit all'interno della page table. Inoltre, sappiamo che esiste il registro CR3 che punta direttamente alla tabella delle pagine (indicandone l'indirizzo fisico). Di conseguenza, il kernel deve conoscere il posizionamento esatto in memoria della page table in modo tale da poter inizializzare correttamente CR3 in fase di startup.

Nella pagina seguente è riportato uno schema del kernel durante lo startup nei sistemi i386.



Come è possibile vedere, la page table iniziale è compresa negli 8 MB in cui il kernel può espandersi durante la fase di startup. Anche la bootmem si trova lì, e serve per tenere traccia delle aree di memoria libere ("free") in quegli 8 MB.

Per quanto riguarda l'indirizzamento lineare dei sistemi i386, all'interno di un address space è possibile esprimere al più 4 GB (232) di indirizzi logici.

- I primi 3 GB sono utilizzati per le informazioni di livello user.
- L'ultimo GB è utilizzato per le informazioni di livello kernel.

Nel caso in cui ad esempio il kernel occupi tutta la memoria fisica a disposizione, si può andare incontro a dei conflitti nel momento in cui diviene necessario materializzare in RAM delle pagine di livello utente; in tal caso, la parte user può utilizzare i frame fisici che il kernel le mette a disposizione.

Struttura page table in i386

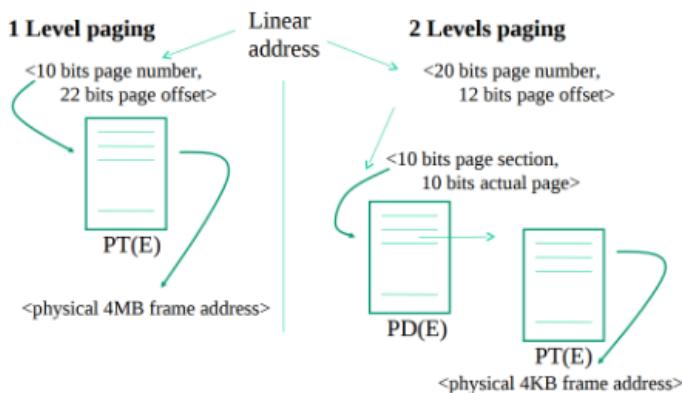
Molto spesso le pagine da 4 MB per l'indirizzamento della memoria risultano essere inadeguate. Per questo motivo, i sistemi i386 mettono a disposizione due modi di effettuare la paginazione:

- **Paginazione a 1 livello:**

Prevede pagine da **4 MB**. Qui gli indirizzi, che sappiamo essere a 32 bit, sono suddivisi in questo modo:
I primi 10 bit identificano la pagina su cui l'indirizzo deve essere mappato (i.e. l'offset della page table che punta a quella pagina);
I restanti 22 bit indicano l'offset all'interno di quella pagina (infatti 2^{22} byte = 4 MB). Il vantaggio di questa organizzazione è dato dalla presenza di un'unica page table; tuttavia, spesso, pagine così grandi non sono così agevoli da utilizzare (*basti pensare che, per allocare una struttura dati di pochi byte, bisogna materializzare ben 4 MB di memoria*): di conseguenza, tale organizzazione è tipicamente usata solo durante la fase di startup.

- **Paginazione a 2 livelli:**

Prevede pagine da **4 KB**. Qui gli indirizzi sono suddivisi in quest'altro modo:
I primi 10 bit (**PDE – page directory entry**) identificano l'offset della page table di primo livello che punta alla page table di secondo livello da esaminare;
I secondi 10 bit (**PTE – page table entry**) identificano l'offset della page table di secondo livello che punta alla pagina su cui l'indirizzo deve essere mappato; infine, i restanti 12 bit indicano l'offset all'interno di quella pagina (infatti 2^{12} byte = 4 KB).
Nulla vieta di avere una applicazione *mista*, in cui, partendo da una page table, per alcune entry andiamo direttamente al frame fisico, per altre puntiamo a tabelle che a loro volta puntano ad un frame fisico.



La page table è necessariamente collocata in memoria in maniera allineata rispetto al frame fisico.

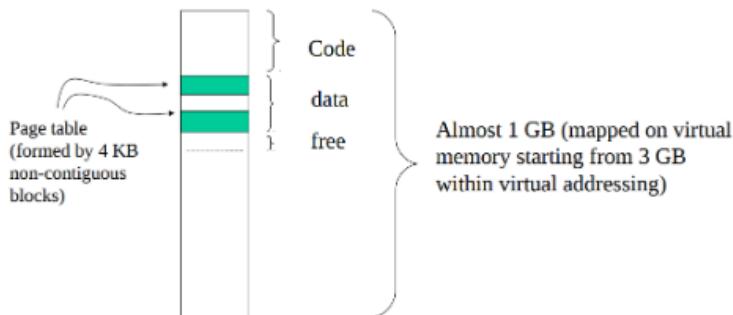
Questo vincolo è dovuto al fatto che il registro **CR3** è in grado di esprimere gli indirizzi di memoria con la granularità dei 4 KB. Comunque sia, quando abbiamo una page table da 4 MB, questa è tipicamente racchiusa in un **unico frame di memoria fisica**, mentre quando abbiamo la paginazione a due livelli, è possibile che ciascuna page table si trovi su più frame fisici non contigui tra loro, purché venga mantenuto l'allineamento rispetto ai frame stessi.

Riassumendo, per passare dalla fase di startup del kernel alla fase steady state, è necessario far fronte alle seguenti problematiche:

- 1) È necessario passare da una granularità delle pagine di 4 MB a una granularità delle pagine di 4 KB.
- 2) È necessario estendere a 1 GB la quantità di memoria logica riservata al kernel.
- 3) È necessario riorganizzare la page table in due livelli separati.
- 4) È necessario identificare le aree di memoria libere tra gli 8 MB già raggiungibili per poter espandere la page table.

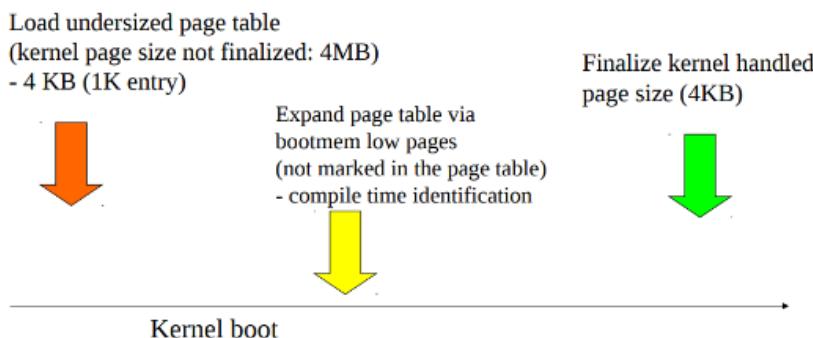
- 5) Non è possibile utilizzare altre facility di memory management al di fuori della paginazione, dato che, come sappiamo, la *core map* e le *free list* esistono soltanto dopo che il sistema è andato in steady state. Dunque, come suggerito in precedenza, per recuperare e sfruttare le aree di memoria libere, si ricorre alla *bootmem*.

Dunque, il layout del kernel 2.4 nei sistemi i386 a steady state è il seguente:



Ora, negli 8 MB iniziali, si hanno anche le page table di secondo livello, esattamente nelle locazioni che prima erano contrassegnate come "free" dalla bootmem. Nel frattempo, è stato anche necessario modificare i bit delle page table che indicano la regola di paginazione utilizzata.

Di seguito è riportato uno schema riassuntivo sull'evoluzione delle page table durante la fase di boot del kernel:



Paginazione nei sistemi i386 in Linux

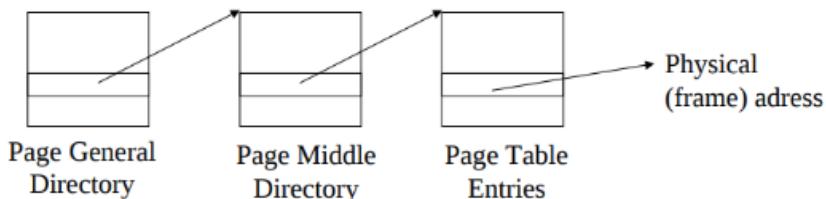
Linux vede in realtà la paginazione come organizzata a *tre livelli*. Ciascun indirizzo è dunque suddiviso nel seguente modo:



Dove:

- **PGD** (page general directory) = bit che identificano l'offset della page table di primo livello che punta alla page table di secondo livello da andare a guardare.
- **PMD** (page middle directory) = bit che identificano l'offset della page table di secondo livello che punta alla page table di terzo livello da andare a guardare.
- **PTE** (page table entry) = bit che indicano l'offset della page table di terzo livello che punta alla pagina di memoria su cui l'indirizzo deve essere mappato.
- **Offset** = bit che indicano lo spiazzamento da applicare all'interno della pagina di memoria.

Il numero di bit di ciascuna di queste quattro porzioni degli indirizzi è variabile.



Tuttavia, sappiamo che i sistemi i386 supportano al più due livelli di paginazione: per motivi di compatibilità, Linux, all'interno dei sistemi i386, **prevede che la sezione PMD degli indirizzi di memoria sia composta da zero bit**. Di conseguenza, si crea il seguente mapping:

- PGD di Linux → PDE di i386
- PTE di Linux → PTE di i386

In Linux è possibile definire il numero di entry da cui è composta ciascuna tabella all'interno del file `include/asm-i386/pgtable-2level.h`:

```
>#define PTRS_PER_PGD    1024
>#define PTRS_PER_PMD    1
>#define PTRS_PER_PTE    1024
```

- Per compatibilità coi sistemi i386, si pone un numero di entry per la page middle directory pari a 1, che corrisponde appunto ad avere 0 bit associati al campo PMD degli indirizzi e, quindi, a non disporre proprio della PMD.
- Poiché *ciascuna page table occupa 4 KB di memoria*, ponendo 1024 entry per ogni tabella, si hanno delle entry grandi **4 byte**.
- Avere 1024 entry per la tabella di primo livello e 1024 entry per le tabelle di ultimo livello implica che possiamo mappare fino a $1024 \cdot 1024 = 2^{20}$ pagine di memoria distinte.

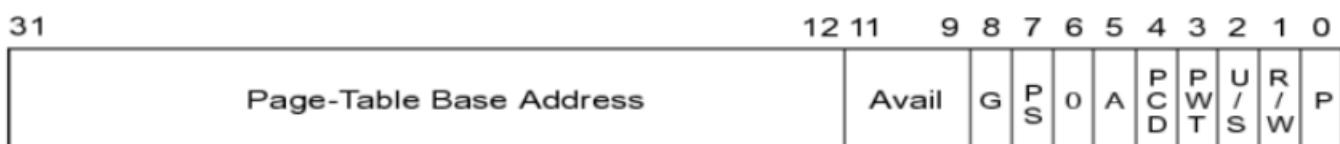
Inoltre, in kernel 2, all'interno del file `arch/i386/kernel/head.S`, viene definito il simbolo `swapper_pg_dir`, che esprime l'indirizzo di memoria virtuale della PGD che viene correntemente utilizzata. Il valore con cui viene inizializzato `swapper_pg_dir` dipende dalle scelte che si fanno a livello di compilazione (e, quindi, da come viene definita l'immagine iniziale del kernel). D'altra parte, in kernel 3 il simbolo definito per questo scopo è `init_level4_pgt`, mentre in kernel 4 e 5 è `init_top_pgt`.

Le page table in sé sono invece definite all'interno del file `include/asm-i386/page.h` come delle struct composte da un solo campo (che esprime il contenuto delle loro entry):

```
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
```

La definizione di tre struct tutte uguali (**i.e. tutte con un unico campo di tipo `unsigned long`*) evita che il programmatore ponga una variabile *x* di tipo `PTE_t` uguale a una variabile *y* di tipo `PGD_t` (operazione ammessa in C se `PTE_t`, `PMD_t` e `PGD_t` fossero semplicemente una ridefinizione del tipo `unsigned long`), il che sarebbe scorretto dato che si tratta di page table significativamente differenti tra loro (ad esempio presentano bit di controllo diversificati).

Entry della PDE nei sistemi i386



- **Page-table base address**: indica l'indirizzo di una tabella di secondo livello oppure l'indirizzo di una pagina da 4 MB.
- **Page-table base address**: indica l'indirizzo di una tabella di secondo livello oppure l'indirizzo di una pagina da 4 MB.

- **G (global page)**: bit ignorato.
- **PS (page size)**: se vale 0, vuol dire che il page-table base address indica l'indirizzo di una tabella di secondo livello (che sappiamo essere di 4 KB); se vale 1, vuol dire che il page-table base address indica l'indirizzo di una pagina di memoria da 4 MB.
- **0 (reserved)**: bit riservato.
- **A (accessed)**: bit che indica se la pagina è stata acceduta (i.e. se il firmware è riuscito a effettuare una traduzione da indirizzo logico a indirizzo fisico); è uno sticky flag, nel senso che viene settato a 1 dal firmware ma può essere resettato a 0 esclusivamente dal software.
- **PCD (cache disabled)**: se vale 0, vuol dire che il caching è abilitato per la pagina (o il gruppo di pagine) corrispondente alla presente entry; se vale 1, vuol dire che il caching è disabilitato.
- **PWT (write-through)**: se vale 0, la cache policy utilizzata per la pagina (o il gruppo di pagine) corrispondente alla presente entry è il write-back; se vale 1, vuol dire che la cache policy utilizzata è il write-through.
- **U/S (user/supervisor)**: se vale 0, la pagina (o il gruppo di pagine) corrispondente alla presente entry gode dei privilegi supervisor; se vale 1, la pagina (o il gruppo di pagine) gode dei privilegi user.
- **R/W (read/write)**: se vale 0, la pagina (o il gruppo di pagine) corrispondente alla presente entry può essere acceduta solo in lettura; se vale 1, la pagina (o il gruppo di pagine) può essere acceduta sia in lettura che in scrittura.
- **P (present)**: bit che indica se la presente entry è valida (i.e. se ci porta effettivamente su una pagina o un gruppo di pagine).

NB: Non c'è nulla all'interno della entry che ci dice se possiamo fare il fetch (lettura) di istruzioni all'interno della pagina (o del gruppo di pagine) corrispondente oppure no. Questo rappresenta un problema di sicurezza nel momento in cui è consentito effettuare il fetch di qualunque cosa all'interno dell'address space.

Entry della PTE nei sistemi i386

31	12 11	9 8	7	6	5	4	3	2	1	0
Page Base Address	Avail	G A T	P A T	D	A	P C D	P W T	U S	R /W	P

- **Page base address**: indica necessariamente l'indirizzo di una pagina.
- **PAT (page table attribute index)**: è una don't care.
- **D (Dirty)**: bit che indica se la pagina corrispondente alla presente entry è stata acceduta in scrittura (e, quindi, è stata modificata); anch'esso è uno sticky flag.

NB: anche qui non c'è nulla all'interno della entry che ci dice se possiamo fare il fetch di istruzioni all'interno della pagina corrispondente oppure no.

NBB: all'interno del file header include/asm-i386/pgtable.h sono definite alcune macro che indicano la posizione dei bit di controllo delle entry della PDE o PTE. Esse possono essere utilizzate per estrarre le relative informazioni di controllo dalle entry della PDE o PTE.

```
>#define _PAGE_PRESENT    0x001
>#define _PAGE_RW        0x002
>#define _PAGE_USER      0x004

>#define _PAGE_ACCESSED  0x020
>#define _PAGE_DIRTY     0x040 /* proper of PTE */
```

Relazione tra page table ed eventi di trap/interrupt

Quando viene eseguita un'istruzione I che vuole accedere a un indirizzo di memoria, in caso di TLB miss, è necessario ricorrere alla page table per effettuare una traduzione tra indirizzo logico e indirizzo fisico.

In tale scenario, il primo controllo che viene fatto è quello sul *presence bit*: se la entry della page table esaminata è valida, allora viene generata una trap che porta alla materializzazione del frame fisico in memoria e alla riesecuzione dell'istruzione I (che, di fatto, è risultata essere offending). A valle di tutto questo, potrebbero essere generate anche ulteriori trap: ad esempio, una seconda trap viene essere sollevata se I è un'istruzione di scrittura e, quando viene controllato il bit R/W, emerge che l'indirizzo di memoria target è accessibile solo in scrittura; in tal caso, avremmo un segmentation fault.

Algoritmo di inizializzazione delle page table in i386 nel kernel 2.4

I seguenti step vengono seguiti ciclicamente:

- 1) Determinare l'indirizzo virtuale da mappare in memoria fisica; chiaramente tale indirizzo (indicato dalla variabile `vaddr`) sarà diverso a ogni iterazione, e il limite massimo da considerare è mantenuto all'interno della variabile `end`.
- 2) Allocare una PTE (una tabella di secondo livello che gestisce 1024 pagine), che verrà collegata alla tabella di primo livello (PDE) che è stata definita a partire dalla page table relativa alla fase di startup (per cui, in effetti, la PDE e la page table relativa alla fase di startup coincidono).
- 3) Popolare le entry della PTE.
- 4) Ora il prossimo indirizzo virtuale da mappare (`vaddr`) sarà 4 MB più avanti rispetto a quello appena mappato.
- 5) Saltare allo step 1 fin tanto che esistono ancora indirizzi virtuali da mappare (ovvero fin tanto che `vaddr < end`).

NB: Ciascuna entry della PDE viene settata per puntare alla PTE corrispondente solo dopo che tale PTE è stata popolata correttamente. Se così non fosse, il mapping della memoria andrebbe perso a ogni TLB miss.

All'interno dell'algoritmo appena descritto, vengono utilizzate le seguenti funzioni:

- `set_pmd()`: è una macro che implementa il settaggio di una entry della PMD (che, nel caso di Linux, corrisponde al settaggio di una entry della PDE).
- `mk_pte_phys()`: è una macro che costruisce un'entry della PTE, che include l'indirizzo fisico del frame target.
- `__pa()`: dato un indirizzo virtuale del kernel, restituisce il corrispondente indirizzo fisico nel caso in cui valga il direct mapping.

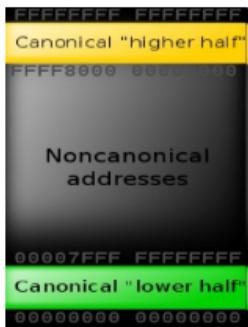
Chiaramente esiste anche l'operazione duale a `__pa()`, che è `__va()`. Se anticipassimo `set_pmd` prima del ciclo presente sopra, sarebbe un errore grave. Questo perché dobbiamo considerare anche il firmware. Se una zona è acceduta, la aggiungo a TLB, e potrei raggiungerla anche senza entry di secondo livello, ma se non ci fosse nel TLB, saltierebbe tutto.

PAE Physical Address Extension

I sistemi x86 a 32 bit considerati finora presentano un limite molto importante: sia per esprimere gli indirizzi lineari, sia per il physical addressing si hanno a disposizione solo 32 bit. Di conseguenza, sia gli address space che la memoria RAM dei calcolatori possono estendersi a un massimo di 2^{32} byte = 4 GB. Col passare del tempo, soprattutto per quanto riguarda la memoria fisica, questo limite ha iniziato a risultare particolarmente stretto. A tal proposito, viene in soccorso la **PAE (Physical Address Extension)**, che consiste nell'aumento da 32 a 36 del numero di bit utilizzati per il physical addressing; perciò, è ora possibile avere una memoria RAM che si estende fino a 2^{36} byte = 64 GB, anche se il limite per gli address space dei processi rimane di 4 GB. Il vantaggio sta dunque nella possibilità di materializzare in memoria fisica pagine relative a un numero maggiore di processi attivi. Questo meccanismo ha però un'implicazione importante: le tabelle delle pagine, per essere in grado di ospitare gli indirizzi fisici estesi, devono avere delle entry di maggiori dimensioni. In particolare, per mantenere la struttura originale delle page table ed evitare così di effettuare modifiche troppo radicali al Makefile del kernel, si è deciso di raddoppiare la dimensione delle entry di tutte le page table e di dimezzarne il numero (da 1024 a 512). Poiché, nella trattazione svolta finora, si hanno due livelli di paginazione, si va incontro a un supporto di $\frac{1}{4}$ dell'address space (un fattore 2 è dato dal dimezzamento del numero di entry della page table di primo livello e un fattore 2 è dato dal dimezzamento del numero di entry della page table di secondo livello). Per compensare questa riduzione, si aggiunge un terzo livello di paginazione e, più precisamente, si inserisce una tabella top level chiamata **page directory pointer table**, che dispone appunto di 4 entry ed è puntata direttamente dal registro CR3. Il bit 5 del registro CR4, invece, indica se la PAE è attivata o meno.

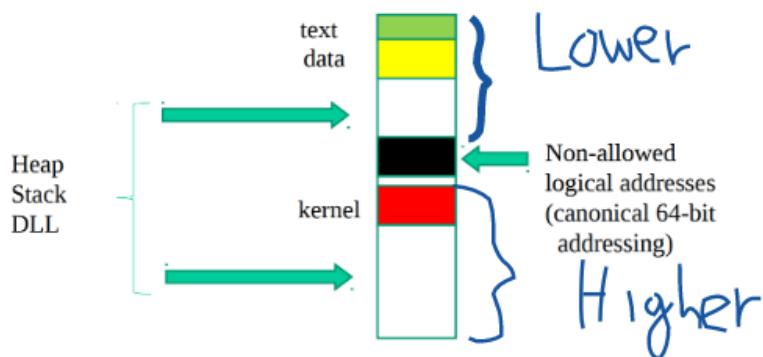
Architetture x86-64

Estendono lo schema PAE mediante il cosiddetto **long addressing mode**, mettendo a disposizione 64 bit per esprimere gli indirizzi di memoria. Perciò, almeno a livello teorico, consentono un indirizzamento di memoria logica di 2^{64} byte. Nelle implementazioni effettive, però, si possono raggiungere al più 248 indirizzi (che comunque consentono di spennare su ben 256 TB). Questi 2^{48} indirizzi sono nella cosiddetta **forma canonica**, per cui sono suddivisi in due metà dette **higher half** e **lower half**: in particolare, tra i 48 bit utilizzabili, il più significativo determina automaticamente il valore degli altri bit ancor più significativi (dal 48 al 63). Di conseguenza, gli indirizzi esprimibili sono quelli con i *17 bit più significativi tutti pari a 0* e quelli con i *17 bit più significativi tutti pari a 1*: tutti gli indirizzi che cadono nel mezzo sono detti non canonici e non sono utilizzabili.



In realtà, non tutti i sistemi operativi permettono di sfruttare l'intero range di 256 TB di memoria logica / fisica esprimibile con 48 bit. Ad esempio, Linux ad oggi mette a disposizione 128 TB per l'indirizzamento logico e 64 TB per l'indirizzamento fisico. Ciò dipende dal setup delle tabelle delle pagine, la parte user di Linux usa la metà, ovvero 2^{47} , lo scopo è ottimizzare la circuiteria del processore, in quanto è stato visto che con tale valore per indirizzare address fisici era la più adatta.

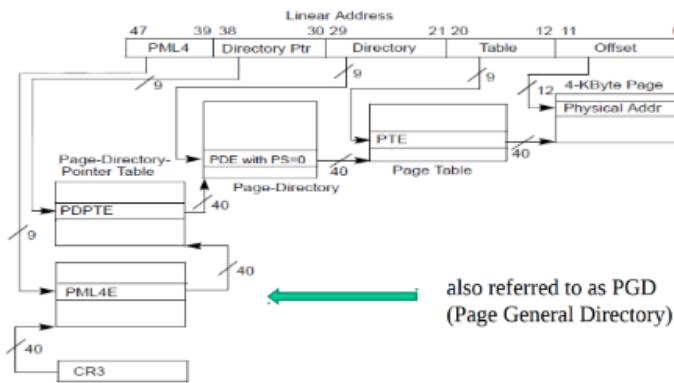
In definitiva, l'address space di un'applicazione in un sistema x86-64 si presenta così:



Normalmente, tutto ciò che riguarda direttamente l'applicazione viene posizionato nella parte alta dell'address space (i.e. la porzione con gli indirizzi più bassi rispetto alla zona in nero non utilizzabile), mentre tutto ciò che riguarda direttamente il sistema (e.g. il VDSO) viene posto nella parte bassa dell'address space.

Page table nelle architetture x86-64

Qui viene introdotta la paginazione a 4 livelli dove il nuovo livello di paginazione è chiamato **Page-Map level** e, in tutti i livelli, ogni pagina è composta da 512 entry. Con questo schema è possibile indirizzare 512^4 pagine di dimensioni pari a 4 KB ciascuna, per cui nominalmente si ha a disposizione una memoria totale proprio di 256 TB. La struttura delle page table è raffigurata qui di seguito:



Nella pagina seguente, invece, è riportato il contenuto delle entry delle page table di tutti e quattro i livelli di paginazione.

- Per quanto riguarda le entry della tabella di primo livello (PML4E), si hanno due possibilità: presence bit = 0 (entry non valida); presence bit = 1 (entry valida che punta a una tabella di secondo livello).
 - Per quanto riguarda le entry delle tabelle di secondo livello (PDPTE), si hanno tre possibilità: presence bit = 0 (entry non valida); presence bit = 1 AND page size bit = 0 (entry valida che punta a una tabella di terzo livello); presence bit = 1 AND page size bit = 1 (entry valida che punta direttamente a una pagina di 1 GB).
 - Per quanto riguarda le entry delle tabelle di terzo livello (PDE), si hanno tre possibilità: presence bit = 0 (entry non valida); presence bit = 1 AND page size bit = 0 (entry valida che punta a una tabella di quarto livello); presence bit = 1 AND page size bit = 1 (entry valida che punta direttamente a una pagina di 2 MB).
 - Per quanto riguarda le entry delle tabelle di quarto livello (PTE), si hanno due possibilità: presence bit = 0 (entry non valida); presence bit = 1 (entry valida che punta a una pagina di 4 KB).

Di seguito sono riportati tutti i campi della PTE per quanto riguarda le architetture x86-64:

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M-1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

Notiamo stavolta che è presente un flag che indica se la pagina di memoria puntata è eseguibile o meno (i.e. se vi si può effettuare il fetch delle istruzioni o meno). Tale flag è **XD** (che sta per **execute disable**), ed è molto importante dal punto di vista della sicurezza poiché *previene che un attaccante inietti del codice eseguibile malevolo all'interno dei segmenti dell'address space che non siano di testo (i.e. dati, stack e così via)*.

Direct mapping vs non-direct mapping in x86-64

Una entry della PML4 può essere associata fino a 2^{27} frame di memoria che, moltiplicati per 4 KB, danno luogo a 2^{29} KB = 2^9 GB = 512 GB. In tal modo, nei chipset più comuni, abbiamo spazio in abbondanza per effettuare il mapping diretto di tutta la RAM disponibile all'interno delle pagine del kernel (ed è quello che si fa tipicamente in Linux). Comunque sia, ogni volta che è necessario, è possibile anche rimappare la stessa memoria RAM in una maniera non diretta.

Huge Pages

Sono le pagine che possono essere puntate direttamente dalle PDE (ovvero quelle da 2 MB). A livello applicativo, possono essere mappate attivando il flag *MAP_HUGETLB* nella chiamata a *mmap()* oppure attivando il flag *MADV_HUGE PAGE* nella chiamata a *madvise()*. È possibile vedere quante huge page sono attualmente in uso nel sistema all'interno di */proc/meminfo* oppure all'interno di */proc/sys/vm/nr_hugepages*. Le pagine da 1 GB, invece, non sono utilizzabili a livello applicativo, bensì soltanto dal kernel.

Prendere una huge page di 2MB, per una certa regione mmappata, è una ottimizzazione perchè ho pagine fisiche (frame) contigue invece che frame diversi. Però dove è l'ottimizzazione? Perchè è meglio avere pagine vicine? Non c'entra il NUMA, ho facility che permetterebbe di avere indirizzi logici contigui e di unire i frame. La risposta è che se lavoro con pagine sparse, la probabilità che ci siano due linee di cache conflittanti è non nulla, mentre è nulla se pagine contigue. (Se ho due pagine diverse, e tocco la prima linea di cache di una, escludo l'altra perchè avrei un conflitto). Se ho due indirizzi fisici *i* ed *i'*, che corrispondono a stessa linea dell'architettura di cache (sottoinsieme della RAM). Con contiguità in memoria fisica, saranno sicuramente su due linee di cache diverse. Devo specificare quante *huge table* sono utilizzabili.

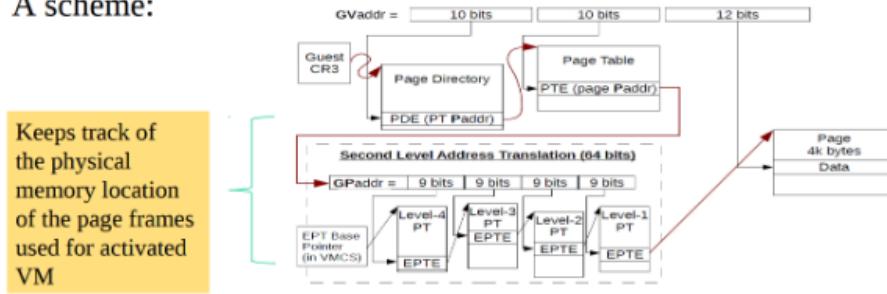
Supporto hardware alla virtual memory

Si usano delle shadow copies, ad esempio il CR3 è del processore virtuale. Allora Guest CR3, ci permette di arrivare su tabella di primo livello, poi secondo etc. L'indirizzo fisico non è un vero indirizzo fisico, bensì logico, e viene passato alla tabella delle pagine del processo virtual machine. Qui ci arriviamo con CR3 vero, e prendiamo il vero indirizzo fisico. Magari la VM la vede con indirizzo 0, ma nella realtà non lo è!

Il problema è che si compiono attività non consone alla sicurezza. **Ad esempio, se arriviamo ad una pagina non valida** (sia quando passo tre le tabelle primo e secondo livello, sia a Second Level Address Transaction), viene

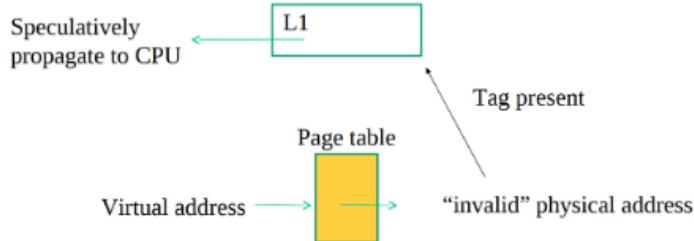
comunque passato in cache L1. Questo perchè “magari” qualcosa in cache L1 c’è, e questo è alla base dell’attacco **L1TF**. Ho thread su VM, ho configurato Page Table per avere indirizzo frame a cui voglio accedere, e ci metto bit di invalidità. **Se vi accedo, niente si mette in mezzo.**

- Intel Extended Page Tables (EPT)
- AMD Nested Page Tables (NPT)
- A scheme:



Attacco L1 Terminal Fault L1TF

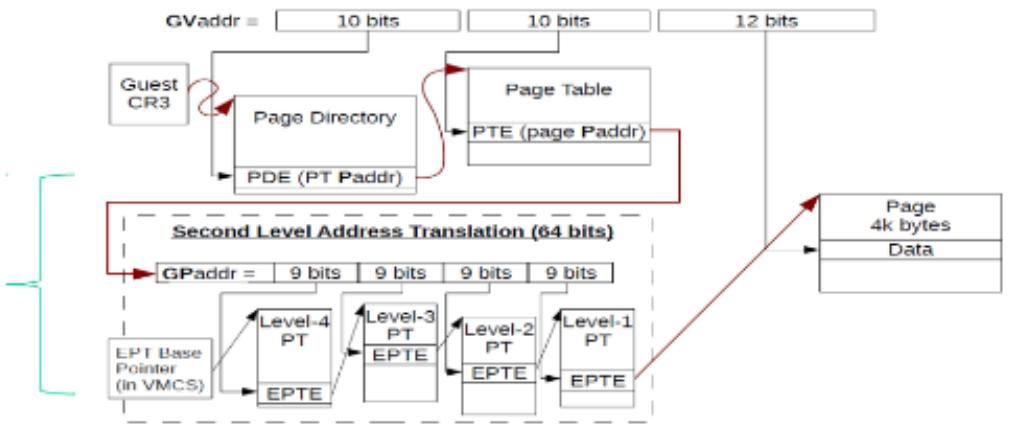
Sappiamo che, se accediamo a una entry di una page table, il primo controllo che viene effettuato è sul *presence bit*: se quest’ultimo è pari a 0 (entry non valida), potrebbe essere possibile eseguire delle istruzioni in modo speculativo sfruttando le informazioni contenute all’interno di quell’entry della page table. Da qui nasce l’**attacco L1TF**, che prevede la seguente idea: se abbiamo una entry di una page table con presence bit pari a 0 (non valida), la entry stessa potrebbe comunque aver propagato i suoi valori (e in particolare il TAG, ovvero i bit associati al presence bit) all’interno dell’architettura di memoria. Se il TAG viene utilizzato come un indice per andare a sporcare lo stato della cache (cache L1), si compie un attacco dello stesso stile di Meltdown: infatti, poiché il TAG appartiene a una entry non valida della page table, risulta essere un indice non accessibile (i.e. accessibile solo speculativamente).



L’attacco va a buon fine nel momento in cui il contenuto della entry invalida della page table mappa su dei dati attualmente salvati in cache. Infatti, una mitigazione che è stata attuata consiste nel fare in modo che il kernel imposti il valore delle entry non valide a valori opportuni che non mappano su dati cachabili. Si tratta di una mitigazione e non della soluzione definitiva perché esiste ancora un caso in cui la vulnerabilità è sfruttabile: quando una macchina virtuale è in esecuzione all’interno della macchina host, è possibile fare una detection delle informazioni all’interno della memoria fisica dell’host a partire dalla macchina virtuale guest.

A scheme:

Keeps track of the physical memory location of the page frames used for activated VM

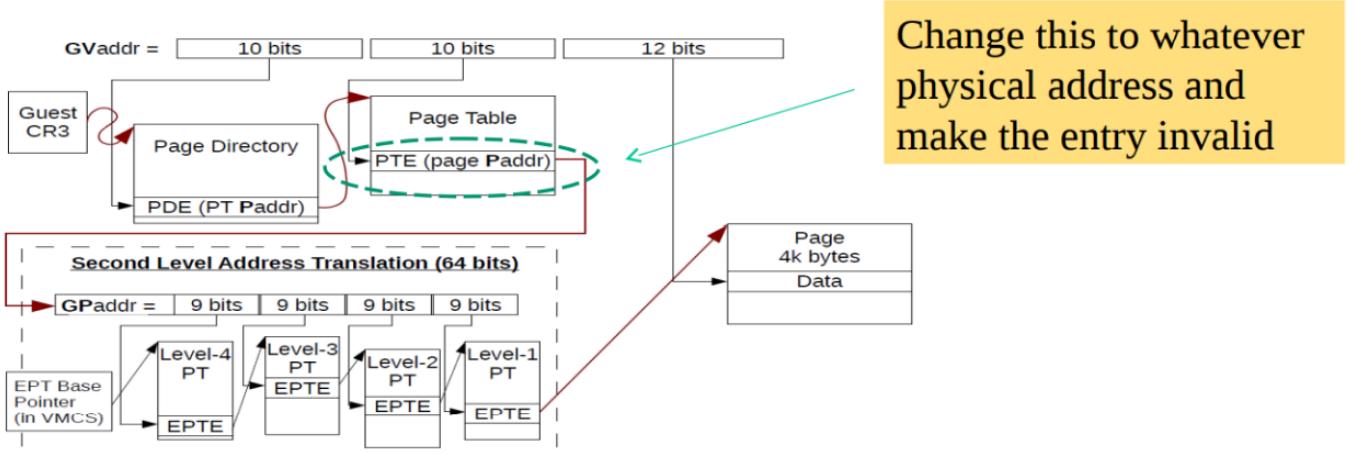


La **parte superiore** dello schema (comprendente Guest CR3, Page Directory e Page Table) è relativa alla paginazione che avviene all'interno della **macchina guest**, mentre la *parte inferiore* è relativa alla paginazione che avviene all'interno del *sistema host*.

Nel momento in cui bisogna accedere alla memoria a partire da un'applicazione che gira nella VM guest, si seguono i seguenti passaggi:

- Si ricorre al registro CR3 della macchina guest per recuperare l'indirizzo della PDE guest.
- A partire da un'apposita entry della PDE, si accede alla corrispondente PTE guest.
- Le entry della PTE guest non mappano direttamente su delle pagine fisiche in memoria, bensì conducono alla tabella delle **pagine top level del sistema host**.
- A partire da questo momento, la traduzione dell'indirizzo avviene secondo lo schema tradizionale basato su quattro livelli di paginazione.

A questo punto, performare un attacco L1TF a partire dalla VM guest è abbastanza semplice:



Supponiamo che in memoria fisica, all'indirizzo x , ci sia un dato cachabile, e consideriamo la entry e_1 della PTE guest. Allora è possibile fare in modo che e_1 sia non valida e abbia un contenuto che, **senza sfruttare la "second level address translation"** (del sistema host), punti proprio all'indirizzo fisico x .

Così, quando un'applicazione che gira all'interno della VM guest tenta di effettuare un accesso in memoria sfruttando proprio la entry e_1 della **PTE guest**, poiché il presence bit è pari a 0, *il processore non percorre i 4 livelli di paginazione relativi al sistema host (nemmeno speculativamente)*, bensì va a verificare speculativamente se il dato associato all'indirizzo x si trova nella cache L1 (in altre parole, se il presence bit vale 0, l'indirizzo che si ottiene dalla **PTE guest** non viene più considerato come indirizzo fisico GUEST bensì come indirizzo fisico HOST). Se sì, allora la macchina virtuale attaccante utilizza quel dato come indice per spiazzarsi in un probe array: in tal modo, sfrutta il side channel dato dalla cache per estrapolare delle informazioni relative all'esecuzione di altri thread o altre applicazioni, le quali possono anche girare in un'eventuale macchina virtuale vittima che vive all'interno

del sistema host. Una condizione necessaria per cui questo attacco vada a buon fine è che l'attaccante e la vittima girino in due hyperthread associati al medesimo CPU-core, in modo tale che condividano la medesima cache L1.

Core map

È una struttura dati che tiene traccia dello stato (e.g. libero vs occupato) dei frame all'interno del sistema e, quindi, ci permette di fare l'allocazione e la deallocazione delle pagine di memoria. Più precisamente, è un array in cui ciascuna entry corrisponde a un frame della RAM. Ciascuna entry viene rappresentata con la seguente struct C:

```
typedef struct page {
    struct list_head list;      /* ->mapping has some page lists. */

    ...
    atomic_t count;           /* Usage count, see below. */

    ...
    unsigned long flags;      /* atomic flags, some possibly
                                updated asynchronously */
    ...
} mem_map_t;
```

- **struct list_head list**: puntatore alla testa di una lista collegata; permette agli elementi della core map di essere collegati tra loro.
- **atomic_t count**: contatore che tiene traccia del numero di page table entry associati al frame in RAM.
- **unsigned long flags**: insieme di flag che indicano lo stato del frame in RAM.

Free list

E' una struttura dati definita nel seguente modo:

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    ...
    int nr_zones; //actually used zones
    ...
    struct page *node_mem_map;
    ...
} pg_data_t;
```

- **struct zone node_zone[MAX_NR_ZONES]**: array contenente le informazioni relative a ciascuna ZONE gestita dalla free list.
- **int nr_zones**: numero di ZONE gestite dalla free list.
- **struct page *node_mem_map**: puntatore alla prima entry della core map legata alla free list. Di fatto, esiste una free list differente per ogni nodo NUMA, e ognuna di esse tiene traccia di un sottoinsieme di pagine libere relativamente a una porzione della core map (in altre parole, possiamo vedere la core map come logicamente suddivisa in più porzioni, ciascuna delle quali è relativa a un nodo NUMA). È per questo motivo che è necessario tenere traccia di quale porzione della core map è associata alla nostra free list.

Ma vediamo ora com'è definita la struct zone che abbiamo come primo campo della struct pglist_data.

```

struct zone {
    ...
    free_area_t    free_area[MAX_ORDER];
    ...
    spinlock_t    lock;
    ...
    struct page   *zone_mem_map;
    ...
}

```

Where we do pick free memory
blocks in a buddy allocator

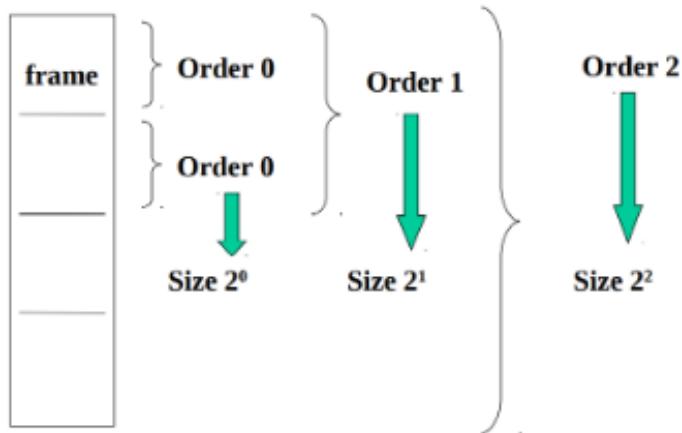
Up to 11 in recent kernel versions
(it was typically 5 before)

- `free_area_t free_area[MAX_ORDER]` = array di aree di memoria libere che possono essere allocate; nelle versioni più recenti del kernel, è un array composto da 11 entry (e non una sola) perché si vuole avere la possibilità di scegliere se allocare una pagina di memoria, oppure due contigue, oppure quattro contigue, e così via. Il sistema che permette di allocare più frame fisici contigui è noto come buddy allocator.
- `spinlock_t lock` = spinlock che serve a gestire correttamente la sezione della core map legata alla ZONE di interesse della free list corrente (i.e. del nodo NUMA in cui ci troviamo).
- `struct page *zone_mem_map` = puntatore alla prima entry della core map legata alla ZONE di interesse della free list corrente (i.e. del nodo NUMA in cui ci troviamo).

Buddy Allocator

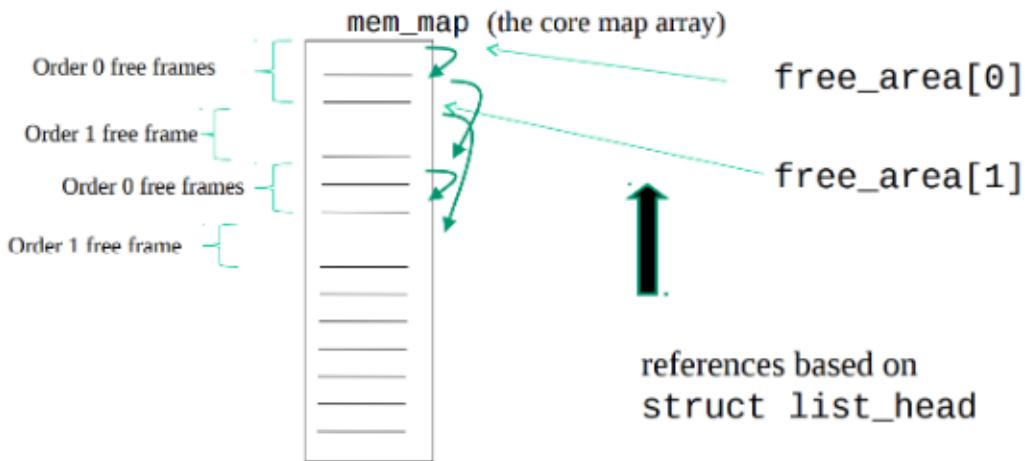
È un *allocatore di memoria fisica* che segue il seguente schema:

I frame possono essere presi singolarmente oppure a gruppi con una dimensione pari a una potenza di 2. I frame singoli costituiscono un gruppo di ordine 0, mentre i frame “accoppiati” costituiscono un gruppo di ordine n (dove 2^n è la dimensione del gruppo in termini di numero di frame). È possibile unire due gruppi contigui dello stesso ordine k in un unico gruppo di ordine k + 1; è altresì possibile suddividere un gruppo di ordine k in due sottogruppi contigui di ordine k - 1.

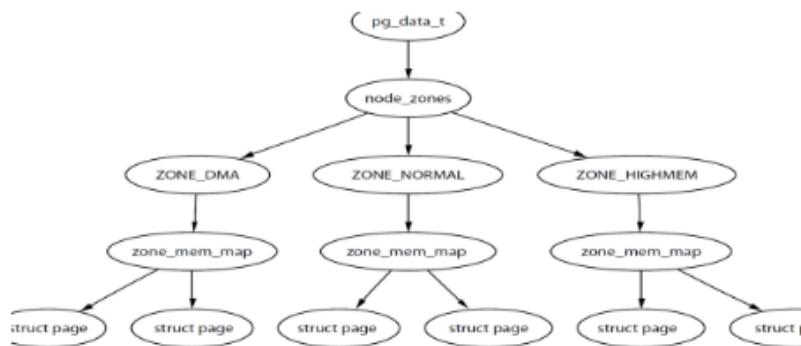


Tornando alla core map e alla free list:

- Per quanto riguarda la **core map**, è organizzata in più liste collegate, dove ciascun elemento di ogni lista è relativo a una entry della core map stessa. In particolare, si ha una lista collegata per gli elementi di ordine 0, una lista collegata per gli elementi di ordine 1, e così via.
- Per quanto riguarda la **free list**, l'array `free_area` definito all'interno della struct zone è organizzato così: la entry i-esima contiene un puntatore alla prima entry della porzione corrente della core map associata a un elemento di ordine i (dove la porzione corrente della core map sarebbe quella porzione della core map associata a una specifica ZONE di uno specifico nodo NUMA).

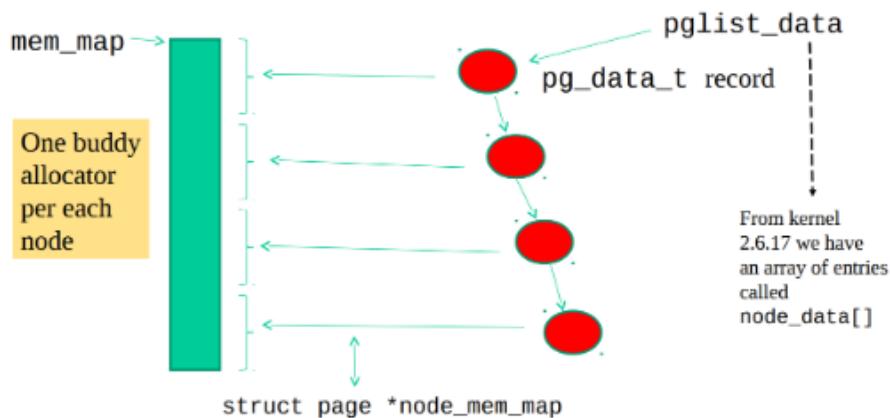


Schema associato ad uno specifico nodo NUMA



Caso di kernel NUMA-aware

Come già detto, nel caso dei sistemi composti da molteplici nodi NUMA si hanno più free list distinte (una per ogni nodo NUMA, appunto). Di conseguenza, sono definite più `struct* pg_data_t` legate tra loro tramite una lista collegata. Tali struct, dunque, hanno anche un campo `node_next` (anch'esso di tipo `struct pg_data_t`) per andare a formare così una lista collegata di tipo `struct pglist_data`. A valle di questa considerazione, lo schema che lega la core map (`mem_map_t`) con le free list (`pg_data_t`) è il seguente:



A partire dalla versione 2.6.17 del kernel i vari `pg_data_t` sono definiti all'interno di un apposito array chiamato `node_data[]`.

Può capitare che un thread in esercizio, a livello user, riceva un interrupt, il quale fa partire un handler, il quale potrebbe maneggiare la memoria verde, o anche chiamare qualcosa per la gestione, e rimanere bloccato. Ma quindi anche il thread rimarrebbe bloccato, quindi a livello kernel dobbiamo fare in modo che le API siano “non anonime”, quindi se l'API può mandarti o meno in blocco. Ciò è alla base dei seguenti concetti:

Contesti di allocazione

Ma quali sono i contesti di allocazione da affrontare nel momento in cui viene richiesta memoria?

- **Process context:** l'allocazione di memoria è causata da una system call o da una trap (e.g. page fault). Se la system call o la trap non è soddisfacibile, il thread interessato viene posto in uno stato di wait. Qui l'assegnazione della memoria ai vari thread utilizza uno schema basato su priorità (possono essere forniti dei frame fisici prima a un thread con una priorità più alta).
- **Interrupt context:** l'allocazione di memoria viene richiesta da un interrupt handler. L'interrupt handler non può essere *mai posto in attesa, neanche se la richiesta è non soddisfacibile*. Stavolta, l'assegnazione della memoria ai vari thread non utilizza uno schema basato su priorità.

In entrambi i contesti di allocazione, si passa all'esecuzione in modalità kernel per effettuare l'allocazione della memoria.

Api del buddy system

- `unsigned long get_zeroed_page (int flags)`: rimuove una sola pagina dalla free list, azzerà il contenuto di tale pagina e ne restituisce l'indirizzo logico. Il parametro flags specifica il contesto di allocazione ed, eventualmente, la priorità che si vuole assegnare al thread corrente per l'assegnazione del frame di memoria.
- `unsigned long __get_free_page (int flags)`: è come la `get_zeroed_page()` con l'unica differenza che non azzerà il contenuto della pagina che restituisce.
- `unsigned long __get_free_pages (int flags, unsigned long order)`: è come la `__get_free_page()` ma, anziché allocare una singola pagina di memoria, ne allocà un gruppo di ordine k (i.e. allocà 2^k pagine). Il parametro *order* specifica proprio il valore k dell'ordine.
- `void free_page (unsigned long addr)`: dealloca una singola pagina di memoria. Il parametro addr indica l'indirizzo logico della pagina da deallocare.
- `void free_pages (unsigned long addr, unsigned long order)`: è come la `free_page()` ma, anziché deallocare una singola pagina di memoria, ne dealloca un gruppo di ordine k (i.e. dealloca 2^k pagine).

NB: se si passano a `free_page()` o a `free_pages()` dei parametri errati o inadeguati, si può andare incontro a una corruzione del kernel. Se prendo un blocco di una certa taglia, non posso rilasciarne un sottoblocco.

Vediamo quali sono i valori principali che possono essere assunti dal parametro flags:

- `GFP_ATOMIC`: si vuole che il chiamante non venga posto nello stato di sleep (per cui ci troviamo nel caso dell'interrupt context).
- `GFP_USER – GFP_BUFFER – GFP_KERNEL`: è ammesso che il chiamante venga posto nello stato di sleep (i tre flag sono elencati in ordine crescente di priorità).

Tutte le API di allocazione che abbiamo visto restituiscono l'indirizzo virtuale di pagine di memoria **directly mapped**. Di fatto, la contiguità delle pagine allocate è garantita sia per gli indirizzi logici che per gli indirizzi fisici. Si lavora con normal memory (*frame directly mapped*), ma se volessi una high memory (*non directly mapped*), devo chiedere ad altro allocatore, che alloca tutto e mette in page table. Però se prendo frame diversi, non posso ritornare un unico indirizzo fisico, e quindi non tutti i kernel permettono tale operazione. Molto spesso ci viene data della normal memory, high memory non molto usata. Nelle release correnti il buddy allocator da indirizzi *directly mapped* (allineata sui "limiti" dei frame), se chiedessi high memory uso altro allocatore, che potrebbe non essere allineata (per termini di sicurezza ad esempio). Anche buddy allocator usa cache, senza far riferimento a free list e meccanismi di lock. La sua cache è un concetto di "quick list". Nel buddy system prelevo alcune pagine, messe in più liste di ordine di livello superiore rispetto alle strutture considerate. Tali liste sono per-cpu, quindi niente lock. Quale allocatore buddy, con macchina NUMA avente più allocator, uso? Soluzione: **mem-policy**.

Infatti, se ci facciamo caso, le API non permettono di specificare il nodo NUMA in cui si vuole allocare la memoria (i.e. non permettono di specificare l'esatta free list che si vuole coinvolgere). Trattiamo API di alto livello che a loro volta invocano un'API di più basso livello che specifica il nodo NUMA che deve essere coinvolto nell'allocazione. Tale API è:

```
struct page *alloc_pages_node (int nid, unsigned int flags, unsigned int order)
```

Il valore di **nid*** (ovvero il nodo NUMA in cui deve avvenire l'allocazione) viene selezionato in base a delle **mempolicy data** (è un METADATO sul thread control block, politiche legate alla gestione della memoria) relative al thread chiamante. Inizialmente tali politiche erano gestite esclusivamente a livello kernel ma successivamente è stata messa a disposizione un'API user level per modificarle, in modo tale che il programmatore possa avere il controllo su quale

nodo NUMA venga coinvolto nelle allocazioni delle pagine di memoria. Analizziamo anche questa API:
`int set_mempolicy (int mode, unsigned long *nodemask, unsigned long maxnode)`

Che cosa vanno a indicare i parametri?

- `int mode`: stabilisce la modalità con cui il kernel deve allocare memoria per conto del software applicativo. Può assumere uno dei seguenti valori:
 - `MPOL_DEFAULT` (prendo sul nodo NUMA corrente, non sempre è ottimale, se un thread inizializza altri thread, questi thread avrebbero dipendenza dalla cpu del thread inizializzante).
 - `MPOL_BIND` (secondo cui viene selezionato un unico nodo NUMA da coinvolgere nelle allocazioni).
 - `MPOL_INTERLEAVE` (secondo cui il nodo NUMA da coinvolgere in ogni allocazione viene selezionato secondo uno schema Round-Robin, usato durante il boot del kernel da idle process).
 - `MPOL_PREFERRED`. Riguardano interrupt context.
- `unsigned long nodemask`: puntatore a una maschera di bit, dove ciascun bit è relativo a un nodo NUMA della macchina. L'i-esimo bit vale 0 se non si vuole che il thread chiamante utilizzi l'i-esimo nodo NUMA per le allocazioni di pagine di memoria, vale 1 altrimenti.
- `unsigned long maxnode`: dimensione della maschera di bit.

L'API appena esaminata è di carattere generale, ma ne esiste anche un'altra che associa una specifica area di memoria logica a una mempolicy:

```
int mbind (void *addr, unsigned long len, int mode, unsigned long *nodemask, unsigned long maxnode,  
unsigned flags)
```

Rispetto a `set_mempolicy()`, qui si hanno due parametri in più:

- `void *addr`: indirizzo di memoria logica a partire dal quale si vuole impostare la mempolicy.
- `unsigned long len`: numero di byte per cui si vuole impostare la mempolicy.

NB: la memoria logica coinvolta in questa API deve essere necessariamente user level.

Infine, è possibile migrare delle pagine di memoria (sempre user level) da un nodo NUMA a un altro mediante quest'altra API:

```
long move_pages (int pid, unsigned long count, void **pages, const int *nodes, int *status, int  
flags)
```

Pagine mai migrate da sole, solo con swap out e swap in, ma ad esempio oggi non c'è più la swap area, ma swap file (quindi tramite file system), però la swap area oggi ha poco senso con le ram che abbiamo.

Che cosa vanno a indicare i parametri?

- `int pid`: ID del processo di cui si vogliono migrare le pagine.
- `unsigned long count`: numero delle pagine che si vogliono migrare.
- `void pages`: puntatore alle pagine di memoria che si vogliono migrare.
- `const int *nodes`: puntatore ai nodi verso cui si vogliono migrare le pagine.
- `int *status`: puntatore all'area di memoria in cui verrà registrato l'esito della migrazione.
- `int flags`: flag che indicano delle restrizioni sulle pagine che si vogliono migrare.

Caso di allocazioni/deallocazioni frequenti

Nel caso in cui si abbiano allocazioni e deallocazioni frequenti di strutture dati “*target-specific*” (= che vengono utilizzate con un obiettivo specifico, vedi ad esempio la page table) non conviene più ricorrere al buddy allocator poiché quest'ultimo lavora in maniera sincronizzata tramite uno spinlock: infatti, si tratterebbe di una soluzione che non scala perché le diverse richieste sull'allocazione / deallocazione della stessa struttura dati verrebbero serializzate. Di base, questo problema viene risolto sfruttando i cosiddetti **buffer pre-reserved**, all'interno dei quali vengono precaricate delle pagine di memoria, in modo tale che siano già pronte nel momento in cui un'allocazione o deallocazione viene richiesta; tali buffer costituiscono degli allocator alternativi che concorrono a rendere il sistema più efficiente e scalabile per quanto riguarda l'aspetto del memory management. Implementati tramite quicklist, cioè freelist di singole pagine, associate ad una cpu, come? Vado a variabile per-cpu e punto alla prima delle pagine, quindi ho una lista di pagine per-cpu.

Quicklist

Sono dei buffer pre-reserved per il caso specifico delle page table. Per gestire questi buffer, si ricorre a particolari API pensate per la paginazione a 4 livelli (pensate cioè per le PGD, PMD, PUD e PTE): `pgd_alloc()`, `pmd_alloc()`, `pud_alloc()`, `pte_alloc()`, `pgd_free()`, `pmd_free()`, `pud_free()` e `pte_free()`. Informalmente, si può affermare che queste API implementano il caching delle pagine di memoria. Le quicklist sono implementate come delle liste **per-core**: è proprio per questa ragione che non c'è bisogno di sincronizzazione per utilizzarle. Prendo cpu, prendo variabile, prendo il pointer, vedo se l'oggetto puntato esiste, se esiste lo scrollo, e mi prendo la memoria. Ci lavoro solo io. Non ci sono istruzioni atomiche, non faccio vedere a nessuno che faccio qualcosa, ho marcato il dato nel TLB, se arriva una interrupt resto io in CPU. Thread può ricevere interrupt, chiama handler, handler ritorna, ma thread rimane in CPU. Potrebbe lasciare la CPU in altri casi. Quindi preemptable != interrompibile.

Possiamo essere interrompibili e non preemptable (non lascia cpu).

Dopo `put_cpu_var(quicklist)` ritorna preemptable, in mezzo non lo è. Informazioni per CPU, non posso lasciarla perchè lavorerei solo in quella CPU. Se lista è vuota, allora prendo `free_pages`.

Comunque sia, a partire dalla versione 4 del kernel Linux, il pre-reserving può essere fatto direttamente con le API del buddy allocator. In versioni ancora più recenti, le quicklist sono proprio uno dei componenti del buddy system, il che permette di utilizzare in modo trasparente le quicklist stesse facendo riferimento alle API relative alla buddy allocation. In ogni caso, le quicklist sono disponibili anche in altre salse per la programmazione di livello kernel. Qui la cache è per-cpu, ma potrei avere anche non per singola cpu.

Buddy system è oggetto condiviso tra le cpu, quindi richiede lock. Le quicklist permettono di lavorare in maniera scalabile, con cache buffer per-cpu, quindi non c'è conflitto con altre cpu. Questo vale per la singola pagina, o multipli di una pagina. Se volessi *meno di una pagina?*

SLAB / SLUB allocator

È un allocatore utilizzato nel caso in cui si voglia avere a che fare con aree di memoria di dimensioni minori rispetto alla pagina. Si parla di cache lato kernel, siano esse generali (“voglio memoria”), o più specifiche.

Il meccanismo di acquisizione da basso livello è basato su *lazyness*. Abbiamo una cache che gestisce una taglia T , creo una seconda cache (e di conseguenza un secondo metadata). Le chiamate sulla seconda cache possono usare la stessa area della prima cache. Una cache per l'allocazione di memoria è un oggetto di memoria pre-riservato. Vediamo la cache come set di informazioni disponibili, senza scendere giù per il buddy system, è layer software superiore. L'attività di pre-reserving viene fatta dal *buddy allocator*, quindi è *directly mapped*. Fa uso delle seguenti API:

- `void *kmalloc (size_t size, int flags)`: alloca un'area di memoria contigua di una dimensione prestabilita e ne restituisce l'indirizzo logico (primo byte linea di cache); l'area di memoria è di livello kernel ed è directly mapped. Il parametro flags indica la priorità dell'allocazione. Mappa su mem-cache virtuale su specifico nome, e nome riflette la taglia.
- `void *kzalloc (size_t size, int flags)`: è come `kmalloc()` con l'unica differenza che azzerà l'area di memoria da allocare.
- `void kfree(void *obj)`: dealloca l'area di memoria di livello kernel associata all'indirizzo logico passato come parametro.

Possiamo vedere le cache con: `sudo cat /proc/slabinfo`

Quando viene invocata `kmalloc()` o `kzalloc()`, l'allocatore restituisce una zona di memoria pre-reserved; inoltre, vengono controllate le mempolicy del thread chiamante per capire qual è il nodo NUMA in cui l'allocazione deve avvenire. È quindi chiaro che, internamente alle chiamate a `kmalloc()` e `kzalloc()`, viene invocata la seguente altra API, che specifica anche il nodo NUMA da coinvolgere nell'allocazione:

```
void *kmalloc_node(size_t size, int flags, int node)
```

Di conseguenza, esistono più allocatori SLAB (uno per ogni nodo NUMA), per cui è ammessa la concorrenza nell'allocazione di memoria in diversi nodi NUMA.

Slab allocator virtuali

È possibile creare dei nuovi SLAB allocator che lavorano con “chunk” di una certa dimensione D (dove il chunk corrisponde alla zona di memoria che verrà messa a disposizione con un'operazione di allocazione). Se però esisteva già un allocatore che lavora con chunk della stessa dimensione D, allora viene creata solo un'istanza virtuale di SLAB allocator che fisicamente mappa su quello già esistente. D'altra parte, è possibile effettuare il destroy di uno SLAB allocator A solo nel momento in cui gli eventuali chunk allocati da A sono stati tutti rilasciati.

Vediamo un po' di API che permettono di manipolare questi allocator: (queste a più basso livello di quelle viste sopra)

- `struct kmem_cache *kmem_cache_create (char *name, size_t size, size_t align, unsigned long flags, void (*ctl)(void *))`: crea un nuovo SLAB allocator. L'ultimo parametro è function pointer (riceve `void*`, indirizzo generico), ci permette di inizializzare l'area di memoria in cui farò le allocazioni, se diverso da `NULL`. Se `NULL`, ed esiste una cache che gestisce la size di chunks, è come se venisse sovrascritta. In questo caso non mi serve un nome specifico, basta poter gestire quella size. Size è la size del chunk. Quando realmente uso quella memoria, viene ritornata, la funzione viene richiamata iterativamente su tutti i chunks della cache.
- `int kmem_cache_destroy (struct kmem_cache *cache)`: distrugge uno SLAB allocator.
- `void *kmem_cache_allocator (struct kmem_cache_t *cache, int prio)`: alloca della nuova memoria tramite l'allocatore specificato come parametro. “Prio” è bloccante o non bloccante.
- `void kmem_cache_free (struct kmem_cache_t *cache, void *ptr)`: dealloca un'area di memoria precedentemente allocata.

Le free-list è una lista per-cpu, la release-list è invece concorrente tra le cpu.

Slab coloring

Quando viene creato un nuovo allocatore fisico, avrà associato un **colore**, che è un codice numerico indicante l'offset del primo buffer/chunk di memoria libero da consegnare quando viene richiesta un'allocazione. In realtà, due SLAB allocator associati alla stessa taglia (dimensione D) possono anche avere dei colori differenti: infatti, se idealmente tutte le pagine di memoria possono essere mantenute in cache ma molti allocator vanno a lavorare sullo stesso offset, si avranno dei conflitti all'interno della cache; di conseguenza, avere molteplici colori porta all'ottimizzazione dell'uso della cache.

Sia ALN l'allineamento del chunk che vengono consegnati dal nostro allocatore. Allora l'offset del primo buffer di memoria libero da consegnare è pari a $D + ALN \cdot COLOR$.

Allocazioni di aree di memoria molto ampie

Nelle versioni più recenti del kernel il buddy allocator è in grado di allocare fino a 2^{11} pagine di memoria contigue ma, precedentemente, il limite era addirittura di 2^5 pagine. Di conseguenza, è possibile avere il bisogno di allocare un'area di memoria di dimensioni superiori rispetto a quanto consentito dal buddy allocator (questo avviene ad esempio quando si vuole montare un modulo del kernel). Per tale necessità si ricorre a un ulteriore allocatore per la memoria kernel: il **vmalloc**, che è in grado di allocare delle pagine che sono *contigue per la memoria logica*, ma non necessariamente per la memoria fisica (per cui non si tratta necessariamente di memoria directly mapped).

Le API che permettono di utilizzare il vmalloc sono le seguenti:

- `void *vmalloc (unsigned long size)`: alloca un'area di memoria con le dimensioni specificate dal parametro in input, e restituisce il relativo indirizzo logico. Si usa per puntare i moduli, non per le interrupt. Qualsiasi struttura dati messa lì dentro non è detto sia directly mapped, dovrei usare buddy allocator in tal caso. Poi potrei usare un allocatore che ritorna questa memoria così definita.
- `void free (void *addr)`: dealloca un'area di memoria precedentemente allocata con una `vmalloc()`.

Confronto tra kmalloc e vmalloc

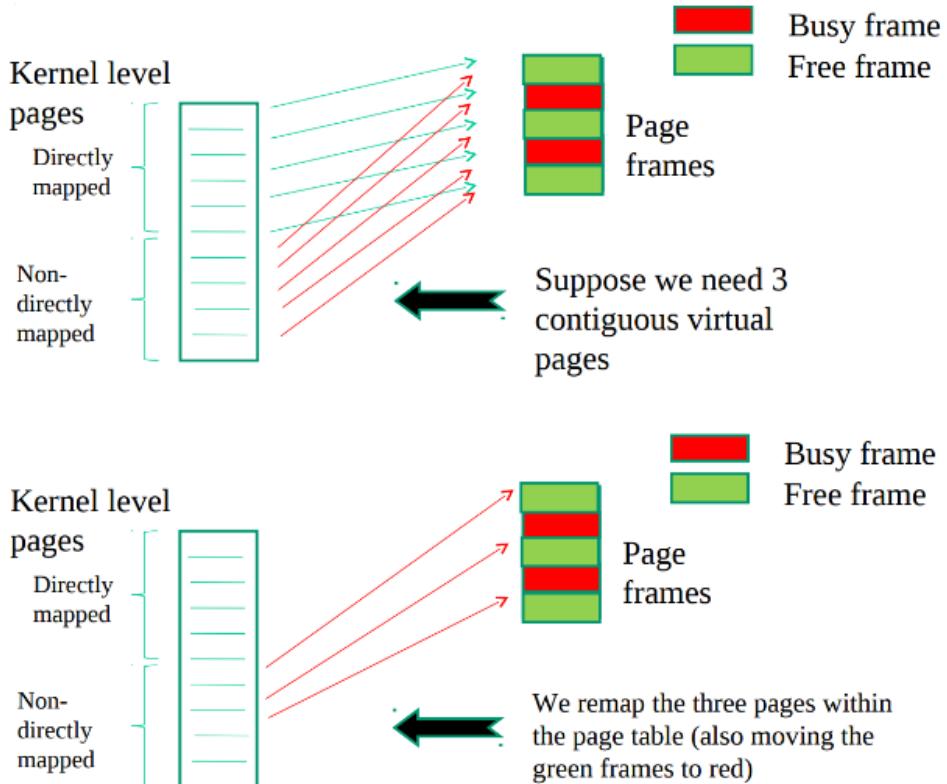
- Dimensioni della memoria che può essere allocata:
 - 128 KB per kmalloc (cache aligned)
 - 64-128 MB per vmalloc
- Contiguità della memoria fisica allocata:
 - Sì per kmalloc
 - No per vmalloc

Effetti sul TLB, interazione con hardware

TLB è l'acronimo di *Translation Lookaside Buffer*, una struttura dati che cacha le associazioni indirizzo logico → indirizzo fisico):

- *Nessun effetto per kmalloc*: di default le pagine di memoria del kernel vengono registrate come directly mapped; a seguito di una loro allocazione mediante lo SLAB allocator (il quale fa riferimento al buddy allocator), queste rimangono certamente directly mapped, per cui il TLB non deve essere sottoposto ad alcun aggiornamento.
- *Effetti globali per vmalloc*: il vmalloc fa anch'esso riferimento al buddy allocator, ma può invocarlo più volte per ottenere tutta la memoria desiderata; questo può portare ad avere memoria fisica non contigua (ovvero pagine di memoria non directly mapped), il che causa la necessità di aggiornare il TLB.

Effetti sul TLB usando vmalloc Vediamo più nel dettaglio cosa può avvenire internamente quando si allocano delle pagine di memoria con vmalloc. Supponiamo di avere 5 **page frames** *non directly mapped* di cui solo la seconda e la quarta risultano occupate (i.e. già allocate), e supponiamo di voler richiedere all'allocatore **vmalloc** **tre pagine di memoria virtuale contigue**. Allora avviene la seguente cosa:



In pratica, affinché vengano restituite al chiamante le prime tre pagine logiche tra quelle non directly mapped, queste tre pagine vengono rimappate in modo tale da corrispondere ai tre frame fisici liberi (anche se questi frame fisici non sono contigui).

NB₁: la richiesta di allocazione di nuove pagine non directly mapped non è l'unica possibile causa del re-mapping: anche la modifica dei permessi di accesso alle pagine stesse può portare al re-mapping, poiché comunque sia si tratta di informazioni che salgono sul TLB.

NB₂: chiaramente, nella realtà, con vmalloc vengono tipicamente rimappati dei blocchi di pagine piuttosto ampi (e non le singole pagine).

Come accennato precedentemente, l'aggiornamento del TLB deve avere una natura globale poiché deve essere visibile a qualunque CPU-core: di fatto, qualsiasi thread (mentre è in esecuzione in modalità kernel) deve essere in grado di raggiungere una qualsiasi pagina di memoria del kernel.

Operazioni implicite vs esplicite sul TLB

Di base, il TLB deve essere modificato (o almeno invalidato):

- Nel momento in cui viene cambiata la page table a cui si fa riferimento (→ aggiornamento del registro CR3)
- Nel momento in cui viene modificata una qualche entry della page table (→ mmap / unmap di alcune pagine di memoria).

Il livello di automazione nella gestione del TLB dipende dall'architettura hardware specifica. Per essere sicuri che la gestione del TLB avvenga correttamente a prescindere dal livello di automazione della sua gestione, si fa uso dei **kernel hook**, che si occupano della gestione esplicita delle operazioni del TLB; gli hook vengono mappati a compile-time a delle *null operations* solo nel caso in cui il TLB è gestito del tutto automaticamente.

Per quanto concerne i processori x86, l'automazione è solo parziale. Infatti: - L'invalidazione del TLB avviene in **modo automatico** solo nel caso in cui viene aggiornato il registro CR3. Se ho due hyperthread su stesso core, invalido il MIO CR3, non quello dell'altro hyperthread. - Eventuali cambiamenti che si hanno all'interno delle singole page table non scatenano l'invalidazione automatica del TLB (per cui in tal caso si deve ricorrere ai kernel hook).

Tipi principali di eventi sul TLB

Classificazione rispetto alla **scala degli eventi** sul TLB possono essere:

- **Globali** se hanno a che fare con indirizzi virtuali accessibili da qualunque hyperthread fisico in *real-time-concurrency* (i.e. indirizzi delle pagine del kernel).
- **Locali** se hanno a che fare con indirizzi virtuali accessibili in *time-sharing concurrency*; la località è in particolare rispetto all'applicazione.

Classificazione rispetto alla **tipologia degli eventi** sul TLB possono essere:

- **Remapping** degli indirizzi di memoria, (i.e. cambio della traduzione da indirizzo logico a indirizzo fisico, tipo `vmalloc`).
- Modifica delle **regole di accesso** agli indirizzi logici (read only vs read/write).

Tipicamente, l'aggiornamento delle informazioni all'interno del TLB va fatto a seguito del flush del TLB (invalidazione non selettiva) o di una sua porzione (invalidazione selettiva).

Ad esempio, a livello globale, flusho tutti i TLB, mentre a livello locale flusho secondo regole, e il tutto parte da chi stava lavorando a livello locale.

Costi del flush del TLB

Per quanto riguarda i **costi diretti**:

- Latenza del protocollo di livello firmware per l'invalidazione delle entry del TLB (che sia essa selettiva o non selettiva).
- Latenza per il coordinamento cross-CPU nel caso in cui il flush sia globale (i.e. coinvolga tutte le CPU della macchina).

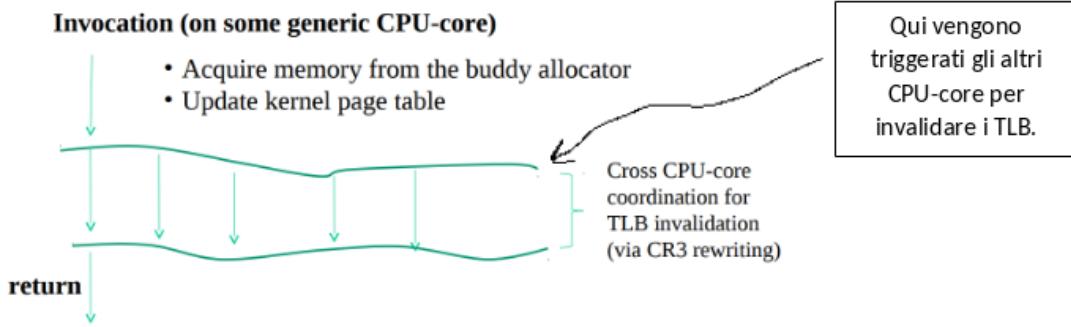
Per quanto invece riguarda i **costi indiretti**:

- Latenza dell'inserimento delle informazioni aggiornate all'interno del TLB in caso di miss (dove il miss è una diretta conseguenza del flush del TLB). Questo costo dipende dal numero di entry del TLB che devono essere rinnovate.

Flush globale del TLB su Linux

- `void flush_tlb_all(void)`: è un'API che effettua il flush di tutte le TLB di tutti i CPU-core attivi nel sistema. Chiaramente è molto dispendiosa, e il costo dipende anche dalle istruzioni macchina offerte dall'ISA dell'architettura che si sta usando.

Nel caso dell'x86, non esiste alcuna istruzione che effettui direttamente il flush di tutti i TLB di tutti gli hyperthread, per cui `flush_tlb_all()` flusha il TLB del CPU-core in cui gira il chiamante e fa sì che vengano triggerati tutti gli altri CPU-core affinché anche loro effettuino il flush. Per far ciò, vengono applicati degli schemi a livello software, il che inficia ulteriormente sul ritardo dovuto all'invocazione di `flush_tlb_all()`. Delle funzioni che potrebbero chiamare al loro interno `flush_tlb_all()` (ma non necessariamente) sono `vmalloc()` e `vfree()`, il cui funzionamento è schematizzato qui di seguito:



Sostanzialmente metto della barriera, perchè tutti devono completare l'operazione. Il che è molto dispendioso in termini di tempo.

Flush parziale del TLB su Linux

- `void flush_tlb_mm (struct mm_struct *mm)`: è un'API che effettua il flush di tutte le entry del TLB che sono relative alla parte user-space del chiamante. Viene invocata solo quando è stata eseguita un'operazione che coinvolge l'intero address space (e.g. dopo che il mapping della memoria è stato duplicato con `dup_mmap()` per eseguire una fork, oppure dopo che il mapping della memoria è stato eliminato con `exit_mmap()` per terminare il processo). Thread che da user a kernel chiama `mprotect()` richiede di aggiornare TLB ad esempio. Mandiamo avvertimento se sta girando su un thread di questo processo. Quindi non aspettiamo che altri completino l'aggiornamento, avvertiamo solamente.
- `void flush_tlb_range (struct mm_struct *mm, unsigned long start, unsigned long end)`: è un'API che effettua il flush delle entry del TLB associate alla zona di memoria compresa nei limiti dati dai parametri start, end. Viene invocata dopo che una regione di memoria è stata spostata (e.g. con una `mremap()`) oppure quando vengono cambiati i permessi di accesso alla zona di memoria (e.g. con una `mprotect()`).
- `void flush_tlb_page (struct vm_area_struct *vma, unsigned long addr)`: è un'API che effettua il flush di una singola pagina di memoria dal TLB. Supporta iterativamente il `tlb_range`, e quindi anche `tlb_mm`. `vma struct` indica che nell'address space c'è una zona start address ed end address, utilizzabili. C'è una lista gestita e aggiornata quando si chiama una `mmap`.
- `invlpg`: è un'istruzione offerta dall'ISA dell'x86 che effettua il flush di una particolare entry del TLB.
- `void flush_tlb_ptables (struct mm_struct *mm, unsigned long start, unsigned long end)`: è un'API che effettua il flush delle entry del TLB associate a una specifica page table. Viene invocata quando tale page table viene dismessa.
- `void update_mmu_cache (struct vm_area_struct *vma, unsigned long addr, pte_t pte)`: è un'API che può essere utilizzata per *precaricare* delle entry sul TLB a seguito di un page fault. In maniera *machine dependent* ripopolare TLB eventualmente presente nell'architettura stessa. Se non le supporta, possono essere esposte ma non fare nulla.

Cross Ring Data Move

Introduzione

Sappiamo che possiamo cambiare il flusso di esecuzione dalla modalità user alla modalità kernel e viceversa. Finora siamo stati abituati a pensare che, per passare le informazioni da una modalità all'altra (e, più in generale, da un ring a un altro), si sfruttano i registri del processore. Tuttavia, molto spesso la taglia dei registri non è sufficiente per ospitare i dati da trasportare da un livello di protezione all'altro. Come facciamo? Se popolassimo i registri con dei puntatori alle aree di memoria contenenti i dati da passare all'altro ring, romperemmo completamente il modello di protezione ring-based: di fatto, per un'applicazione user level sarebbe possibile andare in modalità privilegiata passando al kernel un puntatore a un'area di memoria kernel space; questo si tradurrebbe nella possibilità da parte dell'esecuzione in modalità non privilegiata di scrivere delle informazioni di livello kernel mediante una chiamata a una system call.

La soluzione vera a questo problema dipende da numerosi fattori, tra cui l'effettivo supporto alla segmentazione e la presenza (o assenza) di meccanismi di protezione addizionali che si hanno nell'hardware.

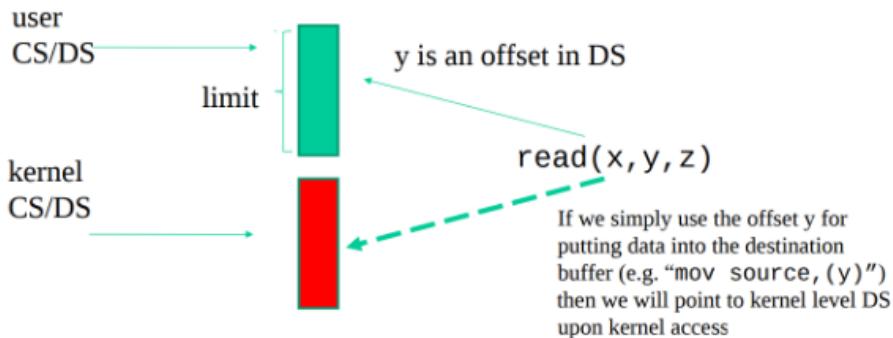
Caso della segmentazione flessibile

È la segmentazione di *x86 protected mode*, in cui è possibile fare in modo che i vari segmenti (come CS, DS) puntino ovunque vogliamo all'interno dell'address space.

- **Vantaggio:** si ha una separazione completa dei segmenti all'interno dell'address space, e questo previene le letture e le scritture illegali nei segmenti del kernel.
- **Problema:** c'è bisogno di un meccanismo per rendere la protezione dei segmenti trasparente al software.

Alla fine, questa soluzione consiste nell'avere all'interno di uno spazio di indirizzamento di un'applicazione un certo numero di segmenti user level che ricoprono gli indirizzi di memoria più bassi e un certo numero di segmenti kernel level che ricoprono gli indirizzi di memoria più alti; c'è dunque un indirizzo nel mezzo al di sotto del quale ricadiamo sicuramente in aree di memoria user space e al sopra del quale ricadiamo in aree di memoria kernel space (ricordiamoci però che questa non è una soluzione adottata da Linux).

Consideriamo ora la system call `read(x, y, z)`, dove x è il canale di I/O da cui si vuole leggere, y è l'offset all'interno del segmento dati (DS) in cui si vogliono riportare i byte letti e z è il numero di byte da leggere. Qui il problema risiede nel fatto che `read(x, y, z)` è una system call che viene invocata quando si è in modalità user ma, poiché poi viene eseguita direttamente dal kernel, y viene trattato come un offset rispetto al segmento DS di livello kernel, in particolar modo se il segmento da utilizzare viene scelto dal compilatore e non dal programmatore (i.e. se il compilatore, nel definire l'istruzione di `mov` dei dati letti verso la locazione indicata da y , sceglie come indirizzo di memoria di destinazione l'offset y rispetto al segmento DS di livello kernel). Questo problema, da capo, rompe il modello di protezione ring-based.



Soluzione La soluzione al problema appena esposto consiste nel rendere “**handcrafted**” (= scritta in maniera machine dependent dal programmatore del software del kernel, quindi non viene fatto automaticamente dal compilatore) i pezzi di codice del kernel per lo spostamento cross ring dei dati. Più specificatamente, è possibile sfruttare un selettor di segmento programmabile per mappare momentaneamente il segmento FS sul segmento DS *user level* e poi applicare il displacement y a FS per il movimento dei dati (alla fine dell'operazione FS viene ripristinato): in tal modo, per la `read()`, viene utilizzata un'area di memoria user space e non più kernel space.

L'operazione appena descritta è di tipo **segmentation fixup**. Chiaramente, comporta dei costi relativi al cambio di stato dei registri del processore (i.e. dei selettori di segmento), che di fatto richiedono degli accessi aggiuntivi alla memoria.

Caso della segmentazione constrained

È la segmentazione di *x86 long mode*, ma anche di x86 protected mode nel caso in cui si usano sistemi operativi che forzano il posizionamento dei segmenti CS, DS, SS ed ES (sia *user level* sia *kernel level*) a offset 0x0. In tale circostanza, mappare FS sul segmento DS *user level* è una soluzione che non funziona più: di fatto, questo porterebbe automaticamente a mappare FS anche sul segmento DS kernel level, per cui alla fine la system call `read()` (come qualsiasi altra system call) utilizzerà comunque una zona di memoria di livello kernel anziché di livello user.

Soluzione L'indirizzo di memoria a cui puntare per uno scambio dati user/kernel è determinato non solo dallo stato del processore, bensì anche dal software del kernel, e la determinazione è attuata per ogni singolo address space che il kernel sta gestendo. In particolare, il kernel dovrà decidere se il pointer che viene passato per identificare la zona di memoria coinvolta nello scambio dati user/kernel è lecito oppure no. Graficamente, si applica un taglio verticale all'address space, non orizzontale, come abbiamo sempre visto.

Per quanto riguarda Linux nello specifico, la soluzione è nota come “**per-thread memory limit**”: qui viene fissato un indirizzo limite dell’address space (**addr_limit**) e, se per eseguire un’operazione di *scambio dati user/kernel* viene utilizzato un buffer di memoria che cade entro **addr_limit**, allora l’operazione è permessa, altrimenti no (quindi non è solo il pointer utilizzato a dover cadere entro il limite, ma anche il valore di *pointer+size*, dove *size* è la dimensione dell’area di memoria su cui si vuole eseguire l’operazione). In realtà, se viene utilizzato un pointer lecito ma una size non lecita (per cui si usa un buffer di memoria che in parte cade entro il limite e in parte no), tipicamente l’operazione viene eseguita solo parzialmente (i.e. solo nella zona di memoria che cade entro il limite). Alcune API facevano controllo sul limite, e se non andava bene, c’erano dei problemi. Oggi, l’**addr_limit** è definito a compile-time. Correntemente, **addr_limit** in Linux è impostato all’indirizzo 0x00007fffffff000, che corrisponde al *lower half* della forma canonica dell’indirizzamento di x86. È possibile leggere il valore di **addr_limit** invocando l’API del kernel **get_fs()**, che accede a una struttura il cui campo **seg** contiene proprio il limite. **Addr_limit** può anche essere aggiornato attraverso l’API del **kernel set_fs(x)**.

Esempio di utilizzo di **addr_limit**

```
unsigned long limit;
limit = (unsigned long) get_fs().seg;
printk ("limit is %p\n", limit);
```

API kernel per lo user/data move

- **unsigned long copy_from_user (void *to, const void *from, unsigned long n)**: copia *n* byte da un buffer di memoria user space (puntato da *void from*) a un buffer di memoria kernel space (puntato da *void to*). Prima, però, verifica se l’operazione è lecita, confrontando i valori di *to*, *n* con **addr_limit**, mentre, in ultima istanza, restituisce il numero di byte che NON è stato possibile copiare (i.e. il residuo). La presenza di *n* ci fa capire che ci muoviamo in un address space, perchè consegniamo un sottooggetto compatibile con l’**addr_limit**.
- **unsigned long copy_to_user (void *to, const void *from, unsigned long n)**: copia *n* byte da un buffer di memoria kernel space (puntato da *void from*) a un buffer di memoria user space (puntato da *void to*). Anch’essa restituisce il numero di byte che NON è stato effettivamente possibile copiare.
- **void get_user (void *to, void *from)**: copia un intero da un indirizzo user space (*from*) a un indirizzo kernel space (*to*).
- **void put_user (void *from, void *to)**: copia un intero da un indirizzo kernel space (*from*) a un indirizzo user space (*to*).
- **long strncpy_from_user (char *dst, const char *src, long count)**: copia una stringa null-terminated lunga al più *count* byteda un indirizzo user space (*src*) a un indirizzo kernel space (*dst*). Restituisce poi il numero di byte che NON è stato effettivamente possibile copiare.
- **int access_ok (int type, unsigned long addr, unsigned long size)**: restituisce un valore diverso da zero se è possibile eseguire un’operazione di movimento dati cross ring su un buffer di dimensione *size* a partire dall’indirizzo *addr*; restituisce zero altrimenti. Il parametro *type* indica la tipologia del buffer (i.e. user space o kernel space).

Nota: Se devo spostare dati con syscall **read**, si chiama kernel, e ci sarà thread a ring0 che muove i dati. Noi controlliamo in base ad **addr_limit**, ma queste zone di memoria sono effettivamente utilizzabili? Infatti, un thread livello kernel potrebbe generare segmentation fault, causata da un user che vuole ad esempio lavorare su una zona non *mappattata*. Chi fa questo check? **Access_ok()**. L’**addr_limit** è dentro il TLB, ogni thread ha il proprio, e quindi unico **addr_limit** per questo thread.

Address space ha parte user e parte kernel. Un thread ha il proprio address space, non può chiedere di andare a lavorare su un altro address space. Se così fosse, ci sarebbe un’altra page table, altro thread control block ... Gli address space sono separati!

E’ possibile farlo a livello kernel, perchè ho controllo sulla page table che tocco. Tale concetto è usabile per memoria condivisa, però dovrei avere zone “uguali”, accessibile anche a livello user.

L’associazione *indirizzo fisico-virtuale* è demandata all’handler del page-fault. Quando chiamo **syscall**, scendiamo a livello kernel, possono comunque capitare page-fault.

E’ diverso dal segmentation-fault, in quanto un **page-fault** è **risolvibile**. Le operazioni possono essere non atomiche.

Service redundancy

L'attività di `check` che viene effettuata nel caso della segmentazione constrained (ma un discorso analogo vale anche per il **segmentation fixup**) deve essere eseguita solo nel momento in cui è previsto un *cross ring data move*, e non quando si effettua un accesso in memoria senza cambiare il livello di protezione. Supponiamo infatti di voler invocare una `sys_read()` (l'operazione di `read()` propria della modalità kernel) mentre siamo in esecuzione in kernel mode. Avere l'attività di check incapsulata in tale system call renderebbe impossibile l'esecuzione della lettura, poiché l'area di memoria coinvolta nella `sys_read()` si trova oltre l'`addr_limit` (com'è giusto che sia). Di conseguenza, quando stiamo lavorando solo a livello kernel, è necessario *bypassare questo check*, in quanto qui l'`addr_limit` non ha ruolo. Per farlo, si utilizza uno schema di **replicazione** (o di **ridondanza**), secondo cui alcune system call vengono di fatto replicate. Ecco qualche esempio:

- `kernel_read()` è una ridondanza per la system call `read()`:
 - Mentre `read()` internamente invoca `sys_read()` che effettua il check...
 - ... `kernel_read()` è funzionalmente identica ma bypassa il check (per cui `kernel_read()` non può essere invocata in alcun modo da un thread in esecuzione in modalità user).

Constrained supervisor mode

Sappiamo che l'operazione di `memcpy()` consiste nel copiare il contenuto di un'area di memoria *A* all'interno di un'altra area di memoria *B*. Se il puntatore relativo all'area di memoria *B* è *tampered* (i.e. è errato), si possono avere dei problemi di sicurezza, perché si può avere un accesso in memoria speculativo che è illecito. Nei sistemi più datati, non ci si poteva far nulla. In quelli più moderni, invece, si ha un supporto addizionale orientato alla sicurezza noto come **constrained supervisor mode**. Si tratta della modalità di esecuzione del kernel con dei vincoli: in operazioni come `memcpy()`, si controlla che si sta lavorando esclusivamente con indirizzi di livello kernel; se questo non è il caso, non è possibile in alcun modo eseguire l'operazione.

Constrained supervisor mode in x86

In x86 si hanno due supporti hardware:

- **SMAP (Supervisor Mode Access Prevention)**: blocca l'accesso ai dati delle pagine user mentre si è in esecuzione al CPL 0.
- **SMEP (Supervisor Mode Execution Prevention)**: blocca il fetch delle istruzioni delle pagine user mentre si è in esecuzione al CPL 0.

Esempio `copy_to_user()`

- Viene effettuato il check su `addr_limit` per verificare se l'operazione è fattibile.
- Viene determinata la quantità di dati che può essere effettivamente copiata. Lo fa `access_ok`, tempistica $O(n)$, perchè si prende Thread Control Block, e successivamente viene scandita la Tabella Management, per vedere se c'è la quantità richiesta.
- Viene disabilitato lo SMAP.
- Viene effettuata la vera e propria copia dei dati.
- Viene riabilitato lo SMAP.

Tutto ciò serve a fixare il problema nato dallo user, il quale passa parametri non corretti. Ma quante volte capita? Secondo alcune statistiche, quasi mai.

Kernel masked SEGFAULT

I check sulla quantità di dati che possono essere coinvolti nelle operazioni di movimento dati cross ring effettuati mediante l'API `access_ok()` presentano un *problema di efficienza*: fanno uso della *memory map* (`*mm`, che comprende i metadati per la gestione dell'address space del thread corrente) del thread in esecuzione e richiedono numerose istruzioni macchina aggiuntive solo per muovere i dati tra lato user e lato kernel. In particolare, l'ispezione di `mm` può avere un costo lineare (e non costante). Per ovviare a questo inconveniente, è stato introdotto il **kernel masked SEGFAULT**. Si tratta di un meccanismo per cui l'unico controllo che viene effettuato prima del movimento dati cross ring è quello sull'`addr_limit` (controllo solo il primo passo), ma poi non si ha alcun check sulla struttura dell'address space: di

conseguenza, se viene rispettato il limite, l'operazione di movimento dati viene eseguita direttamente. Perciò, si ha la possibilità di andare a toccare delle pagine di memoria user space non mappate o con dei permessi di accesso non conformi. Questo è l'unico caso in cui si può scatenare un segmentation fault al livello kernel (e viene comunque causato da un'API di livello user – come `copy_to_user()` o `put_user()` – a cui è stata passata come parametro un'area di memoria non adeguata dal punto di vista del mapping o dei permessi di accesso). In questo modo:

- Nel caso in cui va tutto bene, si ha uno speedup significativo nell'esecuzione dell'operazione di movimento dati.
- Nel caso in cui si ha un segmentation fault, viene attivato un gestore che finalizza l'operazione semplicemente restituendo il numero di byte residui.

Linux Modules p111