

## 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 deterministici, 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 sw, 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 **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 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.

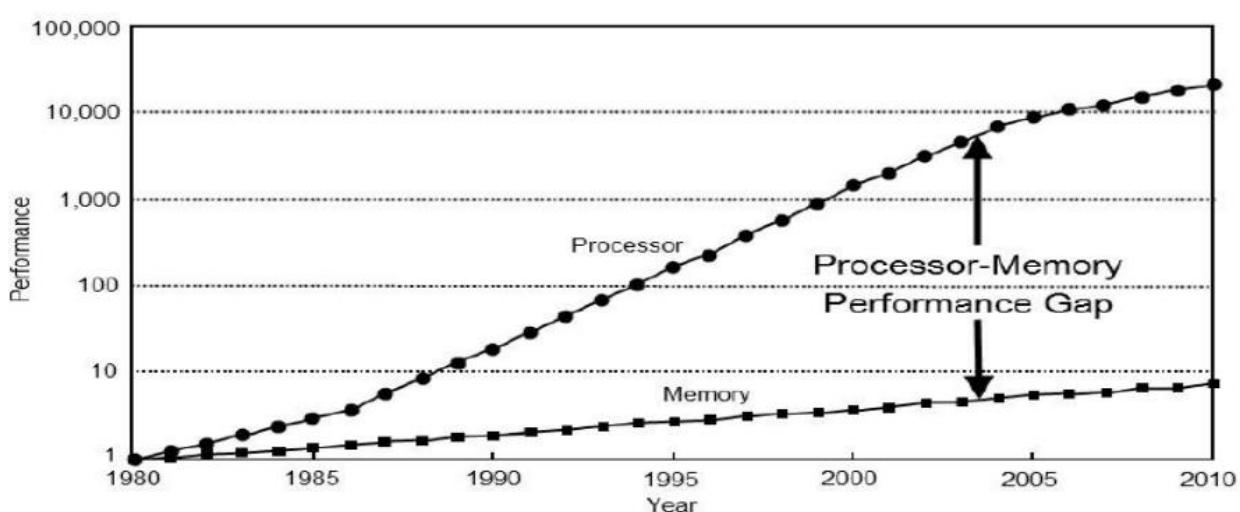
-> Possono dover richiedere a un certo punto una risorsa hardware tuttora occupata da un'altra istruzione, per cui devono rimanere in attesa.

-> 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:

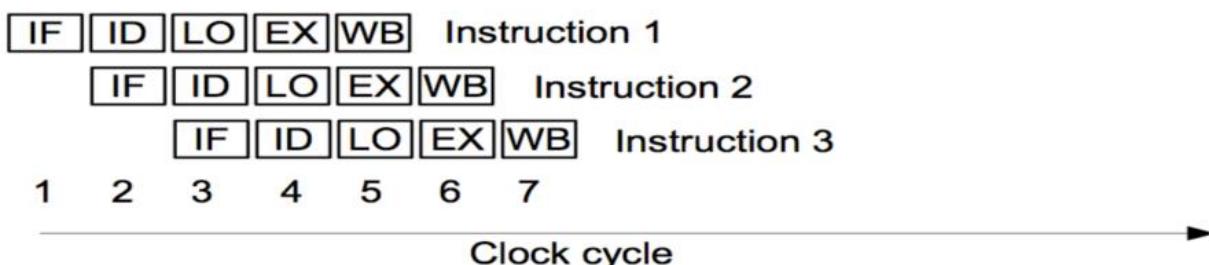
- > Non prevede una separazione temporale tra le finestre di esecuzione delle diverse istruzioni: più istruzioni possono essere in esecuzione contemporaneamente (questo è parallelismo).
- > Le istruzioni che vengono sequenzializzate dal programmatore non sono necessariamente eseguite secondo la stessa 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  $N \times L \times T$ .

-> Con la pipeline si ha un ritardo pari a  $(N+L) \times T$ .

Lo speedup è dunque pari a  $(NxL)/(N+L)$ , 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. ( ad esempio dovrei avere hardware infinito, visto che ogni stage è un pezzo hw).

Ad esempio:

- 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  $i_1$ .

-> **Rischidulazione 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.

-> **Rischidulazione 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 ho effetti reali sull'ISA finché non devo "mostrare" il suo output. Se in pipeline su un thread ho A->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, 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.

### Pipelining vs sviluppo software

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. Esempi di ISA sono ADD, COMPARE, JUMP.

Esempio: 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 poi 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. Qui 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 il **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. (Perchè le istruzioni successive devono ancora essere fetchate ed eseguite, quindi come potrei superarle?)

Ciò che viene dopo a 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.

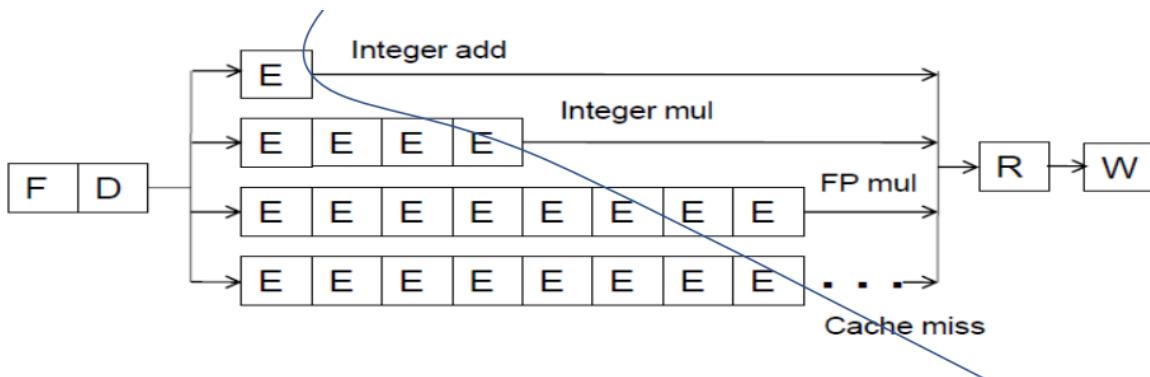
### Pipeline superscalare

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*.

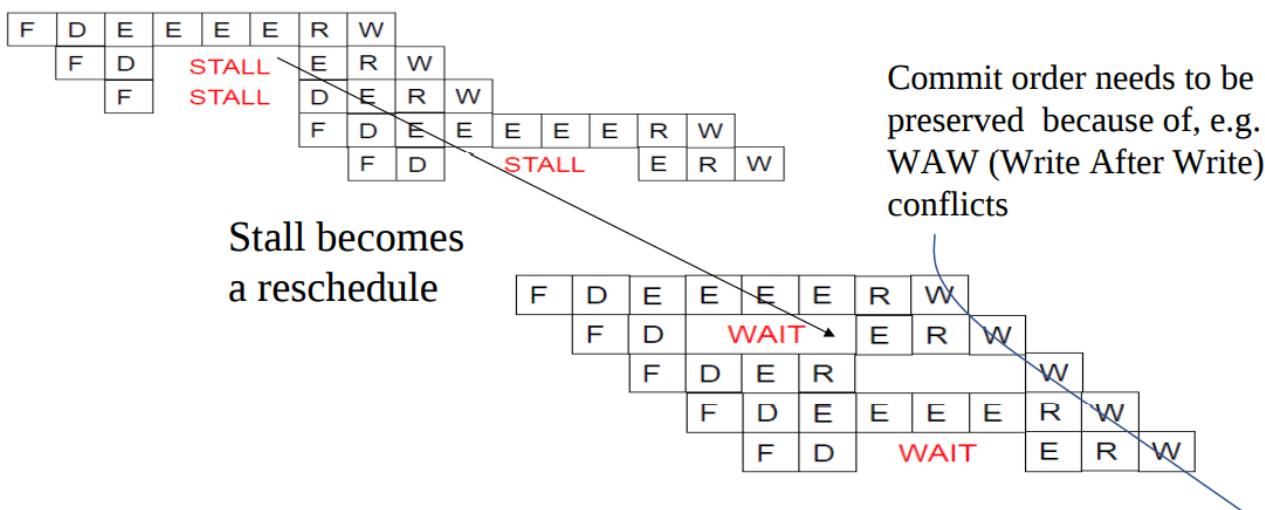
-> **Processare le istruzioni indipendenti** (sia sui dati che sulle risorse hardware) **il prima possibile**, dove un'istruzione indipendente sulle risorse hardware è un'istruzione che non ha bisogno di attendere che un qualche componente si liberi per processare un certo stage x.

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 parallele a X canali", che è come avere una autostrada a X corsie. Con Out-Of-Order Pipeline si evitano "grosse latenze", poichè se "libero una corsia perchè sono veloce" anche gli altri andranno più veloce (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 (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 precede 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 necessariamente dire committato).

#### Dipendenze 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<sub>k</sub> 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.

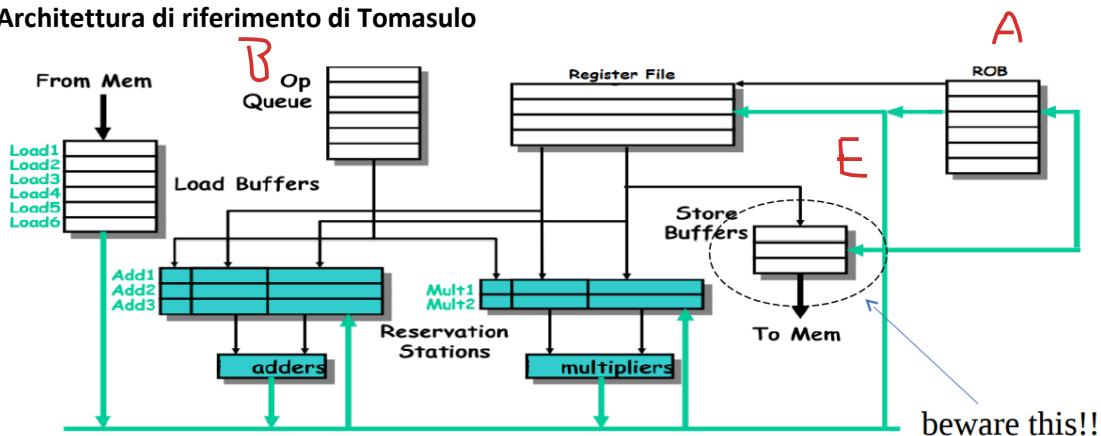
Posso superare solo dopo essere entrato nella pipeline (nelle condizioni discusse nelle pagine precedenti), ma non posso superare prima di entrare in pipeline.

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 sempre 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.

c) Si fa uso di una **cache** per leggere in modo più efficiente i dati provenienti dalla memoria.

d) 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 il problema 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.

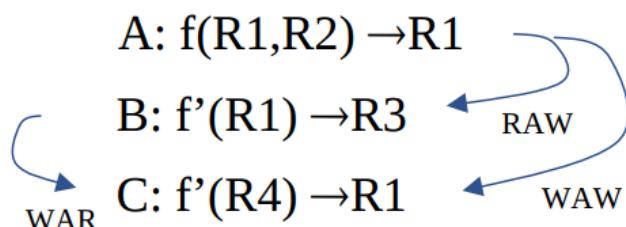
### Esempi di schemi di esecuzione

#### Esempio 1:

Supponiamo di avere tre istruzioni A, B e C tra cui B e C hanno una stessa latenza  $\delta$ , mentre A ha una latenza  $\delta' > \delta$ . Assumiamo inoltre che:

- A esegua l'operazione  $f(R_1, R_2)$  e riporti l'output sul registro  $R_1$ . (in particolare, scrive su un alias di  $R_1$ ).
- B esegua l'operazione  $f'(R_1)$  e riporti l'output sul registro  $R_3$ . (legge da STESSO alias di  $R_1$ )
- C esegua l'operazione  $f'(R_4)$  e riporti l'output sul registro  $R_1$ . (scrive su UN ALTRO alias di  $R_1$ ).

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
add r4,r7,r3
add r3, r2, r7

**after**

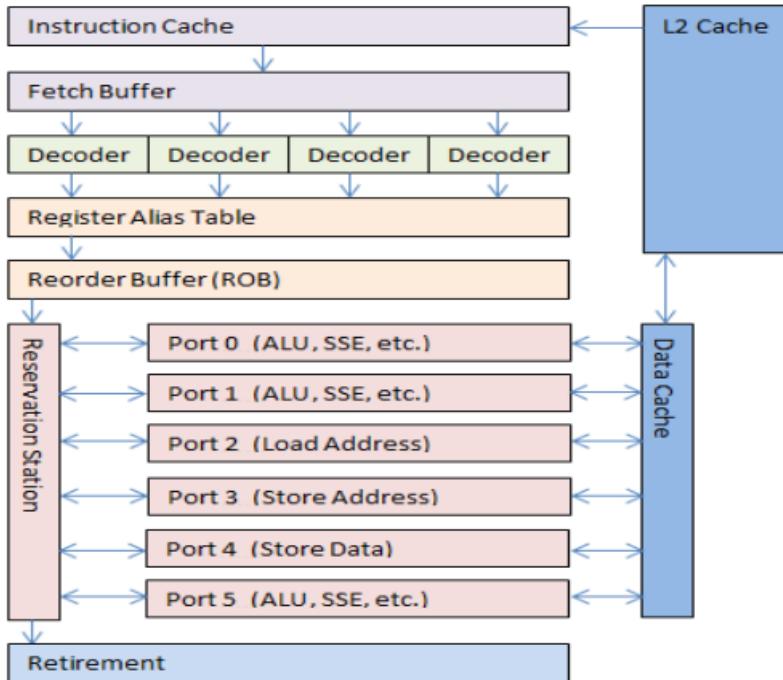
add rob6,r3,4
add rob7,r7,rob6
add rob8,r2,r7

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**.

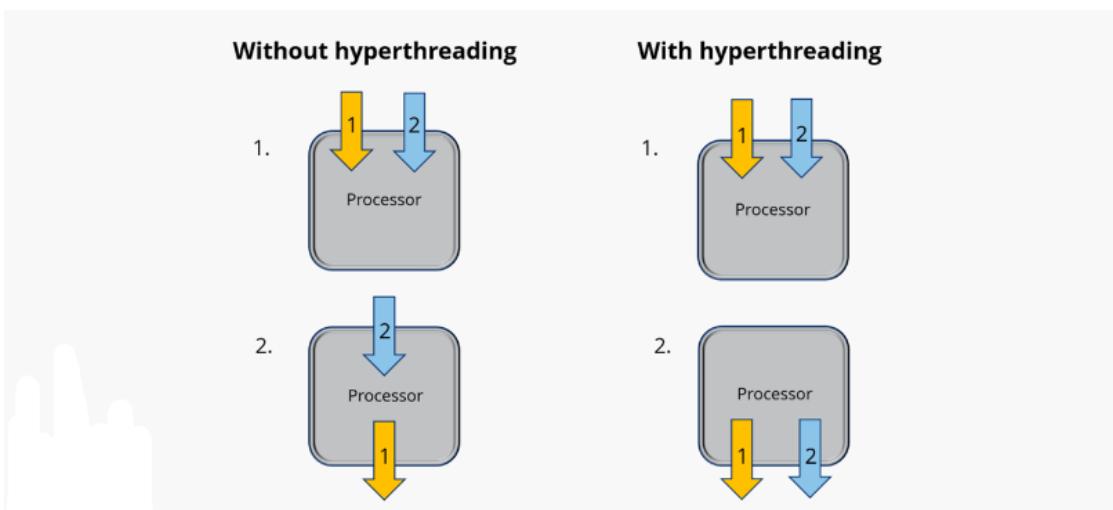
### Organizzazione architetturale dei processori x86 OOO



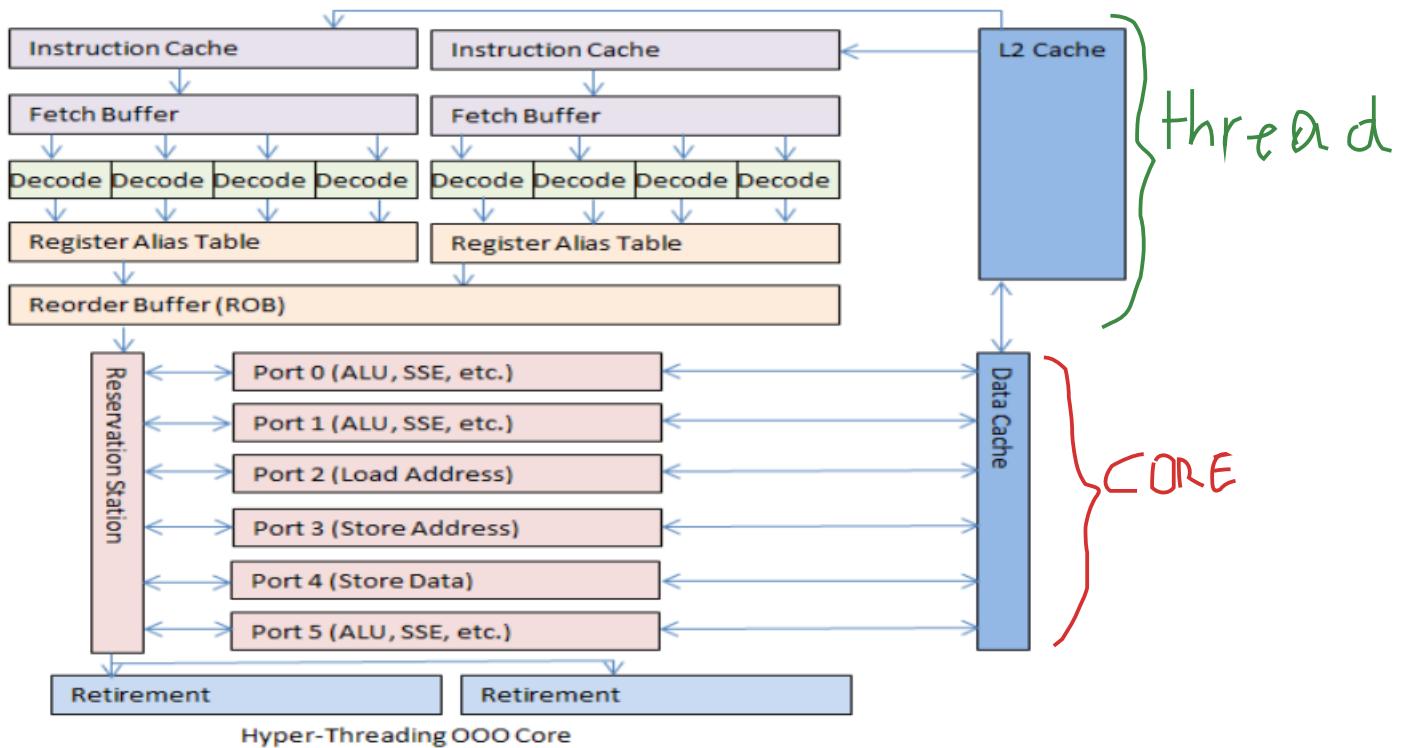
### Processori hyper-threaded

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 spurate. 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 (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 miglioro, perchè 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. 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 (esempio: 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 (avvengono durante esecuzione di un programma, 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  $e_A$ . **Istruzione offending se vuole usare qualcosa che non è usabile. Non genera una trap, perchè 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 **clflush**. Dopodiché, mediante una **mov**, si preleva un **particolare byte x** (che ad esempio può rappresentare un carattere) **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: tale aggiornamento della cache rappresenta proprio il side effect lasciato sullo stato micro-architettonico 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 e lo salvo in un registro (non potrei), accedo a array[registro contenente 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.

```

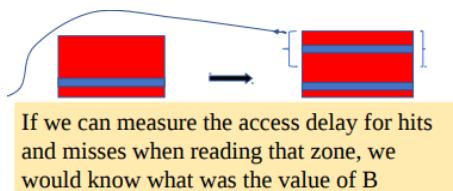
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]

```

This is B

Use B as the index of a page

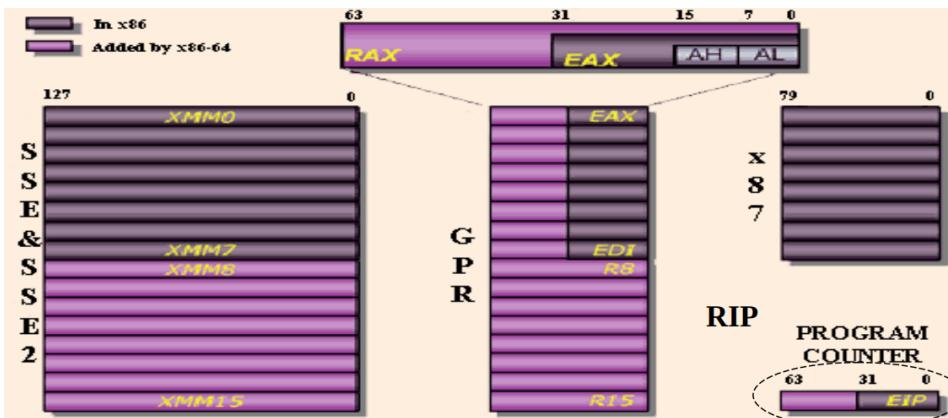
B becomes the displacement  
of a given page in an array



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.

### 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 (..., ..., ...)*

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

Displacement → movl foo, %eax  
Base → movl (%eax), %ebx  
Base + displacement → movl foo(%eax), %ebx  
 movl 1(%eax), %ebx

(Index \* scale, movl (,%eax,4), %ebx  
Base + (index \* scale) + displacement movl foo(%ecx,%eax,4), %ebx

-> **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).

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            ; shifta rax di 0xc = 12, cioè 12 bit a 0 (verso sinistra shifta la parte meno significativa)
                        ; lo spiazzamento dista 2^12 = 4096 byte dal predecessore SEMPRE.
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, perchè 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

1) **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.

2) **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.

3) **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).

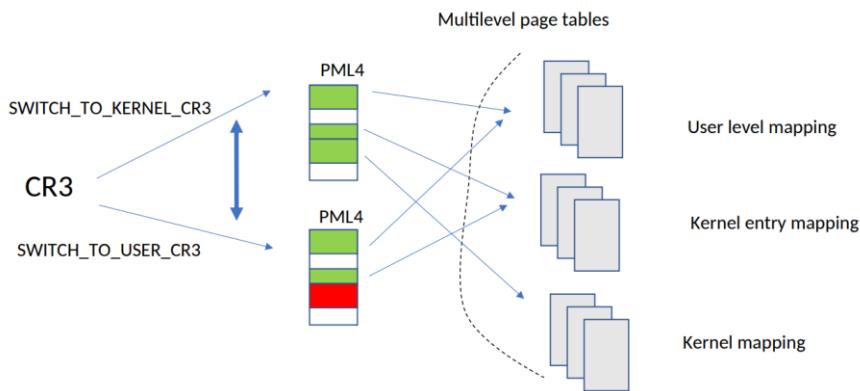
Dall'altro lato, quando un processo gira in modalità kernel, utilizza un'altra page table  $PT_K$  che contiene il mapping degli indirizzi di memoria mancanti in  $PT_U$ . Di fatto, gli indirizzi del kernel space il cui mapping è presente nella page table  $PT_U$  costituiscono la cosiddetta **entry zone** del kernel, che contiene quasi esclusivamente il codice che permette di effettuare lo switch delle page table ( $PT_U \rightarrow PT_K$ ).

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 selettor CR3, ad ogni cambio, azzera la TLB e genera cache miss. Però è un'operazione automatica ed abbastanza semplice. (Alla fine PML4 kernel è PML4 user + altri indirizzi kernel, quindi potremmo pensare che basti aggiungere gli indirizzi kernel alla PML4 user, però automaticamente si pulisce tutto per semplicità).



## 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. E' una zona di cache hardware contenente [indirizzo istruzione, target da usare se jumo], e l'indirizzo istruzione può anche essere il target stesso. E' 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, saprò quella corretta in fase di esecuzione, e quindi è register dependent, perchè 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

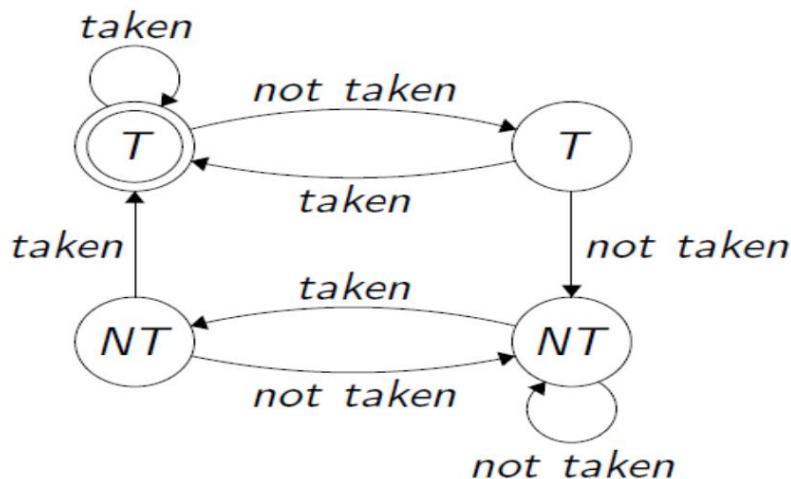
#### Predittori con bit di stato:

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. Caso semplice: *ciclo for, anche se prima o poi uscirò, sbagliando la predizione*.

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 intero itero, 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, perchè 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 blocco di codice:

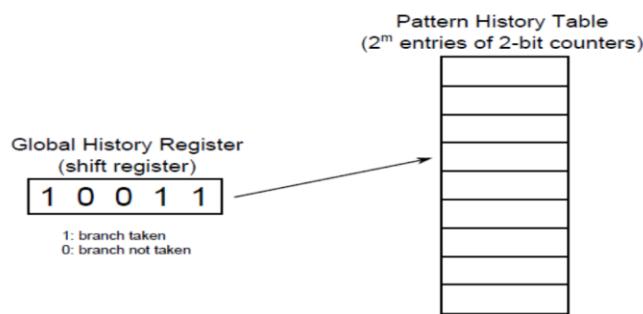
```

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

Ci piacerebbe avere un predittore che, 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 predizione 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 predizione 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 associo una entry nel Patter Hystori 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 = Entry 10, ho sbagliato una volta, resto su 0.

Entry 11: non capita perchè non prendo mai 11 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.

(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, quindi non ho perdite di prestazioni.

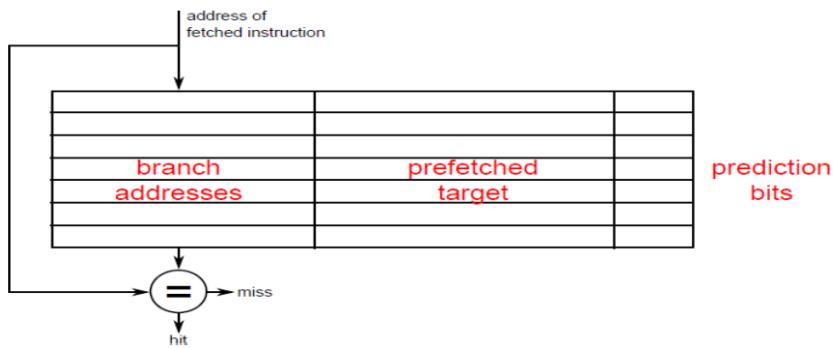
#### **Predittori per i salti indiretti**

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?

### Attacchi 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: (salti condinali)

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.

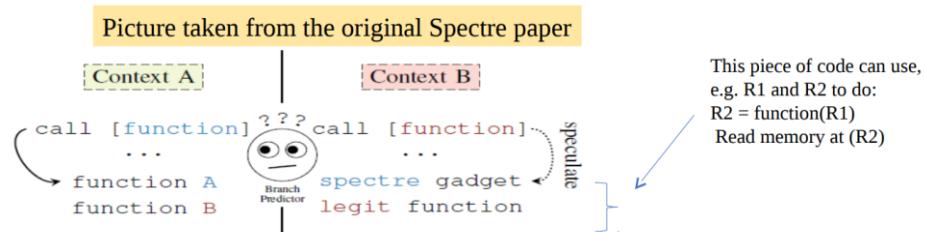
Aver "patchato" Meltdown non implica risolvere anche Spectre. La soluzione di Meltdown risiedeva nel fatto che tale attacco creasse una trap poichè si passava da user a kernel, e c'era il cambio del puntatore alla page table. Qui sono sempre in user mode.

Con più thread passo alla versione v2!

### Spectre 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 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
retpline_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:

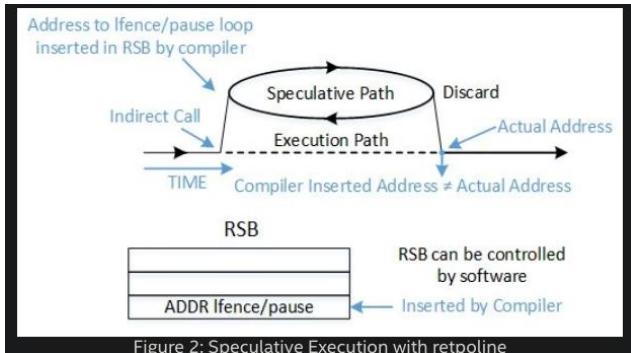
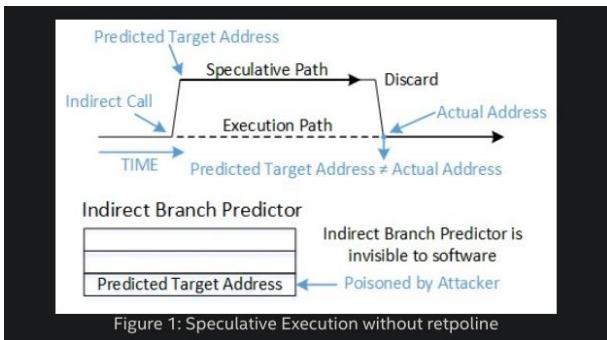
-> Carica l'indirizzo destinazione del salto (*target\_address*) sullo **stack**. (ora è in cima allo stack)

-> Effettua una call di *retpline\_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 *retpline\_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 *retpline\_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 *retpline\_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. È 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.

Entrambe sono operazioni software, spesso si usa la syscall *prcr()* per lavorare col Processor Control. Se lavoriamo con *exec()* (programma chiama un altro programma) funziona uguale, perchè facenti parte dello stesso programma iniziale.

## Introduzione alla Sanitizzazione

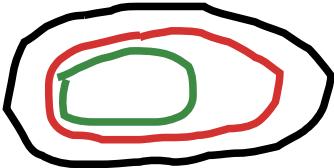
Per Spectre V1 abbiamo visto la possibilità di riusare le patch di Meltdown. V1 mi permetterebbe di leggere dati kernel? Sì, perchè la predizione a livello kernel potrebbe essere basata su predizioni livello user. A livello kernel abbiamo una tabella, che chiamiamo **driver**, in cui per ogni indice 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.

## Sanitizzazione:

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 a una entry della tabella o l'unica locazione di memoria ammissibile e innocua al di fuori della tabella).

### Spiegazione terra-terra:



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 lì 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 sì 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?

Supponiamo di avere un ciclo for estremamente semplice come quello riportato di seguito:

```
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:

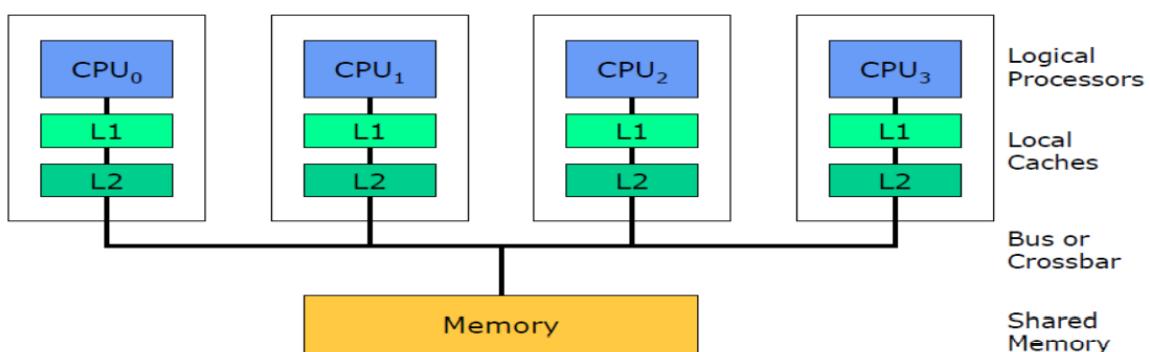
- 1) Aumentare la frequenza del clock.
- 2) Incrementare i componenti hardware a supporto del processore.

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  $VxVxF$ , 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 architettoniche descritte di seguito.

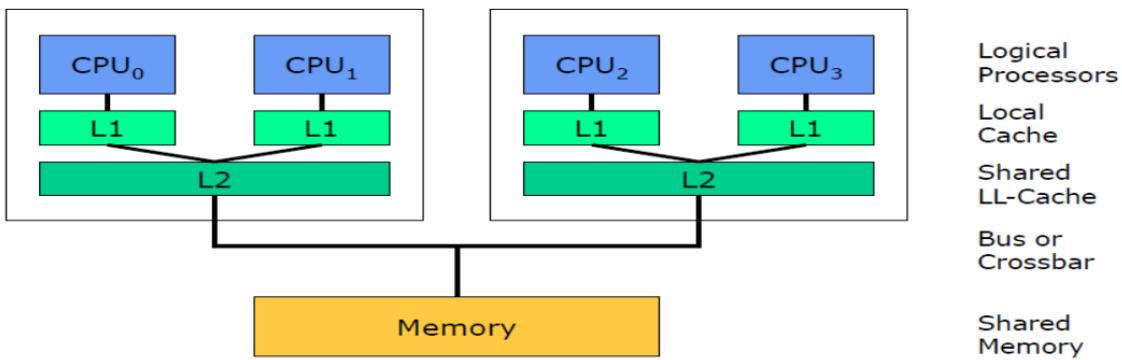
### Symmetric multiprocessors:

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



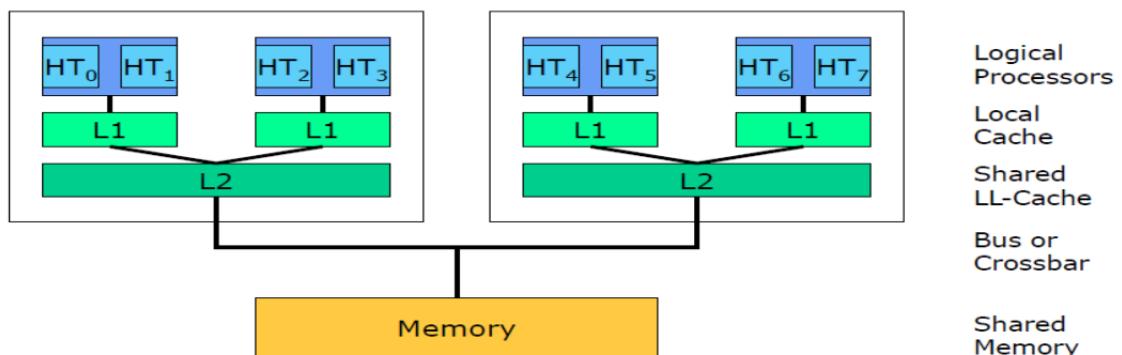
### Chip Multi Processor (CMP) o Multicore:

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 L1,L2...).**



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

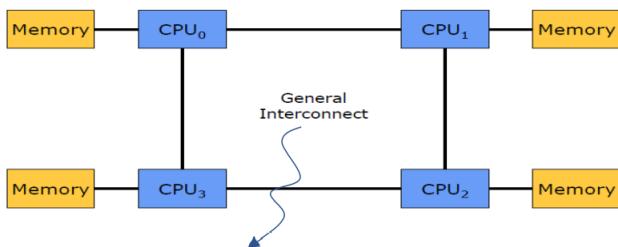
### Non Uniform Memory Access (NUMA):

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<sub>i</sub> 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<sub>i</sub>, 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  $50(\text{hit})+200(\text{miss})$  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  $200(\text{hit})+300(\text{miss})$ . Tempi totalmente diversi, che possono creare **problemi di coerenza nella cache**.



This may have different shapes  
depending on chipsets

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 (CC)

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

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-thorough).
- 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 complete 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.

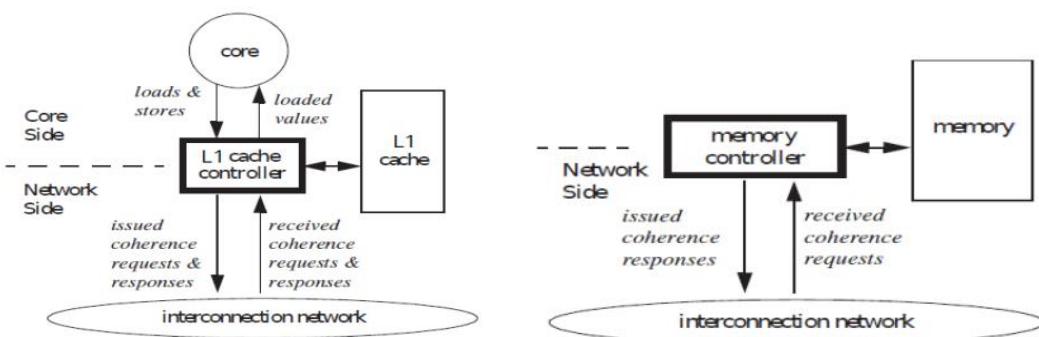
Esistono due famiglie principali di protocolli CC:

-> **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 agganciate 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 L1 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 CPU0 con cache L1 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 di se una copia di un blocco si trova nello stato **modified**, nello stato **shared** o nello stato **invalid**. In particolare:

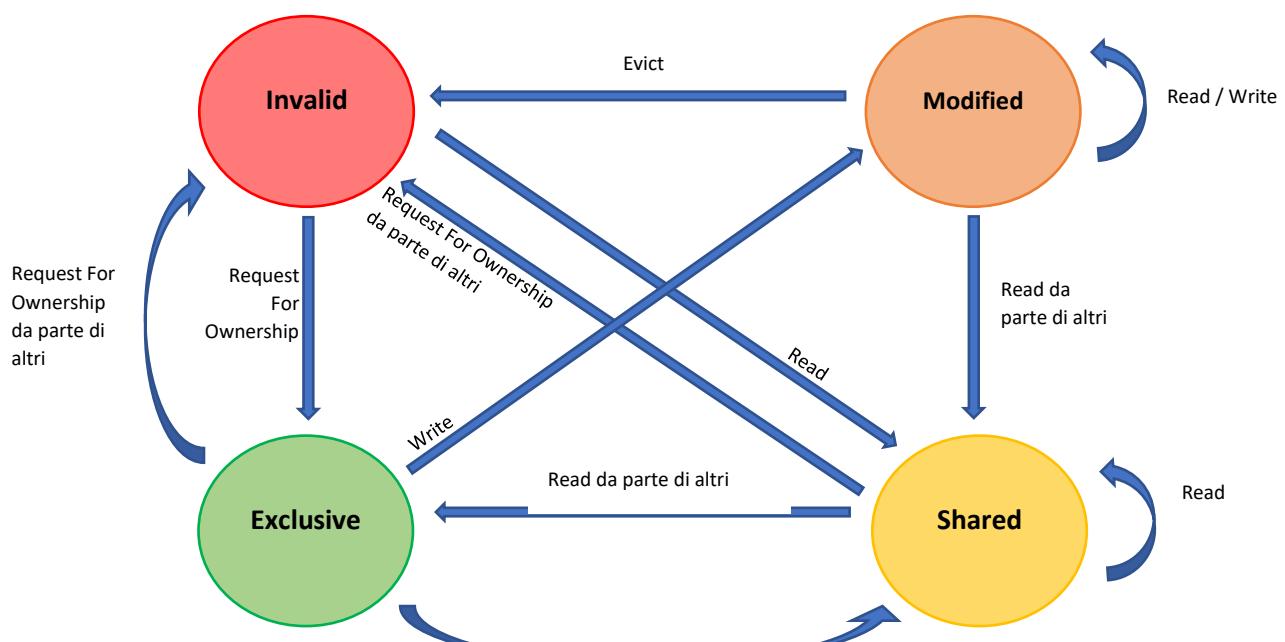
- > Si trova nello stato *modified* se è appena stata sovrascritta.
- > Si trova nello stato *shared* se esistono dei CPU-core che stanno leggendo altre copie del medesimo blocco (per cui esistono più copie valide di tale blocco).
- > Si trova nello stato *invalid* se un'altra copia del medesimo blocco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

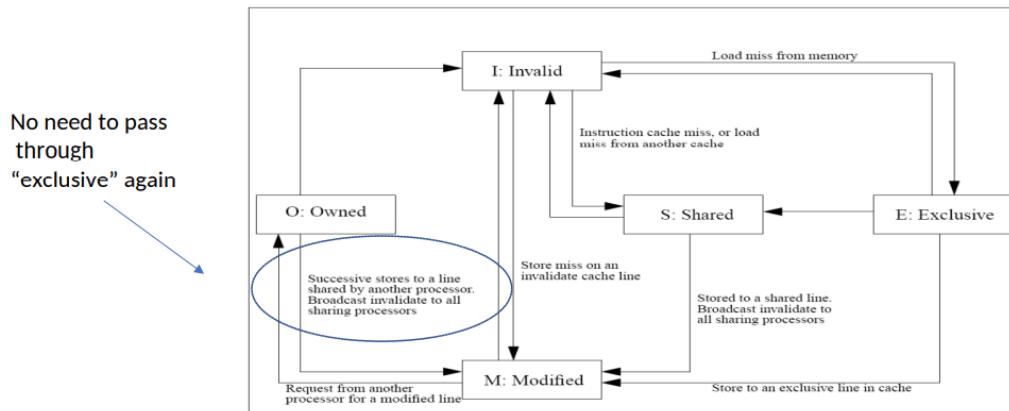


Notiamo che ogni volta che si esce dallo stato modified, si effettua il write back in memoria delle nuove informazioni che sono state scritte. Se volessi solamente leggere? Chi ha l'informazione si trova in exclusive e poi passa a shared. Potrei introdurre un quinto stato per evitare queste due transizioni. 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.

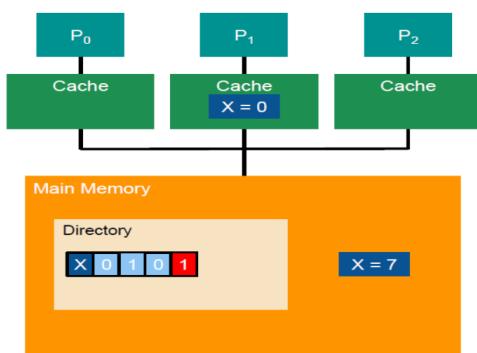
#### Protocolli directory based:

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<sub>0</sub> 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<sub>0</sub> vuole leggere il blocco X, ha un cache miss e il dirty bit è pari a 1.

X viene richiamato dal processore P<sub>j</sub> tale che presence[j]=1. Dopodiché 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<sub>0</sub> vuole scrivere sul blocco X, ha un cache miss e il dirty bit è pari a 0.

P<sub>0</sub> invia un messaggio di invalidazione non a tutti gli altri processori, bensì solo a quelli col bit presence[j] pari a 1. Dopodiché tali presence[j] vengono settati a 0, e presence[0] e il dirty bit vengono impostati a 1.

-> **Caso 4:** P<sub>0</sub> vuole scrivere sul blocco X, ha un cache miss e il dirty bit è pari a 1.

X viene richiamato dal processore P<sub>j</sub> tale che presence[j]=1. Dopodiché 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.

### API per allocare memoria allineata – livello software

Cache impattata quando creo struct ad esempio. Devo applicare politica *Loosely Related Fields*, cioè se un'operazione tocca due zone disgiunte, queste devono essere in due linee diverse di cache. Se fossero su stessa linea di cache, creerei un *False Cache Sharing*. Se le zone solo correlate, allora sono *Strong Related* e uso un'unica linea di cache.

**Esempio:** Spawno due thread, uno legge e l'altro scrive su un vettore di interi (4 byte/entry), operazioni conflittuali. Writer aggiorna  $v[0]$ , Reader legge in una zona  $v[TARGET]$ . Se  $TARGET \geq 16$ , vuol dire che mi sto spaziando, dall'entry iniziale, di  $16 * 4 = 64$  byte, che è *una linea di cache*. Stiamo quindi dicendo che i due thread operano su due linee di cache diverse. Questa run è molto più veloce di un'altra con  $TARGET < 16$ , perchè i thread vanno in conflitto sulla singola linea di cache.

NB: Tutte le operazioni stanno nello **Store buffer**, non in cache, solo con **nfence** vanno in cache. (Nel codice c'è infatti **nfence**).

E' possibile avere una run veloce anche con  $TARGET < 16$ ?

Sì. Se la macchina è 2core/4thread (vedi slide *Baseline OOO, con due motori sotto*), posso, tramite *0x5 ./a.out*, usare due hyperthread su stesso core (5 = 0101, uso lo 0-esimo e il secondo). Perchè lavora meglio? Perchè stanno **lavorando su stessa replica**, e non devono cambiare stato .

(MESI è tra cache "diverse", non nella singola).

Essi accedono alla stessa memoria cache L1 e L2 del core e condividono la stessa copia dei dati.

Senza **affinità**, i thread vengono mossi tra core diversi.

```
POSIX_MEMALIGN(3)           Linux Programmer's Manual          POSIX_MEMALIGN(3)

NAME
    top
    posix_memalign, aligned_alloc, memalign, valloc, pvalloc - allocate
    aligned memory

SYNOPSIS
    top
    #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);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
```

Se ALIGNMENT = 64, sto allineando esattamente a una linea di cache. Parto sicuramente da lì, 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.
<b>Description</b>		
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.		
<b>Operation</b>		
<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>		
<b>Flags affected</b>		
None.		

## CLFLUSH

### Flush Cache Line

Opcode	Mnemonic	Description
0F AE /7	CLFLUSH m8	Flushes cache line containing m8.
<b>Description</b>		
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, 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).		
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, perché 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 inline:

- **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/l** = 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 al).

### Esempio di utilizzo dell'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"
        "rdstc \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<sub>1</sub>: gli attacchi Meltdown e Spectre utilizzano proprio questo costrutto qui.

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

NB<sub>3</sub>: 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:

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

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

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<sub>A</sub> e il thread  $T_B$  dell'attaccante giri sul CPU-core<sub>B</sub>. 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<sub>A</sub> 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 alla 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, e se 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 la visione di come (e in quale ordine) le istruzioni di accesso alla memoria vengono viste dal **processore che le esegue (program order)** dalla visione di 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 private per un 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-core<sub>1</sub> debba eseguire le operazioni  $a_1, b_1$  (dove  $a_1$  viene prima di  $b_1$  nel program order), e supponiamo che CPU-core<sub>2</sub> 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-core<sub>1</sub>.

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 talune 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 ha 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 segua la disciplina FIFO*. (in x86 si, in altre architetture no). Nel caso in cui non la 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 in ogni operazione flushando la pipeline.**

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**, *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 riportato una funzione C contenente un ASM inline che implementa un trylock mediante cmpxchg (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 primo caso, posso ottenere degli abort, e quindi ritento (alg. Linearizzabile). Nel secondo caso, non ho mai abort, ed è consistente.
- > **Read Copy Update (RCU)**. 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.

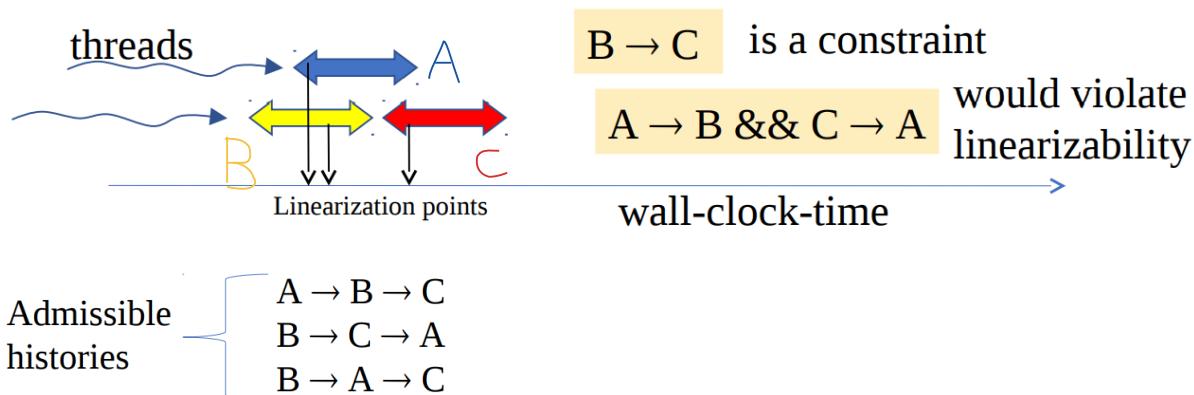
Cosa usare è influenzato dal con cosa sto lavorando!

### 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<sub>1</sub>, T<sub>2</sub>, di cui T<sub>1</sub> invoca la funzione A mentre T<sub>2</sub> 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<sub>2</sub>, quando esegue l'operazione C, riesca a vedere tutto ciò che è stato fatto in B? (sopra è quasi stato dato per scontato!).

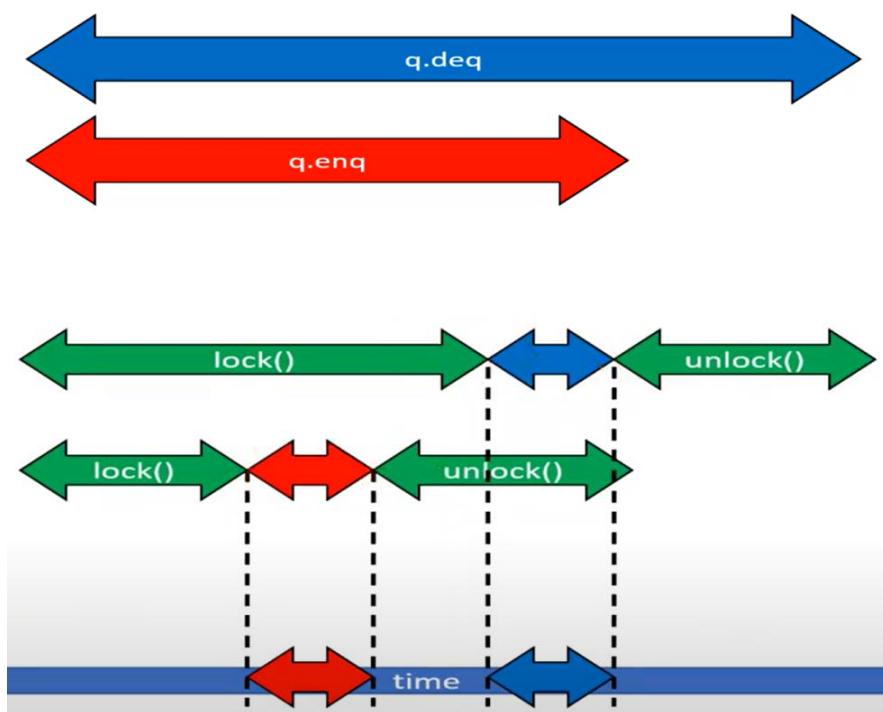
In un caso di *Sequential Consistency* sì (anche se noi non operiamo in questo contesto), altrimenti non è scontato. Ad esempio, il thread potrebbe essere spostato su una CPU diversa, e prima di andarci lo Store Buffer nella vecchia CPU non è stato flushato. Quindi nella nuova CPU non posso recuperare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

### 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.

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.

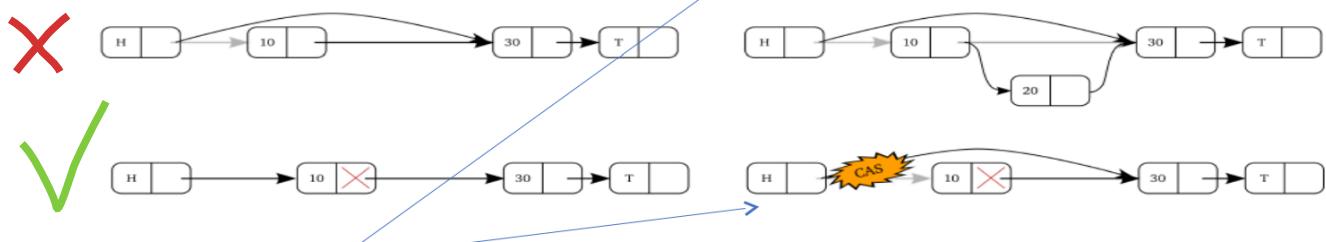
Un grosso vantaggio della sincronizzazione lock-free è che non dà problemi nel caso in cui un thread dovesse crashare. Infatti, qui il comportamento di ciascun thread è indipendente da quello che succede a tutti gli altri thread; nel caso in cui si utilizzi un lock, invece, il crash del thread che detiene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

Di seguito viene riportato un caso d'uso d'esempio per la sincronizzazione lock-free:

- 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 marcare il nodo 10 come da eliminare, per poi dealloccarlo 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 aree sensibili! Farebbe **traversing**.  
(il thread si muove solo sulla lista, non conosce altro!).

In lungo: Ad esempio, 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, un buffer è adibito per contenere l'ultimo eventuale valore che non è stato ancora acceduto da alcun lettore e 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, mentre 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 decrements il counter se ha finito di leggere.

D'altra parte, quando lo scrittore deve effettuare una scrittura, esegue un'`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 iniziate == #lettura terminate), 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.

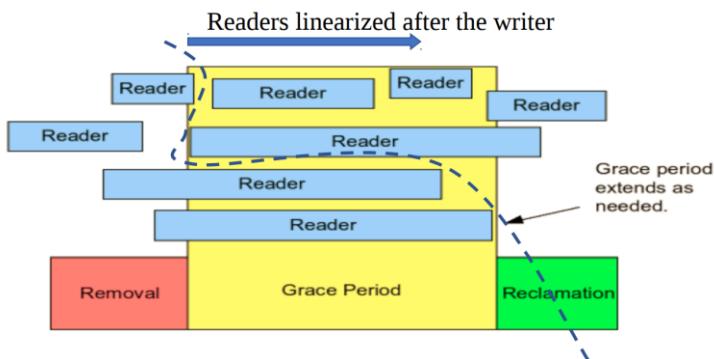
### Read Copy Update (RCU)

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 un numero arbitrario di reader per volta: i reader, per eseguire le letture, non devono attendersi né a vicenda né col writer, mentre 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 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 deallocazione (e il

buffer / nodo viene marcato come “da deallocare”) e l’istante in cui il buffer / nodo viene effettivamente deallocato, 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. **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 fintantochè qualcuno la usa.** Quindi se reader concorrenti, su *cpu diverse*, non vedono che un “pezzo” è stato sganciato, devo aspettarli. Questo perchè 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<sub>i</sub>, significa che sulla CPU<sub>i</sub> 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.

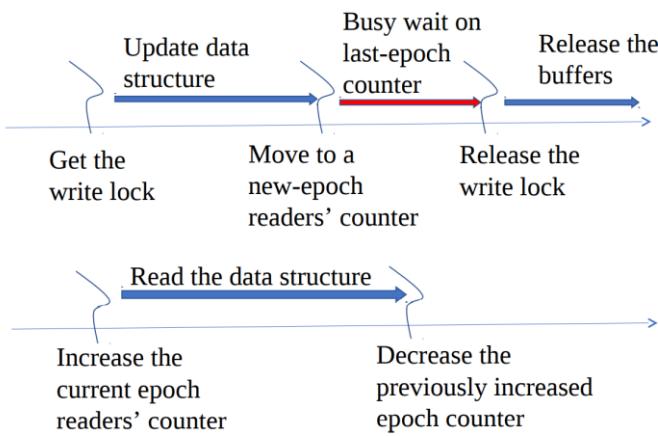
Il grosso svantaggio di questo approccio sta nell’impossibilità di interrompere l’esecuzione dei lettori: se a

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. *la 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 \(MD\)](#).

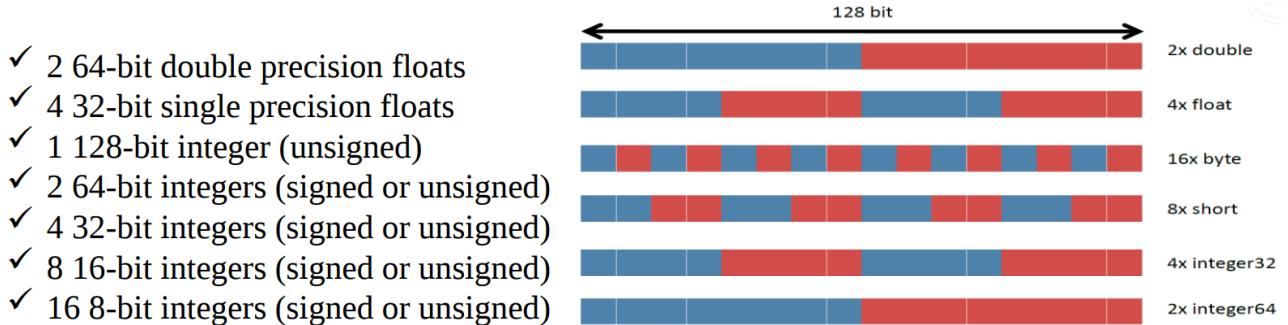
- **MISD (Multiple Instruction Single Data)**: è una forma di calcolo più rara, anche se qualcuno sostiene che un processore speculativo sia MISD; di fatto, i processori speculativi introducono la possibilità di eseguire più istruzioni in parallelo su uno stesso dato, sfruttando anche molteplici istanze 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** (raccolta di dati dalla memoria verso un registro vettoriale) e l'**hardware data scatter** (riversamento di dati dal registro vettoriale verso la memoria). [Tramite flag -O posso implementare calcoli vettoriali senza API](#), esiste 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*, *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 XMM0, XMM1,..., XMM7. Tali registri sono in grado di ospitare:



In SSE2, invece, si hanno 16 registri vettoriali a 128 bit, che sono YMM0, YMM1,..., YMM15.

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 vs 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_epu64(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>

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

### Indirizzamento della memoria – come vedo la memoria?

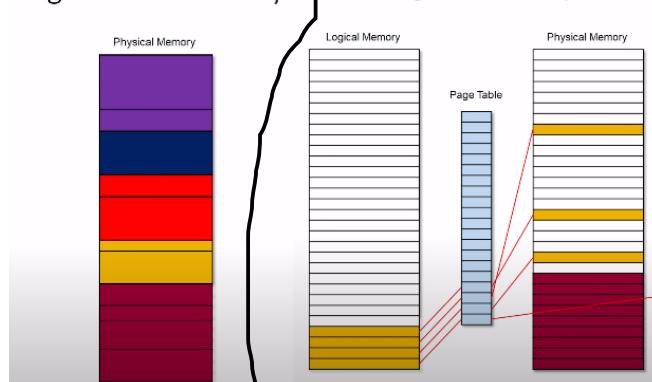
La memoria può essere indirizzata secondo più modalità possibili. Una di queste è data dall'**indirizzamento lineare**, dove si utilizzano appunto gli indirizzi lineari che sono caratterizzati esclusivamente da un offset, indipendentemente da se stiamo operando in memoria fisica (i.e. ram) o in memoria 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 combinate 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  $\Delta$  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  $\Delta$ .

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 che 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).

Segmented Memory      Paged Memory



#### 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 (quindi spreco spazio). Permette al programmatore di organizzare le aree di lavoro.

Con la **sola paginazione** (è nata prima) ragioniamo su pagine (più piccole dei segmenti), e possono essere non contigui 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 programmati 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 e 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 che 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 segment](#))

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. Però 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:  $\text{Segment} * 16 + \text{Offset}$ .
  - > 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 ( $2^{16} \times 2^4$ ), quindi abbiamo  $2^{16} \times 2^4$  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 sono indirizzi logici, altrimenti gli indirizzi lineari corrispondono a quelli fisici. Cioè noi abbiamo un indirizzo lineare (segmento + offset, dato da indirizzamento lineare e segmentazione). Ora, questo indirizzo è esattamente l'indirizzo di memoria fisica? Se non ho paginazione, deve per forza esserlo, perché 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:  $\text{address} = \text{TABLE}[\text{segment}].\text{base} + \text{offset}$ .
- > 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:  
 $\text{address} = \text{TABLE}[\text{segment}].\text{base} + \text{offset}$
- > 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 ne esiste più di una 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)**.

- La 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).

- La 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 thread1 usa LDT1, e thread2 usa LDT2. 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 vs paginazione: a cosa serve la segmentazione, se ho la 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 voglio 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 *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? Be', 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 sulla 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 descresce. 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

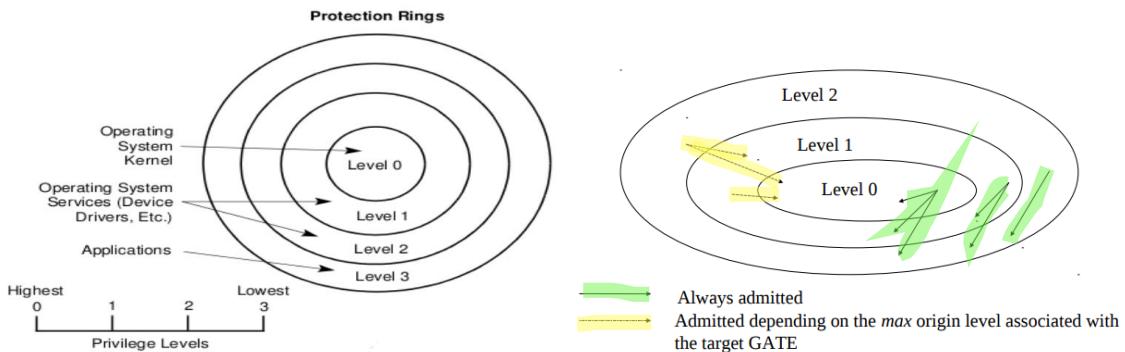
protezione diverso da h, è richiesto un cross-segment jump. In particolare, è sempre ammesso saltare dal livello di privilegio h a un livello di privilegio h+i (con i>0) poiché porta ad abbassare il grado di protezione. Per migliorare i propri privilegi e, quindi, per passare da un livello di protezione 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  $\text{seg.id}$  è l'identificatore del segmento e  $\text{offset}$  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.

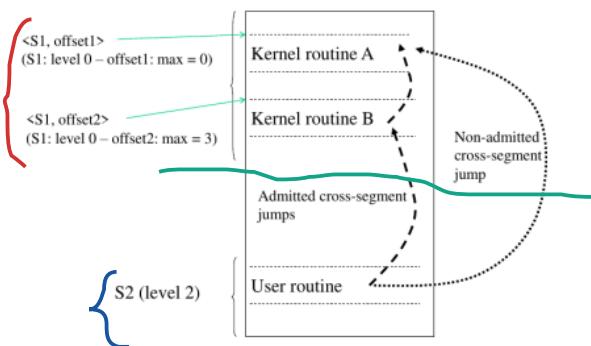
Il meccanismo appena descritto può essere schematizzato col **ring model**. Nei sistemi x86 moderni si hanno esattamente 4 livelli di protezione, che sono schematizzati nella seguente figura:



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 S1. La user routine è invece in un altro segmento S2.

Il GATE kernel routine A ha livello 0 (del segmento), un certo offset rispetto S1, 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.

User entra in Kernel routine B mediante cross-segmento (perchè passo da S2 a S1) e poi da Kernel Routine B a Kernel Routine A mediante intra-segmento (perchè parto da S1 e rimango in S1).

## 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. *Es: 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, perché 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
Base 24:31   Flags		Access Byte	

-> **Base**: indica l'indirizzo base del segmento puntato dalla entry. La GDT ci porta alla base del segmento.

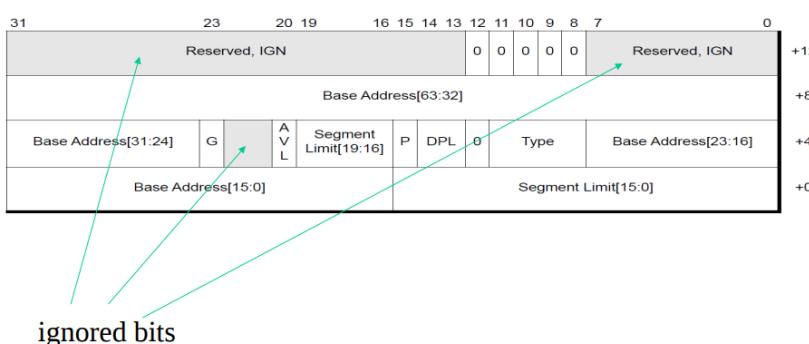
-> **Limit**: indica la dimensione del segmento.

-> **Flags**: tra i flag rientra il **granularity bit**; se vale 0, il campo limit viene espresso in byte, mentre 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.

Per esprimere un segmento utilizzo *base 0:15, base 16:23, e base 24:31*.

-> **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: (64 bit, in realtà 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.
<b>Description</b>		
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.		
<b>Operation</b>		
<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>		
<b>IA-32 Architecture Compatibility</b>		
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.

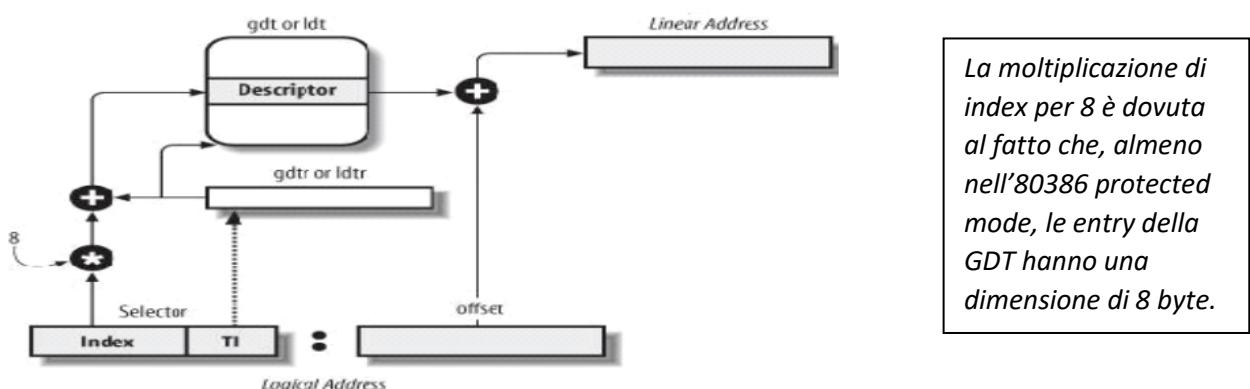
Per memorizzare le informazioni del registro sgdt, necessito di una struttura di tipo packed.

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, a 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 sovrapposizione. Questa scelta è dovuta al fatto che in tal modo è possibile programmare software *molto portatile*, 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

Can we read/write/execute?

Is the segment present?

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

La tabella di cui sopra vale in particolar modo per i sistemi x86 a 32 bit.

Per quanto concerne l'x86-64, il fatto che taluni 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, 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<sub>A</sub>, mentre quando è in esecuzione un thread B, quella entry della GDT può contenere come indirizzo base TLS<sub>B</sub>, cosicché thread differenti, a parità di istruzione macchina, possano accedere a punti della memoria diversi (i.e. a sezioni TLS diverse), ove richiesto.

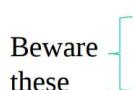
Quindi per FS/GS, associate al contesto del thread nella CPU, vengono aggiornate a seconda del thread in esecuzione, un thread può vedere in GS una certa base, un secondo thread vede in GS un'altra base.

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	Linux's GDT	Segment Selectors
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APM BIOS 32-bit code	0xb8
TLS #3	0x43	APM BIOS 16-bit code	0xc0
reserved		APM BIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 (__KERNEL_CS)	not used	
kernel data	0x68 (__KERNEL_DS)	not used	
user code	0x73 (__USER_CS)	not used	
user data	0x7b (__USER_DS)	double fault TSS	0xf8

- 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 perché 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*). Essi vengono aggiornati 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 lo **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 che specific purpose), e li salvo in un’area puntata da TSS. Mi permette quindi, di salvare un’immagine e caricarne un’altra, in un’operazione simile al **context switch**. Oggi viene usato solo per info 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 partono da base 0.** Solo per SS,DS,ES,CS si parte da 0.

31	15	0
I/O Map Base Address	LDT Segment Selector	T 100
	GS	96
	FS	92
	DS	88
	SS	84
	CS	80
	ES	76
	EDI	72
	ESI	68
	EBP	64
	ESP	60
	EBX	56
	EDX	52
	ECX	48
	EAX	44
	EFLAGS	40
	EIP	36
	CR3 (PDBR)	32
	SS2	28
	ESP2	24
	SS1	20
	ESP1	16
	SS0	12
	ESP0	8
		4
		0
 Reserved bits. Set to 0.		

-> 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 perché 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, perchè 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.

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)	



Ciascun TSS viene **aggiornato** ogni volta si ha context switch, ovvero ogni 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.

#### LTR (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-core<sub>1</sub> abbia un segmento GS con base B e che un thread  $t_2$  in esecuzione sul CPU-core<sub>2</sub> 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 variabile x di tipo intero 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, cioè 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; in tal modo, ogni 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.

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 aggiornare 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). Il parametro addr è invece un puntatore a **unsigned long** nel caso in cui si vuole eseguire una **get** dell'indirizzo base del segmento GS o FS del thread corrente: stavolta, 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" la TLS, in particolare prendendo l'indirizzo TLS di un thread, salvarlo in una variabile (get\_FS) e poi far puntare l'altro thread a questo address (set\_FS), se poi questo ultimo thread scrive, lo farò su dove lavora il primo thread.

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 o in protected mode. 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 encapsulate 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 TS 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.

-> 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 ha 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.

Possiamo chiamare una INTERRUPT in modo SINCRONO tramite la macro INT (interrupt), che usa i GATE, al posto di JMP. Essa permette, tramite i GATE, di passare in kernel mode.

A tal proposito, 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 ammessa è INT 0x80).

- > È la entry 0x2e della IDT per i sistemi Windows (per cui l'istruzione ammessa 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.

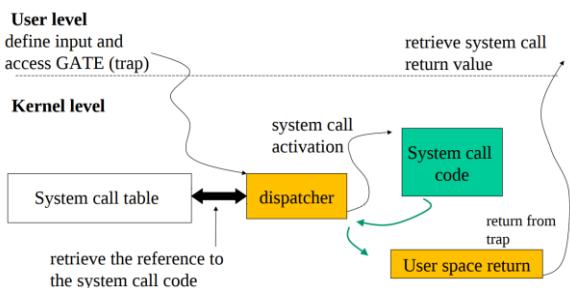
Il blocco di codice del kernel raggiungibile a partire dall'istruzione di INT è detto **dispatcher**, che è un modulo software in grado di triggerare l'attivazione delle system call. Il dispatcher identifica quale *entry* usare.

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) e **STI** (set interrupt bit). La prima elimina la possibilità per gli interrupt di interrompere l'esecuzione del dispatcher, mentre la seconda la ripristina. Questo però 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 vuole 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 quanto riguarda 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 quanto invece riguarda UNISTD\_64, 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 valore di ritorno e nome della funzione di ritorno.

- Il valore numerico (*l'indice*), associato al nome della system call `__NR_##name` (e.g. nel caso della `fork()` è `__NR_fork`), 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.

Ciò viene realizzato tramite la penultima riga del blocco di codice di cui sopra.

- La terzultima riga del blocco di codice, invece, 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)`.

Approfondiremo subito tale aspetto.

```

/* user-visible error numbers are in the range -1 - -124:
 see <asm-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 maggiore o uguale a zero; altrimenti il valore di ritorno è compreso tra -1 e -124. 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). E' "one-shot" perchè ritorna subito qualcosa!

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, di cui i primi due sono in comune con `_syscall0`, type1 è il tipo di dato che la system call accetta in input e 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 `_syscall0`, 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 __syscall6(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); \
}

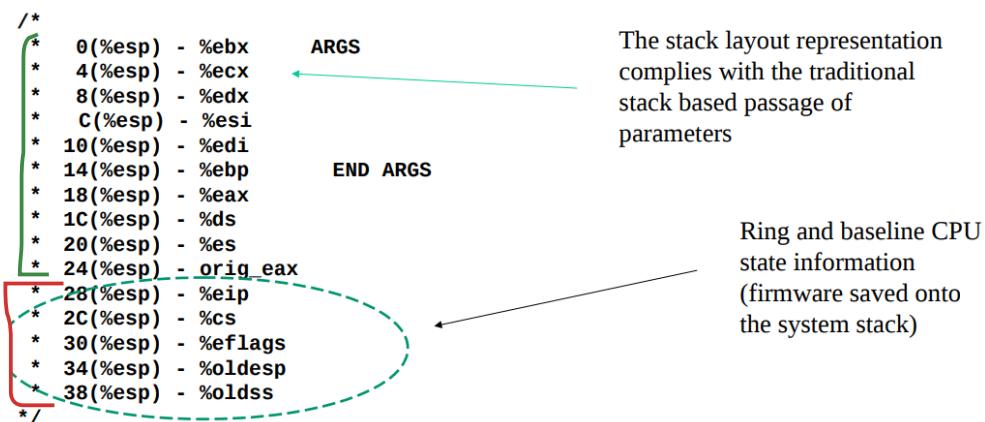
```

- 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](#)).

#### Convenzione di UNISTD\_32 per l'invocazione di 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) e 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 (perché deve solo posizionare *sullo stack tutto ciò che va da ebx a orig\_eax nell'ordine giusto*, i registri restanti vengono salvati sullo stack dal firmware) e che la system call sappia 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\_64 per l'invocazione di 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.

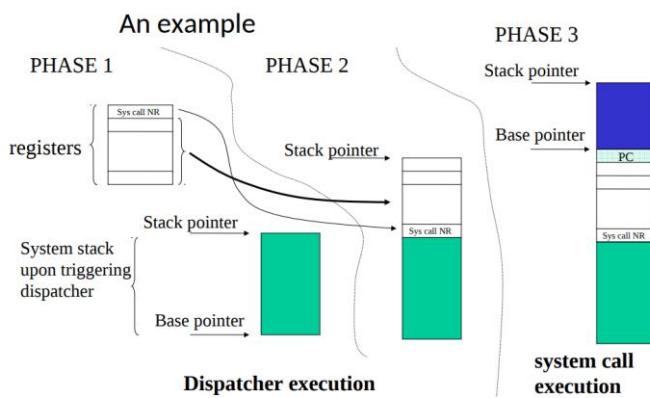
-> I registri rdi, rsi, rdx, r10, r8, r9 contengono, nell'ordine, i parametri da passare alla system call. Il quarto argomento (arg3) può essere inserito nel registro rcx anziché in r10 solo nel caso in cui stiamo lavorando con del codice puramente C (senza ricorrere a costrutti o operazioni machine 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 smezzano 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. Dopodiché 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.

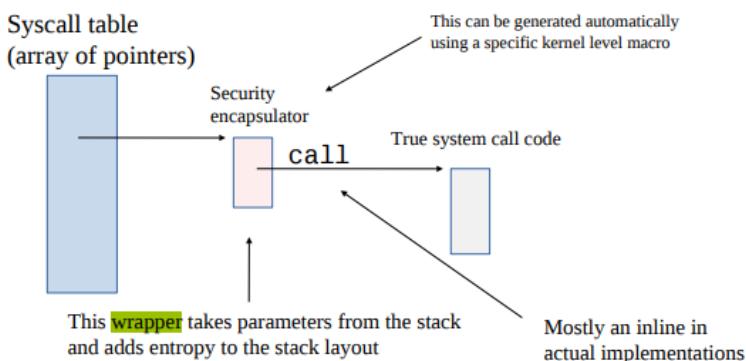


- La porzione colorata in verde è relativa ai registri che vengono salvati nello stack per opera del firmware.
- La porzione bianca è relativa ai registri che vengono salvati nello stack da parte del software (qui in particolare si hanno il codice numerico della system call e i parametri da passare alla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

```
#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);
```

Di seguito è mostrato un altro esempio in cui si fa un override della system call fork() in UNISTD\_32.

```
#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 !!
}
```

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.

### 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: i **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:

- > 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:

- > 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:

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:

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 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/syscall). 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 a tale servizio. Ciò che ne consegue è una trasformazione in blocchi assembly che chiamano *syscall 64bit fast* oppure *int 0x80 32 bit slow*.

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

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);
}
```

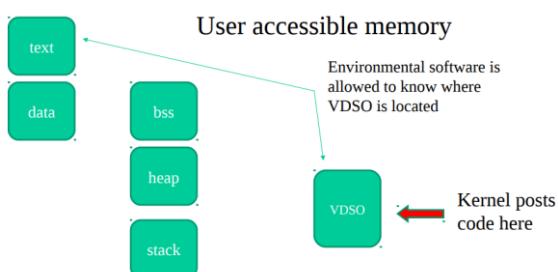
### Virtual Dynamic Shared Object (VDSO)

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 *vds = (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, 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).

## Implementazione del dispatcher attivabile con int 0x80 (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
        cmpb $0,need_resched(%ebx)
        jne reschedule
        cmpb $0,sigpending(%ebx)
        jne signal_return
    restore_all:
        RESTORE_ALL

```

Beware this!!!

Manipulating  
the CPU  
snapshot in  
the stack

-> L'istruzione `cmpl $(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` contiene 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.

-> Prestiamo 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**.

## Implementazione del dispatcher attivabile con syscall (kernel 2.4) –UNISTD64

```

ENTRY(system_call)
    swapgs
    movq %rsp,PDAREF(pda_olrsp)
    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 $PT_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, è un'altra cosa, ma si usa sempre il dispatcher.

-> La seconda e la terza istruzione 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    %rsp, PER_CPU_VAR(rsp_scratch)
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(rsp_scratch) /* pt_regs->sp */
pushq    %r11                /* pt_regs->flags */
pushq    $USER_CS            /* pt_regs->cs */
pushq    %rcx                /* pt_regs->ip */
pushq    %rax                /* pt_regs->orig_ax */

GLOBAL(entry_SYSCALL_64_after_hwframe)
PUSH_AND_CLEAR_REGS rax=$ENOSYS
TRACE_IRQS_OFF
/* IRQs are off. */
moveq    $user_trdi
pushq    %r11
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,
 * 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

```
144 ENTRY(entry_SYSCALL_64)
145     UNWIND_HINT_EMPTY
146     /*
147     * Interrupts are off on entry.
148     * We do not frame this tiny irq-off block with TRACE_IRQS_OFF/ON,
149     * it is too small to ever cause noticeable irq latency.
150     */
151
152     swapgs
153     /* tss.sp2 is scratch space. */
154     movn  %xsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
155     SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
156     movq  PER_CPU_VAR(cpu.current_top_of_stack), %rsp
157
158     /* Construct struct pt_regs on stack */
159     pushq  $_USER_DS          /* pt_regs->ss */
160     pushq  PER_CPU_VAR(cpu_tss_rw + TSS_sp2)      /* pt_regs->sp */
161     pushq  %r11               /* pt_regs->flags */
162     pushq  $_USER_CS          /* pt_regs->cs */
163     pushq  %rcx               /* pt_regs->ip */
164
165 GLOBAL(entry_SYSCALL_64_after_hwframe)
166     pushq  %rax              /* pt_regs->orig_ax */
167
168 PUSH_AND_CLEAR_REGS rax=$-ENOSYS
169
170 TRACE_IRQS_OFF
171
172     /* IRQs are off. */
173     movq  %rax, %rdi
174     movq  %rsp, %rsi
175     call   do_syscall_64        /* returns with IRQs disabled */
176
177 TRACE_IRQS_IRETQ           /* we're about to change IF */
178
179     /* Try to use SYSRET instead of IRET if we're returning to
180     * a completely clean 64-bit userspace context. If we're not,
```

Switch to the kernel view of memory

-> Poco dopo l'istruzione `swapgs` 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.

### 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

Avviene secondo i seguenti step: Qui configuro cose che posso usare su startup Linux, non facciamo mai lo startup di queste cose! Sto solo creando contenuto file system.

- 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, col parametro *allyesconfig* (per cui vengono inclusi tutti i sottosistemi software, portando verosimilmente a dei conflitti) oppure col parametro *allnoconfig* (per cui non viene incluso alcun sottosistema software, ottenendo verosimilmente un kernel con non offre abbastanza servizi).
- 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) **mkintrd -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).

Possiamo effettuare compilazioni di tipo:

- *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*.
- 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).

Esempio: *grep HZ /boot/config-6.5.7-200.fc38.x86\_64* , 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 anatomie (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/kallsyms**, 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/kallsyms**. 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.

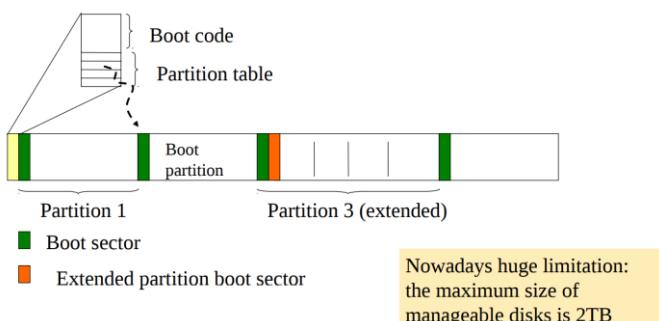
Comunque sia, il primo bootsector contiene il **master boot record (MBR)**, che mantiene del codice eseguibile e una tabella (la **partition table**) con *quattro entry*; ciascuna di queste entry identifica una **partizione** del dispositivo, e il *primo settore di ogni partizione può operare come un 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)	<b>MBR Bootstrap</b> (flat binary executable code)
0x1b4	10	Optional "unique" disk ID <sup>1</sup>
0x1be	64	<b>MBR Partition Table</b> , 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 ( $=9 \times 10^{21}$  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. E' 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 di BIOS/UEFI durante il boot del 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. E' altamente configurabile, posso dirgli di gestire un tot di memoria rispetto a quella totale che ho, ad esempio. All'inizio NON C'E' per-cpu-memory, però con [smp\\_processor\\_id\(\)](#), che richiama cpuid, quindi 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 multi-core:

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 stady state. In questa soluzione, la prima cosa che fa ciascuna CPU è invocare l'istruzione CPUID: se risulta essere la CPU<sub>0</sub>, 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

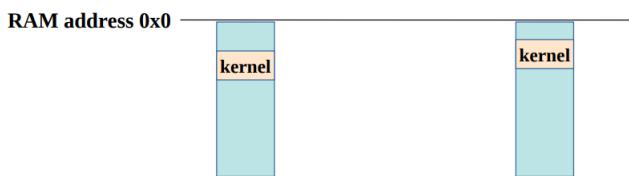
Allo startup, eseguiamo attività inizialmente esterne allo stack kernel, poi faccio setup stato processore e memoria. Il software usa indirizzi logici, e devo settare il supporto agli indirizzi fisici, senza considerare la randomizzazione. Tutto ciò che succede ci porta allo *steady-state*. Inoltre, 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

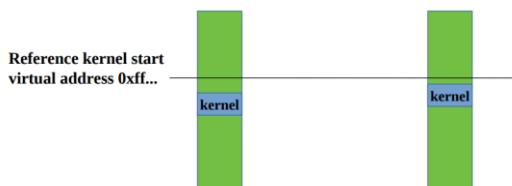
La regione della memoria fisica in cui viene caricata l'effettiva immagine del kernel è determinata dal **bootloader** sfruttando la randomizzazione.

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 parto 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:

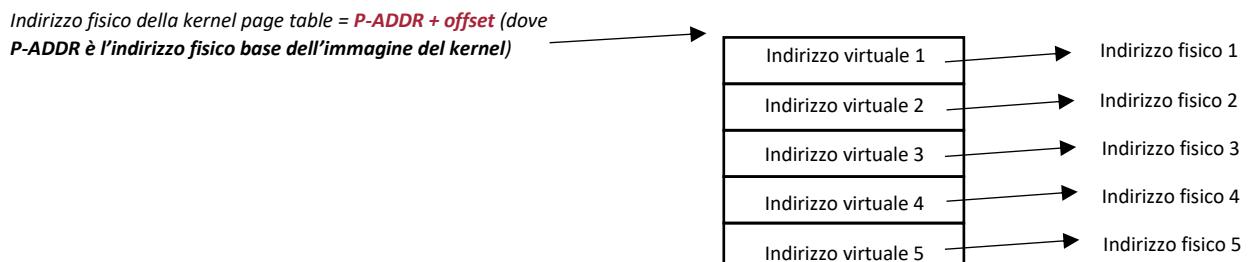


Dunque, nel momento in cui si definisce la page table, è necessario porre attenzione anche nella definizione degli indirizzi logici (da associare poi a quelli fisici).

Possiamo avere PT diverse, e quindi collocazioni diverse.

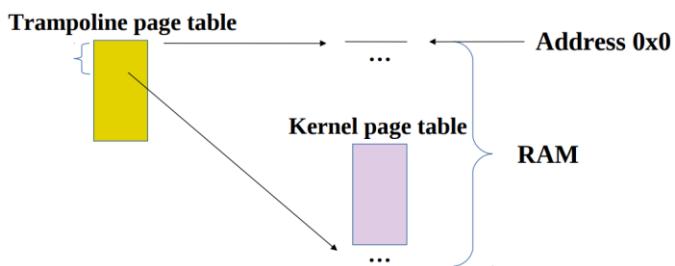
Come possiamo definire una page table (raggiungibile dal kernel) che associa correttamente tutti gli indirizzi logici dell'immagine del kernel ai corrispettivi indirizzi fisici in memoria tenendo conto che si ha la randomizzazione del posizionamento del kernel sia a livello logico che a livello fisico?

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 stady state), inclusa l'identity page table; questa page table preliminare è detta **trampoline page table**.



Quello viola è Identity Page Table kernel.

#### 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  
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, perché 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.

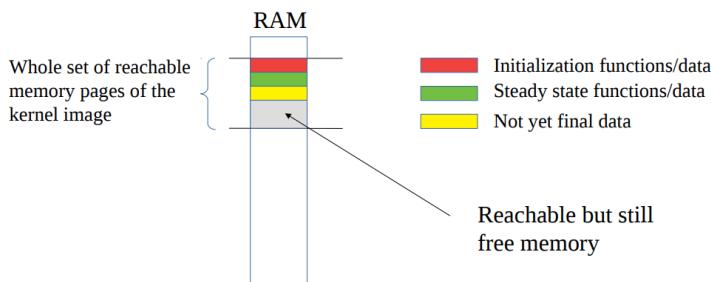
Se non vado in stallo, non leggo le info per via della patch, ma ho annullato la randomizzazione perché ho

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.

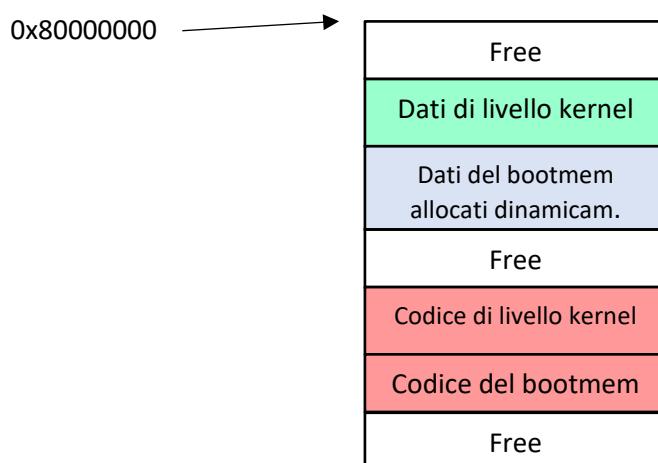
La fase di linking del kernel si occupa di posizionare queste funzioni su specifiche pagine logiche.

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 casciano 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 **startup**, 0xc0800000 → pagina. Le immagini iniziali sono compatte allo stesso modo.

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).

Qui dentro possono esserci delle invocazioni alle funzioni `_init` presenti all'interno della sezione "Codice del bootmem".

## 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

Obiettivi del setup della kernel page table: Ci dice come vediamo lo spazio!

- 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.

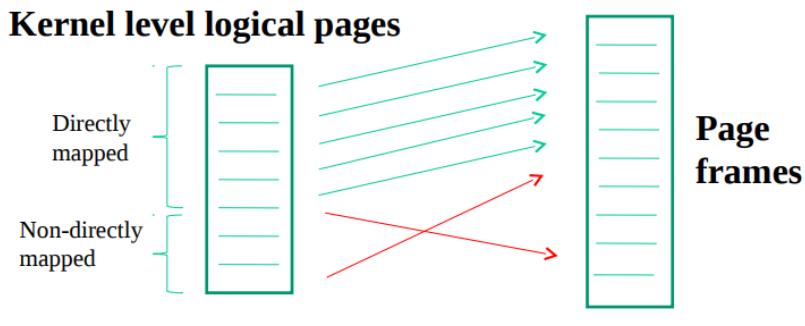
### Pagine di memoria directly mapped:

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:

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

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



A destra memoria fisica.

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 *pagine user*; comprende la sezione della memoria fisica successiva al megabyte 896. Cose user 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->1 sarebbe più facile!

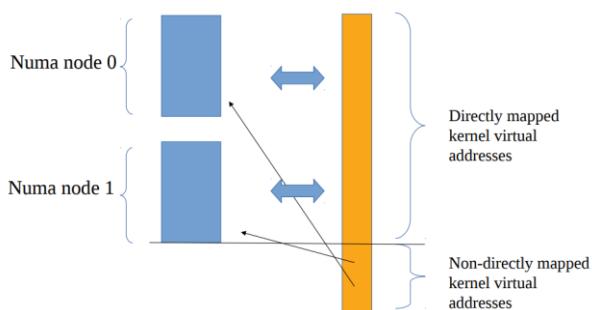
Qui ci mettiamo cose che vogliamo mascherare, come le stack areas dei thread!

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 del kernel e struttura della page table in i386

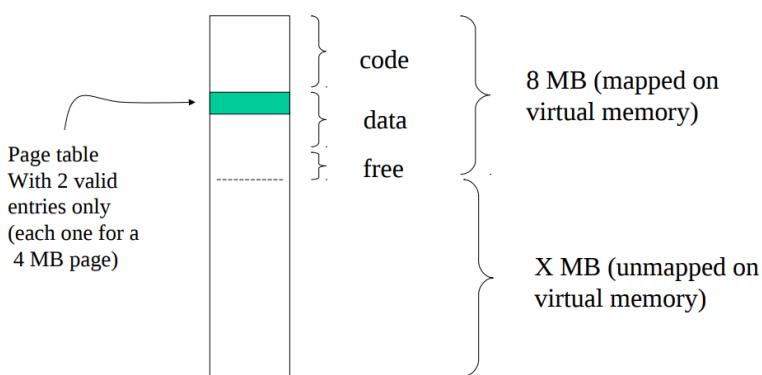
### Organizzazione del kernel in i386:

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 ( $2^{32}$ ) 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.

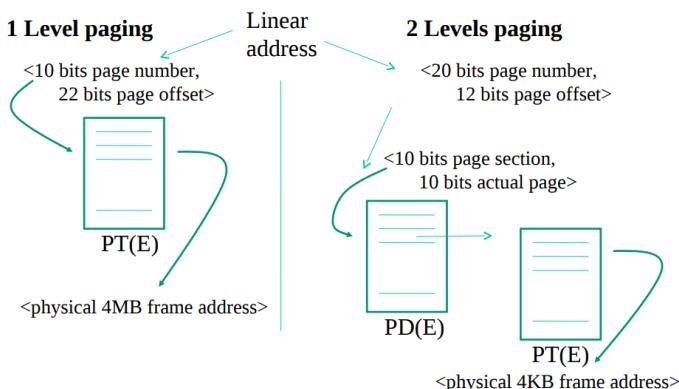
### Struttura della page table in i386:

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:

- 1) **Paginazione a 1 livello:** è esattamente quella che 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), mentre 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.

2) **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.*

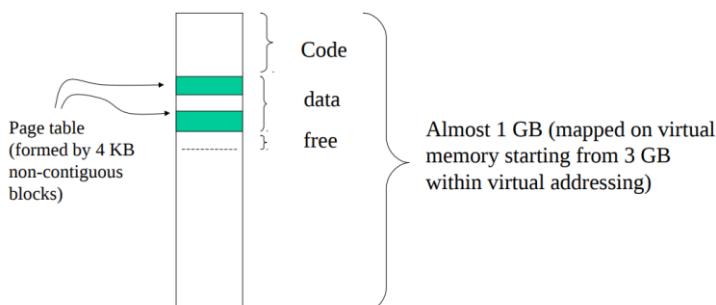


La page table è necessariamente collocata in memoria in maniera allineata rispetto al frame fisico. Questo vincolo è dovuto al fatto che il registro CR3 è in grado di esprimere gli indirizzi di memoria con la granularità dei 4 KB. Comunque sia, quando abbiamo una page table da 4 MB, questa è tipicamente 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:

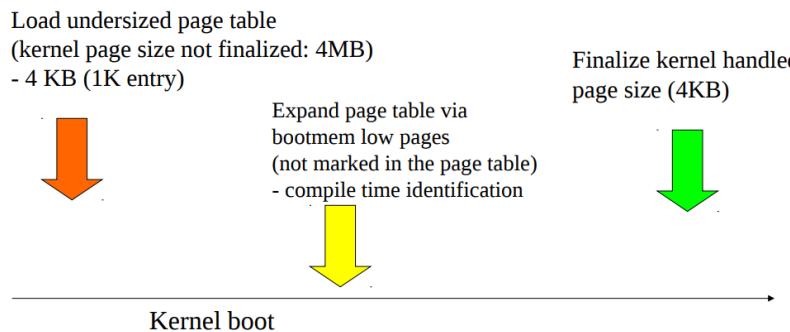
- 1) È necessario passare da una granularità delle pagine di 4 MB a una granularità delle pagine di 4 KB.
- 2) È necessario estendere a 1 GB la quantità di memoria logica riservata al kernel.
- 3) È necessario riorganizzare la page table in due livelli separati.
- 4) È necessario identificare le aree 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:

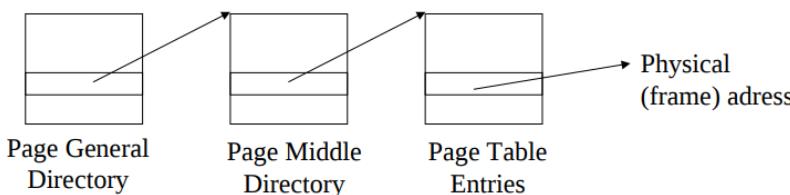
Linux vede in realtà la paginazione come organizzata a *tre livelli*. Ciascun indirizzo è dunque suddiviso nel seguente modo:



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  $1024 \times 1024 = 2^{20}$  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:

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

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

-> **Avail (available)**: sono bit messi a disposizione per un utilizzo custom da parte del programmatore.

-> **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:

Qui analizzeremo solo i campi diversi da quelli relativi alle entry della PDE.

31	12 11	9 8	7	6	5	4	3	2	1	0
Page Base Address	Avail	G A T	P A D	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<sub>1</sub>:** 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.

**NB<sub>2</sub>:** 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_ACSESSED  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 ri-esecuzione 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 (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 perchè 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, salterebbe 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

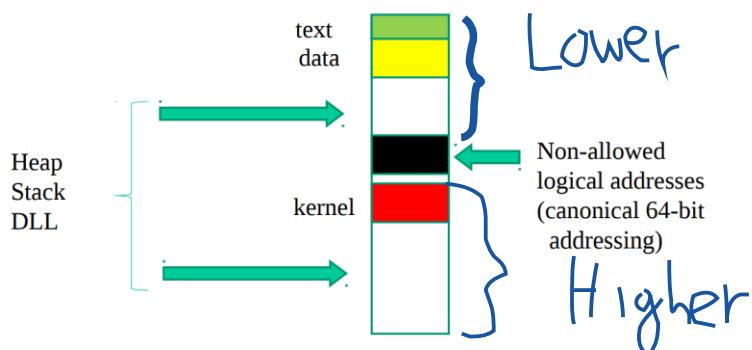
memoria logica di  $2^{64}$  byte. Nelle implementazioni effettive, però, si possono raggiungere al più  $2^{48}$  indirizzi (che comunque consentono di spannare su ben 256 TB).

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.

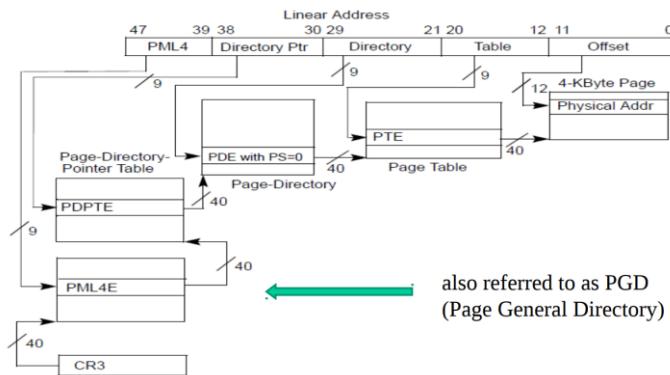
In definitiva, l'address space di un'applicazione in un sistema x86-64 si presenta così:



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:



93

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 (PDPT), 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 nelle architetture 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 *madvice()*.

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.

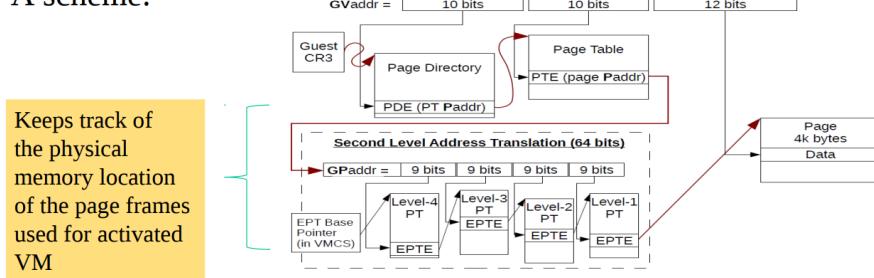
#### **Hardware supported “virtual memory” virtualization**

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.

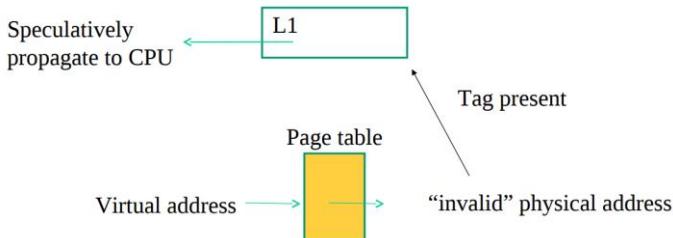
- Intel Extended Page Tables (EPT)
- AMD Nested Page Tables (NPT)
- A scheme:



### 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.

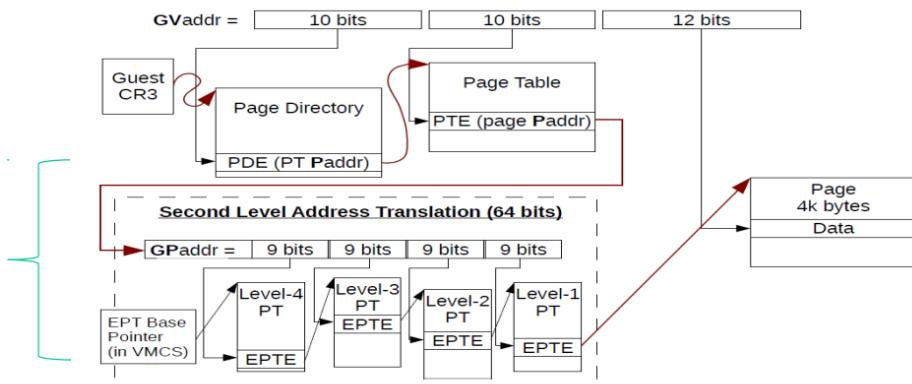
Da qui nasce l’attacco L1TF, che prevede la seguente idea: se abbiamo una entry di una page table con presence bit pari a 0 (non valida), la entry stessa potrebbe comunque aver propagato i suoi valori (e in particolare il TAG, ovvero i bit associati al presence bit) all’interno dell’architettura di memoria. 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

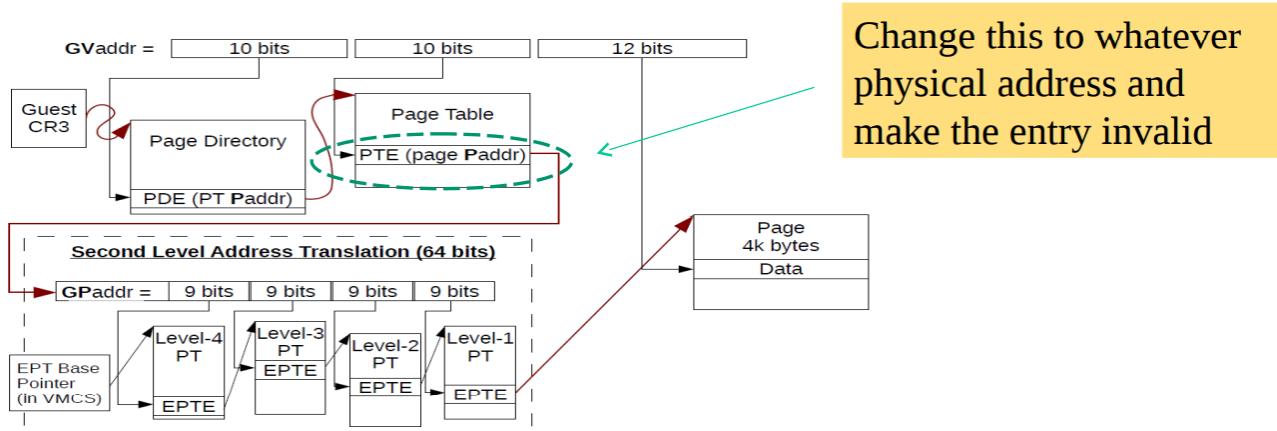


La **parte superiore** dello schema (comprendente Guest CR3, Page Directory e Page Table) è relativa alla paginazione che avviene all'interno della **macchina guest**, mentre la **parte inferiore** è relativa alla paginazione che avviene all'interno del **sistema host**.

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 condividano la stessa 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

È 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;
    ....
}

```

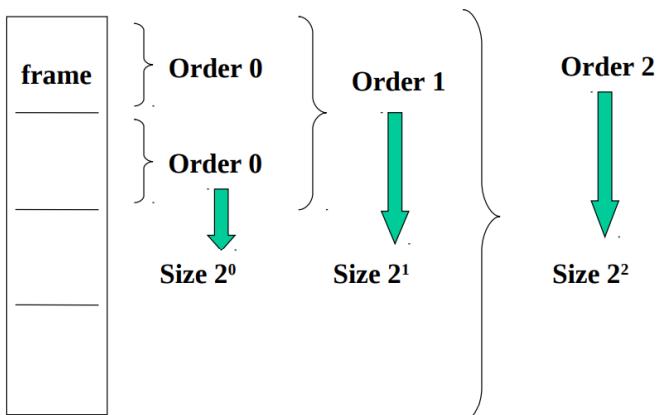
Where we do pick free memory  
blocks in a buddy allocator

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).

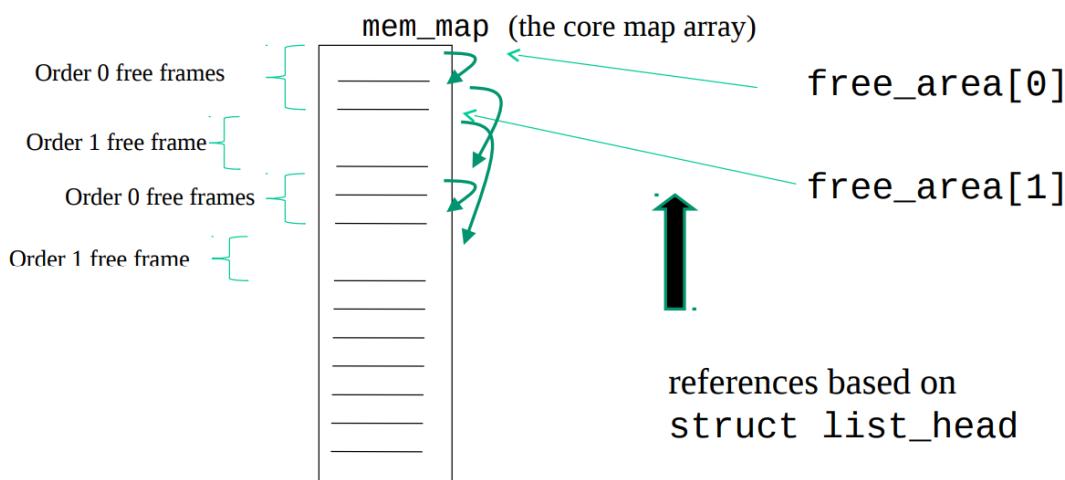
### Buddy allocator

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.

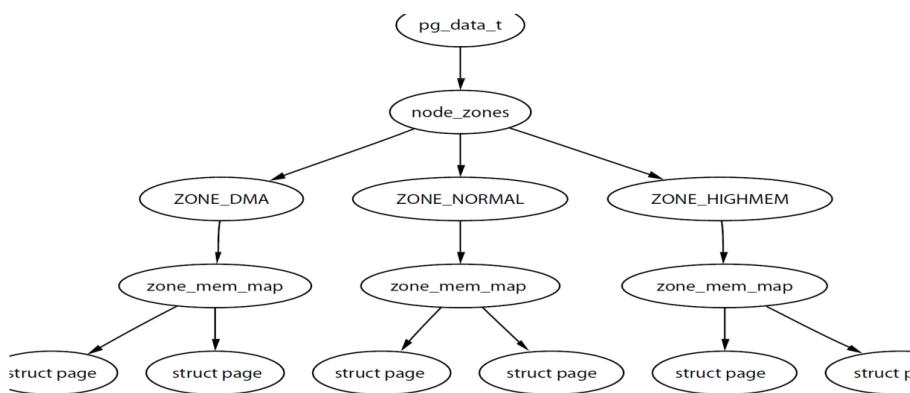


Tornando alla core map e alla free list:

- 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_aera` 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).



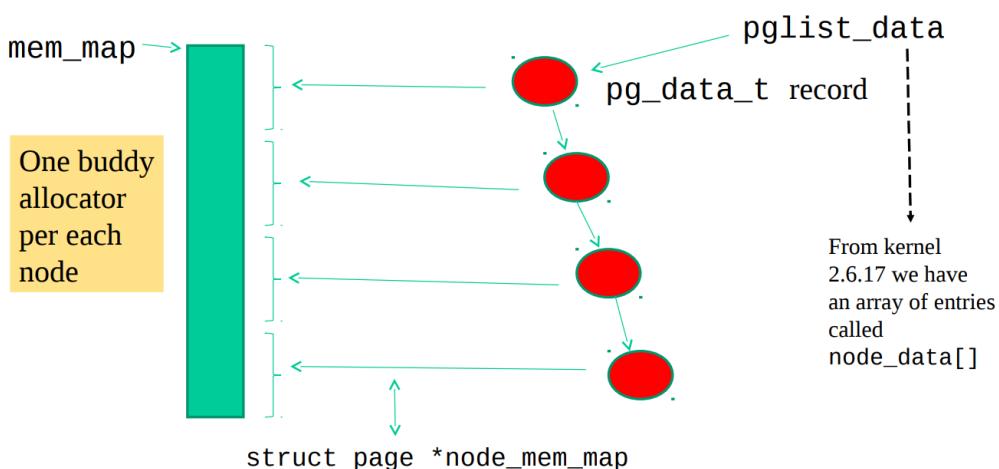
#### Schema associato a uno specifico nodo NUMA:



#### 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, parte 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, azzera 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 azzera 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 alloca un gruppo di ordine k (i.e. alloca  $2^k$  pagine). Il paramtro 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 dealloca 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. Da quale allocatore buddy, e la macchina è NUMA con più allocator, quale uso? Soluzione: **mem-policy**.

Tuttavia, 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). Infatti, si tratta di API di alto livello che a loro volta invoca 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: [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.

**long move\_pages (int pid, unsigned long count, void \*\*pages, const int \*nodes, int \*status, int flags)**

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 scollego, 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 (o 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, alcune generali \(“voglio memoria”\), altre specifiche.](#)

[Il meccanismo di acquisizione da basso livello è basato su lazyness. Abbiamo cache che gestisce taglia T, creo seconda cache \(crea secondo metadato\). Le chiamate sulla seconda cache possono usare la stessa area della prima cache. Una cache per allocazione memoria è oggetto memoria pre-riservato, cache vista come set informazioni disponibili, senza scendere giù per il buddy system, è layer software superiore. Quando viene pre-riservata viene fatta dal buddy allocator, allora 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 azzera 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.

Caratteristiche principali dello SLAB allocator:

- La memoria allocata è sempre allineata alla linea di cache; questo implica che due chiamate distinte a kmalloc() o kzalloc() portano all'allocazione di memoria all'interno di due linee di cache differenti.
- Viene supportata la fast allocation/deallocation: infatti, come abbiamo detto prima, l'allocatore di base restituisce una zona di memoria pre-reserved (i.e. preallocata dal buddy system), se disponibile; se invece è esaurita (come me), si ricorre direttamente al buddy allocator.

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 allocatori: ([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. 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 dei chunk che vengono consegnati dal nostro allocatore. Allora l'offset del primo buffer di memoria libero da consegnare è pari a  $D + ALN * COLOR$ .

### **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

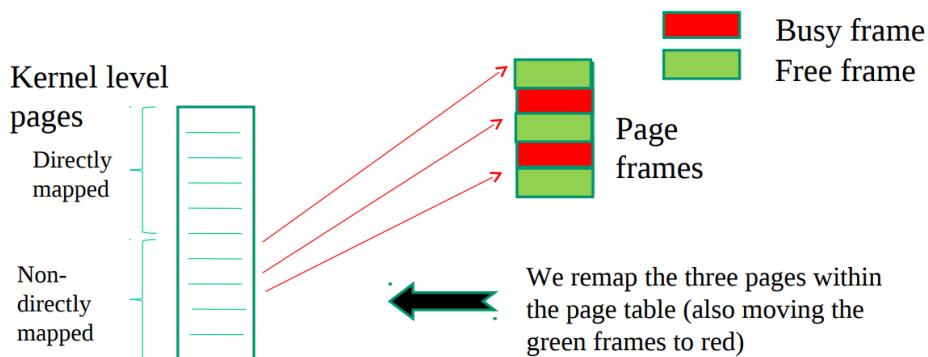
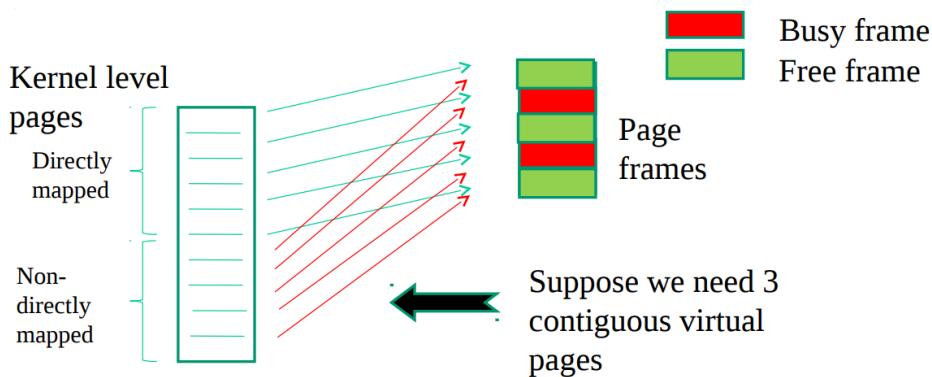
### Effetti sul TLB – interazione con l'hardware, come il TLB

(Translation Lookaside Buffer, 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 all'utilizzo del vmalloc:

Vediamo più nel dettaglio cosa può avvenire internamente quando si allocano delle pagine di memoria con vmalloc. Supponiamo di avere 5 pagine *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. 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).

NB<sub>1</sub>: 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.

NB<sub>2</sub>: 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) e 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:

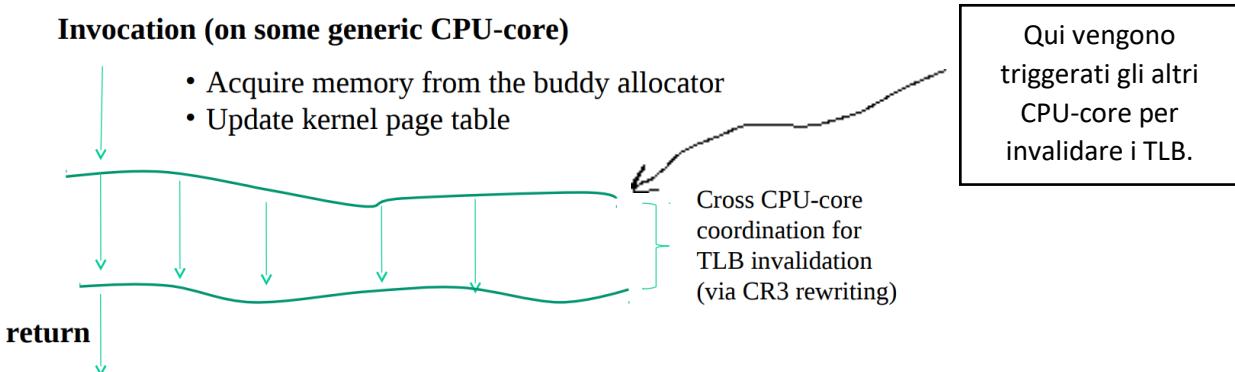
- > Latenza dell'inserimento delle informazioni aggiornate all'interno del TLB in caso di miss (dove il miss è una diretta conseguenza del flush del TLB). Questo costo dipende dal numero di entry del TLB che devono essere rinnovate.

### 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\_pgtables (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

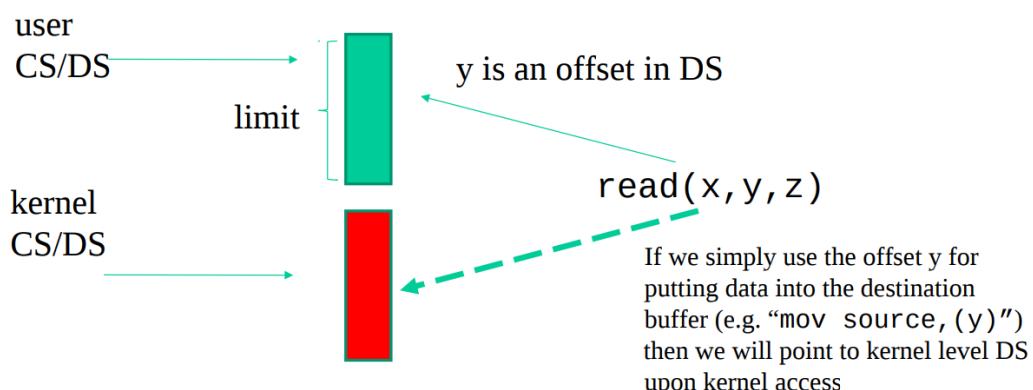
È la segmentazione di x86 protected mode, in cui è possibile fare in modo che i vari segmenti (come CS, DS) puntino ovunque vogliamo all'interno dell'address space.

- 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**” (= scritti 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 che 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 di livello kernel per lo user/kernel 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 AS, 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 byte da 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).

Se devo spostare dati da syscall read, si chiama kernel, e c'è thread a ring0, è lui che muove i dati. Va bene vedere se siamo sopra addr\_limit, ma quelle zone di memoria sono effettivamente utilizzabili? Perchè 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.

Molte di queste API si appoggiano ad access\_ok() per controllare preventivamente se l'operazione di spostamento dati è possibile o meno.

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 poi 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*. 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.

### Esempi:

- **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).
- Analogamente, **kernel\_write()** è una ridondanza per la system call `write()`.

### 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 nell'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.

Si hanno due bit in CR4 (il 21 e il 20) che servono per attivare / disattivare questi due supporti. In particolare, quando lato user viene ad esempio invocata una system call di `read()` e viene passato il controllo al software del kernel, quest'ultimo dovrà essere in grado di accedere a un'area di memoria user level; per questo motivo, prima di eseguire le istruzioni per finalizzare la `read()`, viene disattivato il bit relativo allo SMAP, e viene riattivato alla fine.

#### 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  $O(n)$ , perché prende `Thread Control Block`, poi Tabella management, scandirla e 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

### Introduzione

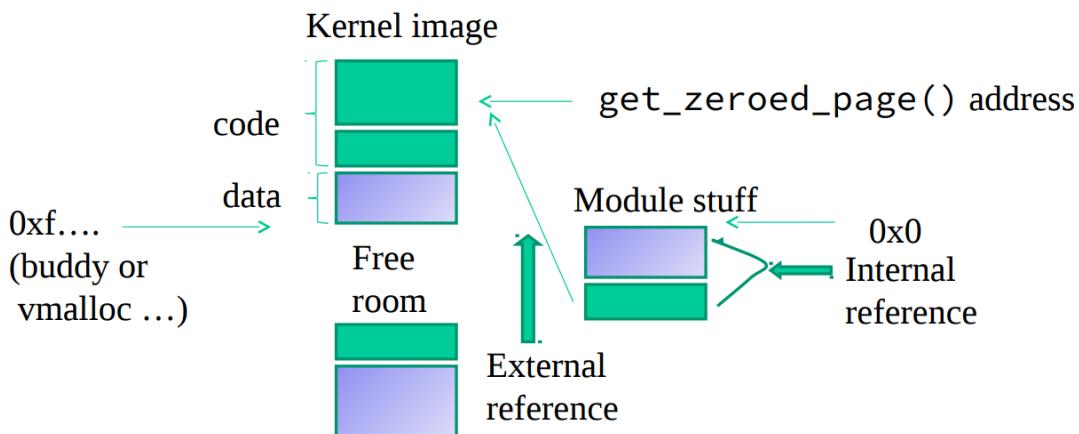
Un modulo Linux è un componente software che può essere aggiunto come parte del kernel (in modo tale da essere incluso all'interno dell'immagine di memoria del kernel) quando quest'ultimo è già in esecuzione. Un vantaggio nell'utilizzo dei moduli sta nel fatto che il kernel non deve essere ricompilato per aggiungere le nuove funzionalità software.

Un possibile utilizzo particolare dei moduli consiste nel gestire lo startup di una configurazione del kernel (vedi ad esempio i moduli riportati sul RAM disk).

### Step da seguire per inserire un modulo

- 1) Abbiamo bisogno di memoria (da allocare col buddy allocator) per caricare in RAM sia i blocchi di codice che le strutture dati inclusi nel modulo (ricordiamo che le pagine del kernel sono sempre materializzate in memoria).
- 2) Abbiamo bisogno di sapere dov'è locata questa memoria in modo tale da risolvere i riferimenti interni del modulo (che siano essi relativi ai dati o relativi al codice). [Abbiamo a che fare con indirizzi logici](#).
- 3) Abbiamo bisogno di sapere dove sono locate in memoria le facility (e.g. system call) già presenti nel kernel che dovranno essere utilizzate dal modulo che stiamo montando.
- 4) Mentre il modulo viene caricato, è necessario effettuare la risoluzione dei simboli all'interno del kernel in modo tale da essere in grado di accedere sia alle facility sia ai dati esterni al modulo.

Vediamo uno schema relativo al montaggio di un modulo:



Nel kernel abbiamo spazio libero in RAM, anche in termini di indirizzi logici per arrivarci in questa zona RAM. Il modulo viene caricato, ha una parte codice (verde), dove ad esempio ha chiamata `get\_zeroed\_page()`, deve cascicare in un punto corretto dell'immagine del kernel. Montare un modulo vuol dire identificare API sottosistema (es: allocatore di memoria, dove sta zona di memoria). Per questo, le [syscall si sono evolute o dismesse](#).

### Chi si occupa del montaggio di un modulo?

- > Fino alla versione 2.4 del kernel, gran parte del lavoro (ma non tutto) era fatto dal livello applicativo. Qui i moduli erano degli ELF con formato .o (erano cioè dei semplici oggetti generabili con gcc). [Non eseguibile, ma include relazioni tra modulo e cose esterne](#). Inoltre, si avevano dei grossi problemi di sicurezza: anche la risoluzione degli indirizzi delle facility e dei dati già presenti all'interno del kernel spettava al livello applicativo, il che vuol dire che il kernel esponeva tutta la sua struttura, vanificando alcuni dei meccanismi di sicurezza che erano implementati, come la randomizzazione del kernel space.
- > A partire dalla versione 2.6 del kernel, gran parte del lavoro viene fatto dal livello kernel. Qui i moduli sono degli ELF con formato .ko (kernel object) e contengono dei metadati in più rispetto al caso

precedente: questi metadati aggiuntivi servono al kernel non solo per effettuare l'allocazione della memoria, ma anche per risolvere l'indirizzamento della stessa. In tal modo, viene risolto il problema di sicurezza che si aveva in precedenza.

### System call suite

#### Fino al kernel 2.4:

- **create\_module**: riserva un buffer di memoria **logica** per il kernel e vi associa un nome, il quale verrà mantenuto dal kernel tra i metadati di gestione dei moduli. L'area di memoria che viene riservata è contigua nello spazio di indirizzamento logico ma non lo è necessariamente in memoria fisica.
- **init\_module**: carica l'immagine del modulo **finalizzata** (con l'indirizzamento della memoria risolto del tutto) all'interno del buffer riservato per il kernel. Dopodiché invoca la funzione di setup del modulo.
- **delete\_module**: invoca la funzione di shutdown del modulo e rilascia dal kernel il buffer dedicato al modulo. **Un modulo senza funzione di shutdown non è rimovibile. La shell ci dice che il modulo è "busy". Un modulo lockato, possono essere molti, è un contatore, non è rimovibile anche con funzione shutdown.** `rmmmod -f` forza tale operazione, ma è rischioso.

#### A partire dal kernel 2.6:

- **create\_module**: non è più supportata.
- **init\_module**: riserva un buffer di memoria per il kernel e vi associa un nome. Dopodiché, carica l'immagine del modulo **non finalizzata** all'interno del buffer (poi spetterà al kernel risolvere l'indirizzamento di memoria e finalizzare l'immagine). Infine, invoca la funzione di setup del modulo.
- **delete\_module**: è esattamente come nelle versioni precedenti del kernel.

#### Parti in comune tra le due generazioni:

- > Funzione per montare un modulo: **int init\_module(void)**, il cui valore di ritorno (intero) indica se il montaggio del modulo è andato a buon fine o meno.
- > Funzione per smontare un modulo: **void cleanup\_module(void)**.
- > **Usage count** (o **reference count**): è un contatore che tiene traccia del numero di entità che stanno utilizzando o che utilizzeranno un determinato modulo. Questo viene incrementato anche nel momento in cui un thread, mentre esegue il codice kernel del modulo, invoca un servizio bloccante e va in stato di wait; questo perché, dopo il servizio bloccante, il thread deve essere in grado di riprendere l'esecuzione all'interno del modulo (in particolare a partire dall'istruzione successiva al servizio stesso). Nel momento in cui lo usage count è maggiore di zero e si tenta di smontare il modulo, si ha un fallimento, a meno che non si forzi esplicitamente lo smontaggio.
- > **Passaggio dei parametri ai moduli**: non è un vero e proprio passaggio di parametri a funzione, bensì i parametri sono passati come valori iniziali di determinate variabili globali definite all'interno del codice sorgente del modulo. Perciò, queste variabili, nel momento in cui vengono definite, devono essere marcate esplicitamente come "parametri di modulo". In altre parole, il valore iniziale che assumono le variabili non è noto a tempo di compilazione, bensì a tempo di montaggio del modulo.

#### Dichiarazione dei parametri ai moduli

- Old style: **MODULE\_PARAM (variable, type)**, dove MODULE\_PARAM è una macro che accetta due argomenti: il nome del parametro (variable) e il tipo del parametro (type).
- New style: **module\_param (variable, type, perm)**, dove il terzo argomento della macro (perm) indica i permessi di accesso al parametro; di fatto, il valore del parametro viene esposto come contenuto di uno pseudo-file all'interno di una determinata sotto-directory della directory **/sys** (più precisamente in **/sys/module**) del virtual file system (dove **/sys** è la versione moderna della directory **/proc**).

In riferimento al *new style*, se i permessi di accesso lo consentono, è anche possibile cambiare il valore dei parametri ai moduli a run-time (che praticamente significa cambiare lo stato del kernel). Tuttavia, in fase di programmazione del modulo, non è possibile specificare dei permessi di accesso troppo larghi perché

altrimenti si avrebbe un errore a tempo di compilazione; è comunque possibile modificare a piacimento tali permessi di accesso successivamente. L'owner è chi ha eseguito il montaggio, tipicamente il root.

### Variante per il tipo array:

La macro che si usa per dichiarare un parametro a modulo di tipo array è **module\_param\_array()**, che accetta quattro argomenti:

- > Nome della variabile di tipo array.
- > Tipo degli elementi dell'array.
- > Indirizzo di una variabile che specifica la dimensione dell'array.
- > Permessi di accesso al parametro.

### Programmi per montare / smontare un modulo

Per montare e smontare un modulo, l'utente non utilizza direttamente le system call `init\_module` / `delete\_module`, bensì utilizza dei programmi che al loro interno invocano tali system call:

- Per il montaggio: **insmod**, che vuole come argomento il file.ko relativo al modulo che si vuole montare. In realtà, è possibile passargli anche degli ulteriori argomenti di tipo `variable=value`, che permettono appunto di inizializzare i parametri del modulo.
- Per lo smontaggio: **rmmod**, che vuole come argomento il nome del modulo da smontare.
- Esiste anche un programma che serve per cercare un modulo all'interno del kernel: **modprobe**.

### Funzionamento di insmod

#### Fino al kernel 2.4:

- 1) Invocazione di `create\_module`.
- 2) Risoluzione di tutti i riferimenti interni ed esterni al modulo; i riferimenti esterni sono esposti in **/proc/kallsyms**, ed è qui che il kernel indica al livello user qual è la sua struttura (i.e. gli indirizzi di memoria dei suoi simboli) vanificando così i meccanismi come la randomizzazione; in versioni più recenti del kernel, il campo "indirizzo" all'interno di /proc/kallsyms viene forzato a 0 per tutti i simboli del kernel, aumentando così il livello di sicurezza.
- 3) Invocazione di init\_module.

#### A partire dal kernel 2.6:

- 1) Invocazione di init\_module.

### Approfondimento sulle system call dei moduli

#### Fino al kernel 2.4:

- > **caddr\_t create\_module (const char \*name, size\_t size)**: tenta di allocare un buffer di memoria per il nuovo modulo, e può essere utilizzata solo dal superuser. Se ha successo restituisce l'indirizzo di memoria kernel a partire dal quale il nuovo modulo risiederà; altrimenti restituisce -1.
- > **int init\_module (const char \*name, struct module \*image)**: carica l'immagine del modulo (già rilocata e finalizzata) ed esegue la funzione init del modulo. L'immagine è composta da un header e dai segmenti code e data veri e propri, dove l'header è descritto dalla **struct module** riportata qui di seguito:

```

struct module {
    unsigned long size_of_struct;
    struct module *next;  const char *name;
    unsigned long size;      long usecount;
    unsigned long flags;     unsigned int nsyms;
    unsigned int ndeps;      struct module_symbol *syms;
    struct module_ref *deps; struct module_ref *refs;
    int (*init)(void); void (*cleanup)(void);
    const struct exception_table_entry *ex_table_start;
    const struct exception_table_entry *ex_table_end;
#ifdef __alpha__
    unsigned long gp;
#endif
};
```

- struct module \*next: puntatore all'header di un altro modulo.
- const char \*name: nome del modulo.
- long usecount: usage count del modulo.
- int (\*init)(void): puntatore alla funzione init del modulo (è un **riferimento interno**).
- int (\*cleanup)(void): puntatore alla funzione cleanup del modulo (è un **riferimento interno**).

NB: come detto in precedenza, i riferimenti esterni vengono risolti tramite la sottodirectory /proc/kallsyms del virtual file system.

-> **int delete\_module (const char \*name)**: tenta di rimuovere un modulo correntemente non utilizzato; se name == NULL, allora verranno rimossi tutti i moduli non utilizzati marcati come auto-clean. Come **create\_module()**, è una system call che può essere usata solo dal superuser. Se ha successo restituisce 0, altrimenti restituisce -1. `Rmmod` usa internamente `delete\_module`, a cui passiamo ulteriori parametri.

#### A partire dal kernel 2.6:

-> **int init\_module (void \*module\_image, unsigned long len, const char \*param\_values)**: carica l'immagine del modulo (non ancora finalizzata) all'interno dello spazio di indirizzamento del kernel, effettua tutte le risoluzioni dei simboli necessari e poi esegue la funzione init del modulo. L'argomento param\_values è una stringa atta a inizializzare i parametri del modulo con delle associazioni di tipo *variable=value*.

-> **int finit\_module (int fd, const char \*param\_values, int flags)**: è come init\_module() con la differenza che preleva l'immagine del modulo da montare da un particolare file identificato dal file descriptor fd. Con i flag, possiamo chiedere l'utilizzo del check sulle signatures dei simboli. Se un modulo vuole usare un sottosistema esterno per chiamare una funzione f() del kernel (quindi esterna del modulo). Magari f() prima ritornava intero, e oggi long. Quindi quale f() viene usata? Così lo controllo.

#### File generati nella creazione di un modulo

#### A partire dal kernel 2.6:

Sappiamo che, a partire dalla versione 2.6 del kernel Linux, il file che si deve ottenere in ultima istanza nella definizione di un modulo deve avere un formato .ko (kernel object). Per dare luogo a un file .ko, si deve comunque partire da un file .o: a tal proposito, viene in aiuto il programma **modpost** che, dal file .o, crea un file sorgente C (con formato .mod) che descrive le sezioni addizionali richieste per il file .ko. Dopodiché, il file .mod viene compilato e viene linkato al file .o originale in modo tale da produrre, finalmente, il file .ko.

#### Macro definibili in un modulo - header

- #include <linux/module.h> include tutte le facility di base che possono essere utilizzate per la

strutturazione dei moduli.

- `#include <linux/kernel.h>` include le funzionalità di base del kernel.
- `#include <linux/smp.h>` include le funzionalità proprie delle macchine parallele come le API per la sincronizzazione.
- `#define __KERNEL__` serve per includere negli header file le parti di codice comprese nella direttiva preprocessore `ifdef __KERNEL__`.
- `#define MODULE` serve per includere negli header file le parti di codice comprese nella direttiva preprocessore `ifdef MODULE`.

### Dettagli sullo usage count

#### Fino al kernel 2.4:

Nelle versioni più ancestrali del kernel, venivano messe a disposizione tre macro all'interno di `module.h` che dichiaravano tre corrispettive API per lo usage count:

- > **`MOD_INC_USE_COUNT`**: incrementa lo usage count di un'unità del modulo che stiamo eseguendo.
- > **`MOD_DEC_USE_COUNT`**: decremente lo usage count di un'unità modulo che stiamo eseguendo.
- > **`MOD_IN_USE`**: verifica se lo usage count è strettamente maggiore di 0. Controllo solo stato contatore.

*NB:* queste API possono operare solo per il modulo corrente, il che lascia una finestra di vulnerabilità: supponiamo che un modulo M contenga una funzione che, nel mezzo, invoca un qualche servizio di un altro modulo M'. È possibile incrementare lo usage count solo dopo essere saltati in M', e questo dà luogo alla possibilità di smontare M' appena prima dell'incremento dello usage count.

Le informazioni sui moduli montati nel sistema (tra cui lo usage count e la quantità di memoria messa a disposizione dei moduli) erano riportate nel file **`/proc/modules`**.

#### A partire dal kernel 2.6:

Nelle versioni più recenti del kernel, per risolvere la problematica descritta poc'anzi, è stata introdotta la possibilità di manipolare anche gli usage count degli altri moduli. Di conseguenza, le API per gli usage count sono diventate le seguenti:

- > **`try_module_get (struct module *module)`**: incrementa di un'unità lo usage count del modulo specificato come parametro. [Passo header del modulo](#).
- > **`module_put (struct module *module)`**: decremente di un'unità lo usage count del modulo specificato come parametro.
- > **`CONFIG_MODULE_UNLOAD`**: verifica se lo usage count è strettamente maggiore di 0.

Ma come otteniamo il puntatore alla struct module nel momento in cui vogliamo manipolare lo usage count del modulo che abbiamo scelto?

- Se vogliamo manipolare lo usage count del modulo corrente, è sufficiente utilizzare la macro **`THIS_MODULE`**.
- In caso contrario, basta ricorrere a un'altra API: **`struct module *find_module (const char *name)`**.

### Simboli del kernel esportati

Un simbolo **esportato** (e.g. il nome di una variabile o di una funzione) è un simbolo che viene reso disponibile e può essere referenziato da qualunque modulo si voglia caricare. Chiaramente, se un modulo M referenzia un simbolo che non è stato esportato, il montaggio di M fallirà.

Per esportare un simbolo, è sufficiente utilizzare l'apposita API che può essere invocata tramite la macro **`EXPORT_SYMBOL`** (definita sempre in `module.h`).

Tutti i simboli esportati dal kernel sono posti all'interno di una particolare struttura dati che può essere acceduta mediante gli pseudo-file. Uno di questi pseudo-file è proprio `/proc/kallsyms`. I simboli possono avere tre stati: **static** (non in kernel table), **available** (in kernel table, non usabile) e **exported** (in kernel

table, usabile dai moduli). Questo per vecchie versioni, oggi, data certa versione kernel, posso vedere la lista dei simboli esportati all'interno di quest'altro file:

`/lib/modules/<kernel version>/build/Module.symvers`

Fino a kernel 5.7, con `void\* kallsyms\_lookup\_name(const char\*)` potevamo chiedere informazioni sulla tabella dei simboli kernel. Permette di cercare ed usare i simboli kernel. Non era per niente sicuro!

Era esposta al kernel probing, kprobe subsystem, per cui dopo 5.7 mi riferivo a lui per usarla!

Cosa è questo sottosistema kProbe?

## Kprobe

È un servizio del kernel che permette di effettuare delle query al kernel e di modificare le API del kernel aggiungendovi un **preambolo** (da eseguire subito prima dell'API vera e propria) e una **coda** (da eseguire subito dopo l'API vera e propria).

È possibile registrare e de-registrare le kprobe mediante le seguenti API:

-> **int \_\_kprobes register\_kprobe (struct kprobe \*p)**: registra una nuova kprobe (passata come parametro) all'interno dell'apposita struttura dati che mantiene le kprobe attive.

-> **int \_\_kprobes register\_kretprobe (struct kretprobe \*p)**: registra una nuova kretprobe (passata come parametro) all'interno dell'apposita struttura dati che mantiene le kprobe attive. Una kretprobe, a differenza di una kprobe, assegna all'API da patchare soltanto la coda e non il preambolo.

-> **static int \_\_kprobes \_\_unregister\_kprobe\_top (struct kprobe \*p)**: rimuove la kprobe passata come parametro dalla struttura dati che mantiene le kprobe attive.

Affinché le kprobe siano effettivamente abilitate, è necessario che la seguente condizione sia verificata: CONFIG\_KPROBES=yes and (CONFIG\_KALLSYMS=yes or CONFIG\_KALLSYMS\_ALL=yes)

**Definizione della struct kprobe: (richiede <linux/kprobes.h>)**

```
struct kprobe {
    struct hlist_node hlist; /* Internal */
    .....
    kprobe_opcode_t addr; /* Address of probe */
    .....
    const char *symbol_name; /* probed function name */
    kprobe_pre_handler_t pre_handler;
        /* Address of pre-handler */
    kprobe_post_handler_t post_handler;
        /* Address of post-handler */
    .....
};
```

La probe può essere disabilitata, ma comunque contata.

Gli ultimi due oggetti sono puntatori a funzione. Se applichiamo probe ad istruzione "i", quando un thread passa su tale istruzione macchina. Prima di passare sull'istruzione si passa per "pre\_handler", poi tocca "i", e dopo "post\_handler". La stessa cosa avviene chiamando symbol name, perché stiamo chiamando probe su prima istruzione macchina a tale funzione. Possono essere null. Spesso usate per vedere stato CPU.

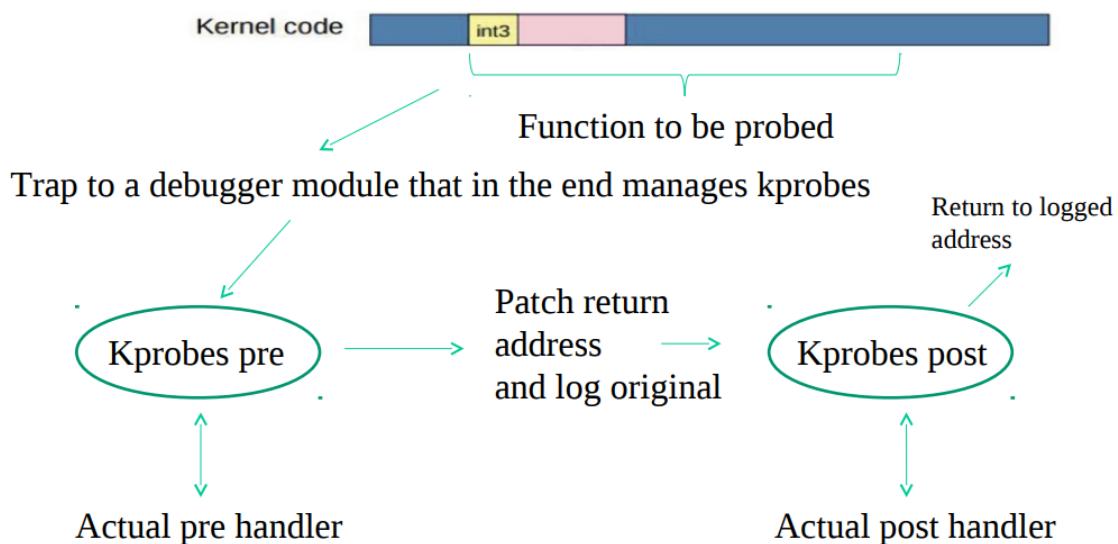
Meccanismo delle kprobe:

Supponiamo di dover eseguire una funzione f del kernel che deve essere "probed" (i.e. patchata a run-time). La prima cosa che si fa è modificare dinamicamente il primo (o comunque i primissimi) byte di f, in modo tale da renderlo un **int3**, (richiede IDT e segmentazione), ovvero un'istruzione che solleva una trap per attivare un **modulo di debug** che, in fin dei conti, gestisce le kprobe. Una volta che il modulo di debug ha preso il controllo, può eventualmente eseguire il preambolo di f, ma poi sicuramente va a cambiare l'indirizzo di ritorno da f sullo stack, inserendovi l'indirizzo di memoria della coda e memorizzando in un log l'indirizzo di ritorno originale (questo è possibile perché il modulo di debug utilizza lo stesso stack di f).

Dopodiché può finalmente essere eseguito il corpo centrale della funzione f; non appena si arriva a

un'istruzione di return, si salta alla coda e solo a posteriori si potrà tornare all'indirizzo di ritorno originale (che viene recuperato dal log). Tra le varie ottimizzazioni, non useremo più int3 (costoso in termini di cicli macchina), bensì un altro approccio.

Di seguito è riportato uno schema riassuntivo di tutto questo funzionamento:



NB: affinché si possa inserire facilmente l'istruzione int3 all'inizio della funzione f, in genere a tempo di compilazione le funzioni del kernel risultano iniziare con delle NOP che, appunto, sono facilmente sostituibili e, quindi, non causano alcun problema.

#### Handler delle kprobe:

Sono definiti nel seguente modo:

-> **typedef int (\*kprobe\_pre\_handler\_t) (struct kprobe\*, struct pt\_regs\*)**: struct kprobe\* contiene l'indirizzo della struttura delle kprobe che indica qual è la kprobe con cui stiamo lavorando; struct pt\_regs\* è un puntatore a un'area di memoria dove potrà essere memorizzato lo snapshot di CPU in modo tale che, dopo l'esecuzione del pre-handler (i.e. del preambolo), potrà essere ripristinato correttamente.

-> **typedef void (\*kprobe\_post\_handler\_t) (struct kprobe\*, struct pt\_regs\*, unsigned long flags)**

#### **Definizione della struct kretprobe (richiede <linux/kprobes.h>):**

Qui ci riferiamo ad un sottosistema, in cui entro ed esco. Prima era specifica istruzione macchina.

```
struct kretprobe {
    struct kprobe_kp;
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    int maxactive; } ; Max active number and counter
    int nmissed; .....;
```

Very similar interface as other probe handlers

Max active number and counter of lost activations

- `int maxactive` = numero massimo di istanze attive di post-handler (i.e. di code) che possiamo avere. Il default è -1, e dipende se kernel è pre-emptable o meno. Nel primo caso è più alto. Se non pre-emptable, è pari al numero delle CPU.

- `int nmissed` = numero di volte in cui non siamo riusciti ad attivare nuove istanze di post-handler perché avevamo già raggiunto il numero massimo.

Il primo intercetta singole istruzioni, qui invece parliamo delle funzioni. Richiede `struct kprobe kp`.  
`entry handler` sarebbe il pre, `handler` sarebbe il post. Possono essere nulli. Il return address della kprob  
che gira in un sottosistema del kernel, deve essere salvato da qualche parte, quindi il sottosistema di  
kprobing gestisce tale aspetto. E' associato a `maxactive`.

### Negazione del probing:

Esistono alcune funzioni del kernel che non possono essere “probed”. Tra queste figurano le API utilizzate come in-line (in cui non è possibile catturare una variazione del flusso di esecuzione mediante un’istruzione int3) e le API che possono essere triggerate (anche indirettamente) dall’esecuzione di una qualche probe. Queste funzioni che non possono essere “probed” sono listate all’interno del seguente pseudofайл:

## /sys/kernel/debug/kprobes/blacklist

**NB: il thread che gira sul pre\_handler e post\_handler è NON PRE-EMPTABLE, quindi non lascia la cpu nè può chiamare servizi bloccanti.** Questo perchè si hanno delle *per-cpu-variables*, tra cui current\_kprobe, che tiene conto del contesto d'esecuzione. Se lascio la CPU, il contesto sarebbe diverso. Inoltre loggo degli address da usare in futuro. Ricordiamo che lo scopo principale era il debugging.

JPROBE

Permette passaggio thread attraverso wrapper, esso vedrà esattamente i parametri passati, senza sfruttare cpu e registri. L'handler, con certa signtura, sarà il wrapper avente stessa signtura del sottosistema da esaminare.

## Versioning del kernel Linux

All'interno dell'header module.h è incluso **version.h**, che comprende delle macro che servono per catturare la versione del kernel come:

-> **UTS\_RELEASE**: restituisce la stringa che indica la versione del kernel all'interno del quale è montato il modulo che stiamo utilizzando (e.g. "4.12.14").

-> **LINUX\_VERSION\_CODE**: restituisce il codice numerico che identifica la versione del kernel all'interno del quale è montato il modulo che stiamo utilizzando. [Utile per vedere se codice adatto.](#)

-> **KERNEL\_VERSION (major, minor, release)**: restituisce il codice numerico che identifica la versione del kernel specificata dai parametri major, minor e release; spesso, nell'effettuare i controlli sulla versione di Linux che si sta utilizzando, si compara questo valore con **LINUX\_VERSION\_CODE**.

Esempio:

Compiler defined outcome

```
#if LINUX_VERSION_CODE > KERNEL_VERSION(x,y,z)
    <whatever you want to specify or include>
#else
    <whatever else you want to specify or include>
#endif
```

Programmer specified outcome

Questo permette di scrivere del software di livello kernel che sia portabile su molteplici versioni del kernel Linux.

## Ridenominazione delle funzioni di startup /shutdown dei moduli

A partire dalla versione 2.3.13 del kernel Linux sono state introdotte delle facility per rinominare le funzioni di startup e shutdown di un modulo:

-> **module\_init** (*my\_init*): genera una routine di startup associata al simbolo *my\_init*.

-> **module\_exit (my\_exit)**: genera una routine di shutdown associata al simbolo my\_exit.

Questo ci permette di rendere il nostro modulo un vero e proprio componente della compilazione del kernel, con una sua funzione di startup ed eventualmente una sua funzione di shutdown.

### Sistema di messaggistica del kernel Linux – perchè esiste sottosistema printk?

Il software di livello kernel può fornire dei messaggi di output in relazione a eventi che occorrono durante l'esecuzione. Tali messaggi possono essere prodotti sia a steady state ma anche durante l'inizializzazione del kernel. Per questo motivo, non è possibile gestire la messaggistica del kernel mediante le system call come sys\_write() o kernel\_write(); piuttosto, si ricorre a un meccanismo che sia perfettamente funzionante anche durante il boot del kernel. In particolare, a ogni messaggio m generato dal kernel, è possibile eseguire fino a due operazioni:

- 1) Riportare m in un apposito **buffer circolare** proprio del kernel (operazione che viene eseguita sempre).
- 2) Riportare m in **console** (operazione che, essendo costosa, viene eseguita dipendentemente dal livello di criticità di m).

Può finire in 1), oppure in 1) e 2).

La funzione che si occupa di produrre i messaggi del kernel in output con questa logica si chiama **printk()**, che è molto simile alla printf(), con le differenze che non può gestire i floating point (kernel lavora su indirizzi) e offre la possibilità di specificare il livello di criticità (o priorità) dei messaggi. Attesa non bloccante, è attiva, abbiamo spinlock.

Il livello di criticità di un messaggio può essere specificato mediante una macro (che viene poi espansa in stringa) che può essere anteposta a tutti gli altri argomenti passati a printk() (inclusa la stringa relativa al messaggio stesso). Le varie macro associate ai livelli di criticità sono definite nel file header kernel.h e sono:

```
#define KERN_EMERG "<0>" /* system is unusable */
#define KERN_ALERT "<1>" /* action must be taken immediately */
#define KERN_CRIT "<2>" /* critical conditions */
#define KERN_ERR "<3>" /* error conditions */
#define KERN_WARNING "<4>" /* warning conditions */
#define KERN_NOTICE "<5>" /* normal but significant condition */
#define KERN_INFO "<6>" /* informational */
#define KERN_DEBUG "<7>" /* debug-level messages */
```

Usage: `printk(KERN\_WARNING "msg to print")`. Posso passarci pointers solo se non crittano messaggi.

Durante il debugging, possiamo disattivarlo (se versione kernel recente). Si usa `'%px` per non avere cifratura, anche se il sistema è crittografato.

Disponiamo anche delle funzioni alias di queste macro:

Name	String	Meaning	alias function
KERN_EMERG	"0"	Emergency messages, system is about to crash or is unstable	pr_emerg
KERN_ALERT	"1"	Something bad happened and action must be taken immediately	pr_alert
KERN_CRIT	"2"	A critical condition occurred like a serious hardware/software failure	pr_crit
KERN_ERR	"3"	An error condition, often used by drivers to indicate difficulties with the hardware	pr_err
KERN_WARNING	"4"	A warning, meaning nothing serious by itself but might indicate problems	pr_warning
KERN_NOTICE	"5"	Nothing serious, but notably nevertheless. Often used to report security events.	pr_notice
KERN_INFO	"6"	Informational message e.g. startup information at driver initialization	pr_info
KERN_DEBUG	"7"	Debug messages	pr_debug, pr-devel if DEBUG is defined
KERN_DEFAULT	"d"	The default kernel loglevel	

The aliases automatically generate the priority string

NB: la stampa degli indirizzi di memoria di livello kernel è un aspetto da considerare con la massima attenzione, specialmente per la sicurezza. Di fatto, causerebbe una diffusione troppo immediata del posizionamento in memoria delle informazioni di livello kernel. Di conseguenza, ogni volta che viene utilizzato il modificatore `%p`, viene stampato l'indirizzo di memoria **cifrato**; per evitare l'encryption, si può comunque usare il modificatore `%px`.

#### Trattamento delle priorità dei messaggi:

Esistono 4 parametri configurabili che determinano quale deve essere l'effettivo trattamento dei messaggi di output. Dati che sono riportati nello pseudofile `/proc/sys/kernel/printk`, sono utilizzabili anche user-level, e sono associati alle seguenti variabili:

- > **console\_loglevel**: è il livello di priorità al di sotto del quale i messaggi sono effettivamente riportati sulla console. Ad esempio, se `console_loglevel` vale 1, soltanto i messaggi con priorità 0 vengono riportati sulla console. Comunque sia, il suo valore tipico è 7. [Cambiabile a runtime](#).
- > **minimum\_console\_loglevel**: è il valore minimo che può assumere la variabile `console_loglevel`.
- > **default\_console\_loglevel**: è il valore di default della variabile `console_loglevel`.
- > **default\_message\_loglevel**: è il livello di priorità di default che viene associato ai messaggi nel momento in cui tale livello non viene espresso esplicitamente all'interno della `printk()`.

#### Gestione del buffer circolare:

Avviene mediante un'unica system call: **syslog (int type, char \*bufp, int len)**, dove:

- int type = codice numerico che indica l'operazione che si vuole eseguire sul buffer circolare.
- char \*bufp = buffer su cui si vuole eseguire l'operazione.
- int len = numero di byte che verranno coinvolti nell'operazione. [Insieme a bufp serve per passare byte da leggere](#).

Il parametro type può assumere uno dei seguenti valori:

**SYSLOG\_ACTION\_CLOSE** (0) Close the log. Currently a NOP.

**SYSLOG\_ACTION\_OPEN** (1) Open the log. Currently a NOP.

**SYSLOG\_ACTION\_READ** (2) Read from the log.

The call waits until the kernel log buffer is nonempty, and then reads at most `len` bytes into the buffer pointed to by `bufp`. The call returns the number of bytes read. Bytes read from the log disappear from the log buffer: the information can be read only once. This is the function executed by the kernel when a user program reads `/proc/kmsg`.

**SYSLOG\_ACTION\_READ\_ALL** (3) Read all messages remaining in the ring buffer, placing them in the buffer pointed to by `bufp`. The call reads the last `len` bytes from the log buffer (nondestructively), but will not read more than was written into the buffer since the last "clear ring buffer" command (see command 5 below). The call returns the number of bytes read.

**SYSLOG\_ACTION\_READ\_CLEAR** (4) Read and clear all messages remaining in the ring buffer. The call does precisely the same as for a type of 3, but also executes the "clear ring buffer" command.

**SYSLOG\_ACTION\_CLEAR** (5) The call executes just the "clear ring buffer" command. The `bufp` and `len` arguments are ignored. This command does not really clear the ring buffer. Rather, it sets a kernel bookkeeping variable that determines the results returned by commands 3 (**SYSLOG\_ACTION\_READ\_ALL**) and 4 (**SYSLOG\_ACTION\_READ\_CLEAR**). This command has no effect on commands 2 (**SYSLOG\_ACTION\_READ**) and 9 (**SYSLOG\_ACTION\_SIZE\_UNREAD**).

**SYSLOG\_ACTION\_CONSOLE\_OFF** (6) The command saves the current value of *console\_loglevel* and then sets *console\_loglevel* to *minimum\_console\_loglevel*, so that no messages are printed to the console. Before Linux 2.6.32, the command simply sets *console\_loglevel* to *minimum\_console\_loglevel*. See the discussion of */proc/sys/kernel/printk*, below. The *bufp* and *len* arguments are ignored.

**SYSLOG\_ACTION\_CONSOLE\_ON** (7) If a previous **SYSLOG\_ACTION\_CONSOLE\_OFF** command has been performed, this command restores *console\_loglevel* to the value that was saved by that command. Before Linux 2.6.32, this command simply sets *console\_loglevel* to *default\_console\_loglevel*. See the discussion of */proc/sys/kernel/printk*, below. The *bufp* and *len* arguments are ignored.

**SYSLOG\_ACTION\_CONSOLE\_LEVEL** (8) The call sets *console\_loglevel* to the value given in *len*, which must be an integer between 1 and 8 (inclusive). The kernel silently enforces a minimum value of *minimum\_console\_loglevel* for *len*. See the *log level* section for details. The *bufp* argument is ignored.

**SYSLOG\_ACTION\_SIZE\_UNREAD** (9) (since Linux 2.4.10) The call returns the number of bytes currently available to be read from the kernel log buffer via command 2 (**SYSLOG\_ACTION\_READ**). The *bufp* and *len* arguments are ignored.

**SYSLOG\_ACTION\_SIZE\_BUFFER** (10) (since Linux 2.6.6) This command returns the total size of the kernel log buffer. The *bufp* and *len* arguments are ignored.

Ma chi è che tipicamente si occupa di effettuare periodicamente il retrieve delle informazioni poste all'interno del buffer circolare? Il **klogd (Kernel Log Daemon)**.

## SYNOPSIS

```
klogd [ -c n ] [ -d ] [ -f fname ] [ -iI ] [ -n ] [ -o ] [ -p ] [ -s ] [ -k
      fname ] [ -v ] [ -x ] [ -2 ]
```

Per quanto riguarda le dimensioni del buffer circolare:

- > Originariamente occupava solo una pagina (4096 byte).
- > A partire dal kernel 1.3.54 occupava due pagine (8192 byte).
- > A partire dal kernel 2.1.113 occupava quattro pagine (16384 byte).
- > Nelle versioni più recenti del kernel occupa molto più spazio.

In ogni caso, il kernel buffer è unico: di conseguenza, si hanno dei costi dovuti al fatto che tale buffer deve essere gestito in maniera sincronizzata. Dunque, è vero che inserire dei messaggi solo nel buffer e non in console riduce un po' di overhead computazionale, ma comunque il costo dell'operazione rimane non banale.

Se utilizziamo la console, invece, la funzione `printk()` non restituisce il controllo finché il messaggio non è stato effettivamente inviato alla console stessa. Per fare un esempio, la consegna di un messaggio su un dispositivo di console seriale che lavora a 9600 bit al secondo rallenta la velocità del sistema di un millisecondo per ogni carattere.

## Funzione panic()

È un'altra funzione che si occupa di riportare messaggi sulla console, tant'è vero che si appoggia su `printk()`. La sua particolarità sta nel fatto che i messaggi che stampa hanno un preambolo prefissato, che è “**Kernel panic**”. Chiaramente si tratta di messaggi col livello di criticità 0.

Quando `panic()` viene invocata, il kernel viene freezato: di fatto, si ricorre a questa funzione solo quando all'interno del kernel avviene qualcosa di veramente strano (e.g. una qualche struttura dati del kernel non è più conforme a quelle che dovrebbero essere le sue caratteristiche).

## KERNEL LEVEL TASK MANAGEMENT

### Introduzione

Un **task** non è altro che una traccia di esecuzione e può essere di tre tipologie:

- Thread in esecuzione in modalità user.
- Thread in esecuzione in modalità kernel.
- Thread che sta eseguendo il gestore di un interrupt.

Quest'ultimo tipo di task è caratterizzato dal **non-determinismo**: di fatto, un thread (sia esso in esecuzione in modalità user, sia esso in esecuzione in modalità kernel) può essere colpito da un interrupt in qualunque istante della sua esecuzione, per cui, in linea di principio, potrebbe switchare verso l'esecuzione del gestore di un interrupt (interrompendo così momentaneamente le vecchie attività) in un qualunque momento che non è predicibile.

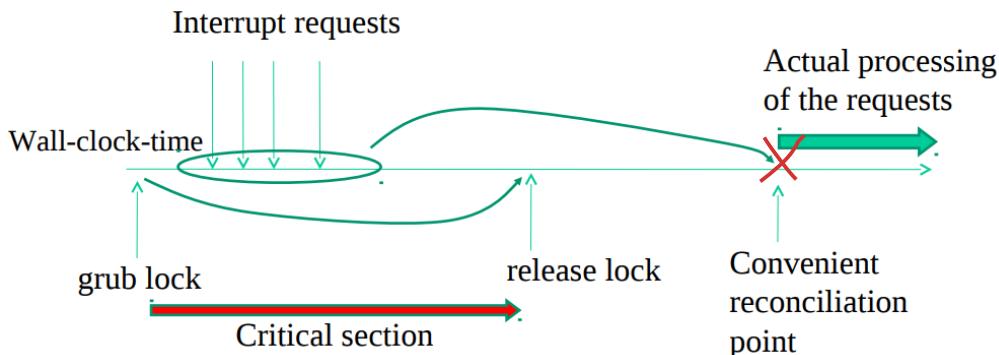
Questo ha un impatto sulle performance del sistema. Facciamo un esempio: supponiamo che due thread A, B debbano accedere a una stessa risorsa condivisa, e supponiamo che A entri in sezione critica (e.g. acquisendo uno spinlock) prima di B. Se A viene colpito da un interrupt e passa ad eseguire l'handler mentre si trova ancora in sezione critica, le attività di B verranno ritardate per effetto dell'allungamento della sezione critica di A.

### Riconciliazione temporale

È una metodologia che è possibile adottare per gestire gli interrupt in maniera efficiente.

Idea: quando sopraggiunge un interrupt, questo non viene processato subito, bensì la sua gestione subisce uno shift temporale: tale fenomeno è detto **work deferring**. Questo può anche portare a un'aggregazione del processamento di più interrupt. Inoltre, è possibile assegnare delle priorità agli interrupt per stabilire l'ordine con cui dovranno essere gestiti. **Quando accetto interrupt, è perché c'è commit su istruzione, la quale determina che tutto ciò, precedentemente a lei nella pipeline, è stato eseguito, e tutto quello successivo a lei, viene squashato.**

Vediamo uno schema:



**NB:** il thread che esegue il gestore degli interrupt può anche essere diverso da quello (o da quelli) colpito dagli interrupt, ed eventualmente viene scelto in base alle convenienze.

La "X" rossa è il punto di riconciliazione temporale, dove processo le richieste di interrupt fatte prima. Ma il dispositivo che le ha richieste, cosa fa? Posso controllare il suo stato? No, perchè faccio tutto dopo. Nel momento delle richieste, qualcosa deve comunque avvenire.

Ma come vengono scelti i **punti di riconciliazione** (i.e. gli istanti temporali in cui vengono gestiti gli interrupt pendenti)?

-> L'istante in cui **si ritorna da una system call**: normalmente non esistono sezioni critiche all'interno delle quali si passa dall'esecuzione in modalità kernel all'esecuzione in modalità user. Tuttavia, questo tipo di punto di riconciliazione prevede che all'interno del sistema operativo siano in esecuzione delle applicazioni

di livello user, e non è una cosa da dare per scontata.

-> L'istante in cui si ha un **context switch**: nel momento in cui un thread A rilascia la CPU a vantaggio di un altro thread B, normalmente non possono esservi delle informazioni bloccate per conto del thread A; in altre parole, in un kernel convenzionale, i thread non rilasciano mai la CPU durante una sezione critica. Per essere sicuri che esista sempre almeno un thread attivo che ogni tanto rilascia la CPU, all'interno del sistema deve esistere sempre il cosiddetto **idle-process**, la cui unica utilità è appunto causare dei context switch.

-> L'istante in cui un determinato thread T, che è stato scelto (o creato apposta) per eseguire l'handler degli interrupt, viene schedulato in CPU (**riconciliazione in process-context**).

L'idea del "fare tutto dopo" richiede comunque attività minimali, altrimenti bloccherei tutto! Vediamo una soluzione:

### **Programmazione top-half / bottom-half**

In realtà, non è pensabile isolare completamente l'arrivo di un interrupt dalla sua gestione effettiva. Per questo motivo, storicamente è stato adottato un approccio noto come **top-half / bottom-half**, secondo cui la *gestione degli interrupt avviene in due livelli logici distinti* (top-half e bottom-half, appunto).

Idea: quando un thread viene colpito da un interrupt deve comunque eseguire delle attività il prima possibile (e.g. inviare un acknowledgement al dispositivo che ha lanciato l'interrupt ma soprattutto schedulare il bottom-half), e queste attività vengono poste nel top-half (*facendo comunque modo che nel top-half vi sia il minor numero di istruzioni macchina possibile*); tutta la parte restante della gestione dell'interrupt viene posta nel bottom-half il quale verrà eseguito in un punto di riconciliazione.

Tipicamente (ma non obbligatoriamente), il top-half viene eseguito in modo non interrompibile.

#### **Bottom-half queue:**

Costituisce il supporto architettonico della programmazione top-half / bottom-half. In particolare, il top-half, per schedulare il bottom-half corrispondente, lo pone all'interno della **bottom-half queue** assieme a tutte le informazioni necessarie affinché l'esecuzione del bottom-half stesso avvenga correttamente.

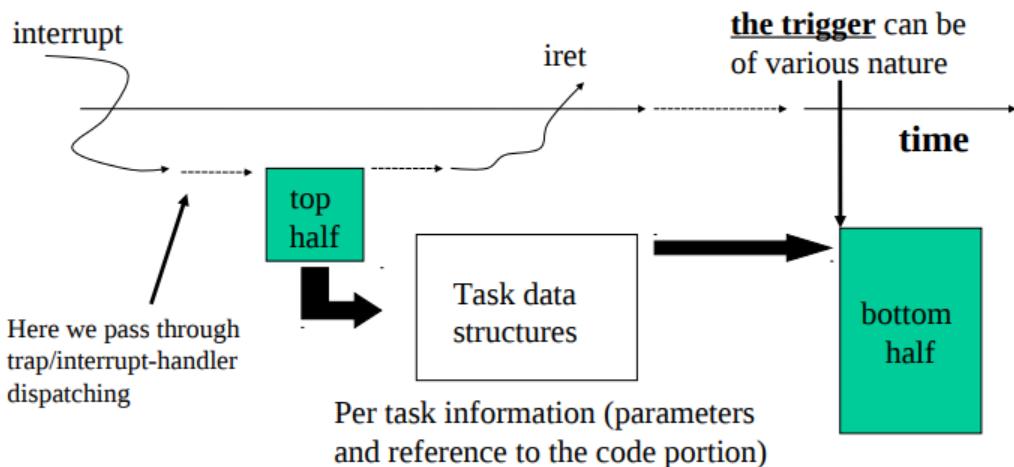
Accodiamo informazioni per il punto di riconciliazione, in particolare blocco bottom-half. Normalmente struttura dati coda, con cui identifico dati e bottom-half.

### **Evoluzione storica della gestione degli interrupt**

Per quanto concerne la gestione degli interrupt, si ha come spartiacque la versione 2.5 del kernel Linux:

- Prima del kernel 2.5 si utilizzavano esclusivamente le **task queue**.
- A partire dal kernel 2.5, al posto delle task queue, sono state introdotte le **softIRQ**, le **tasklet** e **work queue**.

## The historical architectural concept - bottom-half queues



### Task queue

Non è altro che una coda di bottom-half, in cui vengono accodate delle informazioni comprendenti le attività da eseguire per gestire gli interrupt (i.e. i bottom-half) e i parametri da dare in input a queste attività. [Non possiamo in un top\\_half eseguire un bottom\\_half](#).

In realtà, anche nelle versioni più ancestrali del kernel, ne esiste più di una:

- `tq\_immediate`: coda contenente i task che devono essere eseguiti con la cadenza degli interrupt da timer oppure quando si ritorna da una system call. [Cioè quando abbandoniamo la cpu](#).
- `tq\_timer`: coda contenente i task che devono essere eseguiti con la cadenza degli interrupt da timer ma non necessariamente quando si ritorna da una system call.
- `tq\_schedule`: coda contenente i task che devono essere eseguiti in process-context (ovvero quando viene schedulato un thread che deve occuparsi dell'esecuzione dei bottom-half).

È anche possibile definire nuove task queue mediante la macro **DECLARE\_TASK\_QUEUE(queuename)**.

Vediamo nel dettaglio come sono fatti gli elementi che compongono una task queue:

```
struct tq_struct {
    struct tq_struct *next;           //gli elementi costituiscono una lista collegata
    int sync;                         //deve essere uguale a 0 affinché l'elemento sia gestito correttamente cioè buffer riusabile.
    void (*routine) (void *);         //funzione del bottom-half che deve essere eseguita
    void *data;                        //argomento alla funzione del bottom-half
}
```

Si "vede" che è roba vecchia, perchè le aggiunte, non in testa, richiedono tempi  $O(N)$  per l'attraversamento.

### API per le task queue:

- > `int queue\_task (struct tq\_struct \*task, task\_queue \*list)`: inserisce un nuovo task all'interno di una data task queue (list).
- > `void run\_task\_queue (task\_queue \*list)`: esegue tutti i task appartenenti a una data task queue e li sgancia da quella coda.
- > `int schedule\_task (struct tq\_struct \*task)`: schedula un task in modo tale che verrà processato in un punto di riconciliazione successivo. Definiamo un task di default.

**NB:** le task queue predefinite sono gestite (e quindi flushate) dal kernel, ad esempio quando si giunge a un punto di riconciliazione. Le altre task queue, però, necessitano il controllo del programmatore, per cui richiedono l'invocazione esplicita a `run\_task\_queue()` affinché i relativi task vengano eseguiti. In ogni caso,

il programmatore può scegliere liberamente in quale task queue (predefinita o meno) inserire eventuali task.

### Esecuzione dei bottom-half:

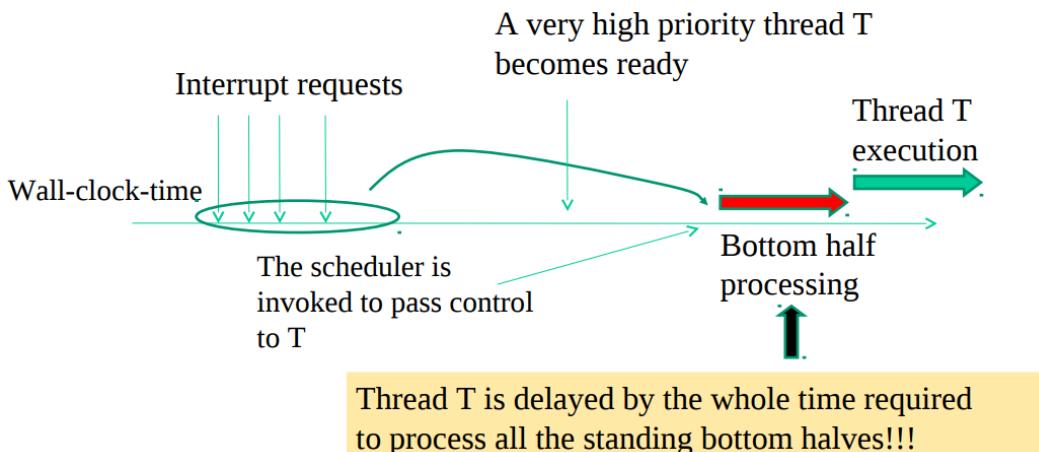
Prima del kernel 2.5, quando erano in vigore le task queue, l'invocazione dei bottom-half da parte del kernel (che avveniva mediante la funzione `do\_bottom\_half()`) occorreva ogni volta venisse eseguita la funzione `schedule()` (i.e. venisse invocato lo scheduler) o la funzione `ret\_from\_sys\_call()` (i.e. si ritornasse da una system call).

Attenzione però: supponiamo di avviare l'esecuzione dei bottom-half  $f_1, f_2, \dots, f_n$  contestualmente alla funzione `schedule()`. Questo vuol dire che  $f_1, f_2, \dots, f_n$  devono essere eseguiti mentre un thread A sta rilasciando la CPU a vantaggio di un altro thread B e, quindi, devono essere presi in carico o dal thread A o dal thread B. Supponiamo ora che qualcuno di questi bottom-half invochi un qualche servizio bloccante: a questo punto si apre la possibilità per il thread B di essere deschedulato a vantaggio di un altro thread C perché è andato in stato di blocco per mezzo del gestore di un interrupt (e questo senza che B abbia svolto alcuna delle sue attività vere e proprie). Siamo dunque andati incontro a una variazione non voluta delle politiche di scheduling e delle priorità dei thread.

**Per ovviare a questo inconveniente, è necessario che i bottom-half comprendano esclusivamente servizi non bloccanti.**

C'è un altro problema: è possibile che, nel cambio di contesto, il thread (B) che va in stato di esecuzione mentre un altro thread (A) rilascia la CPU sia un thread real-time ad alta priorità che deve portare a termine le proprie operazioni con una certa urgenza. Il fatto che i bottom-half vengono eseguiti durante il cambio di contesto ritarda anche di molto l'esecuzione delle operazioni vere e proprie di B, specialmente se i bottom-half pendenti sono numerosi. *Cioè ritardo il thread critico perché devo processare i bottom-half.*

Tale scenario è riportato nel seguente schema:



La causa primigenia dei problemi appena descritti è il fatto che l'esecuzione di tutti i bottom-half (e, quindi, la gestione di tutti gli interrupt) è demandata allo stesso thread, sia nel caso in cui si ha un context switch, sia nel caso in cui si ritorna da una system call. Infatti, questo comporta l'impossibilità di sfruttare la presenza di molteplici CPU-core per la gestione degli interrupt e l'impossibilità di ottimizzare la località delle operazioni e degli accessi ai dati associati agli interrupt (vedi l'asimmetria delle architetture NUMA). *Stiamo localizzando tutti i bottom-half su un unico nodo NUMA. Non controllo né località della attività, né parallelismo.*

Di conseguenza, abbiamo bisogno di:

- Una maggiore scalabilità.
- Una maggiore flessibilità nell'utilizzo delle code. (voglio che avvenga su nodo Numa x, non y né z).
- Una maggiore reattività e predicitività per le attività da svolgere per gestire gli interrupt.

È proprio per questo motivo che, a partire dal kernel 2.5, si è adottato un meccanismo differente per la gestione degli interrupt (in particolare introducendo le softIRQ, le tasklet e work queue).

### Architettura softIRQ

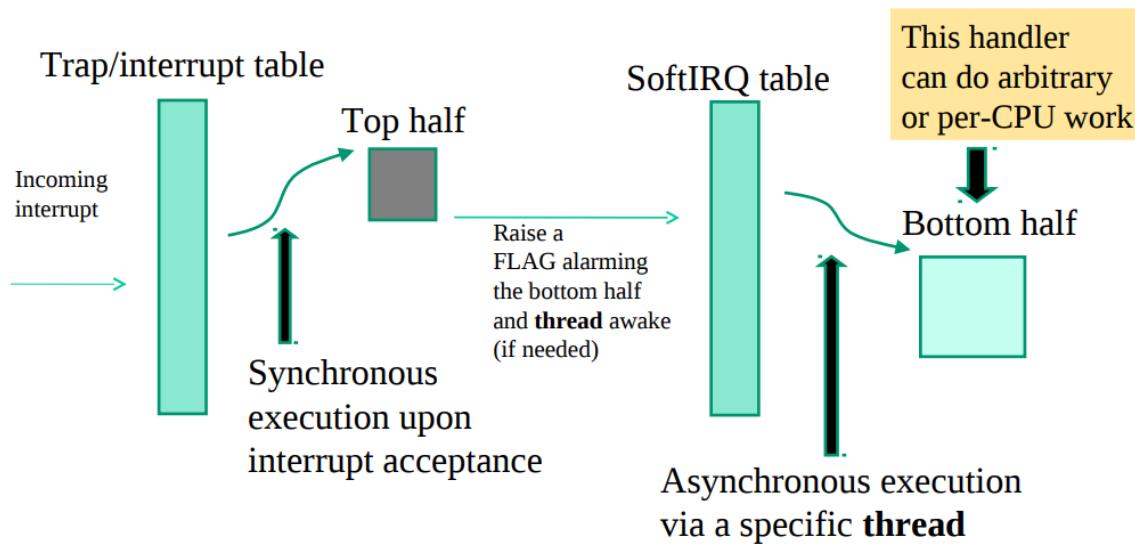
#### Idea:

La dimensione dei top-half viene ridotta ulteriormente. Non devo inserire io in coda qualcosa, so già cosa dovrà essere fatto. Top-half e bottom-half sono quindi noti, se li punto, basta partire dal top-half e flaggare il bottom-half. Non c'è più accodamento.

Ora non è più detto che i top-half debbano inserire in una coda i rispettivi bottom-half: se ciò non avviene, vuol dire che i bottom-half esistevano già da prima da qualche parte e i top-half si limitano a indicare che un determinato bottom-half dovrà essere eseguito (ma appunto non devono crearlo o inserirlo in una coda).

A differenza di prima, non si hanno più delle vere e proprie code di bottom-half, bensì delle code di informazioni che devono essere processate dai bottom-half. Quindi bottom-half, quando parte, potrebbe essere collegato ad una coda, per vedere le cose da fare. Quindi è ancora implementabile, se serve. Tale coda è detta **Tasklets**, accodiamo informazioni di input per il bottom-half, ma non accodiamo più bottom-half.

Vediamo il funzionamento nel dettaglio: quando arriva un interrupt, mediante IDT, viene attivato il relativo top-half che va a lavorare nella **softIRQ table**: in particolare, imposta a 1 un flag di un elemento di questa tabella per segnalare la presenza dell'interrupt e poi sveglia un thread detto **softIRQ daemon**. Quest'ultimo analizza la softIRQ table, vede qual è l'elemento col flag pari a 1 e va a invocare la funzione associata a quell'elemento; tale funzione non è altro che il bottom-half corrispondente all'interrupt che è arrivato.



Questo meccanismo permette di eseguire più bottom-half in parallelo: di fatto, possono esserci più **softIRQ daemon** che vengono svegliati concorrentemente e vengono schedulati su due CPU differenti per svolgere in modo parallelo il loro lavoro. Oggi, ce n'è 1 per cpu, risvegliamo quello associato alla cpu in run. Tale demone ha priorità importante, ma non è la più importante.

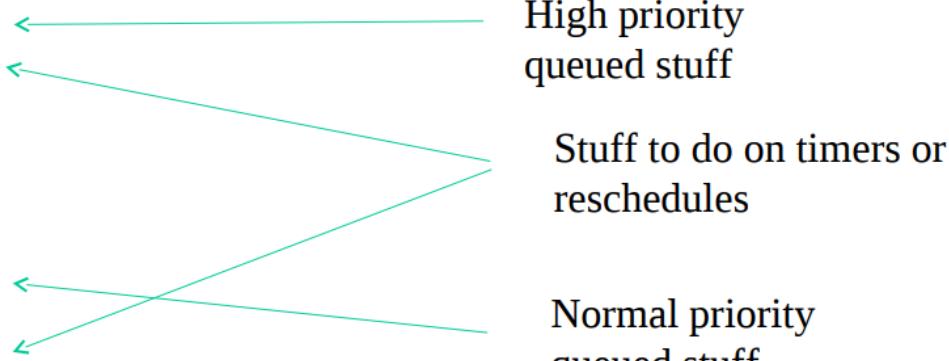
Inoltre, la softIRQ table è diversa dall'interrupt table: è possibile che due top half differenti (attivati da due interrupt differenti) vadano poi a lavorare sulla medesima entry della softIRQ table poiché magari richiedono l'esecuzione dello stesso bottom-half.

#### Struttura della softIRQ table:

Ciascuna entry della softIRQ table è associata a un particolare valore che indica un livello di priorità (dove 0 corrisponde alla priorità massima). I softIRQ daemon, quando vengono svegliati, controllano prima le entry

con priorità migliore e dopo passano alle entry con priorità via via peggiore finché non ne incontrano una col flag settato a 1. I vari livelli di priorità sono definiti in una enum e sono riportati qui di seguito:

```
enum {      HI_SOFTIRQ=0,
            TIMER_SOFTIRQ,
            NET_TX_SOFTIRQ,
            NET_RX_SOFTIRQ,
            BLOCK_SOFTIRQ,
            BLOCK_IOPOLL_SOFTIRQ,
            TASKLET_SOFTIRQ,
            SCHED_SOFTIRQ,
            HRTIMER_SOFTIRQ,
            RCU_SOFTIRQ,
            NR_SOFTIRQS }
```



Badiamo che si ha un softIRQ daemon per ogni CPU-core (inteso anche come hyperthread fisico); più precisamente, ciascun softIRQ daemon ha affinità con un unico CPU-core e viene indicato con `ksoftirq[n]`, dove `n` è l'indice del CPU-core con cui è affine. Per giunta, tramite appositi flag, è possibile stabilire se un softIRQ daemon può processare un determinato bottom-half oppure no. In tal modo, possiamo creare delle affinità tra le softIRQ (i.e. i bottom-half) e i CPU-core; non solo: i vari softIRQ daemon possono processare i bottom-half senza necessità di sincronizzarsi tra loro. Gli interrupt da timer sono più di 1. [Abbiamo API per `HRTIMER\\_SOFTIRQ`](#), ad esempio per assegnazione di CPU per un certo quanto di tempo.

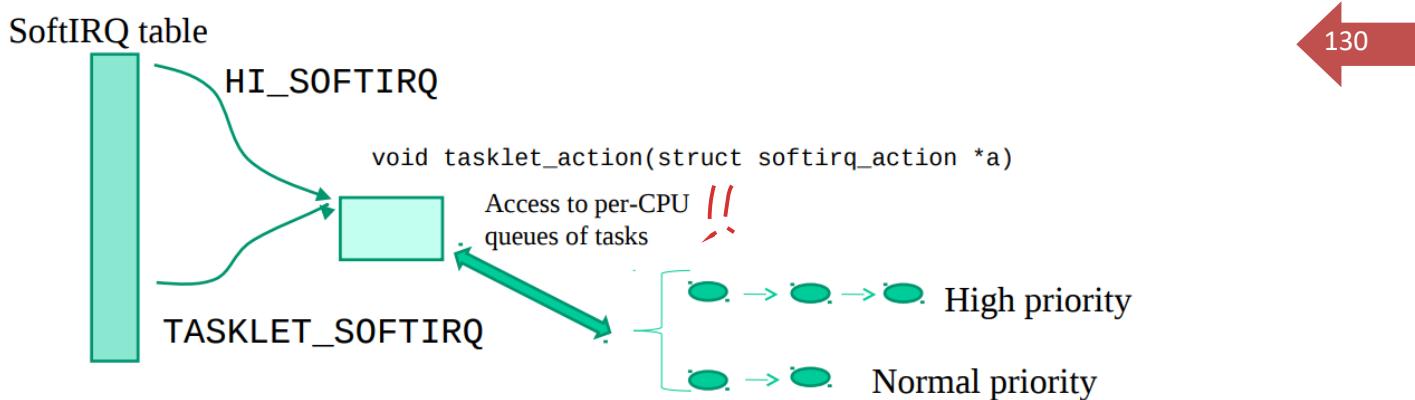
#### Vantaggi dell'architettura softIRQ:

- Consente l'esecuzione multi-thread (parallela) di molteplici task bottom-half. (Grazie ai demoni per-cpu).
- L'esecuzione dei bottom-half non è sincrona con la schedulazione dei thread: ad esempio, se si ha un thread real-time critico (che quindi ha una priorità superiore rispetto ai softIRQ daemon), questo va sempre in esecuzione senza ritardi a prescindere dai bottom-half che sono pendenti.
- Il fatto che possiamo effettuare il binding tra i task da eseguire e i CPU-core incentiva la località delle operazioni e degli accessi ai dati associati agli interrupt all'interno delle architetture NUMA.
- Rimane comunque la possibilità per il programmatore di accodare task per gestire gli interrupt, e a tal proposito è possibile utilizzare alcuni dei bottom-half presenti nell'architettura, come gli `HI_SOFTIRQ` e i `TASKLET_SOFTIRQ`.

#### Tasklet

Sono le strutture dati che mantengono quei task (i quali alla fine non sono altro che normalissime funzioni che gestiscono interrupt) che, all'interno dell'architettura softIRQ, possono essere definiti (e accodati) dal programmatore. Si tratta di task che possono appartenere a due differenti livelli di priorità (`HI_SOFTIRQ` con priorità alta e `TASKLET_SOFTIRQ` con priorità normale) ma, ciononostante, vengono processati utilizzando un unico bottom-half, il quale va dentro una struttura di tasklet per verificare se c'è un qualche task da processare. Tale struttura di tasklet è composta da due code di cui, appunto, una ad alta priorità (`HI_SOFTIRQ`) e una a priorità normale (`TASKLET_SOFTIRQ`). Anche queste due code sono per-CPU, per cui il bottom-half che le governa non deve sincronizzarsi con nessun altro per selezionare il task da mandare in esecuzione.

Nella pagina seguente è riportato uno schema di ciò che è stato appena descritto.



Le funzioni implementate dai task (i.e. dalle tasklet) possono accettare un unico puntatore come parametro e devono ritornare void: di fatto, trattandosi di handler di interrupt, non hanno da restituire niente a nessuno. Il limite delle queues per cpu è dato dal fatto che vanno bene solo se il thread in esecuzione è lo stesso che è interessato a tali queues, non c'è l'idea di poter toccare le code di un'altra cpu.

### **API per le tasklet:**

- > `DECLARE\_TASKLET (tasklet, function, data)`: dichiara una tasklet dal nome indicato dal primo parametro (tasklet) e che punta a una certa funzione function che accetta ‘data’ come argomento.
  - > `DECLARE\_TASKLET\_DISABLED (tasklet, function, data)`: dichiara una tasklet **disabilitata**, nel senso che non può essere correntemente processata. Questo vuol dire che è possibile pre-accodare delle informazioni all’interno della struttura di tasklet per poi marcarle come “attive” in un secondo momento (in modo tale che l’attivazione di una qualche tasklet venga fatta in maniera efficiente; questo è molto importante ad esempio quando vengono implementati i device driver). **NB: Tasklet alla fine è un function pointer.**
  - > `tasklet\_enable (struct tasklet\_struct \*tasklet)`: abilita una particolare tasklet.
  - > `tasklet\_disable (struct tasklet\_struct \*tasklet)`: disabilita una particolare tasklet. **Quindi il demone non lo processa, ma in un secondo momento potrò abilitarlo e permettere il processamento. Il vantaggio è che così ce le ho già pronte.**
  - > `tasklet\_disable\_nosynch (struct tasklet\_struct \*tasklet)`: disabilita una particolare tasklet in maniera asincrona. **Se tasklet è in processing, e funzione è stata chiamata, la disabled normale è bloccante, aspetterei. Con questa, non aspetto, se fallisco ritorna errore.**
  - > `void tasklet\_schedule (struct tasklet\_struct \*tasklet)`: inserisce una particolare tasklet in fondo alla coda con priorità normale.
  - > `void tasklet\_hi\_schedule (struct tasklet\_struct \*tasklet)`: inserisce una particolare tasklet in fondo alla coda con priorità alta.
  - > `void tasklet\_hi\_schedule\_first (struct tasklet\_struct \*tasklet)`: inserisce una particolare tasklet in testa alla coda con priorità alta (per cui si tratta di una tasklet assolutamente prioritaria).

## The tasklet init function

NB: se una tasklet è stata schedulata su uno specifico CPU-core, allora verrà certamente eseguita dal softIRQ daemon affine a quel CPU-core: in altre parole, non può essere migrato verso un altro CPU-core. **Non possiamo bloccare un demone!** Quindi no servizi bloccanti nella coda di tasklet.

### Work queue

La versione 2.5.41 del kernel Linux ha rimpiazzato definitivamente le task queue con le **work queue**. In particolare:

- Gli utilizzatori delle code tq\_immediate (come i driver) hanno dovuto switchare alle tasklet. [Processiamo attività non bloccanti, con un demone, senza troppi problemi.](#)
- Gli utilizzatori delle code tq\_timer sono dovuti passare all'utilizzo diretto dei timer (che approfondiremo in seguito). [Ovvero usano software-queue architecture.](#)
- In tutti i casi in cui le tasklet e i timer sono inappropriati, può essere usata l'interfaccia **schedule\_work()**, che permette di utilizzare le work queue. In particolare, accoda il lavoro da svolgere (i task) verso un apposito demone (chiamato **kworker**); l'esecuzione vera e propria avverrà in modo deferred nel momento in cui il demone sarà schedulato in CPU.

La particolarità della work queue sta nel fatto che, mentre viene processata, gli interrupt continuano a essere abilitati; inoltre, le funzioni chiamate da una work queue possono invocare dei servizi bloccanti (cosa assolutamente vietata per le tasklet), ma ciò rimane comunque sconsigliato.

#### API per le work queue:

- > **schedule\_work (struct work\_struct \*work)**: schedula il lavoro (il task) descritto dalla struct work passata come parametro in input; il lavoro chiaramente viene inserito in una work queue di sistema già esistente. [La work-queue è multi-queue, avente zone su diverse CPU gestite da demoni diversi. Noi inseriamo nella coda in cui abbiamo avviato l'API.](#)
- > **schedule\_work\_on (int cpu, struct work\_struct \*work)**: schedula il lavoro descritto dalla struct work sulla CPU specificata. [Quindi specifico nodo NUMA.](#)
- > **INIT\_WORK (&var\_name, function-pointer, &data)**: inizializza un nuovo lavoro che dovrà essere poi schedulato. Il parametro &var\_name è un puntatore alla struttura che descrive il task, function-pointer è la funzione che deve essere eseguita per portare a termine il task, mentre il parametro &data è un puntatore all'argomento della funzione.
- > **struct workqueue\_struct \*create\_workqueue (const char \*name)**: crea una nuova work queue. [Ritorna il suo puntatore, che posso usare per passare elementi, un thread per CPU. Esiste probabilità di blocco, ad esempio se i thread che devono acquisire risultano bloccati.](#)
- > **struct workqueue\_struct \*create\_singlethread\_workqueue (const char \*name)**: crea una nuova work queue all'interno della quale un solo thread per volta può processare le attività. [Crea multi-coda con un solo demone, che si farà "il giro" di tutta la multicoda. Garantisce la sequenzializzazione, no concorrenza.](#)
- > **void destroy\_workqueue (struct workqueue\_struct \*queue)**: elimina la work queue specificata come parametro.
- > **int queue\_work (struct workqueue\_struct \*queue, struct work\_struct \*work)**: schedula il lavoro descritto dalla struct work inserendolo all'interno della work queue specificata come parametro.
- > **int queue\_delayed\_work (struct workqueue\_struct \*queue, struct work\_struct \*work, unsigned long delay)**: schedula il lavoro descritto dalla struct work inserendolo all'interno della work queue specificata come parametro; tuttavia, l'esecuzione del task può avvenire solo dopo che è trascorso un intervallo di tempo definito dal parametro delay.
- > **int cancel\_delayed\_work (struct work\_struct \*work)**: elimina un job (lavoro) pendente dalla work queue a cui apparteneva.
- > **void flush\_workqueue (struct workqueue\_struct \*queue)**: effettua il flush dell'intera work queue specificata come parametro eseguendone tutti i job.

#### **Creazione dei thread responsabili delle work queue:**

Fino al 2010, i demoni responsabili delle work queue venivano creati mediante le API che servono per

istanziare una nuova work queue. In particolare, nel caso di work queue multi-thread, veniva istanziato un nuovo thread per ogni CPU, mentre nel caso di work queue single thread, veniva istanziato un unico thread. Prendiamo in considerazione le code multi-thread e teniamo conto del fatto che, col tempo, si è iniziato a fare un uso sempre più massivo delle work queue: soprattutto nelle macchine con un numero alto di CPU-core / hyperthread (e.g. 1024) questo ha iniziato a rappresentare un problema enorme, nel momento in cui venivano istanziati NxM thread solo per la gestione delle work queue, dove N è il numero di CPU-core / hyperthread ed M è il numero di work queue istanziate. Di fatto, le conseguenze erano le seguenti:

- Veniva utilizzata moltissima memoria per la creazione dei thread e si poteva andare incontro a un esaurimento dei PID già prima che potessero essere eseguiti dei thread in user space.
- Un numero così elevato di thread concorrenti poteva far sì che i meccanismi di sincronizzazione non funzionassero più bene e, quindi, si verificassero dei deadlock.
- Anche se più job dovevano essere eseguiti su una stessa CPU, solo per il fatto che appartenevano a work queue differenti dovevano essere presi in carico da thread diversi. Di conseguenza, era necessario un context switch per passare da un job all'altro e questo, considerando il fatto che spesso i job (quindi le funzioni da eseguire) sono molto piccoli, causava un rallentamento inaccettabile: in effetti, il tempo impiegato per portare a termine i job era sensibilmente ridotto al tempo totale di esecuzione (comprendente anche i context switch).

A valle di queste considerazioni, nel 2010 è stata eliminata la possibilità di creare thread contestualmente alla creazione di work queue, per cui le API `create_workqueue()` e `create_singlethread_workqueue()` sono state rimpiazzate rispettivamente da `alloc_workqueue()` e `alloc_ordered_workqueue()`. ([quest'ultima processata in maniera sequenzializzata](#)).

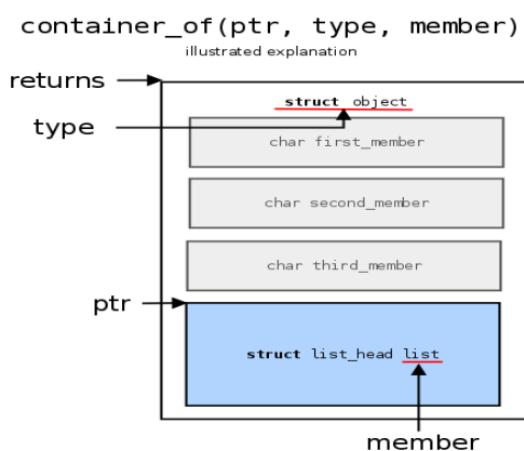
#### Work queue concurrency managed:

A partire dal 2010, dal momento in cui non vengono più esplicitamente creati nuovi thread assieme alla creazione delle code, le work queue sono **concurrency managed**. In particolare, esistono dei pool di thread di sistema per-CPU la cui unica responsabilità è governare le work queue.

I pool di thread sono dinamici: il numero di thread viene dinamicamente incrementato o decrementato in base alla quantità di lavoro pendente all'interno delle work queue.

#### Container\_of():

La routine `container\_of (ptr, type, member)` non è utile solo per le strutture dati relative alle work queue ma anche in generale, e serve per ottenere un puntatore alla struttura dati che contiene il campo di cui viene specificato l'indirizzo. Il parametro `ptr` è l'indirizzo del campo che conosciamo, `member` è il nome del campo che conosciamo e `type` è il tipo di struttura di cui vogliamo ottenere l'indirizzo base.



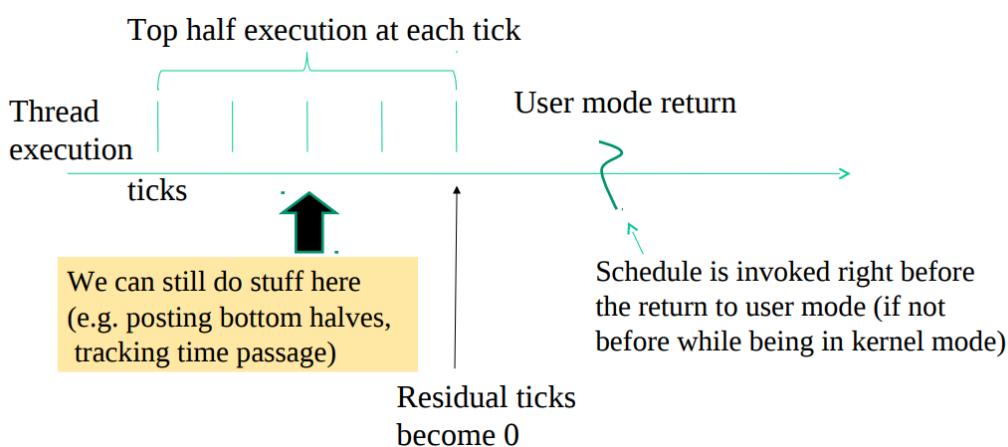
<http://www.linuxwell.com>

NB: esiste Randomized Layout, che non ci permette di trovare il `ptr` a “`struct list_head list`” a mano.

## Interrupt da timer vs CPU scheduling

Quando scade il quanto di tempo che un thread in CPU aveva a disposizione prima di essere deschedulato, un interrupt da timer colpisce quel thread. Ma ora sappiamo benissimo come funzionano gli interrupt, per cui possiamo immaginare che il thread non viene immediatamente schedulato quando arriva l'interrupt da timer. Piuttosto, viene eseguito un top-half minimale che si limita a registrare in un'apposita struttura dati in memoria il fatto che il thread colpito deve essere deschedulato appena possibile; il rilascio vero e proprio della CPU è implementato all'interno del bottom-half e viene eseguito con una logica deferred.

In realtà, ciascun thread non è colpito da un interrupt da timer solo quando deve essere deschedulato, bensì al passaggio di ogni tick. In particolare, a ogni tick, il top-half controlla il valore di "residual ticks" (= quanto di tempo residuo) e lo decrementa, per cui va a segnalare la necessità di deschedulare il thread solo se la condizione "residual ticks == 0" è verificata. [Parliamo di safe places, ad esempio quando un thread ritorna user, cosa che al kernel non interessa più](#). Tramite `schedule()`, dico che io posso lasciare la cpu.



NB: lo schema top-half / bottom-half non è valido quando si parla di preemption dovuta a cause non correlate al passaggio del tempo.

Se sto in sezione critica, e mi arriva interrupt da timer? Nelle vecchie versioni, bloccavo gli interrupt, eseguo, e poi riabilito. Oggi non si può fare, se bloccassi interrupt da timer, non esistono più bottom-half, starei fermando l'architettura. Oltretutto, potrebbe essere richiesto un interrupt timer, quindi in qualche caso sarei limitato.

Badiamo che disabilitare gli interrupt anche solo mentre il thread si trova in sezione critica è molto svantaggioso soprattutto per le macchine multicore. Infatti:

- Si perderebbero alcuni tick e, con questi, la possibilità di tenere traccia dei quanti di tempo con una precisione a grana fine.
- I CPU-core perderebbero momentaneamente la possibilità di lanciare degli interrupt verso altre CPU.

Un altro aspetto a cui bisogna stare attenti è il busy waiting dato dall'istruzione **while (!condition);** (non ha body, attendiamo qualcosa) che **bisogna assolutamente evitare quando si programma a livello kernel**. Non passerà mai in un **safe place**, dipende magari da un altro thread, in un contesto **context switch**. Infatti, è vero che un thread kernel level che fa busy waiting continua a ricevere gli interrupt da timer, ma è altrettanto vero che non va mai a considerare o eseguire i bottom-half, appunto perché è intrappolato nel ciclo while e non è in grado di fare altro; di conseguenza, il thread non può rilasciare la CPU a vantaggio di qualcun altro.

Il problema invece non si pone se l'istruzione **while (!condition);** è utilizzata a livello user, quindi preemptable. Infatti, in tal caso, quando il thread che esegue il ciclo while viene colpito dall'interrupt da timer, passa dall'esecuzione user mode all'esecuzione kernel mode ed è qui che incontra la zona di codice che controlla se ci sono dei thread in attesa di andare in CPU ed eventualmente invoca lo scheduler.

## Timer hardware implementati nei sistemi x86

Sono tutti processati con schema top-half, per controllare nel tempo quello che succede.

- **Time Stamp Counter (TSC)**: è il timer che conta il numero di clock della CPU ed è accessibile mediante l'istruzione rdtsc. A differenza degli altri due elencati di seguito, non ha nulla a che vedere con gli interrupt.
- **Local APIC TIMER (LAPIC-T)**: è un timer che può essere programmato per inviare one shot oppure periodicamente gli interrupt, e solitamente viene sfruttato per tenere traccia del tempo con la granularità del millisecondo. Tipicamente è quello che viene utilizzato per contare i tick rimanenti a ciascun thread prima che esso debba rilasciare la CPU (in altre parole è utilizzato per il time sharing).
- **High Precision Event Timer (HPTE)**: è una suite di timer che, come prima, può essere programmata per inviare one shot o periodicamente gli interrupt, e solitamente viene sfruttata per tenere traccia del tempo con la granularità del nanosecondo. Tipicamente viene usata per gestire delle attività generiche all'interno del sistema.

### LAPIC-T

Il top-half degli interrupt generati dal LAPIC-T esegue le azioni elencate di seguito:

- Marca la task `queue tq\_timer` come pronta per essere flushata (almeno nelle versioni più ancestrali del kernel Linux); tale coda verrà effettivamente processata dal bottom-half.
- Incrementa di un'unità la variabile globale **volatile unsigned long jiffies**, che tiene traccia del numero di tick che sono trascorsi da quando gli interrupt sono abilitati.
- Esegue altro lavoro minimale relativo al passaggio del tempo.
- Verifica se il CPU scheduler deve essere attivato in base a se il thread corrente ha ancora dei tick residui o meno; se lo scheduler deve effettivamente essere attivato, il top-half va ad attivare il flag **need\_resched** all'interno del TCB (Thread Control Block) del thread corrente. Tale flag viene controllato ogni volta che si passa per un punto di riconciliazione e, se il controllo ha esito positivo, viene invocata la funzione `schedule()` per effettuare il cambio di contesto.

Nelle versioni più recenti del kernel, anziché lavorare sulla task queue tq\_timer, il top-half dell'interrupt da timer va ad attivare il flag del task **TIMER\_SOFTIRQ** all'interno della softIRQ table, in modo tale che venga risvegliato un demone che si occuperà poi di effettuare la gestione vera e propria dell'interrupt.

### High Resolution (HR) Timer

Il top-half degli interrupt generati dagli HR Timer esegue le azioni elencate di seguito:

- Attiva il flag del task **HRTIMER\_SOFTIRQ** all'interno della softIRQ table.
- Programma l'istante in cui dovrà venire il prossimo interrupt associato all'HR timer; di fatto, a differenza degli interrupt associati al LAPIC-T, quelli relativi all'HR timer colpiscono il thread in CPU a intervalli non regolari.
- Riporta all'interno del TCB del thread corrente (o in una struttura dati a esso associato) l'informazione per cui il thread stesso deve rilasciare la CPU appena possibile; in altre parole va a richiedere la preemption del thread corrente. [Il thread, quando esegue in ogni punto del kernel che è safe place, la cpu va a qualcun altro con priorità più alta, cioè un demone.](#)

### `usleep()`:

È una system call che permette a un thread di andare a dormire per una quantità di tempo a grana più fine rispetto alla system call `sleep()` (quindi a grana più fine rispetto al millisecondo). Infatti, qui il tempo trascorso a dormire non viene calcolato sulla base del LAPIC-T, bensì sulla base di un HR Timer.

Andando più nel dettaglio, quando viene invocata la `usleep()` vengono seguiti questi step:

- Il thread chiamante passa a eseguire in modalità kernel.
- Il kernel inserisce una richiesta con HR Timer all'interno del log gestito dal softIRQ daemon.
- Viene invocato lo scheduler per passare il controllo a un altro thread t (qui ha inizio la sleep).

- Quando l'HR Timer scade, il thread t viene deschedulato dalla CPU appena possibile per far di nuovo spazio al thread che era andato a dormire.

#### API per gli HR Timer:

- `ktime\_t ktime\_set (long secs, long nanosecs)` : definisce una quantità di tempo con la granularità dei nanosecondi.
- > `void hrtimer\_init (struct hrtimer \*timer, clockid\_t which\_clock, enum hrtimer\_mode mode)` : inizializza un HR Timer, dove: struct hrtimer \*timer è una struttura che specifica la funzione che dovrà essere eseguita dopo la scadenza del timer con il relativo argomento; clockid\_t which\_clock specifica il meccanismo di clock da utilizzare ([MONOTONIC è locale, REALTIME rispetto a tutte le altre macchine del mondo, cioè ntp protocol, tempo può tornare indietro](#)); enum hrtimer\_mode mode specifica la modalità con cui si vogliono specificare i tempi.
- > `int hrtimer\_start (struct hrtimer \*timer, ktime\_t time, enum hrtimer\_mode mode)` : fa partire un HR Timer inizializzato in precedenza specificando la quantità di tempo che deve passare prima che il timer stesso scada.
- > `int hrtimer\_cancel (struct hrtimer \*timer)` : elimina un HR Timer in maniera bloccante. [Non è detto che l'interrupt sia arrivato, quindi se la funzione processata dal daemon è attiva, non posso farci nulla.](#)
- > `int hrtimer\_try\_to\_cancel (struct hrtimer \*timer)` : elimina un HR Timer in maniera non bloccante.

#### Preemption dei thread

Finora siamo stati abituati a pensare che un thread T possa andare in preemption se va in sleep oppure se scade il suo quanto di tempo di utilizzo della CPU. In realtà c'è anche un terzo caso: il thread T può decidere spontaneamente di essere deschedulato dalla CPU. Ad esempio, questo può accadere se T è un thread a bassa priorità e, a un certo punto, un altro thread T' real-time ad altissima priorità entra nello stato ready. In tale evenienza, un interrupt può colpire la CPU dove T è in esecuzione e, con una logica top-half / bottom-half, T imposta a 1 un apposito flag in una struttura dati per-thread. Quando viene raggiunto un **safe place** (i.e. un punto sicuro in cui è possibile rilasciare la CPU senza conseguenze), viene effettuato un check su tale flag e, se è settato a 1, viene invocato il CPU scheduler. I safe place sono implementati all'interno di API come `printk()`, `ret_from_sys_call()` e molte altre.

Comunque sia, è possibile evitare la preemption quando si passa per i safe place sfruttando il cosiddetto **preemption counter**, che è una variabile per-thread. In particolare, quando il preemption counter è maggiore di zero, significa che il thread corrente sta eseguendo delle attività per cui, secondo il programmatore, non dovrebbe essere deschedulato dalla CPU. Questo contatore viene quindi controllato assieme al flag ogni volta che si attraversa un safe place.

#### API per il preemption counter:

- > `preempt_enable()`: decrementa di un'unità il preemption counter e controlla subito se è diventato pari a 0, cosicché, eventualmente, il thread chiamante venga deschedulato dalla CPU. [E' safe place.](#)
- > `preempt_disable()`: incrementa di un'unità il preemption counter.
- > `preempt_enable_no_resched()`: decrementa di un'unità il preemption counter ma senza controllare subito se è diventato pari a 0. [Non è safe place.](#)
- > `preempt_check_resched()`: controlla sia il flag per il de-scheduling, sia il preemption counter, cosicché, eventualmente, il thread chiamante venga deschedulato dalla CPU.
- > `preempt_count()`: restituisce il valore del preemption counter.

#### Preemption vs variabili per-CPU:

La preemption può rappresentare un problema per l'utilizzo delle variabili per-CPU: supponiamo che un thread T sia in esecuzione sulla CPU X e stia utilizzando una variabile per-CPU V, e supponiamo che a un certo punto T venga deschedulato. Se non c'è una particolare affinità tra T e la CPU X, è possibile che in futuro T venga rischedulato su un'altra CPU Y; a questo punto, l'utilizzo della variabile V potrebbe risultare inconsistente rispetto a prima della preemption, il che rappresenta un problema anche dal punto di vista

della correttezza del software eseguito da T.

Tale inconveniente viene risolto disabilitando la preemption (sfruttando il preemption counter) durante ogni esecuzione di un'API di tipo `get_cpu_var()` e riabilitandola durante ogni esecuzione di un'API di tipo `put_cpu_var()`: di fatto, una `get_cpu_var()` prende in uso la variabile per-CPU "cpu\_var", mentre una `put_cpu_var()` la rilascia. In questo modo, il thread T non può essere mai deschedulato tra l'esecuzione di una `get_cpu_var()` e l'esecuzione di una `put_cpu_var()`.

Attenzione però: se il programmatore è noob e tra la `get_cpu_var()` e la `put_cpu_var()` inserisce una chiamata a un servizio bloccante, non ci si può far nulla, e viene reintrodotto il rischio per cui il thread T venga migrato mentre sta utilizzando una qualche variabile per-CPU.

### Thread Control Block (TCB)

Talvolta chiamato anche *Process Control Block (PCB)*, è una struttura dati associata a uno specifico thread T che mantiene:

- Le informazioni che servono per lo scheduler di CPU.
- Le informazioni relative al collegamento che il thread T ha coi sottosistemi esterni al sottosistema di scheduling. TCB diversi possono anche avere dei collegamenti agli stessi sottosistemi o metadati esterni: è in questo modo che è possibile relazionare tra loro più thread; questo è utile, ad esempio, quando i thread appartengono a uno stesso processo.

### CPU-dispatchability

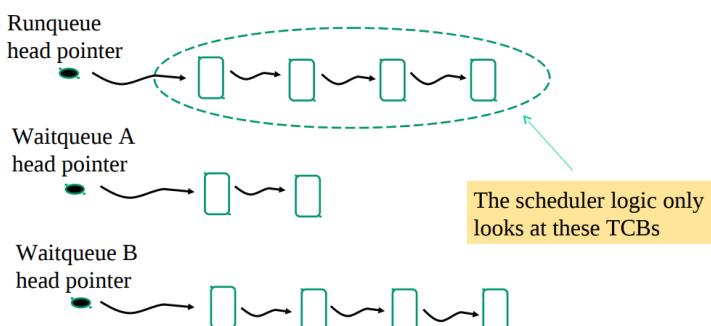
Il TCB indica, a ogni istante di tempo, se il thread può essere **CPU-dispatched**, ovvero se può essere schedulato in CPU. Più formalmente, la CPU-dispatchability è la possibilità per un thread di avere il proprio snapshot di CPU (mantenuto dal proprio TCB) installato su un processore affinché possa essere eseguito correttamente. La funzione di scheduling deve anche considerare il TCB appunto. La dispatchabilità non è determinata dalla logica di scheduling, la quale vede solo quali sono dispatchabili o meno, ma non li può rendere tali. L'oggetto che si occupa di selezionare lo snapshot di CPU da caricare su un processore si chiama **scheduler logic**; esso però non decide quali thread possono essere CPU-dispatched e quali no, bensì esistono altre entità che stabiliscono la CPU-dispatchability.

La struttura dati che mantiene tutti e soli i TCB dei thread CPU-dispatchable è una lista ed è detta **runqueue** (può anche essere multi-queue). Dunque, lo scheduler logic seleziona il thread da schedulare dopo aver scandito tale runqueue.

D'altro lato, tutti i thread non CPU-dispatchable sono mantenuti in altre liste non consultate dallo scheduler logic e chiamate **waitqueue**.

Se ho semaforo, e lo istanzio con syscall, il semaforo crea sua runqueue, non usa altre già esistenti. Dipende dai servizi usati. Le waitqueue sono diverse per ogni sottosistema. La grana deve essere finissima, anche perchè altrimenti dovrei risvegliare tantissimi thread. Le runqueue sono una multi-queue per cpu, quindi logicamente ne ha una, ma in realtà più di una, per gestire oggetti a livelli di priorità diversi.

A scheme



Le operazioni complesse in questa architettura sono il passaggio di un TCB dalla runqueue a una waitqueue e il viceversa.

Consideriamo il passaggio del TCB di un thread A dalla runqueue a una waitqueue. Inizialmente, lo scheduler logic è in esecuzione su un CPU-core all'interno del contesto del thread A. Nel momento in cui A chiama un servizio bloccante, invoca in modo sincrono lo scheduler logic per essere deschedulato, per cui il suo TCB deve ancora trovarsi sulla runqueue. Tuttavia, lo scheduler logic non dovrebbe essere in grado di selezionare il thread A per l'esecuzione in CPU, per cui non dovrebbe essere capace di vedere A nella runqueue.

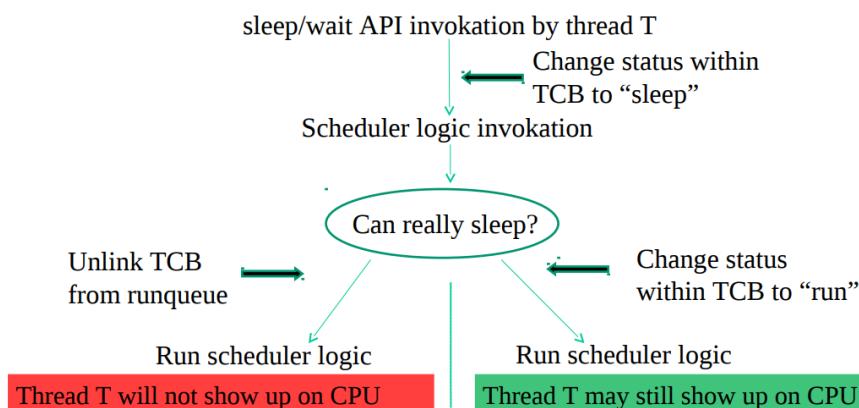
Come facciamo dunque a escludere il TCB del thread A per il processo di selezione da parte dello scheduler logic? Si ricorre ai **servizi sleep/wait di livello kernel**. Essi sfruttano il TCB per governare assieme allo scheduler logic il comportamento effettivo del thread che ha invocato il servizio bloccante. Esistono due possibili esiti dell'invocazione di questi servizi:

- Il TCB del thread viene rimosso dalla runqueue dallo scheduler logic prima che venga effettivamente selezionato il prossimo thread da schedulare in CPU.
- Il TCB del thread rimane presente nella runqueue durante la selezione del prossimo thread da schedulare in CPU. È questo il caso in cui è il thread stesso a decidere mediante lo scheduler logic chi dovrà essere schedulato al posto suo.

Analizziamo nel dettaglio gli step seguiti da un servizio sleep/wait:

- 1) Collegare il TCB del thread chiamante a una qualche waitqueue.
- 2) Marcare il thread come "sleep".
- 3) Invocare lo scheduler logic in modo tale da stabilire se il thread può veramente andare in sleep e, se la risposta dovesse essere no, il thread deve essere nuovamente marcato come "run"; tanto per fare un esempio, se il thread viene colpito da un evento, non può andare a dormire, bensì deve tornare in esecuzione appena possibile per gestire l'evento (e solo dopo può passare allo stato di sleep).
- 4) Quando la sleep termina, scollegare il TCB del thread chiamante dalla waitqueue.

### The timeline



Esistono dunque degli intervalli di tempo in cui un TCB può essere collegato sia alla runqueue che a una waitqueue: di fatto, lo scollegamento dalla runqueue può avvenire solo dopo il collegamento a una waitqueue, mentre lo scollegamento dalla waitqueue può avvenire solo dopo il collegamento alla runqueue.

In particolare, per quanto riguarda il passaggio dalla waitqueue alla runqueue:

- Lo scollegamento dalla waitqueue viene fatto direttamente dal thread che è di nuovo pronto a tornare in esecuzione.
- Il collegamento alla runqueue viene fatto da un altro thread che esegue uno dei seguenti pezzi di codice di livello kernel:
  - Servizio invocato in modo sincrono (e.g. sys\_kill).

- Top half.
- Bottom half.

### Context switch

Il processo di context switch all'interno delle CPU sicuramente include il salvataggio del contesto di CPU del thread deschedulato ( $t_1$ ) all'interno del TCB di  $t_1$  e il ripristino del contesto di CPU del thread schedulato ( $t_2$ ) dal TCB di  $t_2$ . Tuttavia, nella maggior parte delle implementazioni del kernel, noi diciamo che il context switch è realmente avvenuto solo nel momento in cui è stato installato nel processore il valore dello stack pointer del thread entrante (per cui possono anche essere cambiati i valori di tutti i registri ma, se lo stack pointer non punta allo stack del thread appena schedulato, non possiamo ancora dire che il context switch ha effettivamente avuto luogo). Cambiamo stack, cambiamo thread, cambiamo il valore di current (macro legata al valore dello stack pointer), cioè un simbolo che, se usato, punta al TCB del thread corrente, quindi deve cambiare perché diverso thread.

### Struttura del TCB in Linux

Con riferimento alla versione 2.6 del kernel, i campi principali del TCB dei thread sono riportati nella pagina seguente. Il TCB in Linux è una “task\_struct”.

```

> volatile long state
> struct mm_struct *mm
> pid_t pid
> pid_t pgrp
> struct fs_struct *fs
> struct files_struct *files
> struct signal_struct *sig
> volatile long need_resched
> struct thread_struct thread /* CPU-specific state of this task - TSS */
> long counter
> long nice
> unsigned long policy /*CPU scheduling info*/

```

- `volatile long state`: indica lo stato del thread T associato al TCB; può essere modificato sia da T stesso (ad esempio quando non vuole essere più dispatchabile a seguito dell'invocazione a un servizio bloccante) oppure da altri thread (ad esempio quando inviano un segnale a T). Si tratta quindi di un campo che può essere acceduto da chiunque e concorrentemente, per cui deve essere sempre acceduto in memoria e non solo su un qualche registro di processore: è per questo motivo che viene utilizzata la keyword *volatile*, in modo tale che il compilatore non apporti alcuna ottimizzazione su tale campo. Ci dice se siamo dispatchati o no.

- `struct mm\_struct \*mm`: è un puntatore alla **struttura di memory management (mm)** del thread T. È una struttura esterna al TCB di T, per cui anch'essa può essere acceduta tranquillamente sia da T che da altri thread.

- `volatile long need\_resched`: è quel campo del TCB che viene attivato nel momento in cui il numero di tick residui per il thread T (contati tramite il LAPIC-T) diviene pari a 0.

- long counter, long nice, unsigned long policy: mantengono le informazioni per il CPU scheduling. Ad esempio il counter decrementa, mantenendo il tempo rimanente.

Nelle versioni più moderne di Linux, alcune informazioni sono state ridotte a un unico bit e compattate all'interno di bitmask, ma rimangono comunque facilmente accessibili mediante apposite macro/API: ad esempio, il campo need\_resched è diventato il bit **TIF\_NEED\_RESCHED** all'interno di una maschera di bit. Inoltre, sono stati aggiunti nuovi campi nel TCB in modo da rispecchiare correttamente tutte le capability di Linux.

Attualmente, un thread può avere due identificatori, non uno solo!

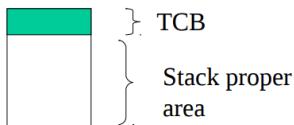
Il pid è associato al container di default (i container sono quelli visti in SDCC), non associabile ad altri.

NB: a questo punto della trattazione è chiaro che senza un TCB o senza uno stack privato un thread non può esistere.

### Allocazione del TCB in Linux

#### Prima del kernel 2.6:

I TCB vengono allocati dinamicamente ogni volta che è necessario. L'area di memoria per il TCB è riservata nella porzione superiore dello stack del thread associato (in particolare per i thread di livello kernel parliamo di kernel level stack):



La dimensione dello stack del thread di livello kernel è pari a due frame dati dal *buddy allocator*, ovvero 8 KB: si tratta di dimensioni molto ridotte (che nelle versioni più recenti del kernel sono aumentate ma non in maniera spropositata), per cui fare delle iterazioni in maniera massiva anziché adottare la ricorsione all'interno dei thread di livello kernel può causare dei problemi.

Per quanto riguarda il thread che si occupa del boot del kernel (che poi risulterà essere l'idle process), gli 8 KB di memoria riservati al suo TCB e al suo stack sono definiti già a tempo di compilazione del kernel, per cui risultano già allocati durante la fase del boot del sistema operativo in modo tale che l'idle process possa lavorare correttamente.

L'organizzazione descritta qui sopra ha delle conseguenze importanti: da una parte, per ottenere due buddy frame, è sufficiente una sola chiamata al buddy allocator. Dall'altra parte, però, se con l'avanzare delle versioni di Linux dovessero aumentare anche di poco le dimensioni del TCB, si ridurrebbe lo spazio a disposizione per lo stack, aumentando anche il rischio di buffer overflow (scrittura delle informazioni dello stack all'interno del TCB); l'unico modo per risolvere il problema consiste nell'aumentare il numero di pagine da allocare col buddy allocator, ma queste devono essere necessariamente una potenza di 2 (per cui si passerebbe da due a quattro, da quattro a otto e così via).

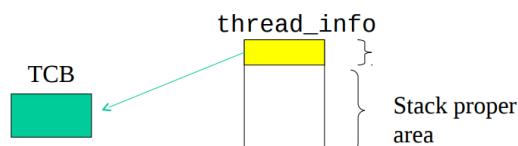
Dal punto di vista implementativo, il TCB (rappresentato dal tipo di dato **task\_struct**) e lo stack del thread sono definiti all'interno di una union. Ad esempio, nel kernel 2.4.37 abbiamo:

```

union task_union {
    struct task_struct task;
    unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
  
```

#### Tra il kernel 2.6 e il kernel 4.8:

Poiché l'accoppiamento tra il TCB e lo stack del thread crea problemi, nella versione 2.6 del kernel si è deciso di separare i due elementi. Nella parte affiorante dello stack rimane comunque un header (una struttura dati chiamata **thread\_info**), all'interno del quale si trova il puntatore al TCB del thread. Perciò, rimane comunque semplice accedere al TCB ma, al contempo, qualunque espansione del TCB non inficia sull'area di memoria dedicata allo stack.



L'header giallo ha ptr a TCB, che sta nella stack area. Se cambio stack area, cambio header. Allora la stack area deve avere una collocazione precisa. Se devo espandere il TCB di pochi byte, posso usare quelli della stack area. Se invece dovessero essere in una zona specifica, allora ci metterei un pointer, però dovrei

definire altri servizi per l'aggiunta e rimozione del collegamento. Se andiamo in overflow, o corrompo **thread\_info**, o quello che c'è sopra in memoria.

La dimensione dello stack del thread di livello kernel è pari a due o più frame dati dal buddy allocator (a partire dal kernel 4 è possibile specificare l'ordine di allocazione con la macro **THREAD\_SIZE\_ORDER**). Ricordiamo che, se il buddy allocator fornisce  $2^k$  pagine di memoria, queste avranno un allineamento rispetto alle  $2^k$  pagine (e non alla singola pagina). È tale caratteristica che rende molto semplice l'ottenimento dell'indirizzo base dello stack del thread e, quindi, del TCB nel caso dei kernel antecedenti al 2.6 e della struttura **thread\_info** nel caso dei kernel compresi tra il 2.6 e il 4.8.

Di seguito sono riportati i campi di **thread\_info** (col kernel 3.19 come riferimento):

```

26 struct thread_info {
27     struct task_struct *task;          /* main task structure */
28     struct exec_domain *exec_domain;  /* execution domain */
29     __u32               flags;        /* low level flags */
30     __u32               status;       /* thread synchronous flags */
31     __u32               cpu;         /* current CPU */
32     int                saved_preempt_count;
33     mm_segment_t        addr_limit;
34     struct restart_block restart_block;
35     void __user         *sysenter_return;
36     unsigned int         sig_on_uaccess_error:1;
37     unsigned int         uaccess_err:1; /* uaccess failed */
38 };

```

### **La macro current**

È un simbolo che ci permette di identificare il TCB del thread correntemente in esercizio in CPU, per cui non è altro che un puntatore a una **task\_struct**.

Fino al kernel 4.8, la macro, per stabilire qual è il thread corrente (e, quindi, per recuperare il TCB corretto), esegue una computazione basata sul valore dello stack pointer sfruttando il fatto che lo stack del thread è allineato secondo le regole del buddy allocator. Chiaramente, questo implica che un cambio dello stack del kernel porta anche a un cambiamento dell'outcome della macro.

Approfondiamo la computazione di **current** nei due sottocasi (prima del kernel 2.6 vs tra il kernel 2.6 e il kernel 4.8):

-> Prima del kernel 2.6: viene preso lo stack pointer e ne vengono mascherati i bit meno significativi (quelli usati per spiazzarsi all'interno dello stack); si ottiene così l'indirizzo della base dello stack che non è altro che l'indirizzo del TCB.

-> Tra il kernel 2.6 e il kernel 4.8: si procede come nel caso precedente ma in più ci si spiazza all'interno di **thread\_info** per recuperare il campo **task** di tipo **task\_struct** (che non è altro che l'indirizzo del TCB).

### La macro current nelle ultimissime versioni del kernel:

Nelle versioni più recenti del kernel, il TCB non è più referenziato da un campo della struttura **thread\_info**, bensì è completamente separato dallo stack del thread. Dunque, **current** ora è un puntatore localizzato nella memoria per-CPU che viene aggiornato ogni volta si ha un rescheduling e può essere recuperato attraverso la funzione **get\_current()**.

Per il buddy allocator è quasi impossibile, data una certa area, prendere delle aree (pagine) superiori ed inferiori di una stack area, per vedere se sta tentando di uscire dalla stack area, questo perché il buddy allocator vede tutta la memoria, ed è spesso frammentata. Anche la stack area non è detto sia continua in memoria fisica.

### **Virtually mapped stack**

Dopo il kernel 4.9 lo stack non è più allineato secondo la logica data dal buddy allocator poiché è diventato possibile sfruttare il **vmalloc()** per riservare allo stack un'area di memoria arbitrariamente grande e non necessariamente contigua a livello fisico. Uno stack siffatto è detto **virtually mapped stack**.

In tal modo non si hanno più vincoli sulla dimensione dello stack e, soprattutto, si marcano la prima e l'ultima pagina allocate con `vmalloc()` come `unmapped`: questo fa sì che, se si eccede con la quantità di informazioni inserite nello stack, il buffer overflow che si genera coinvolge una pagina `unmapped` e, quindi, produce un segmentation fault (senza che vengano sovrascritte in maniera errata zone di memoria valide).

## Runqueue e waitqueue nel kernel 2.4

### Runqueue nel kernel 2.4:

Nella versione 2.4 del kernel abbiamo la definizione di un array di puntatori a `task_struct`, e ciascuna entry di quest'array rappresenta la runqueue di una specifica CPU. Internamente, però, tutte le entry puntano alla testa di una stessa runqueue, che risulta essere unica all'interno del sistema. Di conseguenza, la funzione `schedule()` non deve far altro che scandire l'unica runqueue effettivamente esistente per selezionare il prossimo thread da mandare in CPU.

[Non ho API per metterci/toglierci dalla runqueue, ma ci interfacciamo con layer superiore.](#)

Il thread che si occupa del boot del kernel (che poi risulterà essere l'idle process) viene posto in modo automatico all'interno della runqueue e vi sarà sempre presente.

### Waitqueue nel kernel 2.4:

Nella versione 2.4 del kernel è possibile dichiarare una nuova waitqueue sfruttando la seguente macro:

`DECLARE_WAIT_QUEUE_HEAD(queue)`.

Inoltre, si hanno numerose API che si basano sull'utilizzo di queste waitqueue:

- > `'void interruptible_sleep_on (wait_queue_head_t *q)'` : collega il thread chiamante alla waitqueue specificata come parametro per poi scollarlo dalla runqueue. Il thread rimane a dormire nella waitqueue finché non viene esplicitamente riportato in runqueue oppure finché non viene colpito da un segnale proveniente da un altro thread.

- > `'void sleep_on (wait_queue_head_t *q)'`: è analoga `interruptible_sleep_on()` con l'unica differenza che interpreta i segnali provenienti dagli altri thread come delle don't care.

- > `'void interruptible_sleep_on_timeout (wait_queue_head_t *q, long timeout)'`: anch'essa è analoga a `interruptible_sleep_on()` con l'unica differenza che il thread rimane a dormire nella waitqueue finché non viene colpito da un segnale proveniente da un altro thread oppure finché non scade il timeout specificato come parametro. Siamo colpibili quindi.

- > `'void sleep_on_timeout (wait_queue_head_t *q, long timeout)'`: è analoga all'API precedente con l'unica differenza che interpreta i segnali provenienti dagli altri thread come delle don't care.

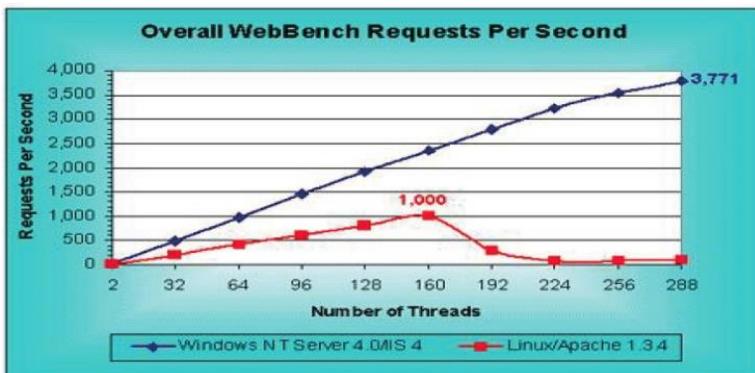
- > `'void wake_up (wait_queue_head_t *q)'`: riporta nella runqueue tutti i thread appartenenti alla waitqueue passata come parametro.

- > `'void wake_up_interruptible (wait_queue_head_t *q)'`: riporta nella runqueue tutti i thread appartenenti alla waitqueue passata come parametro che erano accodati come "interruptible".

- > `'void wake_up_process (struct task_struct *p)'`: riporta nella runqueue solo il thread specificato dal TCB passato come parametro.

Un problema che abbiamo è che `'wake_up()'` e `'wake_up_interruptible()'` non sono per niente selettive, mentre `wake_up_process` è estremamente selettiva nel prendere il thread da riportare nella runqueue. Di fatto, all'interno di un'unica waitqueue potrebbero esserci anche centinaia di thread, e non è ideale poter solo risvegliare o solo un thread o quelle centinaia di thread, senza poter selezionare un sottoinsieme specifico di thread. In effetti, potremmo essere penalizzati dal punto di vista delle performance poiché, per risvegliare un sottoinsieme di thread che soddisfino una specifica condizione, è necessario invocare `wake_up()` (o `wake_up_interruptible()`) per poi rimettere subito a dormire tutti quei thread che non soddisfano quella condizione.

Questa problematica porta al cosiddetto **thundering herd effect**:



### Stati dei thread:

I thread possono trovarsi in uno di questi stati fondamentali:

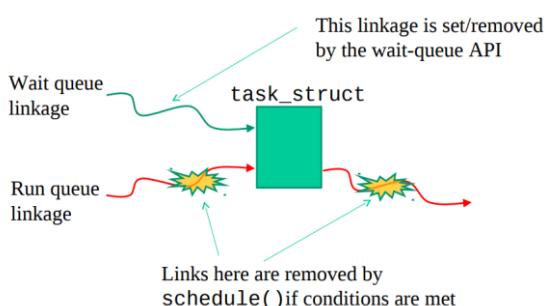
- **TASK\_RUNNING (0):** il thread è nella runqueue ed è osservabile dallo scheduler della CPU.
- **TASK\_INTERRUPTIBLE (1):** il thread è in una waitqueue, dove è stato accodato come "interruptible".
- **TASK\_UNINTERRUPTIBLE (2):** il thread è in una waitqueue, dove non è stato accodato come "interruptible".
- **TASK\_ZOMBIE (4):** il thread ha terminato la sua esecuzione ma il TCB deve ancora esistere in attesa che il codice di uscita del thread venga restituito al parent.

### Implementazione di interruptible\_sleep\_on() / wait vs run queue:

Se mio TCB viene eliminato da runqueue, non posso essere richiamato. Ciò viene fatto in `SLEEP\_ON\_HEAD`, ci mette sulla runqueue ricevuta come parametro. Io, ancora in CPU, devo passare in safe places per lasciare la CPU ad altri. Mi metto come interrompibile, chiamo lo schedule poi per lasciare la CPU. Con Schedule(), prima di analizzare la runqueue e fare context switch, vede se deve eliminare questo TCB, perchè se ho detto che voglio lasciare la cpu, non voglio essere rimesso. Ciò lo specifico in TCB. Lo scheduler passa su runqueue ed eventualmente mi toglie. Con Sleep\_ON\_TAIL, vengo tolto da wait\_queue. Cioè mi tolgo da wait\_queue quando mi rendo conto che sto in esecuzione.

```
#define SLEEP_ON_HEAD          \
    wq_write_lock_irqsave(&q->lock,flags);      \
    __add_wait_queue(q, &wait);                  \
    wq_write_unlock(&q->lock);                  \
\
#define SLEEP_ON_TAIL           \
    wq_write_lock_irq(&q->lock);              \
    __remove_wait_queue(q, &wait);              \
    wq_write_unlock_irqrestore(&q->lock,flags); \
\
void interruptible_sleep_on(wait_queue_head_t *q){ \
    SLEEP_ON_VAR \
    current->state = TASK_INTERRUPTIBLE; \
    SLEEP_ON_HEAD \
    schedule(); \
    SLEEP_ON_TAIL \
}
```

NB: nel momento in cui viene invocato schedule(), il thread chiamante è agganciato sia alla waitqueue per effetto dello SLEEP\_ON\_HEAD, sia alla runqueue: verrà sganciato dalla runqueue proprio dalla funzione schedule(), se determinate condizioni sono rispettate.



## Wait event queue

Rappresentano l'evoluzione delle waitqueue e risolvono il problema legato al thundering herd effect. In particolare, sono delle waitqueue in grado di risvegliare i thread con determinate condizioni: i thread si agganciano a una delle wait event queue specificando l'evento (i.e. la condizione) che dovrà sveglierli. Nel momento in cui si verifica un evento E, ciascun thread della wait event queue si mette momentaneamente in CPU (e quindi in runqueue per opera di un qualche altro thread) e verifica se E corrisponde all'evento in grado di sveglierlo: se sì, termina la sleep, altrimenti si rimette a dormire.

Elenchiamo (senza approfondirle) alcune API che si basano sull'utilizzo delle wait event queue:

### Name

`wait_event_interruptible — sleep until a condition gets true`

### Synopsis

```
wait_event_interruptible (wq,
                        condition);
```

### Arguments

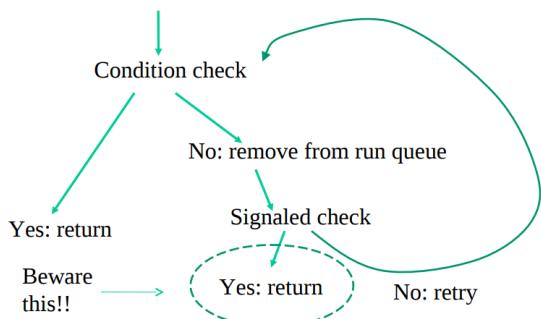
`wq`  
the waitqueue to wait on  
`condition`  
a C expression for the event to wait for

```
wait_event( wq, condition )
wait_event_timeout( wq, condition, timeout )
wait_event_freezable( wq, condition )
wait_event_command( wq, condition, pre-command,
                    post-command)
wait_on_bit( unsigned long * word, int bit,
            unsigned mode)
wait_on_bit_timeout( unsigned long * word, int bit,
                     unsigned mode, unsigned long timeout)
wake_up_bit( void* word, int bit)
```

Ad esempio: `wait_event_command` specifica la waitqueue, cosa fare prima (`pre-command`), cosa fare dopo (`post-command`), quale condizione ci risveglia (`condition`).

Badiamo che il parametro `condition` delle API (macro) sopra elencate non è altro che un costrutto condizionale del linguaggio C (e.g. `i==0`), ed è **passato per valore (e non per riferimento)**. **Non sono API classiche, perchè non modificano nulla, bensì sono delle MACRO.**

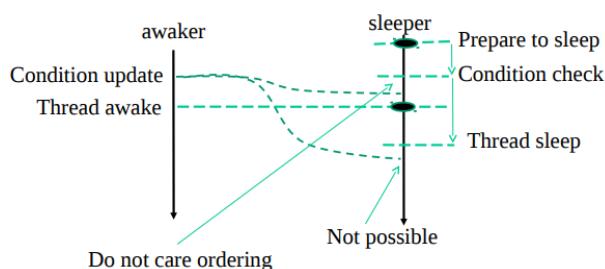
In definitiva, per quanto riguarda le attese interrompibili, lo schema seguito è questo:



Dobbiamo capire se siamo stati segnalati o meno, se siamo segnalati usciamo dal blocco di codice anche se la condizione (condition check) non è vera.

### Linearizzabilità:

Se in un certo momento viene cambiata la condizione di risveglio di un particolare thread T in modo tale che tale condizione diventi vera, è impossibile che il check della condizione poi porti a un falso negativo costringendo così T a rimanere a dormire. La garanzia di questo tipo di consistenza è dovuta al fatto che il TCB di ciascun thread è protetto da uno spinlock (che viene usato ad esempio dalla funzione `schedule()`), per cui può essere acceduto e modificato da un solo thread per volta (anche se, ovviamente, TCB diversi possono essere modificati anche concorrentemente): questo effettivamente porta alla prevenzione del fenomeno di load-store bypass dato dal modello di consistenza TSO adottato nelle architetture come x86.



Ho flusso in esecuzione su CPU che cambia mia condizione di risveglio e fa altre attività, tipo ricollegarmi alla runqueue, con cui mi rendo conto che la condizione di risveglio è soddisfatta e che posso continuare. L'ordine lo vedo io, non è detto che gli altri la vedano così in memoria (a causa degli store buffer). La soluzione è che, quando si fa `thread_awake`, esso usa istruzioni macchina con locking, per prendere target, check condizioni e rilascio. Quando rimettiamo thread su runqueue, il mio store buffer con condizione di risveglio viene flushata, ma questo aggiornamento non può essere perso, perché è stato flushato in memoria. Lavoriamo su struttura dati che porta thread CPU, ma deve vedere anche altri aggiornamenti fatti prima, come? Sincronizzando tutto ciò che c'è sullo store buffer.

### **Il campo mm nel TCB**

È un puntatore alla struttura dati **mm\_struct** che raccorda il thread alle informazioni principali di memory management relative al thread. Tra queste informazioni di memory management troviamo:

- L'indirizzo virtuale della page table usata dal thread (campo **pgd** della **mm\_struct**).
  - Un puntatore a una lista collegata di elementi di tipo **vm\_area\_struct** (campo **mmap** della **mm\_struct**).
- Ciascuno di questi elementi descrive una zona mappata della parte user dell'address space utilizzato dal thread.

Vediamo più nel dettaglio quali sono le informazioni specificate dalla struttura **vm\_area\_struct**:

```

struct vm_area_struct {
    struct mm_struct * vm_mm; /* The address space we belong to. */
    unsigned long vm_start; /* Our start address within vm_mm. */
    unsigned long vm_end; /* The first byte after our end address
                           within vm_mm. */
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot; /* Access permissions of this VMA. */

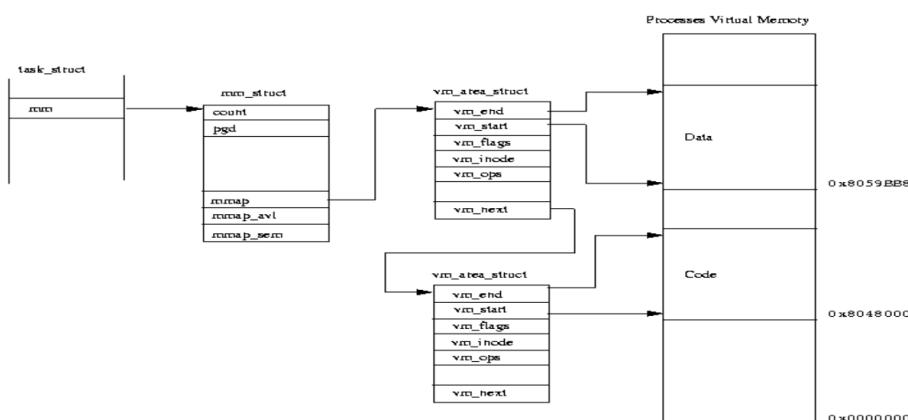
    .....
    /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;
    .....
};

};

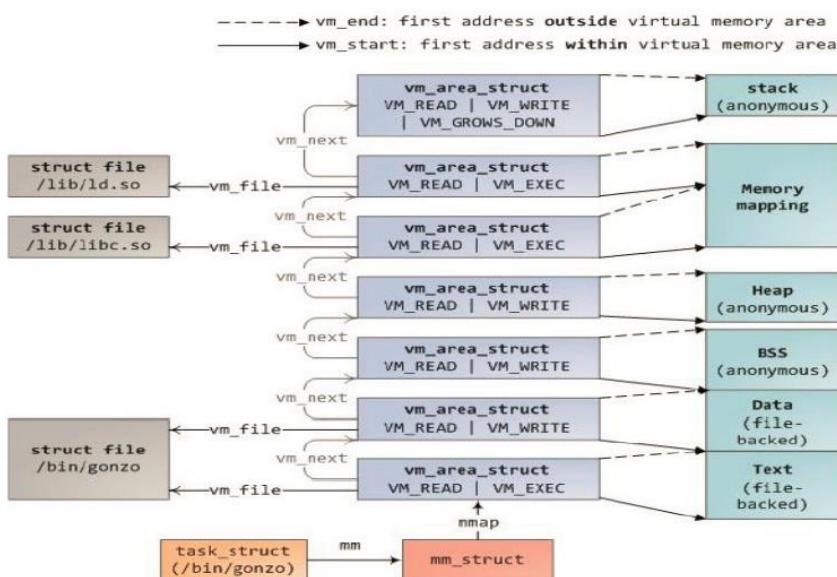

```

- > unsigned long vm\_start: indica dove inizia la zona di memoria (è il primo indirizzo all'interno della zona).
- > unsigned long vm\_end: indica dove termina la zona di memoria (è il primo indirizzo fuori dalla zona).
- > pgprot\_t vm\_page\_prot: indica i permessi di accesso alla zona di memoria.
- > struct vm\_operations\_struct \*vm\_ops: è un puntatore a una struct che indica i gestori delle operazioni (**driver**)che devono essere eseguite a livello sistema sulla zona di memoria in caso di fault (e.g. il gestore dei page fault, che viene specificato dal campo **nopage** di **vm\_ops**). Quando facciamo `mmap`, viene fatto un collegamento ad uno specifico driver la gestione.

## A scheme



## An example



## Identificazione dei thread

Nelle implementazioni moderne del kernel è possibile anche virtualizzare i PID (Process ID). Perciò, ciascun thread può avere più di un PID, di cui uno **reale** (che sarebbe `current->pid`, che già conosciamo) e uno **virtuale**, [valido nello spazio dei nomi in cui si sta muovendo](#).

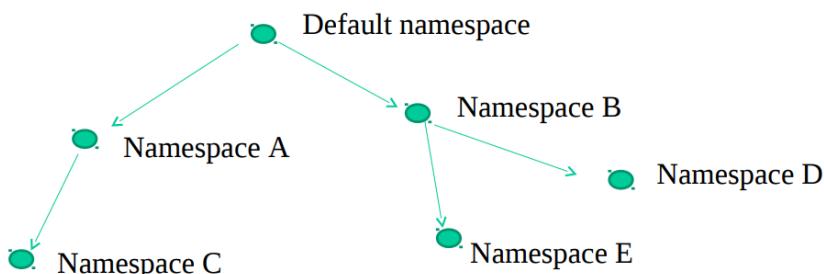
Questo concetto è legato ai **namespace** (spazi dei nomi, ovvero collezioni di nomi di entità): di fatto è possibile associare ciascun thread a un particolare namespace X e, conseguentemente, quel thread avrà un identificatore (virtuale) all'interno di X. Badiamo che se un thread T invoca ad esempio la system call `ppid()` (*parent PID*), allora gli verrà restituito il PID virtuale del parent, ovvero quello che fa riferimento al namespace in cui ci troviamo.

Esiste un namespace principale (**namespace di default**, che è l'unico che esiste allo startup del kernel) che è proprio quello utilizzato per impostare il valore `current->pid` del thread corrente.

Quando viene creato un nuovo thread T, possiamo decidere di muovere T in un altro namespace X rispetto a quello corrente, e X diventerà così il child del namespace corrente. Attualmente in Linux si può avere un massimo di 32 livelli di namespace e tale limite è dato da:

```
#define MAX_PN_NS_LEVEL 32
```

### A representation



Il thread T, la cui creazione porta all'istanziazione di un nuovo namespace X, avrà il PID virtuale pari a 1 nel contesto del namespace X e, sempre all'interno di X, il suo parent avrà il PID virtuale pari a 0.

In ogni caso, tutti i thread, indipendentemente da quale sia il namespace a cui appartengono, avranno un PID definito all'interno del namespace di default (che corrisponde appunto al PID reale).

### ### Visibilità dei namespace

Con riferimento ai servizi del kernel, un thread che vive in un certo namespace X non ha alcuna visibilità dei namespace di livello superiore (= "ancestor"). Ad esempio, non possono essere utilizzate le strutture dati definite per i namespace ancestor, né tantomeno possono essere coinvolti (e.g. tramite una segnalazione di kill) i thread che vivono all'interno di tali namespace ancestor. In effetti, un namespace è una sorta di contenitore che ci va a delimitare le operazioni che possono essere eseguite, e non a caso i container Docker possono essere visti come un tipo di namespace. La creazione di un namespace dipende dalla creazione dei thread, che per definizione sono sempre child, non posso quindi creare un nuovo nodo radice. Con `ps -o pid,pidns` prendo tutti i namespace di tutti i processi visibili dalla shell.

### ### Struttura nsproxy

All'interno del TCB di ciascun thread si ha un campo che corrisponde a una `struct nsproxy \*nsproxy`, che indica quali sono i namespace a cui il thread fa riferimento.

### ### Mapping tra PID e TCB

Molti servizi del kernel fanno uso dell'indirizzo del TCB di un qualche thread (e.g. kill o risveglio dalla

waitqueue), per cui abbiamo bisogno di un mapping tra i PID e gli indirizzi dei TCB (il viceversa chiaramente è semplice dato che esiste il campo pid all'interno del TCB).

Questo mapping è basato su delle strutture dati collegate che sono ovviamente esterne al TCB:

- `extern struct task\_struct \*pidhash[PIDHASH\_SZ]`: è un array di puntatori a task\_struct (i.e. un array di TCB) con un numero di entry pari a PIDHASH\_SZ = 1024. Ciascun thread con un PID pari a x avrà il TCB referenziato dalla k-esima entry dell'array pidhash, dove k è pari a:

$$k = \text{pid\_hashfn}(x) = (x/2^8)^x \&& (\text{PIDHASH\_SX} - 1)$$

- `struct task\_struct \*pidhash\_next`: è il TCB successivo all'interno di una lista collegata. In effetti, i TCB che si possono trovare nell'array pidhash appartengono a loro volta anche a una lista collegata (chiaramente possono esservi più TCB dietro una medesima entry di pidhash).

- `struct task\_struct \*pidhash\_pprev`: è il TCB precedente all'interno di una lista collegata.

A partire dalla versione 2.6 del kernel esiste anche un secondo modo per identificare l'indirizzo del TCB a partire dal PID del thread T: si parte dal namespace X a cui il thread T appartiene e dal PID virtuale di T all'interno di X; dopodiché si sfrutta sempre un sistema di hashing che collega più o meno direttamente il PID virtuale col TCB.

La struttura che si basa su questa seconda soluzione è riportata qui di seguito:

```
struct pid {
    atomic_t count;
    unsigned int level;
    struct hlist_head tasks[PIDTYPE_MAX]      //lists of tasks that use this pid
    struct rcu_head rcu;
    struct upid numbers[1]; //codice numerico su cui eseguire hash
};
```

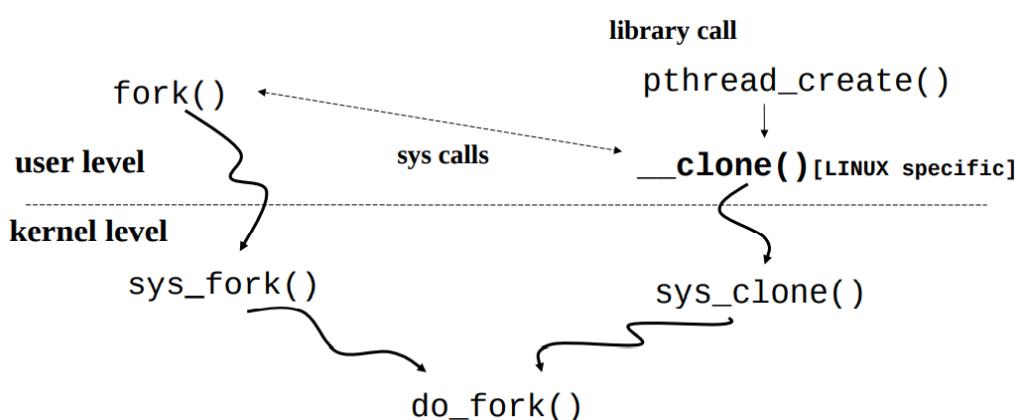
Analogamente alla prima soluzione, ciascuna entry dell'array tasks è relativa a una lista di TCB che hanno uno stesso PID virtuale (ma relativamente a namespace differenti). Inoltre, anche qui l'array è basato su un meccanismo di hashing.

Per utilizzare tale meccanismo di hashing, si ricorre alla seguente API:

- > `struct task\_struct \*pid\_task (struct pid \*pid, enum pid\_type)`: restituisce l'indirizzo del TCB del thread considerato. Notiamo che vuole in input l'indirizzo alla struct pid che può essere ottenuto a partire dal PID virtuale tramite l'API **find\_vpid (pid)**. Di seguito è mostrato un esempio di utilizzo di questa API: `pid\_task (find\_vpid (pid), PIDTYPE\_PID)`, dove il secondo parametro indica che il PID su cui basiamo la query è quello di default.

## Creazione di processi e thread

Avviene secondo il seguente schema:



- **fork()** è una system call che invoca internamente **sys\_fork()**, che è una funzione interna al kernel.
- **pthread\_create()** è una funzione di libreria che invoca la system call **\_\_clone()** (passandovi i parametri configurati in un certo modo), la quale a sua volta chiama internamente **sys\_clone()**, che è un'altra funzione interna al kernel.

- Sia **sys\_fork()** che **sys\_clone()** invocano uno stesso servizio del kernel, che è **do\_fork()**: tale servizio si occupa di istanziare un nuovo flusso di esecuzione, e la modalità con cui lo fa dipende dai parametri che riceve in ingresso, i quali variano a seconda se proveniamo da una **sys\_fork()** oppure da una **sys\_clone()**. In particolare, se proveniamo da una **sys\_clone()**, la **do\_fork()** si limiterà ad eseguire le operazioni necessarie per avviare un flusso di esecuzione appartenente al processo chiamante; se invece proveniamo da una **sys\_fork()**, la **do\_fork()** eseguirà tutte le operazioni necessarie per mettere in piedi un nuovo processo (e.g. definizione di un nuovo address space).

- Alternativamente a **pthread\_create()**, è possibile utilizzare la funzione di libreria **clone()** di livello **user**, che anch'essa invoca a sua volta la system call **\_\_clone()** ma passandovi dei parametri configurati in modo diverso. In particolare, **clone()** ha una segnatura definita in maniera differente in base all'architettura hardware su cui stiamo girando; tuttavia, in tutti i casi, devono sicuramente essere definiti lo stack e il segmento TLS del nuovo thread, ma anche dei flag. Sceglieremo noi la stack area, con **mmap** o qualsiasi allocatore, non lo fa il kernel. L'indirizzo sarà passato come parametro. Il thread inizia kernel e poi, uscendo, torna user, quindi deve sapere l'indirizzo, per questo lo passo. Il return point è quello del thread originale, ma interessa solo alla libreria di livello user. Possiamo settare anche un nuovo TLS, portando l'utilizzo di **FS** in un certo indirizzo.

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */);

/* For the prototype of the raw system call, see NOTES */
```

Alcuni flag che possono essere utilizzati sono riportati qui di seguito:

<b>CLONE_VM</b>	VM shared between processes
<b>CLONE_FS</b>	fs info shared between processes
<b>CLONE_FILES</b>	open files shared between processes
<b>CLONE_PARENT</b>	we want to have the same parent as the cloner
<b>CLONE_NEWPID</b>	create the process/thread in a new PID namespace
<b>CLONE_SETTLS</b>	the TLS (Thread Local Storage) descriptor is set to newtls
<b>CLONE_THREAD</b>	the child is placed in the same thread group as the calling process

- La **do\_fork()** segue questi passaggi:

- 1) Alloca un nuovo TCB.
- 2) Alloca una nuova area di stack di livello kernel.
- 3) Ottiene un PID opportuno per il nuovo thread.
- 4) Deve ereditare la memory map del parent? Dipende dai parametri in input.
- 5) Deve ereditare la vista del file system del parent? Dipende dai parametri in input.
- 6) Deve ereditare la vista dei file del parent? Dipende sempre dai parametri in input.
- 7) Sicuramente deve far sì che il thread child condivida col parent i tick da trascorrere in CPU.

Quest'ultimo punto è fondamentale anche dal punto di vista della sicurezza: se il thread child, anziché dividersi col parent i tick da trascorrere in CPU disponesse di tick nuovi, sarebbe possibile effettuare un attacco in cui si creano thread in maniera massiva in modo tale da monopolizzare la CPU.

Se thread padre ha i tick che diminuiscono, il nuovo thread deve ripartire i tick col padre, altrimenti potrei usare esclusivamente la cpu.

### Primitive di sincronizzazione

- `DECLARE\_MUTEX (name)`: dichiara un nuovo mutex, restituendo un puntatore a una struct semaphore.
- `void sema\_init (struct semaphore \*sem, int val)`: inizializza uno specifico mutex.
- `void down (struct semaphore \*sem)`: acquisisce un token dal mutex specificato in maniera bloccante.
- `int down\_interruptible (struct semaphore \*sem)` : acquisisce un token dal mutex specificato in maniera bloccante; tuttavia, il chiamante potrebbe essere risvegliato durante l'attesa per via di un interrupt.
- `int down\_trylock (struct semaphore \*sem)`: acquisisce un token dal mutex specificato in maniera non bloccante.
- `void up (struct semaphore \*sem)`: rilascia un token al mutex specificato.

Abbiamo anche le API per la gestione degli spinlock:

```
spinlock_t my_lock = SPINLOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
spin_lock(spinlock_t *lock);
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
spin_lock_irq(spinlock_t *lock);
spin_lock_bh(spinlock_t *lock);

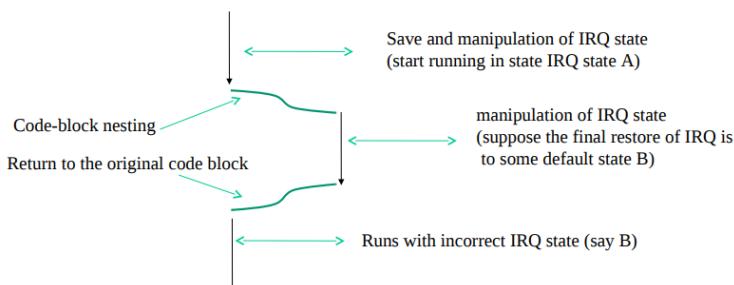
spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock)
spin_unlock_wait(spinlock_t *lock);
```

Quando chiamiamo API per prendere un lock, cosa avviene veramente?

Lo spin locking vorrei eseguirlo in un blocco di codice che non deve essere interrotto, cioè vorrei prendere lock e mettere il processore come non interrompibile. Però che ne so se il processore era già non interrompibile?

Consideriamo **spin\_lock\_irqsave()** e **spin\_lock\_irq()**:

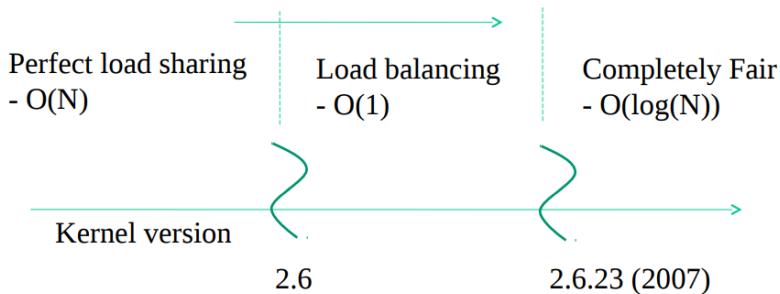
- > `spin\_lock\_irq (spinlock\_t \*lock)`: blocca lo spinlock specificato come parametro.
- > `spin\_lock\_irqsave (spinlock\_t \*lock, unsigned long flags)`: blocca lo spinlock specificato come parametro cambiando anche lo stato di interrompibilità del processore. In particolare, il parametro flags è di input-output (tant'è vero che **spin\_lock\_irqsave()** è una macro) e, di fatto, prima della chiamata assume il valore che deve rispecchiare lo stato finale del processore, mentre al termine della chiamata assumerà il valore che rispecchia lo stato originale del processore; quest'ultimo valore verrà poi riutilizzato come parametro in input per la funzione **spin\_unlock\_irqrestore()** che servirà a rilasciare lo spinlock e a ripristinare il vecchio stato di interrompibilità del processore (in modo tale da garantire un comportamento corretto del sistema).



Abbiamo anche delle varianti delle primitive di sincronizzazione, che ad esempio discriminano i reader dai writer:

- `\_\_RW\_LOCK\_UNLOCKED(xxx\_lock)`: dichiara un nuovo lock di tipo read/write.
- `read\_lock\_irqsave(&xxx\_lock, flags)` : acquisisce un lock in lettura cambiando lo stato di interrumpibilità del processore.
- `read\_unlock\_irqrestore(&xxx\_lock, flags)` : rilascia il lock in lettura ripristinando lo stato d'interrumpibilità del processore.
- `write\_lock\_irqsave(&xxx\_lock, flags)` : acquisisce un lock in scrittura cambiando lo stato di interrumpibilità del processore.
- `write\_unlock\_irqrestore(&xxx\_lock, flags)` : rilascia il lock in scrittura ripristinando lo stato d'interrumpibilità del processore.

### Evoluzione dello scheduler logic in Linux



- **Perfect load sharing**: il carico dei CPU-core è perfettamente bilanciato ma il costo per lo scheduling è lineare rispetto al numero di thread schedulabili.
- **Load balancing**: si cerca di bilanciare il carico dei CPU-core ma non in modo perfetto, e il costo per lo scheduling diventa  $O(1)$ ; qui, rispetto al perfect load sharing, si aggiungono i task real-time.
- **Completely fair**: il costo per lo scheduling dei task real-time (con priorità migliore) rimane  $O(1)$ , mentre il costo per lo scheduling dei task non real-time (con priorità peggiore) passa a  $O(\log(N))$ .

Indipendentemente dal momento, ci sono concetti comuni, come quello dell'epoca.

### Aspetti tradizionali dello scheduler logic:

Abbiamo una linea del tempo, e ad un certo istante  $t$  prendiamo una decisione, che si protrae fino ad un tempo  $t_2$ . Questa è un'epoca. La pianificazione dell'utilizzo dei tick da spendere in CPU è basato sul concetto di **epoca**. Un'epoca termina nel momento in cui tutti i thread che si trovano sulla runqueue hanno esaurito i loro tick (ma, comunque, i thread posti su una qualche waitqueue possono ancora avere dei residui non utilizzati). Nel momento in cui un'epoca finisce, viene ricalcolato il numero di tick da assegnare a tutti i thread per la prossima epoca, compresi quelli posti in qualche waitqueue; in particolare, il numero di tick che ciascun thread otterrà dipende dal suo livello di priorità: più la priorità è elevata, maggiore sarà il numero di tick assegnato.

Per quanto riguarda le priorità, lo schema classico di Posix prevede che le priorità vadano da -20 a +19 (dove -20 è la priorità migliore di tutte e +19 è la peggiore di tutte). È possibile spostarsi tra un livello di priorità e un altro andando a modificare un particolare parametro: la **niceness**.

### Perfect load sharing scheduler

Quando viene invocata la funzione `schedule()`, i TCB che vengono guardati sono tutti quelli relativi ai thread posti nella runqueue. *Quindi non quelle delle waitqueue*. Ciò ha come effetto che qualsiasi thread può andare in esecuzione in qualsiasi istante di tempo su qualsiasi CPU-core. Comunque sia, le decisioni di scheduling si basano sulle priorità dei thread ma anche su determinati indici prestazionali dell'hardware

(e.g. se schedulare un certo thread T favorisce l'architettura di caching, T potrebbe essere selezionato con maggiore probabilità).

Nella versione 2.4 di Linux, l'esecuzione della funzione schedule() si articola in tre fasi distinte:

- 1) Verifica su se il thread corrente (da deschedulare) deve essere rimosso dalla runqueue prima di andare a scandire la runqueue stessa.
- 2) Analisi della runqueue; qui chiaramente la runqueue deve essere unica, altrimenti non potremmo parlare di perfect load sharing scheduler.
- 3) Context switch vero e proprio, in cui il thread selezionato viene posto in CPU.

**Fase 1:** nella parte iniziale della funzione schedule() viene effettuato un check sullo stato del thread corrente: se è interrompibile (i.e. interessato a eventuali segnalazioni in ingresso) ed è stato effettivamente colpito da una segnalazione, allora passa allo stato *running*, per cui non può essere rimosso dalla runqueue; se invece non ha ricevuto alcuna segnalazione, allora viene rimosso dalla runqueue.

**Fase 2:** l'analisi della runqueue si articola a sua volta in più passaggi:

- > `list_for_each (entry, &runqueue_head)`: è una macro che effettua la scansione di una lista circolare ed è l'equivalente di `for (entry=&runqueue_head)->next; entry!=&runqueue_head; entry=entry->next)`
- > `thread = list_entry (entry, struct task_struct, run_list)`: è una macro che permette di risalire all'indirizzo della struttura `task_struct` (i.e. del TCB) che contiene la `run_list` posta all'indirizzo specificato da `entry`; è l'equivalente di `container_of (entry, struct task_struct, run_list)`
- > `if (can_schedule (thread, this_cpu))`: è un controllo su se il thread che stiamo analizzando ora è affine con la CPU su cui ci troviamo ora (i.e. su cui si trova il thread che stiamo per deschedulare). Se questo controllo passa, si va al passaggio successivo, altrimenti si continua a iterare sugli altri thread della runqueue.
- > `int weight = goodness (thread, this_cpu, prev_thread->active_mm)`: calcola la bontà del thread che stiamo analizzando per quanto riguarda l'assegnazione della CPU corrente, anche sulla base di qual è la memory map del thread che stiamo per deschedulare.
- > Se `weight` è maggiore di una certa costante, allora vuol dire che il thread che stiamo analizzando è idoneo per essere schedulato sulla CPU corrente; altrimenti, si procede ad analizzare altri thread all'interno della lista circolare (i.e. della runqueue).

mm vs active\_mm:

Come è stato possibile intuire, il TCB ha due campi di tipo puntatore a struct `mm_struct`:

- **mm**: identifica le informazioni di memory management che caratterizzano l'address space riguardante la parte applicativa del thread.
- **active\_mm**: identifica le ultime informazioni di memory management di livello user che sono state utilizzate da un qualche thread che ha girato / sta girando sulla CPU corrente.

Di conseguenza, per i thread di livello user, `mm` e `active_mm` sono perfettamente equivalenti; per quanto invece riguarda i thread di livello kernel, `mm` è null, mentre `active_mm` potrebbe non esserlo.

Inoltre, per avere delle ottimizzazioni dell'hardware, è bene controllare il campo `active_mm` del TCB dei vari thread nel momento in cui si ha un context switch (siano essi thread di livello user o di livello kernel). Favorendo thread che condividono lo stesso namespace, ottimizziamo le risorse.

Calcolo della goodness:

$$\text{goodness}(p) = 20 - (p->\text{nice}) + (p->\text{counter}) + \text{isPageTableShared} + 15 * \text{wasLastRunningOnSameCPU}$$

- `p->nice`: è la niceness.

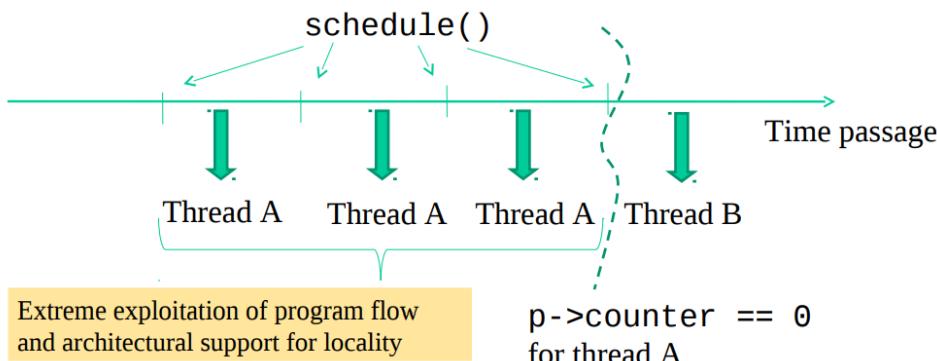
- `p->counter`: sono i tick residui nel quanto di tempo a disposizione per il thread `p` per eseguire in CPU.

- `isPageTableShared`: vale 1 se la page table del thread `p` è condivisa col thread da deschedulare, 0 altrimenti. Cioè tabella delle pagine è condivisa rispetto all'ultima page table in questa CPU.

- `wasLastRunningOnSameCPU`: vale 1 se il thread `p` è stato l'ultimo thread a eseguire su questo stesso CPU-core, 0 altrimenti. Se noi siamo esattamente lo stesso thread (infatti due thread possono condividere namespace ma toccare pagine diverse), noi premiamo se facciamo proprio le stesse cose.

NB: la goodness è forzata al valore 0 nel caso in cui p->counter vale 0.

NB: i 15 punti dati da wasLastRunningOnSameCPU incentivano i cosiddetti **batch di utilizzo dei tick**, in cui uno stesso thread rimane in CPU per diversi tick consecutivi e anche a seguito di talune chiamate alla funzione schedule(). Di fatto, il fenomeno per cui viene selezionato sempre il medesimo thread da far girare in uno stesso CPU-core (al più finché non finiscono i tick per l'epoca corrente) porta all'ottimizzazione della località e, quindi, dell'utilizzo delle risorse hardware della macchina, come ad esempio la memoria cache.



#### Gestione delle epoche:

Sappiamo che ciascuna epoca si conclude quando tutti i thread agganciati alla runqueue hanno esaurito il loro quanto di tempo da trascorrere in CPU. Questo avviene in particolare quando p->counter (il tick counter) raggiunge il valore 0 (goodness pari a 0) per tutti i TCB mantenuti dalla runqueue. Al termine di ciascuna epoca, viene ricalcolato il numero di tick da associare a tutti i thread (compresi quelli attivi ma non agganciati alla runqueue) per la nuova epoca utilizzando la seguente formula:

$$p\text{-}>\text{counter} = [(p\text{-}>\text{counter})/2] + 6 - (p\text{-}>\text{nice})/4$$

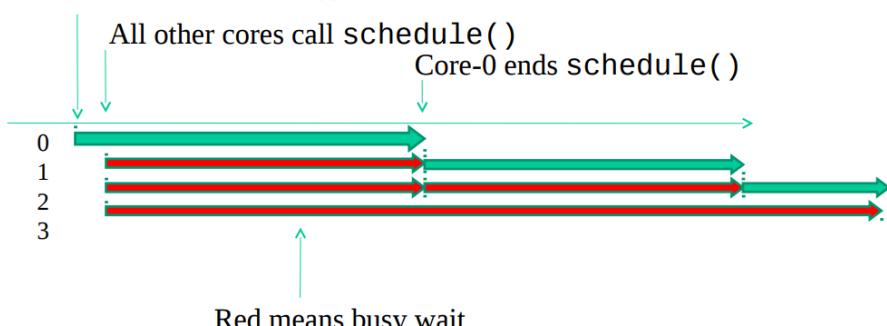
La durata effettiva di ciascun tick dipende dall'architettura che stiamo utilizzando; tipicamente, in Linux, nei sistemi a un solo CPU-core ciascun tick dura 1 ms, mentre nei sistemi multi-core un tick dura ben 4 ms.

#### Problemi di performance:

Il perfect load sharing scheduler presenta alcuni problemi di performance importanti:

- > Quando la funzione schedule() scandisce la runqueue, itera su tutti i thread posti all'interno della coda, compresi quelli che hanno già esaurito i tick nell'epoca corrente: è tale caratteristica che porta lo scheduler ad avere un costo computazionale pari a O(n).
- > All'interno della funzione schedule() è previsto l'utilizzo di un lock per accedere alla runqueue. Ciò implica che la runqueue può essere scandita al più da un CPU-core per volta, il che aggiunge un ritardo di esecuzione maggiore nella funzione schedule(). Chiaramente si tratta di una soluzione che non scala: nel momento in cui si hanno numerosi CPU-core, la probabilità che alcuni di loro invochino schedule() in modo concorrente cresce di molto, per cui gli intervalli di tempo in cui i CPU-core restano in attesa iniziano a diventare importanti. In definitiva, il perfect load sharing scheduler non è adeguato per le architetture con più di quattro CPU-core.

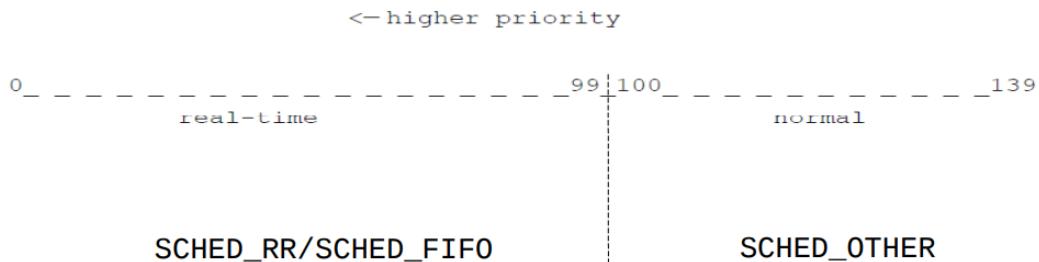
#### Core-0 calls schedule()



## Load balancing scheduler

È uno scheduling che lavora a un tempo costante e prevede l'uso di molteplici runqueue, su cui è difficile avere un accesso concorrente da parte di più CPU-core. Come il perfect load sharing scheduler cerca di tenere bilanciato il workload tra i vari CPU-core, ma lo fa in una maniera più morbida e compatibilmente con le affinità tra i thread e le CPU.

Qui si ha un numero maggiore di priorità rispetto al caso precedente: oltre alle 40 priorità già esistenti, si aggiungono altri 100 livelli riservati ai task real-time:



Quindi 140 priorità, da 0 a 99 real time (le più importanti), da 100 a 139 sono normali, vanno in SCHED\_OTHER. Sono 40 livelli, ovvero la classe originale vista prima.

In particolare, i thread real-time vengono schedulati in CPU con una politica **round-robin** o addirittura **FIFO**: in quest'ultimo caso, non si ha più una logica time-sharing, bensì ciascun task abbandona la CPU per lasciare spazio al successivo solo dopo aver completato l'esecuzione. RR è preemptable, ma è anche real time, quindi posso rilasciare la cpu se passo per dei safe places. Un thread fifo realtime, la userà finchè non chiama servizio bloccante e si mette in waitqueue.

Vantaggi legati alla performance:

La complessità computazionale pari a O(1) del load balancing scheduler è dovuto ai seguenti fattori:

- > Non si ha mai un mix di task eseguibili e task non eseguibili all'interno di una runqueue: quando un thread esaurisce i tick da spendere in CPU per l'epoca corrente, viene sganciato dalla runqueue.
- > I task eseguibili sono a loro volta suddivisi su molteplici runqueue e ciasun CPU-core va a considerare quella runqueue contenente i task più affini a lui.
- > Ciascuna runqueue è a sua volta suddivisa in molteplici sotto-runqueue (una per ogni livello di priorità).

Abbiamo un livello di multicoda per ognuno dei 140 livelli di priorità. C'è replicazione della struttura, non dei contenuti. Un thread può essere solo su una runqueue, quindi c'è una specie di affinità.

Tali caratteristiche permettono ai CPU-core di estrarre direttamente il thread in testa alla sotto-runqueue con priorità migliore all'interno della runqueue di loro competenza, senza dunque effettuare alcuna scansione. Un'app cpu intensive single thread su Linux moderno, potrebbe essere migrata più volte (magari sta in coda perchè c'è un demone in quel monento), avrebbe più senso mettergli un'affinità.

E' possibile avere diversi tick ma stessa priorità per due thread associati ad una stessa priorità.

## Legame tra CPU-core e runqueue

Le diverse runqueue sono completamente separate l'una dall'altra e ognuna di esse è associata a uno specifico CPU-core. Ciò vuol dire che, in condizioni normali, quando un CPU-core esegue lo scheduler logic,

accede alla sua runqueue privata e non si verificano mai delle collisioni. Tuttavia, esistono dei casi particolari in questo non è vero e un CPU-core accede a una runqueue non sua:

- Si deve effettuare il load balancing: ad esempio, un determinato CPU-core non ha più thread eseguibili nella propria runqueue per cui deve andare ad attingere da qualche altra runqueue (badiamo che il load balancing viene fatto anche da un qualche demone di sistema apposito).
- Un thread A correntemente in esecuzione su un certo CPU-core<sub>A</sub> deve risvegliare un altro thread B che però non è affine col medesimo CPU-core<sub>A</sub>: in tal caso, A porrà il TCB del thread B all'interno della runqueue associata a un altro CPU-core<sub>B</sub> (per cui è richiesto che CPU-core<sub>A</sub> acceda alla runqueue di CPU-core<sub>B</sub>), e l'API utilizzata per questa evenienza è **void ttwu\_queue (struct task\_struct \*p, int cpu, int wake\_flags)**.

#### Active queue vs expired queue:

I thread eseguibili, quando esauriscono i loro tick, vengono migrati dall'**active queue** (la runqueue così come la conosciamo) all'**expired queue** (che è appunto la coda dei task scaduti). Nel passaggio dall'active queue all'expired queue si calcola anche il numero di tick da spendere nella prossima epoca, in modo da non avere un costo O(n) neanche per la riassegnazione dei tick ai vari thread attivi. Nel momento in cui l'epoca corrente termina, si esegue un'operazione semplicissima dal costo O(1) che consiste nello scambiare il puntatore all'active queue col puntatore all'expired queue, cosicché l'active queue diventa l'expired queue e viceversa.

#### Wakeup

Con ttwu\_queue, un thread ad altissima priorità su cpu0 può risvegliare altro thread importante su altra cpu1 e mandargli un interrupt.

#### Gestione delle priorità:

Le priorità vengono gestite in maniera differente sui thread real-time e sui thread "user". In particolare:

-> Per quanto riguarda i thread real-time, le priorità sono **statiche** e sono proporzionali al numero di tick assegnati.

-> Per quanto riguarda i thread non real-time, le priorità sono **dinamiche** (in realtà ne hanno anche una statica, quando entrano nella runqueue): di fatto, partono con un livello di priorità iniziale proporzionale al numero di tick assegnati e poi, dopo aver speso una percentuale di tick in CPU, possono subire un cambio di priorità dinamico (tipicamente un miglioramento). Tale miglioramento è deciso direttamente dal sistema operativo: se un thread viene deschedulato dalla CPU perché va nello stato di wait, quando poi torna a essere ready riceve il miglioramento della priorità, altrimenti no. In altre parole, vengono premiati i cosiddetti **thread interattivi**, ovvero quelli che trascorrono una sufficiente quantità di tempo a dormire. Ciò porta anche al fenomeno dell'**inversione di priorità**, per cui un thread A **non real-time** con priorità di base (= priorità statica) pari a X potrebbe superare un altro thread B **non real-time** con priorità di base pari a Y<X (per cui B in principio era più prioritario di A); questo non è molto fair, specie nel momento in cui B viene scavalcato da un numero elevato di thread subendo così una sorta di starvation. Ciò vale per i NON real time, quindi non interferiamo mai con loro. Vale solo per SCHED\_OTHER. I tick sono assegnati per ogni epoca, non sono uguali. Tutti ricevono qualche tick.

Nel load sharing (o time sharing) il numero di tick da assegnare a ciascun thread è basato sulla nozione di **carico (load)**, che è un'informazione mantenuta in un nuovo campo del TCB che è:

```
struct sched_entity {
    struct load_weight load;
    ...
}
```

Dove:

```
struct load_weight {
    unsigned long weight;
```

```
        u32 inv_weight;
    }
```

Il valore weight all'interno di struct load\_weight è quello che determina il carico di ciascun thread e viene stabilito sulla base della niceness.

#### Explicit stack refresh:

Supponiamo di avere una funzione di livello kernel in cui viene letta una variabile x dalla per-CPU memory e, successivamente, un'altra variabile y viene caricata col valore di x. Supponiamo inoltre che, tra le due operazioni precedentemente descritte, il thread venga deschedulato dal CPU-core<sub>A</sub>. Se poi viene rischedulato su un CPU-core<sub>B</sub> diverso, abbiamo un'inconsistenza sul valore di x. Questo può essere ad esempio il caso della funzione schedule() che, per gestire le runqueue, può utilizzare delle informazioni specifiche per la CPU su cui sta girando.

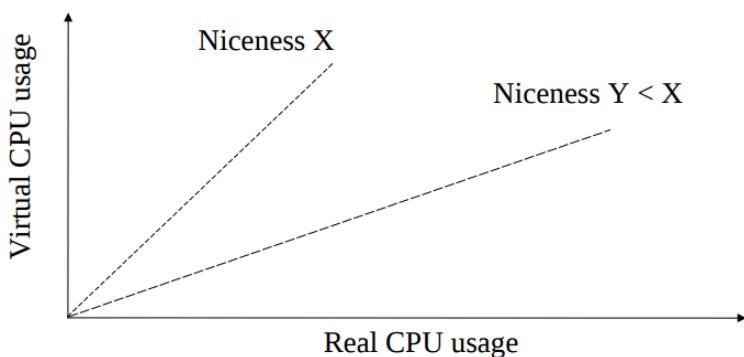
Quello che si fa per evitare problemi è ripopolare le variabili per-CPU (come x nel nostro esempio) ogni volta che si riprende il controllo della CPU; tale meccanismo è noto come **explicit stack refresh**.

#### **Completely fair scheduler**

Rispetto al load balancing scheduler presenta una differenza sostanziale sulla struttura delle runqueue: in particolare, i 100 livelli di priorità riservati ai thread real-time rimangono strutturati con delle sotto-runqueue esattamente come prima (per cui il costo per schedulare un task real-time rimane sempre O(1)), mentre i restanti 40 livelli di priorità vengono collassati in un unico **albero rosso-nero**, la cui navigazione ha chiaramente un costo logaritmico. Tipicamente si prende nodo out-boundary sulla sinistra. La priorità non è basata sul livello, ma in base al path. Deve essere ristrutturato a runtime.

Più precisamente, all'interno dell'albero rosso-nero, i thread sono ordinati in base al **tempo di VCPU (Virtual CPU)**, dove minore è tale tempo (dato dal tempo di CPU diviso il peso), migliore è il posizionamento del thread nell'albero. In questo modo, l'ordinamento effettivo dei vari task all'interno del red-black tree rispecchia le priorità dinamiche molto meglio di quanto lo fa l'euristica implementata nel load balancing scheduler (per cui nessun thread viene veramente penalizzato).

Nel completely fair scheduler viene utilizzata la struttura load\_weight (i.e. il peso) per tenere traccia dell'avanzamento del tempo di VCPU per i vari task. Per ciascun thread T, il tempo di VCPU viene calcolato normalizzando il tempo reale (misurato con la granularità dei nanosecondi) trascorso da T in CPU rispetto al peso assegnato a T. In altre parole, dato il tempo reale trascorso in CPU, il tempo di VCPU dipende dal peso del thread T; in particolare, maggiore è il peso, minore risulta essere il tempo di VCPU.



Grazie al meccanismo appena descritto, si previene lo scenario in cui un certo thread non real-time (ma comunque altamente prioritario) venga sistematicamente superato da altri thread di base meno prioritari ma che vanno nello stato di wait molto più spesso. Se VCPU è alto (niceness è scarsa) è più difficile sorpassare, e dipende da come ci si comporta in CPU, non nella runqueue. Quindi, questo nuovo ordinamento dipende anche dal tempo di uso di CPU, ne un thread ne usa pochissima, ha senso farlo passare avanti.

## Kernel thread

### Kernel 2.4:

Nella versione 2.4 del kernel, per generare un nuovo thread di livello kernel, è sufficiente invocare la funzione **kernel\_thread()** che, internamente, invoca a sua volta la funzione **arch\_kernel\_thread()**. In particolare, quest'ultima è una funzione ASM e ci permette di distinguere in maniera machine dependent qual è il thread parent e qual è il thread child a seguito dell'esecuzione. Come è possibile immaginare, **arch\_kernel\_thread()** chiama internamente **sys\_clone()**. A valle di tutto questo, il thread child, dopo aver stabilito di essere effettivamente il child, chiama lui stesso con un'istruzione di call la funzione da cui lui dovrebbe iniziare l'esecuzione. Prima era facile, solo INT 0X80.

Per capire chi è il thread parent e chi è il thread child, all'interno della funzione **arch\_kernel\_thread()** si sfruttano i registri esp, esi. Per essere più precisi, di seguito è riportata l'implementazione di tale funzione:

Il primo 0x80 è clone, e lo chiama il thread parent, il secondo 0x80 è exit, e lo chiama il thread child.

```
int arch_kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
    long retval, d0;

    __asm__ __volatile__(
        "movl %%esp,%%esi\n\t"
        "int $0x80\n\t"           /* Linux/i386 system call */
        "cmpb %%esp,%%esi\n\t"   /* child or parent? */
        "je 1f\n\t"              /* parent - jump */
        /* Load the argument into eax, and push it. That way, it does
         * not matter whether the called function is compiled with
         * -mregparm or not. */
        "movl %4,%eax\n\t"
        "pushl %%eax\n\t"
        "call *%5\n\t"           /* call fn */
        "movl %3,%0\n\t"          /* exit */
        "int $0x80\n"
        "1:\n\t"
        :"=&a" (retval), "=S" (d0)
        :"0" (__NR_clone), "i" (__NR_exit),
        "r" (arg), "r" (fn),
        "b" (flags | CLONE_VM)
        : "memory");

    return retval;
}
```

Chiamata a \_\_NR\_clone

Queste istruzioni vengono eseguite solo dal child; la call invoca la funzione associata al child.

### API più recenti:

Ora dobbiamo usare syscall, fare dei controlli etc..

-> struct task\_struct \*kthread\_create (int (\*function)(void \*data), void \*data, const char name[]): è la nuova API per creare un nuovo thread di livello kernel e accetta come parametri la funzione che quel thread deve eseguire, l'argomento di quella funzione e il nome del thread. La cosa interessante è che il thread che stiamo creando viene subito posto a dormire su una waitqueue, per cui è necessario attivarlo esplicitamente. Per quanto riguarda le segnalazioni, è possibile colpire il thread creato solo se lui stesso o chi lo ha generato ha attivato questa possibilità attraverso la funzione **allow\_signal(int signal\_code)**; colpire il thread ha come unico effetto quello di svegliarlo (se già non era in runqueue), ma non gli viene consegnato alcun messaggio; infine, l'invio al thread di un segnale di kill avrà effetto solo nel momento in cui il thread stesso verificherà se ha ricevuto una qualche segnalazione mediante un meccanismo basato su polling. Ogni thread così creato, nel suo TCB ha info di controllo che gli dicono che non è killabile, allora deve aggiornare queste informazioni (come abilitare la possibilità di essere colpito tramite segnalazioni).

-> struct task\_struct \*kthread\_create\_on\_cpu (int (\*function)(void \*data), void \*data, unsigned int cpu\_id, const char name[]): è come kthread\_create() ma, in più, definisce anche un'affinità del thread creato col CPU-core specificato come parametro.

### Caso delle macchine single-core

Le macchine single-core tradizionali prevedono soltanto:

-> Le **trap**, che sono eventi sincroni legati all'esecuzione del software. Il flusso di esecuzione stesso genera un'interruzione sul flusso di esecuzione.

-> Gli **interrupt**, che sono eventi asincroni provenienti da dispositivi hardware esterni. E' esterno al flusso di esecuzione.

Il modo classico di gestire gli eventi consiste nell'eseguire l'apposito codice del sistema operativo **sull'unico CPU-core del sistema** nel momento in cui tali eventi vengono accettati. Già così si consegna la consistenza dello stato dell'hardware anche nelle applicazioni multi-thread: di fatto, tutti i thread sono subito in grado di vedere i cambiamenti dello stato della CPU. Il cambio di stato è osservabile da tutti, e se necessario alla correttezza dell'esecuzione, è garantito a tutti.

#### Esempio:

Consideriamo la trap data dalla system call munmap(), che unmappa determinate pagine di memoria. Il gestore di tale trap, oltre ad aggiornare la page table del processo chiamante, deve anche andare a invalidare il TLB presente nel processore. Fatto questo, poiché il processore è unico, qualunque thread che verrà schedulato vedrà il TLB correttamente invalidato, per cui avrà la visione della memoria aggiornata.

### Caso delle macchine multi-core

Tornando all'esempio della munmap() (togliamo cose dall'AS, e ciò porta ad update page table, e quindi anche del TLB), nel caso dei sistemi multi-core il discorso è più complesso. Supponiamo di avere due thread  $t_0, t_1$  che girano rispettivamente su CPU-core<sub>0</sub> e CPU-core<sub>1</sub> e utilizzano il medesimo address space (perché magari sono thread di uno stesso processo). Supponiamo inoltre che  $t_0$  sia coinvolto nella trap data dalla system call munmap(). Allora, CPU-core<sub>0</sub> (in particolar modo il TLB di CPU-core<sub>0</sub>) passa da uno stato A a uno stato B, mentre  $t_1$  non si accorge della trap, per cui CPU-core<sub>1</sub> resta nello stato iniziale A; quindi, se al thread  $t_1$  serve conoscere lo stato della memoria logica, si ritroverà delle informazioni scorrette (non aggiornate).

Di conseguenza, sono necessari meccanismi per propagare i cambi di stato dei CPU-core; tali meccanismi possono essere di due tipologie:

- **Hardware**: i gestori di trap e interrupt sono implementati direttamente in protocolli a livello del firmware; questa soluzione va bene nel momento in cui, dato un particolare evento, la sua gestione deve essere deterministica. Altra limitazione è data dal fatto che, threadA non sa cosa stia usando threadB, quindi deve, per sicurezza, comunicare l'invalidazione del TLB a tutti, che porta ad altre invalidazioni di altri TLB associati ad altri thread.

- **Software**: gli eventi vengono propagati e gestiti dal sistema operativo.

### IPI (Inter Processor Interrupt)

Costituisce un terzo tipo di evento oltre alla trap e all'interrupt, e può triggerare la richiesta di esecuzione di specifiche porzioni di sistema operativo su molteplici CPU-core: in particolare, un CPU-core è la sorgente dell'IPI, mentre uno o più CPU-core possono essere i destinatari dell'IPI. Si tratta di un evento sincrono per il CPU-core sorgente, mentre è asincrono per i CPU-core destinatari.

In un certo senso, l'IPI è utilizzato per definire un protocollo di tipo request / reply, in cui la request non è altro che una richiesta da parte del CPU-core sorgente di effettuare un cambio di stato, mentre la reply è l'esecuzione vera e propria del cambio di stato da parte dei CPU-core destinatari.

Gli IPI richiedono un supporto a livello firmware per generarli, ma poi vengono effettivamente processati a livello del software.

Esistono almeno due livelli di priorità per gli IPI:

-> **Priorità alta**: porta al processamento immediato degli IPI.

-> **Priorità bassa**: porta all'accodamento degli IPI che poi verranno processati nel momento più opportuno, in stile top half / bottom half (quest'ultimo viene processato dopo).

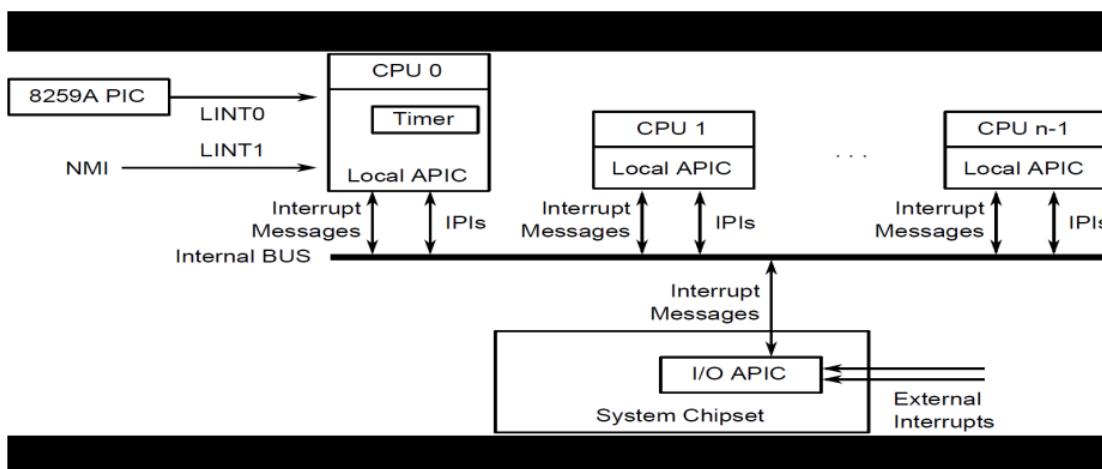
### Supporto hardware nelle macchine x86

Nei sistemi x86 il sistema di base che permette la gestione degli interrupt si chiama **APIC (Advanced Programmable Interrupt Controller)**, che offre a ciascun CPU-core un'istanza locale detta **LAPIC (Local APIC)**; il LAPIC-T di cui abbiamo già parlato è proprio un componente del LAPIC. Il LAPIC offre anche degli pseudo-registri che possono essere usati per postare degli IPI (i.e. inviare degli IPI ad altri CPU-core).

Inoltre, è necessario un bus ad-hoc (**APIC bus**) per far viaggiare le richieste IPI tra un LAPIC e un altro (i.e. tra un CPU-core e un altro). La cpu0 è retrocompatibile rispetto al PIC, e il suo uso comporta l'esclusione di altri aspetti.

APIC, oltre ai vari LAPIC, è costituito da un ulteriore componente, l'**I/O APIC**, che si occupa di accettare gli interrupt esterni (quelli classici) e di inoltrare i messaggi di interrupt ai CPU-core interessati, sfruttando sempre l'APIC bus. Possiamo ricevere interruzioni da dischi SATA, ad esempio. Non ci serve nel lavoro con gli IPI, bensì ci bastano solo i LAPIC.

Di seguito è mostrato uno schema riassuntivo dell'architettura di APIC:



APIC più precisamente supporta fino a 16 tipi di interrupt interni (generati da un qualche CPU-core e rivolto ad altri CPU-core) e non più di 256 tipi di interrupt esterni (generati da un dispositivo hardware esterno). Gli interrupt interni sono riassunti nella seguente tabella:

<b>IRQ 0</b>	System timer. Reserved for the system. Cannot be changed by a user.
<b>IRQ 1</b>	Keyboard. Reserved for the system. Cannot be altered even if no keyboard is present or needed.
<b>IRQ 2</b>	Second IRQ controller. See below for explanation.
<b>IRQ 3</b>	COM 2(Default) COM 4(User)
<b>IRQ 4</b>	COM 1(Default) COM 3(User)
<b>IRQ 5</b>	Sound card (Sound Blaster Pro or later) or LPT2(User)
<b>IRQ 6</b>	Floppy disk controller
<b>IRQ 7</b>	LPT1(Parallel port) or sound card (8-bit Sound Blaster and compatibles)
<b>IRQ 8</b>	Real time clock
<b>IRQ 9</b>	ACPI SCI or ISA MPU-401
<b>IRQ 10</b>	Free / Open interrupt / Available
<b>IRQ 11</b>	Free / Open interrupt / Available
<b>IRQ 12</b>	PS/2 connector Mouse / If no PS/2 connector mouse is used, this can be used for other peripherals
<b>IRQ 13</b>	Math co-processor. Cannot be changed
<b>IRQ 14</b>	Primary IDE. If no Primary IDE this can be changed
<b>IRQ 15</b>	Secondary IDE

## Nomenclatura/ identificazione degli interrupt :

-> **IRQ** = codice associato a una richiesta di interrupt. In altre parole è il codice che indica qual è la linea su cui l'interrupt è mandato in esercizio. Vengono da I/O APIC.

-> **INT** = interrupt line per come è vista dal sistema operativo. Essenzialmente, si ha che INT = F(IRQ), dove F() è una funzione tipicamente hardware specific; nel caso dei processori x86 abbiamo che INT = IRQ+32, il che vuol dire che le prime 32 linee di interrupt sono riservate a scopi diversi dalla gestione degli interrupt e, in particolare, alle **trap** predefinite dell'architettura hardware.

### I/O APIC:

Tiene traccia del numero di CPU-core che si trovano nell'architettura, che corrisponde al numero massimo di CPU-core che possono ricevere uno stesso interrupt. Può effettuare una selezione dei CPU-core specifici verso cui inoltrare ciascuna richiesta di interrupt. Dipendentemente dalla natura dell'interrupt, la politica di selezione dei CPU-core può essere:

- **Fissa**: la richiesta di interrupt viene inoltrata verso un singolo CPU-core predefinito.

- **Logica**: la richiesta di interrupt può coinvolgere diversi CPU-core, dove ne viene selezionato uno per volta secondo uno schema round-robin.

### Interfaccia Linux per APIC:

-> **/proc/interrupt** fornisce il resoconto della consegna degli interrupt ai vari CPU-core.

-> **/proc/irq<IRQ num>/smp\_affinity** indica qual è l'affinità dei vari interrupt coi CPU-core.

Il setup di questi pseudo-file avviene durante la fase di boot del kernel e, almeno in un primo momento, è osservabile nel buffer dmesg (che, successivamente, potrebbe essere sovrascritto).

Allo startup di Linux, Linux stesso decide la configurazione dell'architettura, inclusi gli handler esatti, collegati a tabella core. Cosa non banale. Ad esempio, potremmo confluire tutti gli interrupt nella CPU0. Con `cat /proc/interrupts` possiamo vedere tale gestione degli interrupt, come gli shutdown dei TLB, oppure `Function call interrupts`, che ci permette di avvertire tutte le CPU per fargli eseguire una funzione specifica, senza cambiare TLB, (ne è un esempio `cpuid`, che è per-cpu-variabile, e ogni cpu scrive su un'area il proprio id, flushando la pipeline. Con questo supporto di interrupt IPI, facciamo il setup degli id per-cpu).

### Ulteriori dettagli sulla IDT (Interrupt Descriptor Table)

Sappiamo che è una tabella contenente gli entry point (i GATE) a tutti i gestori degli interrupt.

Nelle macchine x86 (sia protected che long) è composta da 256 entry, che corrispondono proprio al numero massimo di IRQ vector (interrupt esterni) che è possibile generare con la I/O APIC; ciascuna entry è ampia 2x32 bit nelle macchine a 32 bit (protected mode) ed è ampia 2x64 bit nei sistemi x86/64 (long mode).

Analizziamo brevemente le 256 entry della IDT:

Vector range	Use	Back here in a while
0-19 (0x0-0x13)	Nonmaskable interrupts and exceptions	
20-31 (0x14-0x1f)	Intel-reserved	
32-127 (0x20-0x7f)	External interrupts (IRQs)	
128 (0x80)	Programmed exception for system calls (segmented style)	
129-238 (0x81-0xee)	External interrupts (IRQs)	
239 (0xef)	Local APIC timer interrupt	
240-250 (0xf0-0xfa)	Reserved by Linux for future use	
251-255 (0xfb-0xff)	Inter-processor interrupts	

The mixture changes with kernel releases (e.g. 255 is spurious)

L'ultima è spuria. Le entry nella IDT devono essere valide, altrimenti non si sa l'indirizzo dell'oggetto da

attivare, portando ad errore sul flusso di esecuzione corrente. Se un thread livello kernel riceve INT, che targhiamo con codice P (es: 0x80), allora il sottosistema di controllo usa tale codice per deferenziarsi sulla tabella. Allora, nel caso 255 spuria, si registra solo la chiamata ad un handler, senza fare null'altro. Quindi è riutilizzabile. Posso avere altre entry allo stesso handler spurio. Quindi è come una nysis\_call. Cioè ogni entry mi deve portare da qualche parte, quelle spurie mi portano ad uno stesso handler, chiamato handler spurious.

Registro idtr:

È il registro packed che mantiene le seguenti informazioni:

- L'indirizzo virtuale (o meglio, lineare) della IDT. (pointer)
- Il numero delle entry correntemente valide nella IDT. (taglia)

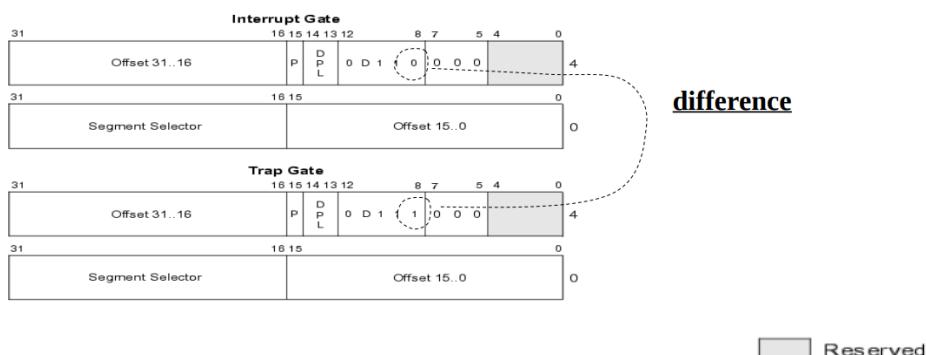
```
Struct desc_struct{
    unsigned long a,b;
}
```

Normalmente, ne viene usata solo una, puntata da tutti. Potremmo comunque usarne altre.

Idtr può essere acceduto mediante le istruzioni **LIDT** (istruzione di load) e **SIDT** (istruzione di store); in particolare, la prima di queste due istruzioni, effettuando una scrittura all'interno del registro idtr, è in grado anche di cambiare la IDT in uso sia per uno specifico CPU-core che per tutti i CPU-core. Quest'ultima operazione risulta essere molto importante nel momento in cui si vuole semplicemente cambiare il contenuto di una entry della IDT ma tale modifica non può essere svolta in modo atomico, per cui potrebbe portare a dei problemi di sincronizzazione con gli altri thread che girano in altri CPU-core. Di fatto, la soluzione a questo problema consiste nel definirsi da capo una nuova IDT identica all'originale eccetto per la entry che si voleva modificare; dopodiché, è possibile modificare il registro idtr facendolo puntare alla nuova IDT e dar luogo a un IPI in modo tale da propagare l'aggiornamento anche agli altri CPU-core.

IDT in x86 protected mode:

Qui ciascuna entry della IDT viene descritta tramite la struttura **struct desc\_struct** che è composta da due campi di tipo **unsigned long**. Analizziamoli:



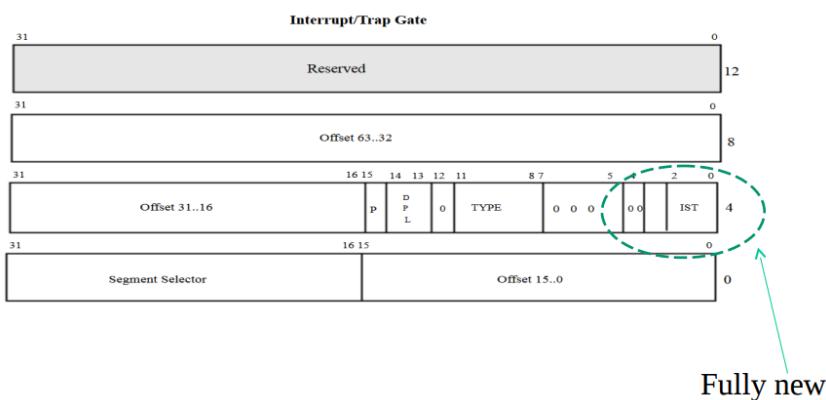
-> **DPL (Descriptor Privilege Level)**: indica il livello di privilegio minimo che bisogna avere per accedere al gestore dell'interrupt associato a questa entry della IDT.

-> **Offset**: identifica il posizionamento del gestore dell'interrupt.

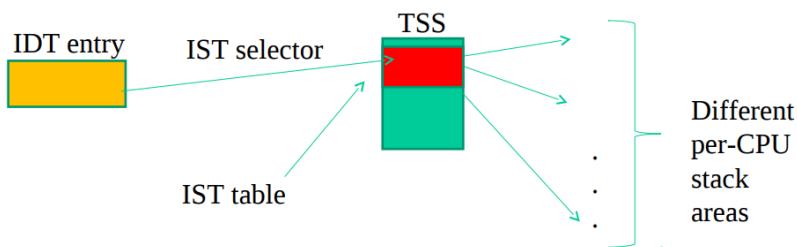
-> **Segment selector**: indica il segmento in cui è possibile trovare il gestore dell'interrupt.

IDT in x86 long mode:

Qui ciascuna entry della IDT è strutturata nel seguente altro modo:



-> **IST (Interrupt Stack Table)**: è un codice a 3 bit associato a una particolare tabella di stack; ciascuno degli 8 valori possibili identifica uno degli stack che possono essere utilizzati nella gestione dell'interrupt (7 + quello di default). In x86 protected mode questo meccanismo basato su molteplici stack non è previsto, il che è problematico nel momento in cui si genera uno stesso interrupt in maniera innestata: di fatto, alla seconda occorrenza dell'interrupt, il gestore coinvolto dovrà lavorare sullo stesso stack usato per la prima occorrenza dell'interrupt, ovvero sullo stack di livello kernel del thread correntemente in esecuzione.



La macro **HML\_TO\_ADDR (offset\_high, offset\_middle, offset\_low)** converte i tre campi *offset* della entry della IDT nell'indirizzo lineare vero e proprio corrispondente.

Un nuovo campo presente, è quello di IST “interrupt stack table”. Per ogni cpu attiva, c’è registro che ci permette di arrivare su TSS, metadati basici per thread in cpu, come area di memoria dello stack pointer. Le stack area, dove le prendiamo? C’è allocatore apposito con kmalloc. La cosa interessante è che vi possiamo scrivere un codice, che ci manda in TSS per prelevare la entry dello stack pointer da usare. Lo stack pointer può puntare a stack area “trampolino”, altrimenti non avremmo visibilità di altre stack area senza questo IST. Se pari a 0, si usa lo stack target classico, usiamo quello del thread.

### Macro per settare le entry della IDT

Caso di x86 protected mode:

- > **set\_trap\_gate (displacement, &symbol\_name)**: inizializza una entry della IDT associata a una trap in modo tale che l’accesso a tale entry sia concesso soltanto al livello di privilegio pari a 0.
- > **set\_intr\_gate (displacement, &symbol\_name)**: inizializza una entry della IDT associata a un interrupt in modo tale che l’accesso a tale entry sia concesso soltanto al livello di privilegio pari a 0.
- > **set\_system\_gate (displacement, &symbol\_name)**: inizializza una entry della IDT associata a una trap (più esattamente a un punto di accesso al kernel) in modo tale che l’accesso a tale entry sia concesso a tutti i livelli di privilegio (i.e. dal 3 in giù).

Il parametro displacement di queste tre macro indica la entry della IDT su cui si vuole lavorare, mentre il parametro &symbol\_name identifica il displacement del segmento (a partire da 0x0) in cui è posizionato il gestore della trap o dell’interrupt. Quindi, l’indirizzo per colpire l’offset deve essere anticipato da &.

Caso di x86 long mode (kernel 3):

Qui si hanno delle macro analoghe a quelle definite per i sistemi x86 protected mode:

- > `set_system_intr_gate (displacement, &symbol_name)`
- > `set_system_trap_gate (displacement, &symbol_name)`
- > `set_trap_gate (displacement, &symbol_name)`

Caso di x86 long mode (kernel 4-5):

Qui viene fatto tutto con un'unica API:

-> `static inline void native_write_idt_entry (gate_desc *idt, int entry, const gate_desc *gate)`: non fa altro che implementare una `memcpy()` per modificare la entry della IDT; il parametro `gate` contiene le informazioni da riportare all'interno della entry.

### Entry della IDT disponibili e riservate

DT ALLOCATA a tempo di compilazione. Prime 32 entry fisse Intel, handler vedo io. Processano eventi che l'hardware riconosce, come page fault e divisioni per 0.

In x86 le prime 32 entry (da 0 a 31) sono riservate agli handler utilizzati per gestire gli **eventi predefiniti**, che principalmente sono trap (e.g. divisione per 0, page fault). Tutte le altre entry, invece, sono disponibili per qualunque scopo legato alla programmazione di sistema: ad esempio, sappiamo che la entry 0x80 viene usata tradizionalmente per l'accesso al livello kernel attraverso l'invocazione alle system call.

Badiamo che, per alcune entry riservate, i task implementati dal microcodice di accettazione della trap (i.e. il firmware) generano un **codice di errore** che dovrà essere passato all'handler della trap. Di conseguenza, l'handler ha bisogno di essere strutturato in una maniera tale da essere *aware* della produzione del codice di errore. A tal proposito, l'handler dovrebbe essere lanciato da un **dispatcher** (che vedremo meglio tra poco) che viene lanciato a sua volta da un **preambolo** e ha il compito di passare all'handler informazioni come lo snapshot di CPU e il codice di errore stesso (che indica il motivo per cui la trap è stata generata; ad esempio, per quanto riguarda l'handler dei page fault, indica la causa che ha scatenato il page fault). D'altro canto, per le entry della IDT non riservate, non deve essere generato alcun codice di errore.

La tabella IDT deve stare sempre in zona OPEN. Come cambio IDT su tutti i processori?  
(è unica, ma voglio aggiornare per tutti il riferimento).

Usiamo IPI, facciamo eseguire la funzione del modulo di aggiornamento, per portarci sulla nuova tabella.

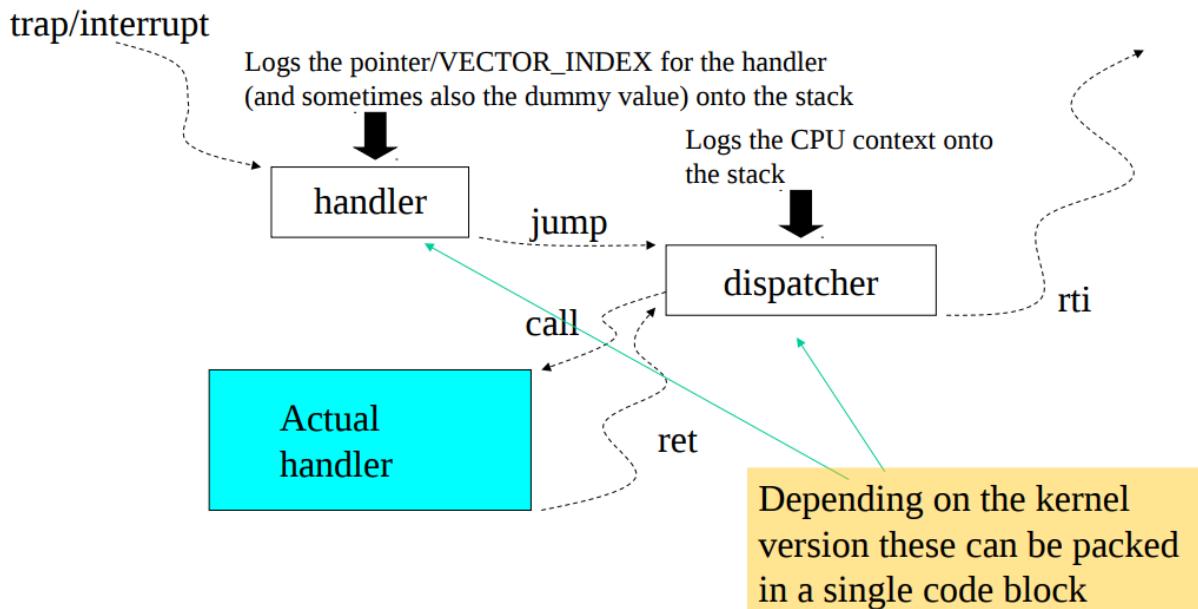
### Gestione modulare degli handler

Gli handler di trap / interrupt vengono gestiti mediante un **dispatcher** unico in tutto il sistema. Vediamo esattamente come.

Inizialmente si ha un **preambolo** (un handler di primo livello) che memorizza nella stack area il codice di errore relativo all'evento oppure, nel caso in cui non debba essere generato alcun codice di errore, memorizza un valore dummy (viene scritto a prescindere qualcosa in modo tale che lo stack abbia sempre la stessa struttura, il che semplifica operazioni come la deallocazione dello stack stesso); dopodiché riporta sullo stack anche l'indirizzo della funzione che implementa un handler di secondo livello. Infine, invoca un oggetto intermedio che è proprio il dispatcher. Quest'ultimo va a introdurre ulteriori informazioni all'interno della stack area (come lo snapshot di CPU) e va a invocare l'handler di secondo livello; l'handler esatto di secondo livello che viene invocato dipende dalla funzione che era stata riportata sullo stack dall'handler di primo livello, e ciò dipende a sua volta da qual è stato l'evento che si è generato. L'handler di secondo livello, infine, utilizza le informazioni salvate nella stack area per effettuare la gestione vera e propria dell'evento; tipicamente fa uso dell'error code e di un parametro di input (un puntatore a dei dati) che era stato precedentemente posto sulla stack area dal dispatcher.

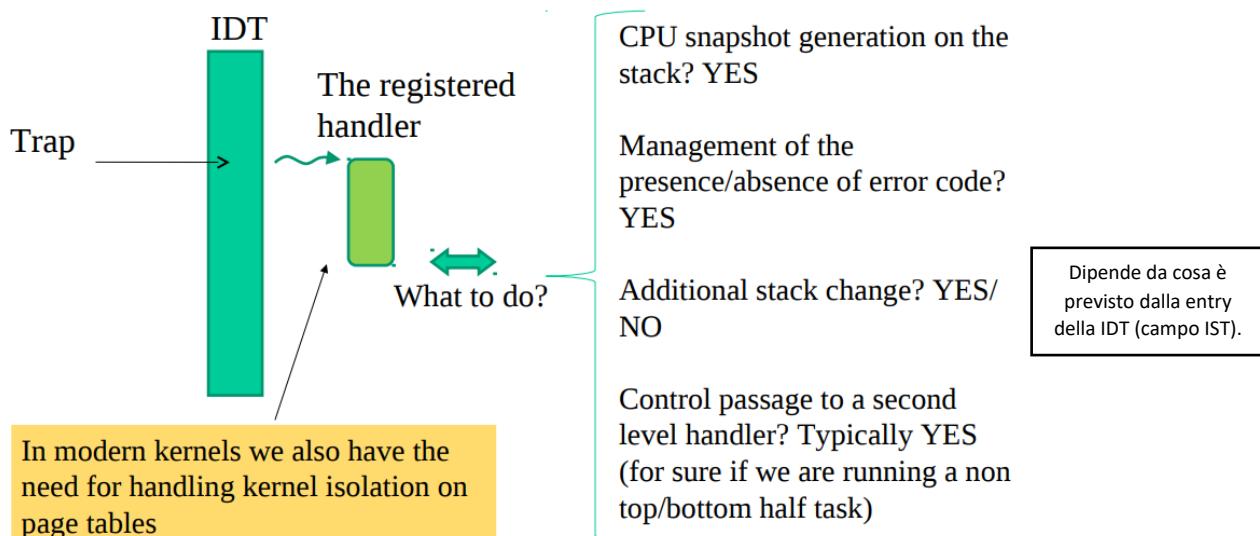
La gestione modulare appena descritta risulta vantaggiosa dal punto di vista della semplicità del codice: ad esempio, permette di avere il preambolo e il dispatcher machine dependent, mentre l'handler di secondo livello viene scritto col linguaggio C. Si ha così una separazione tra i moduli machine dependent e i moduli machine independent.

Di seguito è riportato uno schema riassuntivo del meccanismo appena spiegato:

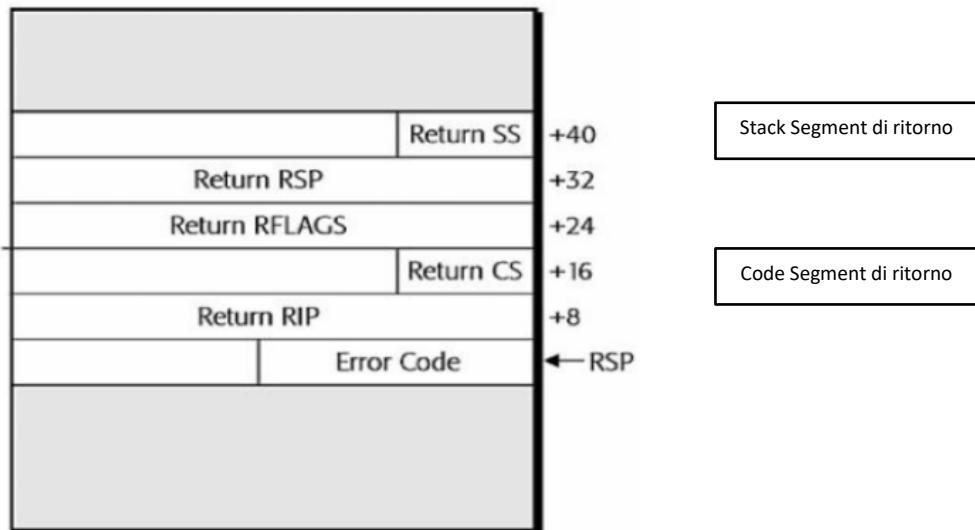


Il gestore, a più alto livello, riconosce il codice che ha fatto partire il gestore, e il pointer ad una fotografia del software.

Recap sulle azioni degli hadler di primo livello:



Dettagli sulla conformazione dello stack usato dall'handler:

Dettagli sullo snapshot di CPU:

-&gt; Caso di i386:

```
struct pt_regs {
    long ebx;          long ecx;
    long edx; long esi;
    long edi; long ebp;
    long eax; int xds; int xes;
    long orig_eax; long eip; int xcs;
    long eflags; long esp; int xss;
}
```

Orig\_eax contiene il codice di errore (se c'è) oppure il valore dummy (0) che lo sostituisce.

-&gt; Caso di x86 long mode:

```
struct pt_regs {
    unsigned long r15; ... unsigned long r12;
    unsigned long bp;
    unsigned long bx; /* arguments: non interrupts/non tracing syscalls only save up to here */
    unsigned long r11; ... unsigned long r8;
    unsigned long ax;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long orig_ax; /* end of arguments */ /* cpu exception frame or undefined */
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss; /* top of stack page */
}
```

Esempio (page fault handler):

L'handler di secondo livello del page fault è la funzione **do\_page\_fault (struct pt\_regs \*regs, unsigned long error\_code)**, dove regs è un puntatore all'area di memoria generata dal software machine dependent (i.e.

dal preambolo e dal dispatcher). I tre bit meno significativi di error\_code specificano il tipo di page fault occorso secondo le seguenti regole:

- Bit 0: vale 0 se la pagina non è stata trovata, vale 1 se c'è stato un problema relativo al livello di protezione della pagina.
- Bit 1: vale 0 se il fault è avvenuto in lettura, vale 1 se il fault è arrivato in scrittura.
- Bit 2: vale 0 se la pagina coinvolta è di livello kernel, vale 1 se la pagina è di livello user.

### **Installazione di nuovi handler di trap / interrupt**

Se si vuole definire un nuovo handler di trap / interrupt all'interno del kernel, si scrive una funzione f machine independent all'interno di un nuovo modulo da montare. Tuttavia, se supponiamo di lavorare con la PTI (Page Table Isolation) attivata, non è possibile scrivere l'indirizzo di f direttamente all'interno di una entry libera della IDT: infatti f, trovandosi nel nuovo modulo del kernel, non può essere visibile al livello user poiché il suo indirizzo non figura all'interno della page table di livello user; inoltre, quando si accede alla IDT, si è ancora in user mode, e convenzionalmente lo switch tra la page table di livello user e la page table di livello kernel avviene tra l'handler di primo livello (i.e. il preambolo) e il dispatcher. Di conseguenza, affinché l'inserimento del nuovo handler abbia successo, è necessario adottare un meccanismo di **binary patching**, in cui si prende in considerazione l'indirizzo di un handler di secondo livello già esistente che viene posto nella stack area da parte di un certo handler di primo livello e lo si sostituisce con l'indirizzo dell'handler (di secondo livello) che stiamo introducendo noi (quindi essenzialmente stiamo patchando l'handler di primo livello). Ma esistono handler di secondo livello che possono essere sostituiti? Sì, e sarebbero i gestori degli **interrupt spuri**, che sono interrupt associati alle linee di interruzione che non sono realmente in uso.

### **Approfondimento sugli IPI**

La gestione immediata degli IPI è consentita nel caso in cui non richiede l'accesso a delle strutture dati condivise tra più CPU-core. Un esempio di questo scenario è dato dal segnale di panico, che infatti porta al blocco immediato di tutti i CPU-core.

Oltre a trasmettere segnali di panico, gli IPI vengono sfruttati anche per altri scopi principali:

- Esecuzione parallela di una stessa funzione su tutti i CPU-core. In uno scenario del genere, la funzione in oggetto è detta **funzione smp**. Qui la entry della IDT che viene coinvolta è la **CALL\_FUNCTION\_VECTOR** e può essere usata chiamando la funzione **smp\_call\_function()**.
- Cambio dello stato dei CPU-core (e.g. aggiornamento dei TLB). Qui una entry della IDT che può essere coinvolta è la **INVALIDATE\_TLB\_VECTOR** e può essere usata chiamando la funzione **invalidate\_interrupt()**.
- Richiesta a un qualche CPU-core di invocare la funzione **schedule()** per cambiare il proprio thread in esecuzione. Questo scenario può verificarsi ad esempio quando un thread  $t_0$  che gira sul CPU-core $_0$  a un certo punto risveglia un thread  $t_1$  che è prioritario ma ha affinità solo col CPU-core $_1$ ; allora  $t_0$  invia un IPI verso il CPU-core $_1$  per far eseguire la funzione **schedule()** sul CPU-core $_1$ . Qui la entry della IDT che viene coinvolta è la **RESCHEDULE\_VECTOR** e può essere usata chiamando la funzione **reschedule\_interrupt()**.

#### API per gli IPI:

- > **send\_IPI\_all()**: invia un IPI a tutti i CPU-core (incluso se stesso).
- > **send\_IPI\_allbutself()**: invia un IPI a tutti i CPU-core escluso se stesso.
- > **send\_IPI\_self()**: invia un IPI solo a se stesso.
- > **send\_IPI\_mask()**: invia un IPI a un sottoinsieme di CPU specificato da una bitmask.

#### Sequenzializzazione della gestione degli IPI:

Per quanto riguarda la sincronizzazione, non ci sono particolari problemi ad esempio per gli eventi corrispondenti alle entry della IDT INVALIDATE\_TLB\_VECTOR e RESCHEDULE\_VECTOR: infatti, quando è richiesto il cambio di stato dei CPU-core, ciascun core lavora esclusivamente su dei registri o su dei dati locali (e.g. il TLB), mentre quando è richiesta la preemption su un certo CPU-core, viene semplicemente

acceduto il TCB del thread in esecuzione su quel core e nient'altro.

Per quanto invece riguarda CALL\_FUNCTION\_VECTOR, la situazione è un po' più complicata: poiché è previsto che i CPU-core eseguano concorrentemente una stessa routine f del kernel, la funzione smp\_call\_function() inserisce all'interno di un'apposita area di memoria l'indirizzo di f e l'argomento da passare a f (che, come al solito, può essere un puntatore a dei dati). Dopodiché, i CPU-core colpiti dall'IPI accederanno concorrentemente in lettura a tale area di memoria, la quale, quindi, non potrà essere riutilizzata da un altro IPI fin tanto che non sarà terminata l'interazione corrente tra i CPU-core. A valle di queste considerazioni, viene introdotta la necessità di sequenzializzare la gestione degli IPI.

Di seguito è riportata una versione un po' datata di smp\_call\_function():

```

207 int smp_call_function(void (*_func)(void *_info), void *_info, int wait)
208 {
.....
215     /* Can deadlock when called with interrupts disabled */
216     WARN_ON(irqs_disabled());
217
218     spin_lock_bh(&call_lock);
219     atomic_set(&scf_started, 0);
220     atomic_set(&scf_finished, 0);
221     func = _func;
222     info = _info;
223
224     for_each_online_cpu(i)
225         os_write_file(cpu_data[i].ipi_pipe[1], "C", 1);
226
227     while (atomic_read(&scf_started) != cpus)
228         barrier();
229
230     if (wait)
231         while (atomic_read(&scf_finished) != cpus)
232             barrier();
233
234     spin_unlock_bh(&call_lock);
235     return 0;

```

Beware this!!

È bene che qui le interruzioni siano disabilitate: se il CPU-core su cui ci troviamo non riesce a prendere il lock perché già occupato, vuol dire che qualche altro CPU-core sta tentando di generare degli IPI.

Il parametro wait indica se il chiamante deve attendere che tutti i CPU-core reagiscano all'IPI o meno.

L'attività della function call deve essere rapida, quindi top-half.

Se prendiamo lock a tempo t, a t1 mandiamo interprocessor interrupt con un certo codice, e parte function call, a t2 qualcuno lo riceve, va nell'area di memoria condivisa (in the scheme) e vede funzione da chiamare e parametro da passare. A t3 segno in area condivisa la mia unità per indicare che sono partito. Non aspetto il completamento, per farlo dovrei mettermi in busy waiting su altra area. Ma non è un buon modo, dovremmo sempre fare cose minime e poi chiamare work queue. Questo perchè altri fanno cose che non sappiamo!

Effetti addizionali degli IPI:

Consideriamo l'IPI associato a RESCHEDULE\_VECTOR: sappiamo che porta alla preemption del task in esecuzione sul CPU-core target. Ciò ha effetti sia sulla correttezza e la consistenza dell'esecuzione del task, sia delle performance del sistema operativo.

Per quanto riguarda la **consistenza**, potremmo avere problemi nel caso in cui il task prelazionato lavori su delle variabili per-CPU. Facciamo un esempio:

```

struct _the_struct v[NR_CPUS];
v[smp_processor_id()] = some_value;
/* task is preempted here... */
something = v[smp_processor_id()];

```

We may be targeting different entries

Di conseguenza, è necessario che il task non abbandoni mai la CPU nell'intervallo di tempo in cui utilizza una stessa variabile per-CPU.

Per quanto invece riguarda le **prestazioni**, potrebbero essere compromesse nel momento in cui il task di riferimento sta eseguendo **smp\_call\_function()**. Di fatto, se il task è non interrompibile, abbiamo un problema nel momento in cui non riesce a prendere il lock, che è il caso in cui un qualche altro thread T in esecuzione su un altro CPU-core aveva già preso il lock: infatti, se T tenta di colpire con un IPI il CPU-core su cui gira il nostro task non interrompibile, andiamo incontro a un deadlock. Tuttavia, se il nostro task è prelazionabile, si rischia che venga buttato fuori dalla CPU mentre tiene il lock occupato, il che allunga di molto la durata della sezione critica e, di nuovo, potrebbe causare dei deadlock.

Per risolvere il problema, bisogna fare in modo che, durante l'esecuzione di **smp\_call\_function()**, il thread sia interrompibile ma non prelazionabile. Ciò è possibile grazie al contatore atomico per-thread che abbiamo già visto (il **preemption counter**): se il contatore vale zero, vuol dire che il thread è prelazionabile, se è maggiore di zero no. Ricordiamo che questo counter può essere incrementato mediante l'API **preempt\_disable()**, mentre può essere decrementato tramite l'API **preempt\_enable()**.

A valle di queste considerazioni, la funzione **smp\_call\_function()** si è evoluta nel seguente modo:

```
int smp_call_function(void (*func) (void *info), void *info, int wait)
{
    preempt_disable();
    smp_call_function_many(cpu_online_mask, func, info, wait );
    preempt_enable();
    return 0;
}
```

Internal structure with  
preemption awareness

Io, da una CPU, chiedo ad altri delle cose, e io non voglio migrare su altre CPU. Non usiamo **smp function call** in maniera massiva, perchè alla fine ha il suo costo.

#### Nota sulla scalabilità:

Nelle macchine con moltissimi CPU-core l'invio di IPI può essere un'operazione molto dispendiosa, soprattutto nel momento in cui si va in broadcast (vedi la **smp\_call\_function()**) o nel momento in cui il chiamante si mette in attesa che i CPU-core target abbiano terminato la gestione dell'IPI. Perciò, è bene fare uso degli IPI solo quando è strettamente necessario.

## VIRTUAL FILE SYSTEM

### Rappresentazioni del file system (FS)

- In RAM: è una rappresentazione parziale o completa della struttura e del contenuto corrente del file system.
- On device: è una rappresentazione non necessariamente aggiornata della struttura e del contenuto del file system.

Un concetto importante è quello del **file system volatile**, che è un file system rappresentato pienamente in RAM, tant'è vero che è pensato per essere temporaneo: nel momento in cui si effettua lo shutdown viene rimosso.

Per quanto riguarda l'accesso e la manipolazione dei dati, abbiamo due parti del sistema operativo:

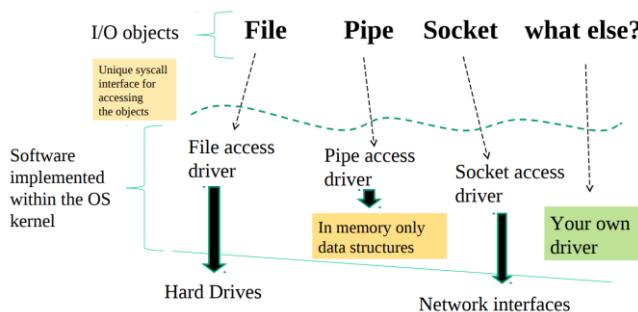
- **Parte indipendente dal FS (FS-independent)**: è un layer del software che può essere invocato all'interno del kernel anche da parte degli altri sottosistemi e permette di lavorare sugli oggetti di I/O appartenenti al file system (e.g. i file o le directory) in modo indipendente dal file system stesso; infatti, tali oggetti sono rappresentati all'interno di strutture dati fatte apposta per il layer del software. In altre parole, si tratta di un layer che espone delle API per lavorare sugli oggetti di I/O del file system in modo trasparente e indipendente da come il file system rappresenta gli oggetti di I/O stessi.
- **Parte dipendente dal FS (FS-dependent)**: è un layer che consiste in un insieme di moduli utilizzati per accedere e manipolare degli oggetti di I/O che sono specifici per un certo file system. Verosimilmente, il layer indipendente dal FS si appoggia su quello dipendente dal FS.

Dunque, un qualunque oggetto del file system è rappresentato in RAM mediante delle specifiche strutture dati e mantiene un riferimento a una tabella di function pointer che consente di identificare l'indirizzo in cui sono posizionate le operazioni per quello specifico oggetto. In particolare, ciascuna tabella implementa un **driver** nel sistema (una tabella di function pointers), che dispone di una sua interfaccia per essere invocato: il software in grado di utilizzare l'interfaccia dei driver è FS-independent, mentre le operazioni sugli oggetti di I/O vere e proprie (che implementano internamente i driver) sono FS-dependent.

### Introduzione al virtual file system (VFS)

I dispositivi (i.e. i device, che non sono altro che oggetti di I/O) possono essere **visti come file** dal software che utilizza le API del virtual file system. Le API vanno a lavorare sullo stato dei dispositivi sfruttando le strutture dati usate per rappresentare in memoria i dispositivi stessi. L'aggiornamento dello stato di questi device può eventualmente riflettersi in un cambio di stato di qualche componente hardware.

Esempi di dispositivi che già conosciamo sono i **file** veri e propri, le **pipe**, le **FIFO** e le **socket**. Vediamo uno schema generale:



Dallo schema è possibile notare che:

- > Per una particolare operazione (e.g. read, write), viene esposta al software un'unica system call per tutti i dispositivi; tale system call, internamente, può invocare driver differenti (che implementano l'operazione in modo diverso) in base ai parametri ricevuti in input, in particolare in base al tipo di dispositivo che si sta usando (un file, una pipe,...).

-> È possibile definire dei nostri dispositivi di I/O che si appoggiano su dei nostri driver.

-> I file e le socket, se subiscono un cambio di stato, riflettono la modifica su un componente hardware (rispettivamente su un **hard drive** e su un'**interfaccia di rete**); tuttavia, ciò non avviene per le pipe e le FIFO.

Esempio sui file come oggetto di I/O:

I file sono mantenuti su un hard drive.

Ma quali sono i moduli software di cui abbiamo bisogno per gestire i file sull'hard drive?

1. Una funzione per leggere il **device superblock (superblocco del dispositivo)** per determinare quali file esistono e dove si trovano i loro dati. Il device superblock è un blocco del file che mantiene i metadati del file stesso.

2. Una funzione per leggere i blocchi dei file per portarli all'interno di un buffer cache.

3. Una funzione per flushare i blocchi aggiornati sul dispositivo (i.e. all'interno dei file veri e propri).

4. Un insieme di funzioni per lavorare effettivamente sui dati (cachati in memoria) e per triggerare l'attivazione delle tre funzioni elencate qui sopra (che servono più che altro a far interagire il software di livello applicativo coi dispositivi).

#### Block-device driver vs char-device driver:

Per risolvere la lista di quattro punti di cui sopra, abbiamo due tipi di driver: i **block-device driver** e i **char-device driver**. In particolare:

- I primi tre punti della lista vengono risolti col block-device driver perché prevedono funzioni che *leggono e scrivono i blocchi dei file*;
- L'ultimo punto, invece, collega le attività da eseguire col char-device driver poiché si tratta di attività di *prelevamento e di consegna di caratteri*. Usato per le tastiere!

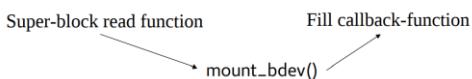
In realtà, anche i dispositivi stessi comprendono i **block device** e i **char device**: i primi offrono operazioni che possono essere eseguite sui blocchi, mentre i secondi offrono operazioni che possono essere eseguite sui singoli caratteri o singoli byte.

Se abbiamo *f* file system incluso in *F* file system, *F* può vedere *f* come sequenza di byte. C'è il layer ad alto livello del FS, che per eseguire tale operazione, userà un char device (che include tutte le operazioni attinenti ai file). Il driver di alto livello (char device) si appoggerà ad un driver di più basso livello (block device), per leggere blocchi e rappresentare tale oggetto in memoria.

Come accennato in precedenza, i driver (tra cui quelli introdotti poc'anzi, ovvero il block-device driver e il char-device driver) sono essenzialmente delle tabelle di function pointer che puntano all'implementazione effettiva delle operazioni che possono essere eseguite sull'oggetto target. Il fatto che driver differenti implementano in maniera diversa una stessa operazione implica che una stessa entry di tabelle di function pointer diverse punta a funzioni differenti. Il punto cruciale da risolvere rimane dunque il come permettere al VFS di determinare qual è il driver da eseguire quando viene invocata una data system call.

#### **File system type in Linux**

Un file system type (tipo di file system) è un insieme di oggetti di I/O con determinate caratteristiche, e tali caratteristiche sono diverse per file system type differenti. Per poter gestire un file system type, abbiamo bisogno di una **funzione di lettura del superblocco** (una funzione che, insieme alle altre funzioni di lettura del superblocco, è mantenuta nel kernel del sistema operativo) che serve a *interpretare i metadati* di tale insieme di oggetti di I/O. In particolare, tale funzione sfrutta delle API di livello kernel (come `mount\_bdev()`) per effettuare il setup di una porzione del VFS; queste API accettano come parametro una funzione di callback e la invocheranno con lo scopo di finalizzare la materializzazione in memoria del superblocco del dispositivo e, in particolare, di riempire il contenuto del superblocco stesso.



Quando montiamo FS (quindi so gestirlo e posso chiamare superblock per rappresentarlo in memoria) viene richiesto un punto di montaggio, il quale necessita di un RAM FS (o radice FS), che è il primo FS che permette di montare tutti gli altri. Dopo, RAM FS viene smontato.

### Magic number:

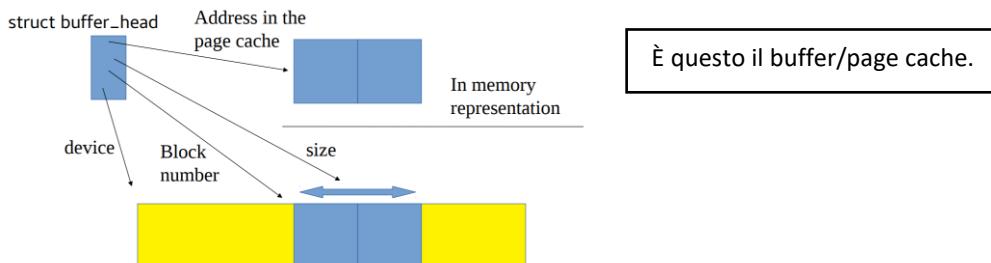
È un codice numerico che identifica il tipo di file system e, quindi, stabilisce qual è la struttura dei relativi oggetti di I/O e come questi possono essere acceduti. La funzione di lettura di superblocco, prima di eseguire le sue operazioni (e.g. montaggio di una porzione del VFS), va proprio a controllare il magic number che identifica il file system type associato al dispositivo (o ai dispositivi) di interesse e, se si tratta di un magic number conforme con lei, allora può procedere con le operazioni.

Il comando **file [-s] /dev/{device-name}** permette di estrarre il magic number dal dispositivo specificato e riporta le informazioni riguardanti il relativo file system type.

### Buffer/page cache

È un'area di memoria (un buffer) dove sono mantenuti i blocchi dei dispositivi e serve a gestire le operazioni di lettura e scrittura. Per gestire questi blocchi, Linux offre la struttura dati **struct buffer\_head** che è costituita dai seguenti dati principali:

- **\*b\_data** = puntatore all'area di memoria (all'interno di un determinato dispositivo) da cui sono stati estratti i dati acceduti in lettura o verso cui i dati devono essere scritti.
- **b\_size** = dimensione del buffer.
- **\*b\_bdev** = block device su cui stiamo lavorando.
- **b\_blocknr** = numero che identifica il blocco caricato nel buffer.

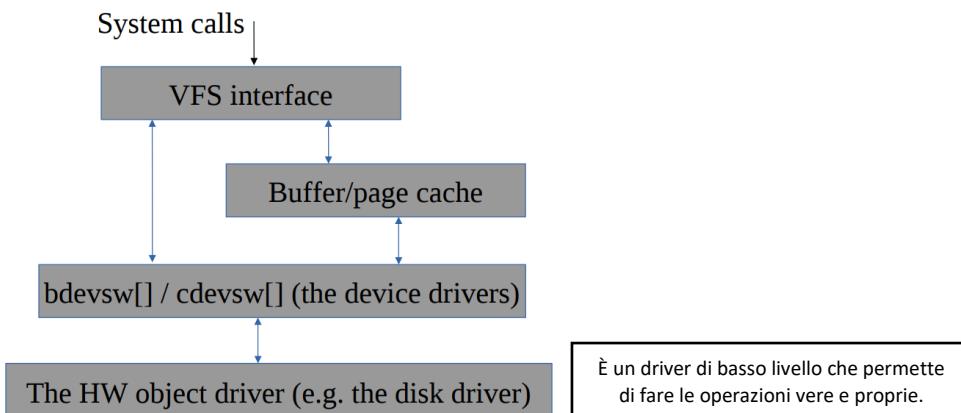


API per leggere / scrivere blocchi di un dispositivo:

- > **`__bread()`**: recupera un puntatore alla struttura `buffer_head` associata al blocco (o insieme di blocchi) che ha il numero `b_blocknr` specificato e la dimensione `b_size` specificata. Lettura blocco dispositivo.
- > **`sb_bread()`**: recupera la dimensione di un certo blocco (o insieme di blocchi) guardando all'interno del superblocco.
- > **`mark_buffer_dirty()`**: marca il buffer/page cache come *dirty* settando a 1 il bit **BH\_Dirty bit**; il buffer aggiornato verrà riportato sul disco in un secondo momento.
- > **`brelse()`**: libera la memoria usata dal buffer, ma solo dopo che eventualmente ha riportato sul disco gli aggiornamenti del blocco (o insieme di blocchi) presente nel buffer.
- > **`map_bh()`**: indica l'associazione tra il blocco puntato e il settore del dispositivo in cui esso si trova.

NB: MANCA PARTE SUPERBLOCCO + api (`mount_bdev`, `mount_single..`) o comunque da trattare un po' di più

## Schema riassuntivo sul layering



### File regolari vs dispositivi

Un qualunque file regolare può essere visto come un block device che ospita un file system. Per associare correttamente questo ruolo al file abbiamo bisogno di montare il file system corrispondente utilizzando un block-device driver specifico, che è il **-o loop driver**. In tal modo, siamo in grado di creare uno stack di “file system device”.

Il concetto di driver è usato in modo ampio in Linux. Preso un file regolare di un FS, abbiamo delle file operations associate ad FS (quindi dipende dal FS a cui appartiene il file). Non è l'unico modo per operare su entità rappresentante stream di byte. Si usa un altro approccio:

Usiamo un driver (tabella di puntatori a file operations) per pilotare attività su stream \*generico\*. Quindi lo stream potrebbe essere anche una pipe. Lo stesso driver è usabile per cose diverse quindi. Il file appartiene al FS, e abbiamo driver, però possiamo associarci un driver di più alto livello, che vede il file come fosse dispositivo a blocchi, trattandolo come dispositivo a blocchi, con granularità pari al blocco. Si passerà sempre al driver più specifico, però il concetto è che possiamo vederlo sia come byte (driver specifico) sia come blocchi (driver più generico, “the device drivers”). Con questo punto di vista, possiamo montarci dentro un file system. Possiamo creare una vista sui dati, gestendole come preferiamo.

### Funzione di lettura del superblocco nei RAM file system

L'associazione di ogni file system type a una funzione di lettura del superblocco è valida anche per i file system le cui informazioni non sono mantenute all'interno di un dispositivo (RAM file system). Anche qui questa funzione si occupa di montare una porzione del VFS, ma non lavora con un driver esterno bensì con delle operazioni codificate direttamente al suo interno.

### Inode e dentry

- Inode = struttura dati del VFS che rappresenta un oggetto di I/O (specificando determinate informazioni come i permessi di accesso).
- Dentry = struttura dati che, tra le varie cose, assegna un nome a uno specifico oggetto di I/O ed è associata all'inode del medesimo oggetto di I/O.

Due API per creare queste strutture dati sono riportate qui di seguito:

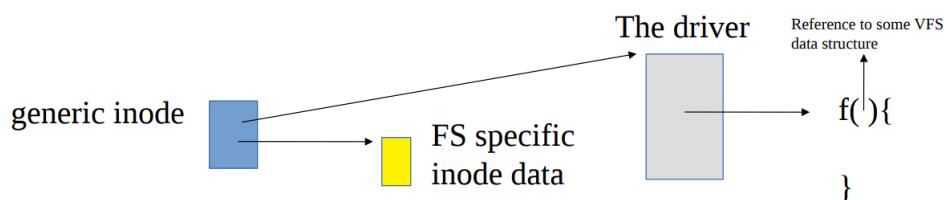
- > **struct inode \*new\_inode (struct super\_block \*sb)**: alloca un *inode generico* assegnando un ID al superblocco passato come parametro; l'inode generico sarà associato allo specifico oggetto di I/O relativo al superblocco.
- > **struct dentry \*d\_make\_root (struct inode \*root\_inode)**: crea una dentry che risulterà essere associata

all'*inode-root*, il quale non è altro che l'inode generico che rappresenta il punto di montaggio di uno specifico file system F (ovvero rappresenta la directory *root* di F).

Tipicamente l'inode generico mantiene dei campi generici usati dal VFS e dei campi che servono a referenziare dati specifici del file system in cui ci troviamo.



Dopodiché sappiamo già che un driver non è altro che una tabella di puntatori a funzione mantenuta all'interno del VFS. Anche il driver è referenziato dall'inode generico del nostro oggetto di I/O. Le funzioni puntate dalla tabella (dal driver) accettano come parametro l'indirizzo di una struttura dati generica del VFS; a partire da tale indirizzo è possibile accedere (più o meno direttamente) ai dati specifici per il file system in cui ci troviamo.



Un thread generico chiama syscall I/O, e riceve codice “c” del canale I/O per lavorarci. Tale codice, a partire da current, identifica nell’array il pointer all’i-node, su cui lavoriamo. Esso ha puntatore al driver per questo specifico FS, che richiamerà specifica operazione nel software. In input, possiamo passare struttura a VFS, e quindi ritornare in “generic inode”, da cui arrivare a “FS specific inode data”. In questa funzione possiamo anche fare operazioni su buffer cache, o superbocco (puntato proprio da i-node).

Il root i-node è quello associato alla directory (cioè un file), qui parliamo di un i-node generico.

### Startup del VFS

Durante la fase di boot del kernel, il thread che si occupa dell'inizializzazione del kernel (i.e. l'idle process) invoca la funzione **`vfs_caches_init()`**, che ha lo scopo di inizializzare tutta la zona di memoria che serve per gestire il VFS e chiama internamente la funzione **`mnt_init()`**. Quest'ultima, a sua volta, invoca:

-> **`init_rootfs()`**: inizializza il **Rootfs**, che è il tipo di file system di “root”, rappresenta un punto di ancoraggio per tutti gli altri FS (i.e. FS di root applicativo) e si trova *unicamente in RAM*, usato per agganciargli altre cose.

-> **`init_mount_tree()`**: monta (istanzia) il Rootfs (vedremo successivamente i dettagli).

In realtà, il kernel Linux non supporta solo il Rootfs ma anche altri file system type, come **Ext** (in diversi flavour). Tra l'altro, in linea di principio, affinché il kernel possa essere eseguito, non ci sarebbe nemmeno bisogno che vi sia un qualche file system montato: questo significa che tutte le funzionalità che servono a far funzionare il kernel sono implementate direttamente all'interno del codice del kernel stesso e non devono essere cercate nei dispositivi. Per far ciò, dovremmo usare kernel daemons, senza FS, escludendo il supporto a questo.

### Strutture dati per rappresentare i file system type

Sappiamo che, per ogni file system type, dobbiamo avere all'interno del kernel una funzione f per andare a leggere il superbocco di quel file system type (appunto la funzione di lettura del superbocco). Queste (e altre) informazioni sono mantenute all'interno di una lista collegata in cui ciascun nodo è relativo a uno

specifico tipo di file system ed è rappresentato dalla struttura dati **struct file\_system\_type** che, almeno per quanto riguarda le versioni più dorate del kernel, viene descritta qui di seguito:

```
struct file_system_type {
    const char *name;
    int fs_flags;
    ...
    struct super_block *(*read_super) (struct super_block *, void *, int);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    ...
};
```

Moved to the **mount** field  
in newer kernel versions

- > **const char \*name** = nome che identifica il file system type.
- > **struct super\_block \*(\*read\_super) (struct super\_block \*, void \*, int)** = funzione di lettura del superbocco.
- > **struct file\_system\_type \*next** = nodo successivo della lista collegata.

Nelle versioni più recenti del kernel, la struct **file\_system\_type** si è evoluta nel seguente modo (vedere pagina successiva):

```
struct file_system_type {
    const char *name;
    int fs_flags;
    ...
    ...
    struct dentry *(*mount) (struct file_system_type *,
                            int, const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    ...
}
```

Beware this!!

- > **struct dentry \*(\*mount) (struct file\_system\_type \*, int, const char \*, void \*)** = la nuova funzione di lettura del superbocco. A differenza di quella precedente, accetta un parametro in più (il **const char \***) che indica qual è il dispositivo a partire da cui deve essere montato il file system; inoltre, notiamo che il tipo di ritorno, da **struct super\_block \***, è diventato un **dentry\***.

Mount è funzione di lettura de superbocco, che sfrutta degli intermediari. Quindi prima chiamiamo superbocco, poi intermediario (ne è un tipo **mount\_bdev()**) che fa setup della page cache; seleziona anche il driver corretto e mette in memoria delle informazioni, associandogli un i-node. Quando l'i-node è in memoria, viene poi chiamato fill callback-function.

Per agganciare un nuovo nodo alla lista collegata dei file system type, si ricorre alla seguente funzione:  
**int register\_filesystem (struct file\_system\_type \*)**.

Questa funzione, se ci pensiamo, viene invocata da **init\_rootfs()** per inserire il Rootfs all'interno della lista collegata dei file system type:

```
int __init init_rootfs(void){
    ...
    register_filesystem(&rootfs_fs_type);
    ...
}
```

### Check di livello user sui file system gestiti dal kernel

I file system correntemente gestiti dal kernel vengono listati accedendo il file **/proc/filesystems**. Il campo

**nodev** dell'output ci indica i file system che sono gestiti come dei file system in memoria; qui, ad esempio, tipicamente troviamo **sys** e **proc**.

### Struct usate per mantenere in memoria la rappresentazione dei FS

- > struct vfsmount: fornisce delle informazioni sul FS come qual è il suo parent. Per ogni file system gestito.
- > struct super\_block: mantiene i metadati di base del FS. Per ogni file system gestito.
- > struct inode: per ogni file che stiamo gestendo.
- > struct dentry: per ogni file che stiamo gestendo.

*Vfsmount* e *super\_block* mantengono delle informazioni proprie di un file system, per cui sono a istanza unica nell'ambito del file system. Al contrario, *inode* e *dentry* sono tali per cui esiste una copia diversa per ogni file o directory che si trova nel file system.

#### Vfsmount:

Fino alla versione 3 del kernel Linux la struct vfsmount era definita nel seguente modo, e ci permette di relazione un file system con un altro, quindi se uno è montato su un altro. Creiamo visione ad alto livello per i file system, contenenti le dipendenze. Abbiamo relazione anche rispetto al super blocco.

```
struct vfsmount {
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent;           //puntatore alla struct vfsmount del FS parent
    struct dentry *mnt_mountpoint;
    struct dentry *mnt_root;               //root del nostro FS
    struct super_block *mnt_sb;            //puntatore alla struct super_block relativa al nostro FS
    struct list_head mnt_mounts;
    struct list_head mnt_child;            //lista dei FS children
    atomic_t mnt_count;                  //contatore atomico del numero di FS che dipendono dal nostro FS
    int mnt_flags;
    char *mnt_devname;                   //nome del dispositivo dove si deve lavorare per il nostro FS
    struct list_head mnt_list;
}
```

Dal kernel 4 in poi, invece, la struct vfsmount è molto più compatta:

```
struct vfsmount {
    struct dentry *mnt_root;              //root del nostro FS
    struct super_block *mnt_sb;           //puntatore alla struct super_block relativa al nostro FS
    int mnt_flags;
} __randomize_layout;
```

La feature `__randomize_layout` è supportata dal plugin **randstruct** e può essere applicata alle struct per motivi di sicurezza, in particolare quando le informazioni contenute nelle strutture sono sensibili.

Il meccanismo di `__randomize_layout` è il seguente: supponiamo di avere una struttura composta dai campi A, B, C. Allora, in memoria questi tre campi possono presentarsi in un ordine diverso ed eventualmente con del padding tra uno e l'altro. In tal modo, pur conoscendo l'indirizzo base della struttura, risulterà difficile accedere ai vari campi con un semplice spiazzamento. Poiché il posizionamento esatto in memoria dei campi della struct è deciso a *compile time* del kernel, il layout della struttura stessa varia tra un'istanza del kernel e l'altra (i.e. tra una macchina e l'altra). Di conseguenza, se utilizziamo l'operatore 'freccia' (->) o l'operatore 'punto' su una struttura randomizzata sulla stessa istanza del kernel in cui è avvenuta la

compilazione, allora si riesce ad accedere ai campi desiderati; se invece utilizziamo questi operatori sulla stessa struttura randomizzata su un'istanza del kernel differente, allora lo spiazzamento che si effettua è il medesimo ma in generale non si riesce ad accedere ai campi desiderati perché si trovano in una posizione diversa. I seed pseudo-random sono comunque forniti, perché potrebbe essere necessario l'accesso a qualche struttura mappata randomicamente. Secondo Torvalds, non è troppo utile come cosa.

In definitiva, il plugin randstruct può essere sfruttato mediante la feature `__randomize_layout` ma viene anche usato di default per tutte le struct composte esclusivamente da puntatori a funzione (come i driver); in questo secondo caso, per disabilitare la randomizzazione della struttura, si può ricorrere alla keyword `__no_randomize_layout`.

### Super\_block:

```
struct super_block {
    struct list_head    s_list;        /* Keep this first */
    dev_t                s_dev;         /* search index; _not_ kdev_t */
    ...
    unsigned long        s_blocksize;
    loff_t               s_maxbytes;   /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations *s_op;
    ...
    unsigned long        s_magic;
    struct dentry        *s_root;
    ...
    struct list_head    s_mounts;     /* list of mounts */
    struct block_device *s_bdev;
    ...
    void                 *s_fs_info;    /* Filesystem private info */
    ...
    const struct dentry_operations *s_d_op; /* default d_op for dentries */
    ...
    struct user_namespace *s_user_ns;
    ...
} __randomize_layout;
```

-> `const struct super_operations *s_op`: puntatore al driver relativo al `super_block`.

-> `struct dentry *s_root`: puntatore alla `dentry` di root.

C'è concetto di namespace anche nel montaggio, quindi thread che vivono in altro namespace non vedono questo montaggio. L'oggetto di I/O è la radice del FS.

### Dentry:

Struttura dati che, dato oggetto in memoria, mantiene il nome dell'oggetto. Nei FS, i metadati tipicamente non hanno un nome (quindi nell'i-node non ho il nome), quindi introduco le `dentry` per colmare questo gap. `Dentry` per i nomi, i-node per i metadati. Sia un directory sia un file hanno entrambe.

```
struct dentry {
    ...
    struct dentry *d_parent; /* parent directory */
    struct qstr d_name;
    struct inode *d_inode; /* Where the name belongs to */
    unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
    ...
    const struct dentry_operations *d_op;
    struct super_block *d_sb; /* The root of the dentry tree */
    ...
    void *d_fsdta; /* fs-specific data */
    ...
    struct list_head d_child; /* child of parent list */
    struct list_head d_subdirs; /* our children */
    ...
} __randomize_layout;
```

-> `struct dentry *d_parent`: puntatore alla `dentry` della directory parent del nostro file o directory.

-> `struct inode *d_inode`: puntatore all'i-node associato alla nostra `dentry`.

Abbiamo info per puntare a specifico FS, ovvero `'d_fsdta'`.

### Inode:

```

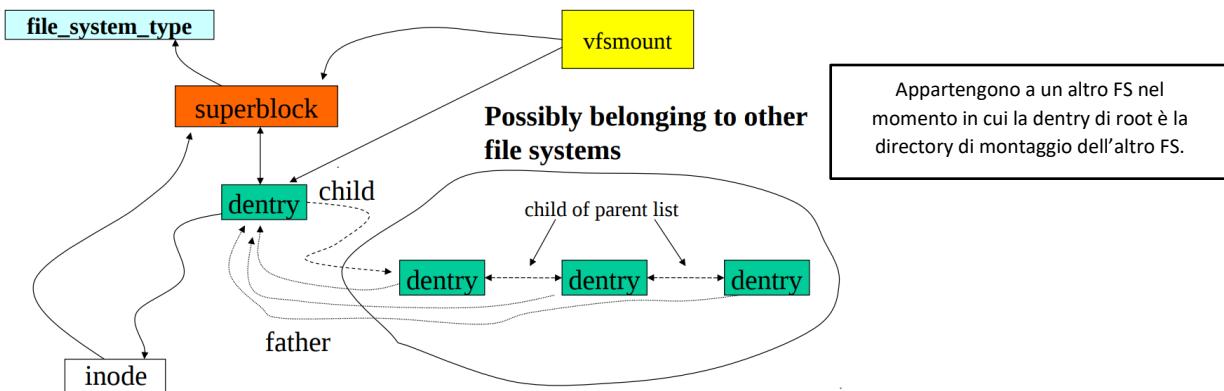
struct inode {
    umode_t          i_mode;
    unsigned short   i_opflags;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int     i_flags;
    ...
    const struct inode_operations *i_op;
    struct super_block *i_sb;
    ...
    loff_t           i_size;
    ...
    spinlock_t       i_lock; /* i_blocks, i_bytes, maybe i_size */
    ...
    union {
        const struct file_operations *i_fop; /* former ->i_op->default_file_ops */
        void (*free_inode)(struct inode *);
    };
    ...
    void            *i_private; /* fs or device private pointer */
} __randomize_layout;

```

177

Identifica driver operazioni su i-nodes (metadati) e file associati ad i-node (contenuto).

### Schema riassuntivo:



### Istanziazione di Rootfs allo startup di Linux

Viene effettuata dalla funzione `init_mount_tree()` e si articola nelle seguenti attività:

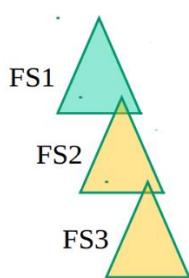
- 1) Allocazione delle quattro strutture dati per Rootfs (**vfsmount**, **super\_block**, **inode** dell'oggetto root di Rootfs e **dentry** dell'oggetto root di Rootfs).
- 2) Collegamento delle strutture dati.
- 3) Setup del nome "/" per l'oggetto root del file system.
- 4) Collegamento tra Rootfs e l'idle process.

I primi tre task vengono svolti dalla funzione **do\_kern\_mount()** invocata da `init_mount_tree()`, mentre l'ultimo task viene svolto dalle funzioni **set\_fs\_pwd()** e **set\_fs\_root()** sempre invocate da `init_mount_tree()` (dove **pwd** sta per *process working directory*).

### FS vs namespace

Noi abbiamo i namespace anche a livello di gestione del VFS. Ciò implica che i thread possono avere delle visioni completamente scorrelate dei dati del VFS; in altre parole, abbiamo la possibilità di definire delle zone del VFS separate tra loro, in cui una zona può essere esposta a taluni thread, un'altra zona può essere esposta ad altri thread e così via. Il montaggio è associato al namespace a cui appartiene. Il montaggio avviene in un namespace di montaggio, ma è condiviso con altri namespace. E' sbagliato dire che viva nel namespace associato al descrittore del thread corrente.

Entriamo più nel dettaglio della questione. Una qualsiasi directory (o comunque un qualsiasi punto) di ciascun file system può essere utilizzata come punto di ancoraggio su cui montare altri file system. È possibile così costruire un albero di FS.



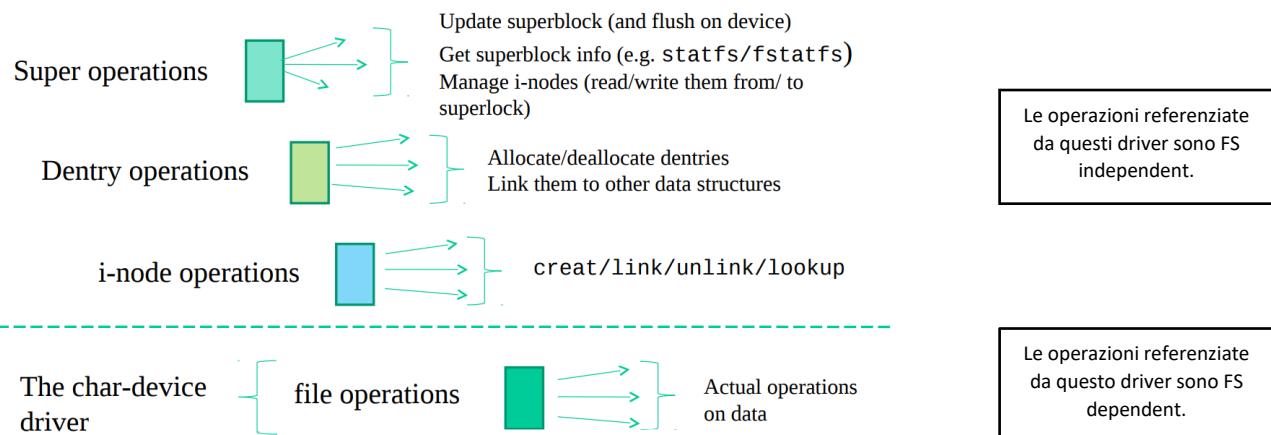
Nella figura qui accanto, ad esempio, possiamo immaginare di avere FS1 = Rootfs che, di base, non viene mai reso visibile agli altri thread. Perciò, la radice dal punto di vista dei thread è data da FS2 che tipicamente corrisponde al FS di root applicativo. Comunque sia, è possibile fare in modo che vi siano FS visibili ad alcuni thread ma non ad altri mediante l'utilizzo dei namespaces. In particolare, è possibile definire un nuovo namespace che, per esempio, abbia come file system di riferimento FS3; FS3 viene così montato a partire da un altro file system (e.g. FS2) ed è visibile esclusivamente al thread root per quel namespace e ai suoi thread figli.

L'unica eccezione è data dall'evenienza in cui il file system FS3 venga montato con l'opzione **SHARED**, secondo cui FS3 non deve essere visibile solo al namespace all'interno del quale viene effettuata l'operazione di montaggio, bensì anche agli altri namespace. L' **unshare()** fa questo:

Abbiamo un thread vivente in `mount_namespace`, con tale comando passa a `mount_namespace1`, creando questo nuovo oggetto e avendolo associato al thread. I thread lavorandi su `mount_namespace` vedranno ciò che avviene in `mount_namespace1`, ma possiamo anche privatizzare ciò.

È anche possibile generare un nuovo thread attraverso la funzione `clone()` che, tra i vari parametri, accetta dei flag; se tra questi flag specifichiamo **CLONE\_NEWNS**, stiamo imponendo che il nuovo thread apparterrà a un nuovo namespace, per cui ciò che succederà col nuovo thread non impatterà il namespace corrente.

### Vista di insieme dei driver



Badiamo che, per poter utilizzare le operazioni corrette all'interno del char-device driver di un determinato oggetto di I/O, è necessario prima passare per i driver FS independent, che permettono appunto di accedere ai metadati del file system e dell'oggetto di I/O necessari.

### Collegamento tra TCB e VFS

Il TCB di ciascun thread T ha il campo **struct fs\_struct \*fs** che punta alle informazioni che associano il thread T con il virtual file system. Su Linux si ha un unico file system, tutti gli altri sono montati in cascata (non c'entra nulla il concetto dei namespaces di montaggio diversi). Windows è diverso, avendo vari file system (disco C, E, F,...)

Nella versione 2.4 del kernel tale struttura è definita come segue:

```

struct fs_struct {
    atomic_t count;
    rwlock_t lock;
    int umask;
    struct dentry * root, * pwd, * altroot;
    struct vfsmount * rootmnt, * pwdmnt,
                      * altrootmnt;
};
  
```

- > **struct dentry \*root**: è la dentry della directory radice del VFS dal punto di vista del thread T (i.e. è il punto più alto del FS visibile al thread T).
- > **struct dentry \*pwd**: è la dentry della directory corrente.
- > **struct vfsmount \*rootmnt**: indica il file system relativo alla directory radice per il thread T.
- > **struct vfsmount \*pwdmnt**: indica il file system relativo alla directory corrente.
- > **atomic\_t count** conta i thread correnti.
- > **altroot** indica altro root nella gerarchia, normalmente per fare operazioni particolari.

Dal kernel 3 al kernel 4.7 la struttura è stata modificata nel seguente modo:

```

8 struct fs_struct {
9     int users;
10    spinlock_t lock;
11    seqcount_t seq;
12    int umask;
13    int in_exec;
14    struct path root, pwd;
15 };

```

Possiamo osservare che i campi **struct dentry \*root** e **struct vfsmount \*rootmnt** sono stati raggruppati nel campo **struct path root**, mentre i campi **struct dentry \*pwd** e **struct vfsmount \*pwdmnt** sono stati raggruppati nel campo **struct path pwd**.

Nelle versioni più recenti del kernel, infine, è stata aggiunta la feature `__randomize_layout`:

```

struct fs_struct {
    int users;
    spinlock_t lock;
    seqcount_t seq;
    int umask;
    int in_exec;
    struct path root, pwd;
} __randomize_layout;

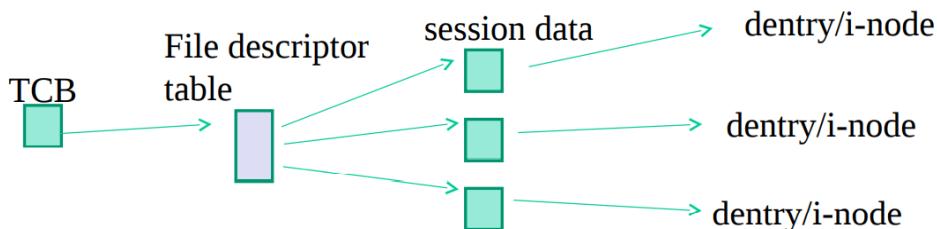
```

Towards more security

### File descriptor table

È una struttura dati che è definita nel TCB e costruisce una relazione tra i **canali di I/O** (che sono codici identificativi all'interno della tabella) e gli **oggetti di I/O** su cui stiamo correntemente lavorando mediante le rispettive **sessioni di I/O**. Abilita quindi la ricerca veloce delle strutture dati usate per rappresentare gli oggetti di I/O e le sessioni di I/O, dove la chiave di ricerca è l'ID dei canali. Per dire che una sessione è valida per un thread, c'è una tabella raggiungibile mediante TCB. Questa tabella è la FDT, l'indice dell'array è codice canale IO, e poi si chiama driver, puntato da i-node. La FDT è in struttura dati puntata da TCB, puntata da struttura **files\_struct**.

In definitiva, i metadati di un oggetto di I/O e, quindi, i relativi dentry e inode possono essere acceduti a partire dalla struttura dati associata alla sessione di utilizzo di quell'oggetto di I/O. La sessione è a sua volta accessibile a partire dalla entry corrispondente del file descriptor table.



Di seguito è riportata la struct che implementa il file descriptor table (sebbene ne sia mostrata una versione un po' datata, è comunque rappresentativa perché nelle versioni più recenti del kernel le modifiche sono minimali).

```

struct files_struct {
    atomic_t count;
    rwlock_t file_lock; /* Protects all the below
Nests                                inside tsk->alloc_lock */
    int max_fds;
    int max_fdset;
    int next_fd;
    struct file ** fd; /* current fd array */
    fd_set *close_on_exec;           bitmap for close on exec flags
    fd_set *open_fds;              bitmap identifying open fds
    fd_set close_on_exec_init;
    fd_set open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};

-> struct file *fd_array [NR_OPEN_DEFAULT]: array di puntatori a strutture di sessione; ha un numero massimo di entry dato dal valore di NR_OPEN_DEFAULT.
-> struct file ***fd: se fd_array ha una dimensione sufficiente per puntare a tutte le strutture di sessione necessarie, allora fd punta proprio a tale array; altrimenti, fd punta altrove, per far sì che tutte le strutture di sessione siano raggiungibili (trattasi di un meccanismo di scalabilità).
-> fd_set *close_on_exec: bitmap che indica quali canali devono essere chiusi contestualmente a un'operazione di exec (cambio della parte applicativa dell'address space del thread corrente).
-> fd_set *open_fds: bitmap che indica quali entry della tabella puntata da fd sono libere e quali no.
-> atomic_t count: contatore dei thread che condividono la struttura; quest'ultima, di fatto, può essere ereditata dal thread generato da un'operazione di clonaggio.

```

#### Struct file:

È una struttura definita nel file descriptor table e descrive la sessione di un determinato oggetto di I/O. Notiamo **next\_fb**, che ci dice da dove iniziare a cercare una bitmap per vedere quale entry sia libera. E' O(1) perchè si segna l'ultima chiusura.

Per quanto riguarda le prime versioni del kernel, la struttura è definita come riportato nella pagina seguente:

```

struct file {
    struct list_head f_list;
    struct dentry *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    loff_t f_pos;
    unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct f_owner;
    unsigned int f_uid, f_gid;
    int f_error;
    unsigned long f_version;
    /* needed for tty driver, and maybe others */
    void *private_data;
    /* preallocated helper kiobuf to speedup O_DIRECT */
    struct kiobuf *f_iobuf;
    long f_iobuf_lock;
};

```

- > **struct dentry \*f\_dentry:** identifica l'oggetto di I/O in questione.
- > **struct file\_operations \*f\_op:** driver delle operazioni che possono essere eseguite sull'oggetto di I/O in questione.
- > **atomic\_t f\_count:** contatore del numero di riferimenti alla struttura.
- > **mode\_t f\_mode:** modalità con la quale possiamo lavorare sull'oggetto di I/O nella sessione corrente (e.g. se abbiamo un file read-write ma noi lo apriamo in modalità di sola lettura, f\_mode specifica soltanto il

permesso di lettura).

-> **loff\_t f\_pos**: offset all'interno dell'oggetto di I/O dove stiamo lavorando col driver f\_op.

Nelle versioni 3, 4 e 5 del kernel, invece, la struct file si è evoluta nel seguente modo:

```

775 struct file {
776     union {
777         struct llist_node    fu_llist;
778         struct rcu_head      fu_rcuhead;
779     } f_u;
780     struct path          f_path;
781 #define f_dentry   f_path.dentry
782     struct inode        *f_inode; /* cached value */
783     const struct file_operations *f_op;
784
785     /*
786     * Protects f_ep_links, f_flags.
787     * Must not be taken from IRQ context.
788     */
789     spinlock_t       f_lock;
790     atomic_long_t    f_count;
791     unsigned int     f_flags;
792     fmode_t          f_mode;
793     struct mutex     f_pos_lock;
794     loff_t           f_pos;
795     struct fown_struct f_owner;
796     const struct cred *f_cred;
797     struct file_ra_state f_ra;
798
799     .....
800     __randomize_layout;;

```

Now we have randomized layout and a few fields are moved to other pointed tables

### Randomized from kernel 4.8

#### API per il VFS di Linux

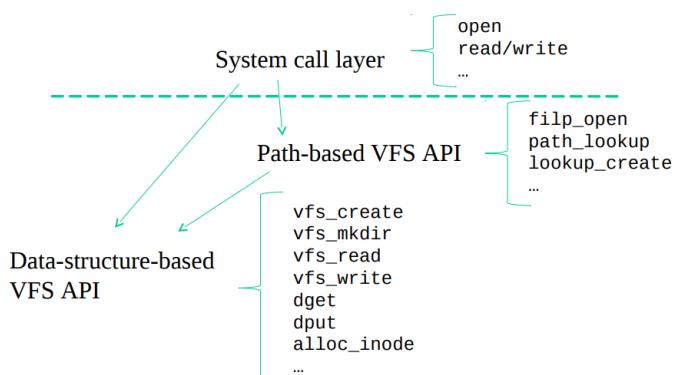
Le API che insistono sul VFS di Linux possono essere attivate da parte di una system call (per cui parliamo di **system call layer**). In particolare, le system call possono occuparsi del setup di una sessione oppure dell'accesso / della manipolazione di dati associati a un determinato canale di I/O.

Tuttavia, quando entriamo in modo kernel, possiamo attuare delle operazioni sulle strutture dati del VFS in modi differenti:

-> **Path-based VFS layer**: qui si hanno API che accettano come parametro di input il path dell'oggetto di I/O su cui si vuole lavorare.

-> **Data structure-based VFS layer**: qui si hanno API che accettano come parametro di input il puntatore alla struttura dati su cui si vuole lavorare.

In particolare, quando si va a lavorare per la prima volta con un oggetto di I/O X, dal system call layer si passa per il path-based VFS layer per eseguire delle operazioni col path di X e ottenere il puntatore a una struttura dati associata a X, e solo poi si arriva al data structure-based VFS layer per operare direttamente con la struttura dati. Tuttavia, tutte le volte successive in cui si va a lavorare su X, è possibile passare direttamente dal system call layer al data structure-based VFS layer.



### Esempi di API path-based:

-> **struct file \*filp\_open (const char \*filename, int flags, int mode)**: effettua il setup di una sessione di I/O associata all'oggetto di I/O identificato dal parametro filename (che di fatto è un path).

In particolare, quando viene invocata la system call **open()**, internamente viene chiamata proprio la **filp\_open()**, che a sua volta andrà a lavorare sui driver dell'oggetto di I/O target: ad esempio, si va sulle inode operation per eseguire una **lookup()** con lo scopo di stabilire se esiste davvero un oggetto identificato dal path dato in input a **open()** e, quindi, a **filp\_open()**.

### Esempi di API data structure-based:

-> **int vfs\_mkdir (struct inode \*dir, struct dentry \*dentry, int mode)**: crea un inode e lo associa con la dentry specificata come parametro.

-> **int vfs\_create (struct inode \*dir, struct dentry \*dentry, int mode)**: crea un inode collegato alla struttura dati puntata dal parametro dentry.

-> **static \_\_inline\_\_ struct dentry \*dget (struct dentry \*dentry)**: acquisisce una determinata dentry, incrementando il relativo reference counter (e, finché tale reference counter non torna a zero, la relativa dentry e tutte le dentry parent non possono essere deallocate).

-> **void dput (struct dentry \*dentry)**: rilascia una determinata dentry, decrementando il relativo reference counter.

-> **ssize\_t vfs\_read (struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*pos)**: effettua un'operazione di lettura su un oggetto di I/O sfruttando direttamente un pointer alla tabella che implementa la sessione (senza dover passare per i canali di I/O).

-> **ssize\_t vfs\_write (struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*pos)**: effettua un'operazione di scrittura su un oggetto di I/O sfruttando direttamente un pointer alla tabella che implementa la sessione.

### **Major number**

È un numero che ci permette di relazionare gli oggetti di I/O ai driver. In particolare, i driver sono registrati all'interno della cosiddetta **device-drivers table**: allora il **major number** di un driver D non è altro che il displacement all'interno di questa tabella che contiene le informazioni relative a D.

Supponiamo di dover istanziare in memoria la dentry e l'inode di un file. Allora dobbiamo:

- Identificare il char-device driver per operare sul file.
- Collegare la dentry e l'inode al char-device driver.

Queste operazioni hanno a che vedere col major number: di fatto è possibile assegnare un nuovo major number (i.e. un nuovo driver) al file, oppure far sì che l'assegnazione del major number avvenga in automatico, prendendo esattamente il major number di un file appartenente al medesimo file system. In particolare, questa seconda strada viene presa nel momento in cui la dentry e l'inode del nuovo file dipendono da un altro file già presente all'interno del file system; infatti, in tal caso, il file nuovo condividerà con quello vecchio il driver, e i due file sono detti **omogenei**.

### **Nodi del VFS e minor number**

A questo punto ci poniamo la seguente questione (problema dell'isolamento dei dati): come facciamo a fare in modo che il driver di due oggetti di I/O omogenei vada a toccare dati differenti a seconda dell'oggetto target? Nel caso in cui gli oggetti di I/O sono dei file regolari, allora sono gli inode a indicare direttamente al driver quali sono i blocchi di dati su cui lavorare. In caso contrario, la situazione si fa un po' più complicata e richiede che prima introduciamo dei concetti preliminari.

Il campo **umode\_t i\_mode** all'interno della **struct inode** mantiene un'informazione che indica il tipo di inode, che può essere:

- Una directory.
- Un file.
- Un char device.

- Un block device.
- Una pipe (o FIFO).

Concentriamoci sul **char device**: è un dispositivo che, mediante l'uso di API di scrittura / lettura, invia / riceve un flusso di byte. Nonostante esponga le medesime API rispetto, ad esempio, a un file regolare, si tratta di un oggetto eterogeneo rispetto al file regolare stesso, per cui dispone di un driver differente. Dunque, se abbiamo un file system F all'interno del quale si trova una directory d, e noi vogliamo montare un char device all'interno di d, stiamo praticamente inserendo all'interno di F un oggetto di I/O eterogeneo rispetto alle varie directory e ai file regolari appartenenti a F.

Tutti gli oggetti di I/O che si trovano esattamente nella situazione appena descritta per il char device di esempio sono detti **nodi**. La funzione **sys\_mknod()** consente la creazione di un inode associato a un nodo. In particolare, l'inode contiene il campo **kdev\_t i\_rdev** che mantiene le informazioni legate al major number e al **minor number** del dispositivo: mentre il major number, come già sappiamo, identifica il driver associato al dispositivo, il minor number identifica quali sono i dati su cui andare a lavorare nel momento in cui viene invocata un'operazione sul dispositivo (i.e. come deve essere implementata l'operazione sul dispositivo). I nodi con lo stesso major number (i.e. associati allo stesso driver) possono avere dei minor number differenti, ed è così che risolviamo il problema dell'isolamento dei dati. Tuttavia, non è escluso che più nodi abbiano stesso major number e stesso minor number: in tal caso, andremo a lavorare sui medesimi dati a prescindere da se invochiamo un'operazione di scrittura / lettura su uno di questi nodi oppure su un altro. E' come quando identifichiamo i blocchi con l'i-node.

Poichè esistono char device e block device driver, abbiamo due tabelle, e quando creiamo un certo oggetto ci riferiamo ad una delle due tabelle. In un FS regolare tutto ciò non c'è, perchè non ci interessano major number: quando lo apro, c'è l'allocazione per la gestione e collegamento con i driver del file system. Tutti i file nella directory `dev`, allo startup, sono nodi, pilotati da driver esterno.

Consideriamo la system call **mknod()** che, appunto, invoca internamente **sys\_mknod()** e serve a creare un nuovo nodo:

-> **int mknod (const char \*pathname, mode\_t mode, dev\_t dev)**: il parametro dev indica il dispositivo da creare specificandone il major number e il minor number; il parametro mode indica la modalità con cui si vuol operare sull'oggetto (S\_IFREG=regular file, S\_IFCHR=char device, S\_IFBLK=block device, S\_IFIFO=FIFO, che verà rilocata, non essendo un file system); il parametro pathname, infine, indica il path dell'oggetto che stiamo creando.

#### Device number:

Non sono altro che il major number e il minor number. Nelle macchine x86, sono rappresentati come bitmask: in particolare, il major number corrisponde al byte meno significativo all'interno della maschera di bit, mentre il minor number corrisponde al secondo byte meno significativo.

La macro **MKDEV(ma,mi)** corrisponde a una variabile che impacca major number e minor number.

#### Uso dei minor number nei driver:

Le funzioni appartenenti a un driver accettano in input un puntatore a una **struct file**, che consente loro di conoscere la sessione con cui stanno lavorando. A partire dalla sessione è possibile risalire alla dentry e all'inode, da cui si deduce a sua volta il minor number. Di conseguenza, è possibile eseguire delle operazioni basate sul minor number.

Tanto per fare un esempio, possiamo avere un array dove ciascuna entry rappresenta un'istanza di un certo dispositivo (i.e. punta a specifici dati). Nel momento in cui creiamo un nuovo device D all'interno di un file system, possiamo associarvi il minor number X, che corrisponderà alla entry X-esima dell'array: così, quando chiamiamo un'operazione su D, si utilizza il minor number per accedere alla entry corrispondente del nostro array e manipolare così i dati corretti.

## Tabella dei char device (chrdevs)

Fino al kernel 2:

È una tabella le cui entry sono delle **struct device\_struct**:

```
struct device_struct {
    const char * name;
    struct file_operations * fops;
};

static struct device_struct chrdevs[MAX_CHRDEV];
```

Per i char device è possibile effettuare la registrazione e la de-registrazione del driver:

- > **int register\_chrdev (unsigned int major, const char \*name, struct file\_operations \*fops)**: registra un driver per un char device descritto dai parametri name e fops; il driver è invece identificato dal major number dato dal parametro major (che, tra l'altro, corrisponde alla entry di indice 'major' della tabella chrdevs). Se major = 0, è il kernel che sceglie la entry da usare, tornandoci il major number scelto.
- > **int unregister\_chrdev (unsigned int major, const char \*name)**: rilascia un driver per un char device precedentemente registrato.

Per quanto riguarda i block device (blkdevs), si hanno la medesima struct e le medesime API.

MAX\_CHRDEV è numero massimo di device, quindi drivers, gestibili.

### A partire dal kernel 3:

La tabella dei char device è qui una tabella le cui entry sono delle **struct char\_device\_struct** (vedere pagina successiva):

```
#define CHRDEV_MAJOR_HASH_SIZE 255
static struct char_device_struct {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    char name[64];
    struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Pointer to file-operations is here

Rispetto alla versione precedente, sono stati aggiunti dei nuovi campi:

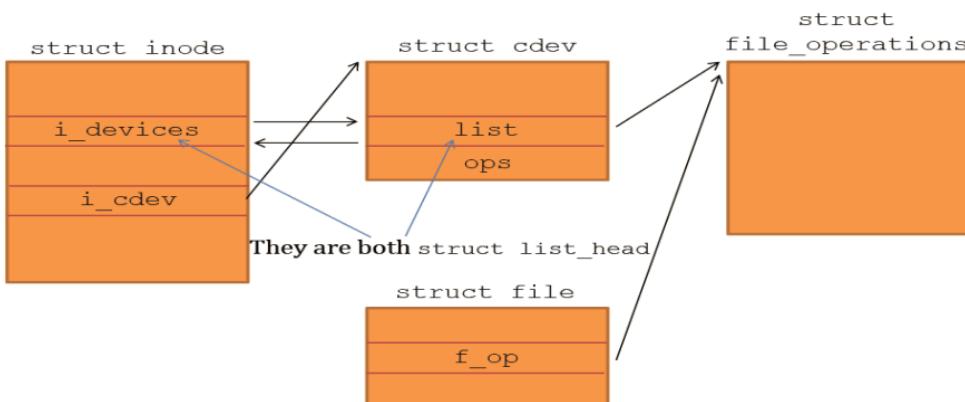
- > **struct char\_device\_struct \*next**: consente di costruire una lista collegata di char\_device\_struct.
- > **unsigned long major**: major number da assegnare al char device.
- > **unsigned int baseminor**: valore minimo del minor number che può essere assegnato alle istanze del char device. Se scriviamo `10`, operazioni su oggetti con minor number = `9` non sono consentite.
- > **int minorct**: numero di minor number differenti che possono essere assgnati alle istanze del char device (per cui i minor number possibili vanno da baseminor a baseminor+minorct-1).

I campi baseminor e minorct servono per verificare agevolmente se l'inode di un certo char device riferenzia un minor number che casca nell'intervallo dei valori ammissibili o meno (senza accedere necessariamente al driver). Queste cose le vede il software di livello superiore del FS su cui il device è collegato, evitando quindi di mettere nel driver tali controlli.

Inoltre, il campo **struct cdev \*cdev** sostituisce la struct file\_operations. La conseguenza è che la struct char\_device\_struct non punta più direttamente al driver, bensì punta a una struttura dati intermedia

(struct cdev) che referenzia a sua volta il driver (i.e. la struct file\_operations).

In definitiva, il collegamento tra le varie strutture dati è schematizzato qui di seguito:



Le file\_operations sono in una tabella collegata a cdev, legata a sua volta su inode.

Per quanto riguarda le funzioni di registrazione e de-registrazione del driver:

- > register\_chrdev() viene rimappata in **int \_\_register\_chrdev (unsigned int major, unsigned int baseminor, unsigned int count, const char \*name, const struct file\_operations \*fops)**, che ha in più i parametri baseminor e count (=minorct).
- > unregister\_chrdev() viene rimappata in **void \_\_unregister\_chrdev (unsigned int major, unsigned int baseminor, unsigned int count, const char \*name)**.

### Struct file\_operations

Nelle versioni meno recenti del kernel, ha il layout riportato qui di seguito (mentre nelle versioni recentissime c'è qualche campo in più):

```

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode*, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
                     unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                     unsigned long, loff_t *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                        loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                      unsigned long, unsigned long);
};
  
```

NB: è possibile sfruttare direttamente i driver per definire nuove operazioni a livello kernel senza modificare la system call table (in particolare tali operazioni si rifaranno dalle system call già esistenti che operano a livello del VFS, come la read() e la write()).

Con queste operazioni, aggiungiamo elementi chiamati senza syscall, bensì con operazione di I/O. Linux lo fa spesso per inserire elementi a livello kernel. Una syscall read e write chiamabile in tal modo è altamente configurabile, quindi non è detto che il risultato sia quello di una classica read o write.

## **Parte finale del boot del kernel**

## Kernel 2.4:

L'ultima funzione che viene invocata all'interno dell'esecuzione di `start_kernel()` è `rest_init()`, che lancia il thread **INIT** con una chiamata a `kernel_thread()` e tenta di schedulare un qualche altro thread in CPU al posto del chiamante (idle process) con una chiamata a `cpu_idle()` (anche se l'idle process rimane comunque sulla runqueue). Il thread INIT si occupa di montare il file system di root applicativo. Finalizza il boot di Linux, ovvero il suo file system per montare programmi basici iniziali.

```
static void rest_init(void)
{
    kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
    unlock_kernel();
    current->need_resched = 1;
    cpu_idle();
}
```

A partire dal kernel 3:

Rispetto alle versioni precedenti del kernel (che non sono NUMA aware), a partire dal kernel 3, all'interno di `rest_init()`, viene invocata la funzione **`numa_default_policy()`**, che cambia la mem-policy utilizzata da round robin a **first touch**. In particolare, round robin porta a selezionare secondo una regola circolare i nodi NUMA in cui materializzare le pagine di memoria, e infatti è una politica sfruttata durante la fase di init per avere il kernel spalmato equamente tra tutti i nodi NUMA; first touch, invece, fa sì che le pagine vengano materializzate nel nodo NUMA più vicino alla CPU che ospita il thread in esecuzione.

```
static noinline void __init_refok rest_init(void)
395 {
396     int pid;
397
398     rcu_scheduler_starting();
399     /*
400      * We need to spawn init first so that it obtains pid 1, however
401      * the init task will end up wanting to create kthreads, which, if
402      * we schedule it before we create kthreadd, will OOPS.
403 */
404     kernel_thread(kernel_init, NULL, CLONE_FS);
405
406     .....
407     .....
408     numa_default_policy();
```

## Switch off round-robin to first-touch

La funzione mount\_root():

È una funzione invocata dal thread INIT ed è strutturata nel seguente modo:

All'interno della funzione `create_dev()`, che viene invocata da `mount_root()`, si ha una `sys_mknod()`, che crea un nuovo nodo. In particolare, qui stiamo operando all'interno di Rootfs e, se osserviamo i parametri, notiamo che il dispositivo che stiamo mettendo in piedi avrà come nodo radice la directory `/dev/root` che, appunto, è una sottodirectory di `/dev` (che, a sua volta, è child di “`/`”, il device di Rootfs). La directory `/dev/root`, essendo un nodo all'interno di Rootfs, sarà associata a un major number e a un minor number, dove il major number identifica il block-device driver che ci permetterà di prelevare i blocchi del file system di root.

#### La funzione `init()`:

È la prima funzione invocata dal thread INIT ed è strutturata nel seguente modo:

```
static int init(void * unused){
    struct files_struct *files;
    lock_kernel();
    do_basic_setup(); ← registering drivers
    prepare_namespace();

    .....
    if (execute_command) run_init_process(execute_command);
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");
    panic("No init found. Try passing init= option to kernel.");
}
```

-> **do\_basic\_setup()**: registra dei driver utilizzando il RAM file system associato alla compilazione del kernel. Più precisamente, il RAM file system viene montato all'inizio della funzione per poi essere smontato nel momento in cui i driver sono registrati.

-> **prepare\_namespace()**: prepara il namespace di default. In particolare, crea la directory `/dev`, la directory `/root` (che non c'entra nulla con `/dev/root`) e il nodo `/dev/console` (che è un char-device a cui devono essere assegnati un major number e un minor number); `/dev/console`, in quanto nodo, risulterà essere un oggetto eterogeneo rispetto agli altri elementi (le directory) presenti in Rootfs. SPAZIO DEI NOMI SUL FILE SYSTEM, non è il classico namespace.

Dopodiché, `prepare_namespace()` invoca proprio la funzione `mount_root()`, per poi terminare effettuando il montaggio vero e proprio del file system di root applicativo (funzione `sys_mount()`) e cambiando la directory di root di riferimento (funzione `sys_chroot()`).

Di seguito sono mostrati i dettagli implementativi di `prepare_namespace()`:

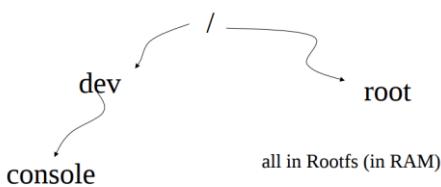
Sul file system solo in RAM, `dev` e `root`, su cui aggancio un altro file system e la `dev/console`. Poi monto

```
void prepare_namespace(void){
    .....
    sys_mkdir("/dev", 0700);
    sys_mkdir("/root", 0700);
    sys_mknod("/dev/console", S_IFCHR|0600,
              MKDEV(TTYAUX_MAJOR, 1));

    .....
    mount_root();
out:
    .....
    sys_mount(".", "/", NULL, MS_MOVE, NULL);
    sys_chroot(".");
    .....
}
```

-> Dopo l'invocazione a `prepare_namespace()`, si effettuano alcuni tentativi nel lanciare un programma applicativo: se uno di questi viene avviato con successo allora è tutto ok, altrimenti si va in `panic()`.

Stato del VFS prima dell'invocazione a mount\_root(): RAM\_FS



### La funzione mount\_block\_root():

```

static void __init mount_block_root(char *name, int flags) {
    char *fs_names = __getname(); char *p;
    get_fs_names(fs_names);
retry: for (p = fs_names; *p; p += strlen(p)+1) {
    int err = sys_mount(name, "/root", p, flags, root_mount_data);
    switch (err) {
        case 0: goto out;
        case -EACCES: flags |= MS_RDONLY; goto retry;
        case -EINVAL:
        case -EBUSY: continue;
    }
    printk ("VFS: Cannot open root device \'%s\' or %s\n",
           root_device_name, kdevname(ROOT_DEV));
    printk ("Please append a correct \"root=%s\" boot option\n");
    panic("VFS: Unable to mount root fs on %s", kdevname(ROOT_DEV));
}
panic("VFS: Unable to mount root fs on %s", kdevname(ROOT_DEV));
out:   pathname(fs_names);
sys_chdir("/root");
ROOT_DEV = current->fs->pwdmnt->mnt_sb->s_dev;
printk("VFS: Mounted root (%s filesystem)%s.\n",
       current->fs->pwdmnt->mnt_sb->s_type->name,
       (current->fs->pwdmnt->mnt_sb->s_flags & MS_RDONLY) ?
       " readonly" : "");
}

```

-> **get\_fs\_names (fs\_names)**: a partire dalla lista dei file system che siamo in grado di gestire, prende tutti i nomi dei FS e li impacca all'interno di un array.

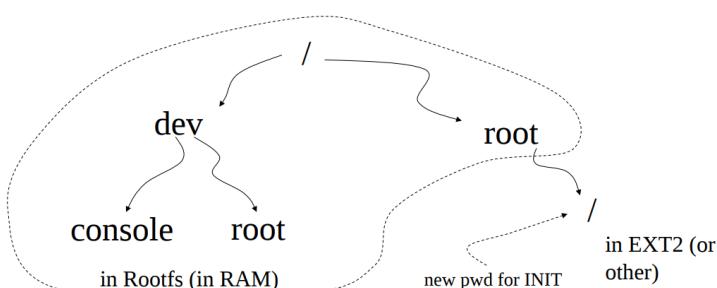
-> Dopo l'invocazione a `get_fs_names()`, si itera all'interno dell'array e si tenta di invocare l'operazione di montaggio dei file system tramite una **sys\_mount (name, "/root", p, flags, root\_mount\_data)**. L'iterazione termina nel momento in cui l'operazione di montaggio per uno di questi file system va a buon fine oppure quando i file system finiscono; nel primo caso vuol dire che la funzione di lettura del superblocco di uno di questi FS ha stabilito che il superblocco è compliant col file system type specificato come terzo parametro di `sys_mount()` (→ è tutto ok e il file system montato risulterà essere il file system di root applicativo), mentre nel secondo caso si va in `panic()`.

### La system call mount():

È la system call che si basa sull'implementazione lato kernel di `sys_mount()`.

-> **int mount (const char \*source, const char \*target, const char \*filesystemtype, unsigned long mountflags, const void \*data)**: qui il parametro interessante è `data`, che è file system specific; ad esempio, può essere un pointer a un'area dati che può essere utilizzata dalla funzione di lettura del superblocco del file system che stiamo montando.

Stato del VFS dopo l'invocazione a mount\_root():



Qui vengono montati FS applicativi, che sono FS esterni (EXT2 ne è un esempio).

Inoltre, qui viene anche riportata la directory /dev, che ci permette di usare i driver e di inserire nuovi nodi / device.

## Descrizione di open(), close(), read(), write()

### Open():

Di seguito sono riportati gli step da seguire quando viene invocata la open():

- 1) Ottenere un file descriptor **libero** mediante current->files->fd. L'operazione va a buon file se esiste almeno un file descriptor effettivamente inutilizzato.
- 2) Ottenere la dentry e l'inode dell'oggetto di I/O da aprire tramite una filp\_open(). L'operazione va a buon fine se l'oggetto di I/O esiste e se dispone dei permessi necessari per essere aperto secondo le modalità specificate.
- 3) Se entrambe le operazioni precedenti hanno avuto successo, allora il file descriptor ottenuto nel punto 1 viene linkato alla dentry dell'oggetto di I/O da aprire.

### Close():

Di seguito sono riportati gli step da seguire quando viene invocata la close():

- 1) Rilasciare la dentry dell'oggetto di I/O che stiamo chiudendo dal file descriptor tramite una filp\_close().
- 2) Rilasciare il file descriptor a partire da current->files->fd.

### Read() / write():

Di seguito sono riportati gli step da seguire quando viene invocata la read() o la write():

- 1) Ottenere un riferimento alla dentry dell'oggetto di I/O target tramite il file descriptor.
- 2) Ottenerne un riferimento alla struttura file\_operations (che identifica il driver con cui dobbiamo lavorare) a partire dalla struttura relativa alla sessione (che, a sua volta, è raggiungibile dal file descriptor).
- 3) Invocare l'operazione desiderata all'interno di file\_operations.

## File system proc

È uno pseudo-file system nel senso che non mantiene informazioni che sono registrate su un qualche dispositivo di memoria di massa, bensì informazioni accessibili esclusivamente in RAM. In particolare, le informazioni di cui tiene traccia riguardano:

- > I processi attivi.
- > L'intero contenuto della memoria.
- > Il settaggio di alcuni aspetti di livello kernel (e.g. il numero corrente di moduli montati).

In pratica, proc permette alle applicazioni user-level di colloquiare direttamente con il kernel ed eventualmente di modificare delle impostazioni di livello kernel.

I file più importanti all'interno di /proc sono listati qui di seguito:

- **cpuinfo** contains the information established by the kernel about the processor at boot time, e.g., the type of processor, including variant and features
- **kcore** contains the entire RAM contents as seen by the kernel
- **meminfo** contains information about the memory usage, how much of the available RAM and swap space are in use and how the kernel is using them
- **version** contains the kernel version information that lists the version number, when it was compiled and who compiled it
- **net/** is a directory containing network information
- **net/dev** contains a list of the network devices that are compiled into the kernel. For each device there are statistics on the number of packets that have been transmitted and received
- **net/route** contains the routing table that is used for routing packets on the network
- **net/snmp** contains statistics on the higher levels of the network protocol
- **self/** contains information about the current process. The contents are the same as those in the per-process information described below
- **pid/** contains information about process number *pid*. The kernel maintains a directory containing process information for each process
- **pid/cmdline** contains the command that was used to start the process (using null characters to separate arguments)
- **pid/cwd** contains a link to the current working directory of the process
- **pid/environ** contains a list of the environment variables that the process has available
- **pid/exe** contains a link to the program that is running in the process
- **pid/fd/** is a directory containing a link to each of the files that the process has open
- **pid/mem** contains the memory contents of the process
- **pid/stat** contains process status information
- **pid/statm** contains process memory usage information

#### Registrazione e creazione di proc:

La funzione **proc\_root\_init()**, che viene invocata da **start\_kernel()**, si occupa di registrare il file system proc e di crearne un'istanza effettiva; può fare anche altro, come ad esempio creare le seguenti sottodirectory di proc: **net**, **sys**, **sys/fs**.

#### Struct proc\_dir\_entry:

È una struttura dati che ci permette di identificare cosa è possibile fare su un determinato file F che appartiene a proc. La sua implementazione (almeno per quanto concerne le prime versioni del kernel) è riportata qui di seguito:

```

struct proc_dir_entry {
    unsigned short low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;
    nlink_t nlink;      uid_t uid;      gid_t gid;
    unsigned long size;
    struct inode_operations * proc_iops;
    struct file_operations * proc_fops;
    get_info_t *get_info;
    struct module *owner;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;
    write_proc_t *write_proc;
    atomic_t count;      /* use count */
    int deleted;         /* delete flag */
    kdev_t rdev;
};

};

```

-> **const char \*name**: è il nome del file F.

-> **struct inode\_operations \*proc\_iops & struct file\_operations \*proc\_fops**: sono puntatori a driver di operazioni.

-> **read\_proc\_t \*read\_proc & write\_proc\_t \*write\_proc**: sono puntatori a due funzioni che, insieme, costituiscono una sorta di minidriver eventualmente da usare al posto di proc\_fops. In particolare, stanno qui per semplificare la vita del programmatore: supponiamo di definire non il driver proc\_fops bensì le operazioni read\_proc e write\_proc. Allora quello che viene realmente utilizzato è un driver di default: quando viene invocata una funzione di read per il file F, tale funzione va all'interno del driver di default e invoca un modulo di default che chiama alcune operazioni di preambolo prima della read ma anche alcune operazioni di coda dopo la read.

La struttura proc\_dir\_entry, nelle ultimissime versioni del kernel, si presenta in quest'altro modo:

```

struct proc_dir_entry {
    .....
    const struct inode_operations *proc_iops;
    union {
        const struct proc_ops *proc_ops;           ← for a file
        const struct file_operations *proc_dir_ops; ← for a directory
    };
    const struct dentry_operations *proc_dops;
    ...
    proc_write_t write;
    void *data;
    .....
} __randomize_layout; ← improvement of security

```

Nelle `proc\_ops` potremmo avere una write, che però è anche definita fuori da `proc\_write\_t write`, in questo caso viene definita una priorità di utilizzo.

In definitiva, la struct proc\_dir\_entry descrive ciascun elemento del file system proc in termini di:

- Nome.
- Inode operations.
- File operations.
- Funzioni di read e di write specifiche per quell'elemento.

#### Montaggio di proc:

Il file system proc non viene necessariamente montato durante il boot del kernel ma viene istanziato solo

se viene configurato. Tipicamente viene montato dal thread INIT, e l'operazione di montaggio può essere configurata come un task runtime oppure può essere effettuata sulla base delle informazioni fornite da **/etc/fstab**.

Badiamo che, poiché si tratta di un file system che vive esclusivamente in RAM, è a **istanza singola**, per cui non necessita di un block-device driver che prelevi le sue informazioni. Di conseguenza, quando viene montato proc, deve essere specificato come parametro esclusivamente il tipo di file system ma non un particolare device. Eseguire una Mkdir su fsproc, comporta la creazione nella cache di memoria, ogni oggetto che la rappresenta (dentry, inode...) viene allocata. Non si è però parlato di dati, per questo esiste anche `\*proc\_create\_data()`.

#### API per gestire directory ed entry di proc:

-> **struct proc\_dir\_entry \*proc\_mkdir (const char \*name, struct proc\_dir\_entry \*parent)**: crea una nuova directory col nome specificato dal parametro name e all'interno della directory puntata dal parametro parent.

-> **static inline struct proc\_dir\_entry \*proc\_create (const char \*name, umode\_t mode, struct proc\_dir\_entry \*parent, const struct file\_operations \*proc\_fops)**: crea un nuovo elemento (una nuova entry) all'interno di proc. Name è il nome dell'elemento; mode indica i permessi di accesso all'elemento; parent è la entry parent dell'elemento (se vale NULL, stiamo piazzando l'elemento nella root di proc); proc\_fops è la struttura in cui verranno create le file operations per il nuovo elemento.

-> **static inline struct proc\_dir\_entry \*proc\_create\_data (const char \*name, umode\_t mode, struct proc\_dir\_entry \*parent, const struct file\_operations \*proc\_fops, void \*data)**: è del tutto analoga a proc\_create() ma in più specifica anche un puntatore ai dati dell'oggetto di I/O che stiamo creando.

Per quanto riguarda le operazioni di read e write, il file system proc dispone delle stesse funzioni rispetto a un qualunque file system gestito dal VFS:

-> **ssize\_t \*read (struct file \*, char \*, size\_t, loff\_t \*)**.

-> **ssize\_t \*write (struct\_file \*, const char \*, size\_t, loff\_t \*)**.

Tuttavia, nel caso particolare in cui i campi read\_proc e write\_proc della proc\_dir\_entry non sono NULL, le operazioni di read e write effettive sono proprio quelle registrate da questi due campi e differiscono dalle operazioni di read e write standard:

-> **typedef int (read\_proc\_t) (char \*page, char \*\*start, off\_t off, int count, int \*eof, void \*data)**: il file system proc, per consegnare le informazioni al chiamante, le impacca in un buffer e le trasferisce mediante una copy\_to\_user(). Di base, il buffer è dato dalla pagina di memoria puntata dal parametro page ma, nel caso in cui 4K byte non dovessero bastare, si utilizza un'altra area di memoria referenziata dal (doppio) puntatore start, il quale, a seguito dell'esecuzione della funzione, comunicherà al chiamante l'indirizzo di memoria realmente utilizzato per registrare le informazioni prelevate dal file system proc.

-> **typedef int (write\_proc\_t) (struct file \*file, const char \*buffer, unsigned long count, void \*data)**.

Prendendo sempre in considerazione la funzione di lettura (read\_proc\_t()), una descrizione schematica di tutti i parametri e del valore di ritorno è riportata di seguito:

Char* page	A pointer to a one-page buffer. (A page is PAGE_SIZE bytes big)
char** start	A pass-by-reference char * from the caller. It is used to tell the caller where is the data put by this procedure. (If you're curious, you can point the caller's pointer at your own text buffer if you don't want to use the page supplied by the kernel in page.)
off_t off	An offset into the buffer where the reader wants to begin reading
Int count	The number of bytes after off the reader wants.
Int* eof	A pointer to the caller's eof flag. Set it to 1 if the current read hits EOF.
void* data	Extra info you won't need
return value	Number of bytes written into page

### File system sys

È un file system disponibile a partire dal kernel 2.6 che è simile in spirito a proc ma si basa su strutture dati differenti. Rappresenta un modo alternativo per il kernel per esporre o impostare le sue informazioni mediante le operazioni di I/O comuni. Sempre FS in memory.

Qui abbiamo che:

- I **kernel object** rappresentano le directory.
- Gli **object attribute** rappresentano i file regolari.
- Le **object relationship** rappresentano i link simbolici ai file regolari.

#### Kernel object:

È un'entità che permette di:

- > Identificare oggetti individuali.
- > Identificare gruppi di oggetti.
- > Identificare la tipologia degli oggetti.
- > Associare una stessa tipologia a molteplici oggetti.
- > Identificare le gerarchie delle informazioni.

Di riflesso, tutte queste operazioni sono possibili all'interno del file system sys.

La struct che implementa il kernel object è riportata qui di seguito:

```
struct kobject{
    const char          *name;
    struct list_head    entry;
    struct kobject      *parent;
    struct kset         *kset;
    struct kobj_type   *ktype;
    struct kernfs_node *sd;
    /*sysfs directory entry*/
    struct kref         kref;
    .....
    .....
}
```

Reference counting

- Name = nome del kernel object.
- Parent = kernel object parent (da cui dipende il *nostro* kernel object). Il gruppo può essere lo stesso, per unificare i metadati di gestione.
- Kset = insieme di kernel object a cui appartiene il nostro kernel object (appartengono allo stesso kset i kobject che hanno il medesimo parent). A quali set appartengono i kernel object.
- Ktype = tipologia del kernel object, che identifica anche le operazioni che si possono fare.
- Kref = reference counter per il kernel object.

In particolare, la struct kobj\_type descrive l'aspetto operativo del kernel object ed è fatta così:

```
struct kobj_type{
    void (*release)(struct kobject*); // Called when reference counting reaches the value zero
    struct sysfs_ops *sysfs_ops;
    struct attribute **defaultAttrs; // We can have multiple attributes
}
```

Actual operations to be executed on the object

- DefaultAttrs = attributi propri del nostro kernel object; praticamente corrispondono ai file regolari che possiamo trovare all'interno della directory corrispondente al nostro kernel object.
- void \*release (struct kobject \*) = funzione che viene invocata per deallocare il nostro kernel object quando il relativo reference counter raggiunge il valore 0.
- Sysfs\_ops = operazioni effettive che possono essere eseguite sul kernel object.

Per quanto riguarda sysfs\_ops, è una struct composta semplicemente da due metodi (in pratica è un mini-driver):

```
struct sysfs_ops {
    /* method invoked on read of a sysfs file */
    ssize_t (*show) (struct kobject *kobj,
                    struct attribute *attr,
                    char *buffer);

    /* method invoked on write of a sysfs file */
    ssize_t (*store) (struct kobject *kobj,
                      struct attribute *attr,
                      const char *buffer,
                      size_t size);
}
```

- show(): mostra un determinato attributo del kernel object specificato. Quanto mostro? Una pagina, allocata dal layer FS. Questo ci fa capire che tali operazioni sono di "piccole dimensioni".
- store(): registra delle informazioni per un determinato attributo del kernel object.

### Cosa possiamo fare coi kernel object?

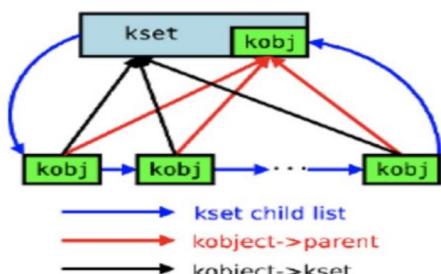
Possiamo rappresentare i dati che possono essere usati dal software per tenere traccia dello stato corrente sia delle entità logiche che di quelle fisiche. Ad esempio, abbiamo la rappresentazione di:

- > Sottosistema di bus USB.
- > Sottosistema di char devices.
- > Sottosistema di block devices.

Un kernel object può appartenere al più a un unico sottosistema, e un sottosistema deve necessariamente contenere kernel object identici.

In Linux si utilizza proprio la struct kset per raggruppare insieme più kernel object all'interno del medesimo sottosistema.

Di seguito è mostrata una rappresentazione dei collegamenti che i kernel object hanno con il kset e tra loro:



Comunque sia, un kernel object può essere associato o non associato a un elemento del file system sys: se lo è, allora si trova in un kset, altrimenti no.

#### API su kernel object e kset:

- > **int kobject\_add (struct kobject \*kobj, struct kobject \*parent, const char \*fmt ...)**: aggiunge il kernel object specificato come parametro a un particolare kset.
- > **void kobject\_del (struct kobject \*kobj)**: rimuove il kernel object specificato come parametro dal kset a cui apparteneva.
- > **void kset\_init (struct kset \*kset)**: inizializza un nuovo kset.
- > **int kset\_register (struct kset \*kset)**: registra un determinato kset.
- > **void kset\_unregister (struct kset \*kset)**: de-registra un determinato kset.
- > **struct kset \*kset\_get (struct kset \*kset)**: incrementa il reference counter di un determinato kset.
- > **void kset\_put (struct kset \*kset)**: decrementa il reference counter di un determinato kset.
- > **void kobject\_set\_name (struct kobject \*kobj, char \*name)**: assegna un nome a un determinato kernel object.

#### Eventi verso lo user space:

All'interno della struttura dati kset è registrato un function pointer chiamato **kobject\_uevent**. Quando avviene un evento che viene tracciato dal kernel, quest'ultimo utilizza la funzione referenziata da kobject\_uevent per avviare un'applicazione user space per consegnare al lato user delle informazioni riguardanti l'evento che è occorso.

Un esempio classico è l'inserimento di un dispositivo USB all'interno della macchina: in tal caso, viene avviato un programma user space che notifica l'utente dell'inserimento e gli chiede cosa vuole fare.

#### API core del FS sys:

- > **int sysfs\_create\_dir (struct kobject \*k)**: crea una nuova directory nel FS sys.
- > **void sysfs\_remove\_dir (struct kobject \*k)**: rimuove una determinata directory dal FS sys.
- > **int sysfs\_rename\_dir (struct kobject \*k, const char \*new\_name)**: rinomina una directory nel FS sys.
- > **int sysfs\_create\_file (struct kobject \*k, const struct attribute \*attr)**: crea un nuovo file all'interno del kernel object specificato.
- > **void sysfs\_remove\_file (struct kobject \*k, const struct attribute \*attr)**: rimuove un determinato file dal kernel object specificato.
- > **int sysfs\_update\_file (struct kobject \*k, const struct attribute \*attr)**: aggiorna un determinato file nel kernel object specificato.

NB: la struttura attribute specifica il nome, l'owner e i permessi d'accesso del file:

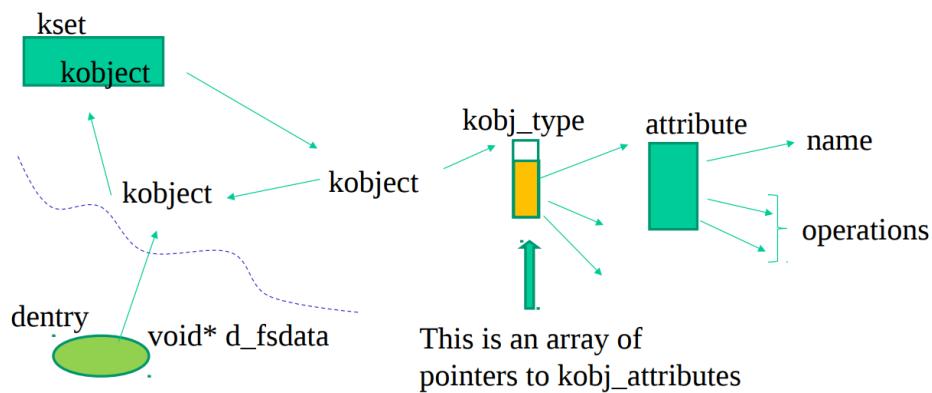
```
struct attribute {
    char *name;
    struct module *owner; // specifica dove sono posizionati i moduli che
                          // operano su queste entità che creiamo
    mode_t mode;
};
```

Inoltre, la struct attribute è incapsulata a sua volta all'interno della **struct kobj\_attribute** che, tra l'altro, punta alla funzione di show() e alla funzione di store() per quanto riguarda l'operatività dell'attributo:

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj,
                   struct kobj_attribute *attr, char *buf);
    ssize_t (*store)(struct kobject *kobj,
                    struct kobj_attribute *attr,
                    const char *buf, size_t count);
}
```

## Architettura di insieme:

È raffigurata nella pagina successiva.



## Creazione di file device nel FS sys:

**/sys/class** è un device file che internamente ospita il riferimento ad altri device file. Per creare un device file in questa specie di “directory”, è possibile ricorrere ad API come **class\_create()** e **device\_create()** che, infatti, restituiscono un puntatore a una struct class. I dispositivi devono essere collegati alle classi, con le varie info nel kernel arriviamo fino ai driver registrati ai major numbers di linux. Creiamo device con certo nome e major number. L’oggetto apparirà anche sul file system, e sarà pilotabile con un driver.

## SOFTWARE SECURITY ASPECTS

### Idee di base

- > I sistemi e le applicazioni devono essere utilizzabili esclusivamente dagli utenti legittimi.
- > L'accesso ai sistemi deve essere garantito sulla base di un meccanismo di autorizzazione e secondo alcune regole stabilite da un qualche **amministratore**. Scopriremo come ne servano due di admin (ad esempio, del mio pc sono io l'admin, ma c'è anche chi ha progettato il sistema Linux su cui scrivo).
- > Un sistema non utilizzabile da nessuno è a tutti gli effetti un sistema inutile, e un attaccante potrebbe voler raggiungere tale condizione effettuando un **attacco DOS (Denial of Service)**.

### Attacco DOS:

Può essere basato sul flooding di:

- Connessioni TCP.
- Pacchetti UDP / pacchetti di livello applicativo.
- Messaggi di richiesta su protocolli application specific.

Si tratta di un tipo di attacco banale dal momento che viene gestito effettuando un trade-off tra l'utilizzo corrente delle risorse del server e l'accettazione delle richieste che arrivano al server (dove il numero di richieste accettate in un certo intervallo di tempo deve essere al di sotto di una certa soglia di tolleranza): all'attaccante basta fare in modo che venga rigettato tutto, anche le richieste provenienti dagli utenti legittimi.

Dal punto di vista del server, è essenziale avere dei metodi che gli permettano di identificare il traffico buono che gli arriva da quello inviato da un malintenzionato, e tale identificazione deve essere effettuata "on the fly" e in modo efficiente. Di conseguenza, abbiamo bisogno di meccanismi che rendano scalabile il software per l'identificazione del traffico, come:

- > Sfruttamento del multicore.
- > NUMA awareness.
- > Algoritmi di sincronizzazione non bloccanti (vedi RCU).

### **Accessi legittimi e illegittimi ai dati**

Il termine "legittimo" include molti concetti nell'ambito IT, molti dei quali correlati all'accesso ai **dati** e a **porzioni di codice**. Le porzioni di codice sono particolarmente critiche poiché l'accesso ai dati viene implementato proprio da un pezzo di codice: di conseguenza, un accesso illegittimo a una porzione di codice può causare un accesso illegittimo a dei dati.

### Approcci per conseguire la sicurezza:

A grandi linee sono tre:

- 1) Crittografia.
- 2) Autenticazione / abilitazione (e.g. per l'esecuzione di porzioni di codice).
- 3) Sicurezza potenziata dei sistemi operativi.

Ciascuno di questi tre approcci si concentra su un aspetto specifico della sicurezza; possono essere combinati insieme per migliorare il livello di sicurezza complessivo di un sistema IT.

### Accessi illegittimi R/W ai **dati** che abbiamo già visto:

I problemi non sono nel codice, bensì nell'hardware a disposizione e cosa fa. Ne sono esempi:

- > Side channel.
- > Branch miss-prediction (Spectre).
- > Speculazione lungo i path di esecuzione in cui vengono lanciate eccezioni (Meltdown).
- > Speculazione sul TAG-to-value (LT1 terminal).
- > Hackeraggio delle strutture dati del kernel (e.g. system call table).

Le contromisure che abbiamo visto finora per questi attacchi sono:

- Randomizzazione dell'address space / delle strutture dati.
- Ispezione dei signature dei programmi (in modo tale che si eviti che vengano inserite istruzioni pericolose per l'integrità del codice e dei dati).
- Cifratura di stream di dati, blocchi di dispositivi, pagine di memoria e dati generici come le password.

Per effettuare la cifratura delle password (in particolare una cifratura **one-way**, perché di fatto si tratta di una funzione hash crittografica), si ricorre all'API **char \*crypt (const char \*key, const char \*settings)**, che fa uso di un **salt**; key è la password in chiaro, mentre settings specifica l'algoritmo di cifratura da utilizzare (e.g. MD5, SHA-256, SHA-512) e il salt.

Sempre a proposito delle password, all'interno dei sistemi Linux il database delle password è mantenuto su due file distinti: **/etc/passwd** e **/etc/shadow**; il primo è accessibile a qualunque utente e viene consultato da alcuni comandi di base (e.g. uid) per recuperare informazioni di base sugli utenti del sistema (magari utenti diversi hanno diritti diversi sui file), mentre il secondo è accessibile esclusivamente all'utente root e mantiene informazioni critiche come le password cifrate. Non siamo in contesto di **Reference Monitor**, che vedremo successivamente.

#### Accessi illegittimi al codice che abbiamo già visto:

- > Miss-speculation.
- > Hackeraggio delle strutture del kernel (e.g. system call table).
- > Side effects sull'hardware a seguito dell'esecuzione speculativa di istruzioni.

Le contromisure che abbiamo visto finora per questi attacchi sono:

- Quelle elencate precedentemente a proposito degli accessi illegittimi ai dati.
- Correzioni esplicite dei valori che, ad esempio, servono per spiazzarci all'interno di strutture dati (vedi la sanitizzazione, che previene lo scenario in cui ci si spiazza a indirizzi arbitrari ed esterni alla struttura dati target per accedere, seppur speculativamente, a chissà che cosa in memoria).
- Impedimento di montare moduli del kernel, in modo tale da essere costretti a utilizzare un'istanza del kernel che sia "trusted" (di fatto i moduli aggiuntivi possono implementare qualunque tipo di attività). Ma attenzione: chi è che esegue il lavoro di montare un modulo del kernel? Non lo fa propriamente un umano, bensì un pezzo di codice eseguito da un particolare thread. Possiamo trovare un thread sacrificale, di cui alteriamo il control-flow per fare quello che vogliamo noi. Il problema non è l'utente root, bensì il fatto che ci siano thread attivi che possano essere alterati per fare ciò. La protezione non deve essere solo da parte del root, ma anche da un'altra parte, che è l'intro per il **reference monitor**.

Questa attività, come molte altre, dovrebbe essere svolta solo da alcuni thread (che hanno determinati permessi) e non da altri. Approfondiamo l'aspetto sull'esecuzione di porzioni di codice non ammesse da parte di taluni thread introducendo il **buffer overflow**.

#### **Buffer overflow**

È una tecnica che porta un thread a far uso di locazioni di memoria non legittime, compresi alcuni pezzi di codice. In particolare, i blocchi di codice possono:

- > Essere già presenti all'interno dell'address space accessibile dal thread.
- > Essere iniettati dall'attaccante.
- > Essere ottenuti componendo insieme delle frazioni di funzioni (dei gadget): ad esempio, tramite appositi meccanismi, si può eseguire un pezzetto della funzione  $f_1$  e poi saltare a un pezzetto della funzione  $f_2$ , dove la somma di questi due pezzetti costituisce un blocco di codice illegittimo per il nostro thread.

Dal punto di vista tecnico, il buffer overflow porta il contenuto di una qualche locazione di memoria a essere sovrascritto da un qualche altro valore mediante un meccanismo non conforme alla logica applicativa prevista dal programma. Il danno minimo che può essere causato da un buffer overflow è un

segmentation fault, mentre il danno massimo è la possibilità di accesso a una qualunque risorsa (porzione di codice o dati).

Uno scopo tipico del buffer overflow consiste nel far sì che il thread devii dal flusso di esecuzione previsto mediante una manipolazione illecita dello stack.

Consideriamo ad esempio lo stack frame associato a una certa funzione f. In fondo a tale stack frame si trova l'indirizzo di ritorno di f. Secondo un flusso di esecuzione legittimo, f dovrebbe poter manipolare sullo stack solo informazioni come i parametri e le variabili locali. Un attaccante può fare in modo che venga modificato il valore di ritorno in modo tale che, al completamento dell'esecuzione di f, non venga correttamente restituito il controllo al chiamante, bensì si salvi verso una zona di codice arbitraria.

### Esempio 1:

Supponiamo di avere la seguente funzione:

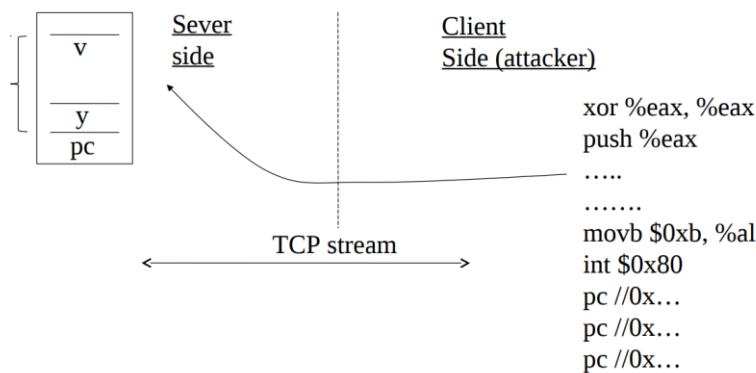
```
void do_work(int x){
    char v[SIZE];
    int y;
    ....
}
```

Una possibilità per l'attaccante è spiazzarsi all'interno dell'array v con un offset maggiore di SIZE in modo tale da raggiungere l'indirizzo dello stack dove è posto il program counter (pc) ed effettuare qui delle operazioni di scrittura.

Questo meccanismo si chiama **stack exploit**. Se l'indirizzo di ritorno che viene inserito nel program counter è relativo a una zona di codice iniettata, allora il meccanismo è detto **stack exploit with payload**.

### Esempio 2:

Vediamo una variante "stack exploit with payload" dell'attacco appena spiegato (che può essere performato anche su un server remoto): qui i dati che vengono immessi nell'array di caratteri v possono costituire una vera e propria zona di codice, mentre il program counter (indirizzo di ritorno) può essere modificato in modo tale da puntare all'inizio della zona di codice.



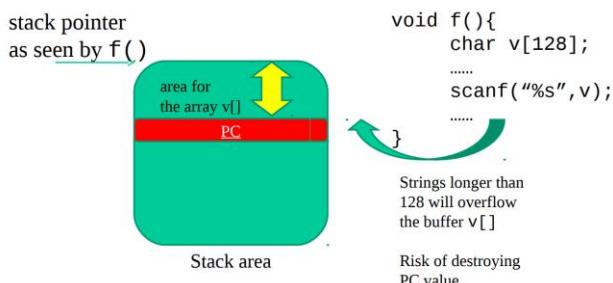
L'unico problema per l'attaccante è che la posizione dello stack è randomizzata, per cui è molto difficile indovinare l'indirizzo esatto da inserire all'interno di pc. La soluzione che si adotta consiste nell'aggiungere tante istruzioni di NOP all'inizio del pezzo di codice da iniettare, in modo tale che ci siano delle buone probabilità di inserire all'interno di pc un indirizzo dello stack che caschi all'interno del pezzo di codice ma prima delle istruzioni che eseguono il lavoro vero e proprio desiderato dall'attaccante (i.e. un indirizzo che cada in mezzo alle NOP).

### Scanf e gets:

Un modo facilissimo per performare questo tipo di attacchi è sfruttare delle funzioni standard come **scanf** e **gets**:

- Scanf acquisisce dei dati in input senza effettuare alcun controllo sulla quantità di questi dati: se si danno in input abbastanza dati, questi eccederanno la dimensione del buffer a disposizione causando così un buffer overflow.

- Gets ha la stessa problematica di scanf, con l'aggravante per cui, se viene data in input una stringa, questa viene acquisita tutta per intero anche se contiene eventuali separatori come "\n".



Di fatto, la gets è ora deprecata. Per quanto invece riguarda la scanf, un primissimo rimedio è stato definirne una nuova versione che non accetta più come parametro direttamente l'area di memoria in cui inserire i dati acquisiti, bensì un puntatore all'area di memoria (e.g. un char \*\*): in tal modo, ci pensa la scanf internamente ad allocare una zona di memoria in cui inserire i dati.

In realtà, è stata anche introdotta un'altra funzione di libreria, la **scanf\_s()**, che accetta come parametro aggiuntivo il numero massimo di byte da acquisire in modo tale che, se il puntatore all'indirizzo in cui inserire i dati in input è corretto, si prevenga il rischio di buffer overflow; tuttavia, se invece il puntatore è scorretto (ad esempio a causa di un bug del software), le funzioni di libreria sicure non ci risolvono il problema.

#### Canary tag:

Rappresenta una possibile soluzione ai buffer overflow, e non è altro che un valore (un tag) read only che, per ogni stack frame, viene posizionato tra il program counter (indirizzo di ritorno della funzione f) e le altre informazioni relative alla medesima funzione f. Nel momento in cui termina l'esecuzione di f, prima di restituire il controllo al chiamante, viene controllato il valore del canary tag: se è rimasto immutato rispetto a quando f ha preso il controllo, allora si assume che va tutto bene; in caso contrario vuol dire che c'è una possibilità per cui sia stato manomesso anche il program counter e si interrompe l'esecuzione dell'applicazione. È una informazione binaria.

Tuttavia, il canary tag non è ancora la soluzione definitiva: un attaccante può anche trovare un modo per manomettere il valore del program counter senza toccare il canary tag. Comunque sia, in gcc l'uso del canary tag è attivo di default: per disattivarlo, basta compilare il programma col flag **-f no-stack-protector**.

#### XD flag:

Un'altra possibile contromisura consiste nel proteggere le pagine di memoria (in particolar modo quelle dello stack) dal fetch (i.e. dall'esecuzione). Ciò è possibile settando opportunamente l'XD flag di ciascuna entry delle page table. In tal modo non è più possibile installare codice eseguibile sulla stack area.

In x86 tutto era fetchabile, ora no. Le JVM hanno JIT, prendono codice non eseguibile in byte code, e lo trasformano in codice eseguibile.

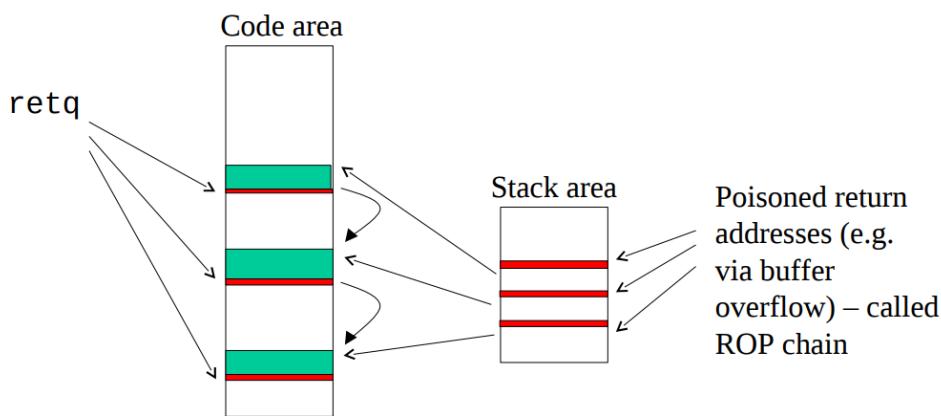
Purtroppo, non siamo ancora safe: ad esempio, l'attivazione di un nuovo programma (come una shell!) tramite una exec viene implementata utilizzando pochissime istruzioni macchina. Di conseguenza, è molto probabile avere tutte queste istruzioni macchina già all'interno dell'address space, anche se sono sparse. Ciò consente a un attaccante di fare un lavoro di patch in modo tale da eseguire queste istruzioni sparse proprio con lo scopo di fare una exec, dando luogo a un path di esecuzione non originariamente previsto per l'applicazione. In particolare, il meccanismo che permette una cosa del genere è detto **Return Oriented Programming (ROP)**. L'idea è che vogliamo eseguire alcune istruzioni, e le cerchiamo (o singolarmente o in blocchi) all'interno dell'AS, e mediante gadget cerchiamo di arrivarci per eseguirle, e magari passare il

controllo ad un altro gadget che continua nel nostro intento. Portiamo thread ad eseguire al di fuori del suo control flow graph, ovvero in questi gadget, questo perchè anche il Kernel presenta dei bug. Possiamo trovare tutto? Tendenzialmente si, in quanto il Kernel è Turing-Completo. Non so le istruzioni dove sono (memoria randomizzata), però magari conosco la distanza tra le istruzioni.

### Return Oriented Programming (ROP)

Prevede che, anziché manomettere l'indirizzo di ritorno di un unico stack frame, ne vengano manomessi diversi, in modo tale che ciascun return address scritto dall'attaccante trasferisca il controllo a una particolare zona di codice che esegua determinate istruzioni prima di invocare una nuova return e saltare così alla zona di codice successiva. Ciascuno di questi pezzetti di codice preceduti da un'istruzione di return è detto **gadget** (si tratta di un termine che avevamo già incontrato mentre descrivevamo Spectre) e non è una funzione intera, bensì solo una frazione.

Di seguito è mostrato uno schema del ROP:



A peggiorare la situazione sta il fatto che i sistemi x86 e x86-64 non impongono un particolare allineamento delle istruzioni in memoria. Di conseguenza, una stessa zona di codice può essere usata in modi differenti tramite ROP. Tanto per fare un esempio, supponiamo di avere da qualche parte nell'address space un pezzettino di codice composto dai seguenti byte: **55 48 89 e5 b8 00 00 00 00 5d c3**

Qui è possibile leggere questo pezzettino di codice a partire dal primo byte oppure a partire dal terzo byte.

-> Se leggiamo a partire dal primo byte abbiamo:

```
push %rbp
mov %rsp, %rbp
mov $0x0, %rax
pop %rbp
ret
```

-> Se invece leggiamo a partire dal terzo byte abbiamo:

```
in $0xb8, %eax
add %al, (%rax)
add %al, (%rax)
pop %rbp
ret # una ret mi permette di annidare gadget
```

### Contromisure per il ROP:

1) **Return Address Protection (RAP):** ogni funzione viene compilata con un preambolo e una coda. Nel preambolo l'indirizzo di ritorno viene memorizzato in modo cifrato all'interno di rbx, che a sua volta viene salvato nello stack; come chiave di cifratura può essere sfruttato il valore di un qualunque registro non utilizzato dalla funzione. Nella coda, invece, il valore cifrato in rbx viene decriptato e confrontato col program counter posto sullo stack, e si esegue effettivamente l'istruzione di return solo se i due valori confrontati sono uguali. Il buffer overflow è solo sulla memoria, NON SUI REGISTRI.

Tuttavia, questa contromisura può essere bucata nel momento in cui l'attaccante legge il valore di rbx ed è in grado di individuare qual è il registro che ospita la chiave di cripitura. Inoltre, si tratta di un meccanismo che ha un costo non minimale, soprattutto per le funzioni più brevi.

2) **Shadow stack**: anche qui ogni funzione viene compilata con un preambolo e una coda. Nel preambolo l'indirizzo di ritorno viene memorizzato in una stack area a parte, detta **shadow stack area**. Nella coda l'indirizzo di ritorno viene riportato sullo stack principale.

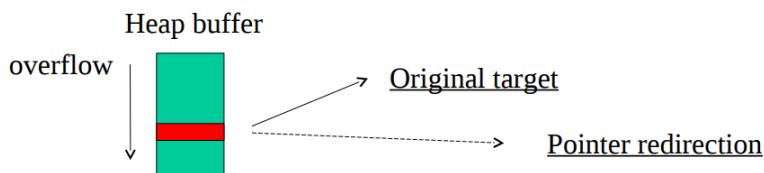
Neanche questa contromisura è particolarmente sicura: l'attaccante può comunque avere la possibilità di attaccare la shadow stack area. Inoltre, si ha sempre un costo in termini di cicli macchina aggiuntivi da eseguire per salvare il return address sulla shadow stack area e per installarlo poi sullo stack originale.

3) **Branch predictor**: nel momento in cui si chiama una return all'interno di una funzione, si utilizza un branch predictor di livello hardware per stabilire quali sono i punti del codice verso cui ci si aspetta di saltare.

Tuttavia, questa tecnica non copre i casi in cui si ha un cambiamento asincrono del control flow (e.g. quando deve essere gestito un segnale Unix, quest'ultimo restituisce il controllo al kernel con una return senza che il kernel abbia effettivamente invocato un'istruzione di call). Inoltre, sono richieste delle patch molto importanti alle system call affinché esse possano analizzare lo stato del predittore.

### Heap overflow

È un tipo di buffer overflow che non va a manomettere gli indirizzi di ritorno posti sullo stack, bensì i puntatori a funzione che si trovano nell'heap. Ciò porta a eseguire un gadget scelto dall'attaccante (o anche del codice iniettato dall'attaccante) nel momento in cui si invoca una *call R*, dove R è il registro che ospita il puntatore a funzione che era stato manomesso.



#### Contromisure per l'heap overflow:

1) **Check sul valore del function pointer**: prima di invocare la funzione puntata da un certo function pointer P posto nell'heap, si effettua un controllo sul valore di P: se questo appartiene a un determinato insieme di indirizzi legittimi allora va bene, altrimenti l'istruzione di call viene abortita.

Questa contromisura, analogamente a tutte le altre, porta a un aumento della complessità computazionale ogni volta che si utilizza un function pointer memorizzato nell'heap. Ogni app multi-thread il check e l'utilizzo di informazioni, se non atomici, saltano. Ciò avviene, magari mediante bug, dopo che ho controllato l'indirizzo (il quale è ok, ancora non modificato) e poco prima di saltarci (è qui che applico la modifica).

2) **Sanitizzazione della memoria**: ciascun buffer all'interno dell'heap viene incapsulato all'interno di una **surrounding area**. Ogni surrounding area contiene un tag read only che viene controllato ogni volta che si effettua un accesso al buffer: finché rimane immutato, gli accessi al buffer effettuati finora sono leciti. Si tratta comunque di una soluzione utile per le analisi offline ma non praticabile quando si lanciano effettivamente i programmi poiché impone che ogni singolo accesso all'heap coinvolga un'operazione di controllo, il che porta a un crollo inaccettabile delle prestazioni; la situazione si aggrava quando abbiamo un'esecuzione concorrente di più thread che devono sincronizzarsi tra loro: se un thread viene ritardato dalle operazioni di controllo dell'heap, eventuali thread che dipendono da lui vengono a loro volta penalizzati. Si usa il flag gcc `fsanitize=memory, in cui istruzioni come malloc vengono sostituite da istruzioni simili (magari una malloc ma con buffer agli estremi per protezione).

## Reale danno provocato dai buffer overflow

Nel più tipico dei casi un buffer overflow può causare dei danni legati al livello di privilegio dell'applicazione che si sta sfruttando. Se l'applicazione ha il SETUID-root (i.e. esegue nei panni dell'utente root, come ad esempio il programma sudo), allora l'attaccante può anche prendere il pieno controllo del sistema ed effettuare delle operazioni (ad esempio all'interno del file system) che non sarebbero previste in un utilizzo lecito del sistema.

## User ID in Unix

All'interno del kernel, lo username degli utenti è soltanto un placeholder. L'informazione che identifica veramente qual è lo user per conto del quale un programma sta girando è lo **UID (user ID)**; analogamente, abbiamo il **GID (group ID)** per identificare i gruppi di utenti.

Ciascun thread, in qualunque istante di tempo, è associato a tre UID/GID:

- Real: è l'utente per conto del quale il thread fa le sue attività.
- Effective: è l'utente di cui il thread ha le capability correnti; ad esempio, se l'utente real è 3 e l'utente effective è 0 (root), vuol dire che il thread attualmente può operare nel sistema come root.
- Saved: è l'utente che viene memorizzato nel momento in cui si cambia l'effective; in particolare, il vecchio effective diventa saved in modo tale che sia possibile ripristinare l'effective in un secondo momento.

Chiaramente, se si modifica l'effective per due volte di seguito senza mai ripristinarlo, il primissimo valore dell'effective andrà perso (e non sarà dunque possibile recuperarlo).

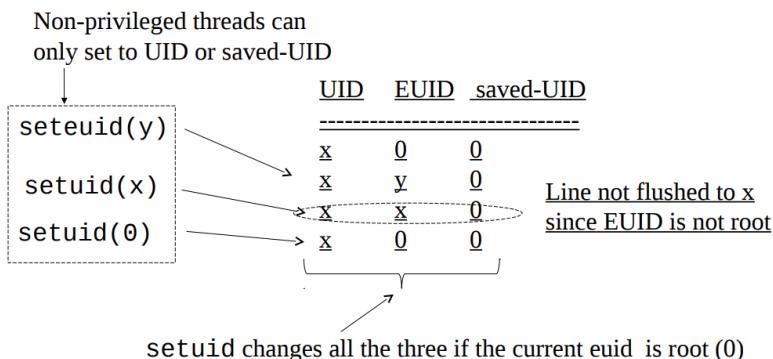
### System call per la gestione di UID/GID:

- > **setuid()** / **seteuid()**: operazioni di set.
- > **getuid()** / **geteuid()**: operazioni di get.
- > Si hanno delle system call analoghe per il GID.

Seteuid() è un'operazione reversibile poiché si limita a modificare l'EUID (effective user ID) memorizzando il vecchio effective in saved UID.

Setuid(), invece, se viene invocato da un thread con EUID diverso da zero, allora ha solo la possibilità di impostare l'EUID o al valore di real UID o al valore di saved UID; altrimenti, si tratta di un'operazione irreversibile poiché sovrascrive tutti e tre gli UID (real, effective e saved) associati al thread chiamante.

### Esempio:



**NB:** mediante setuid() / seteuid() è anche possibile prendere i panni di un utente che non è effettivamente registrato all'interno di /etc/passwd. Se EUID=0 (quindi root) possiamo chiamare seuid(x)

### I comandi su e sudo:

Sono entrambi setuid-root e, quando vengono utilizzati, richiedono all'utente di inserire una password. In particolare:

- *Sudo* imposta il real UID a root per poi eseguire a sua volta un ulteriore comando target.
- *Su* imposta il real UID a un qualunque user target specificato dall'utente.

In definitiva, per passare a eseguire un qualsiasi comando come root, è prima necessario passare per un programma che sia SETUID-root.

### Linux capability

Le capability introducono un terzo modo di operare che si interpone tra root e non-root. Di conseguenza, se un qualche thread ha bisogno di fare qualcosa che non è ammesso per i non-root, non deve per forza diventare un thread root.

Le capability possono anche essere viste come un approccio per costruire dei **protection domain (domini di protezione)**, grazie ai quali è possibile stabilire quali permessi assegnare ai vari thread.

Root thread (ID = 0)	Non-root thread with capability (ID != 0)	Non-root thread (ID != 0)
All kernel level security checks are bypassed	Some kernel level security checks are bypassed	All kernel level security checks are executed

L'esecuzione come user root porta ad avere tutte le capability possibili. Il flag **SECBIT\_KEEP\_CAPS** determina se queste capability vengono mantenute tutte anche a seguito di una chiamata a setuid(). Tale flag può essere configurato con l'aiuto della system call **prctl()**. Oggi ce ne sono 73, ma il problema è che difficile gestirle all'unisono.

#### Capability mask:

Esiste una maschera di 32 o 64 bit che viene usata per determinare se un thread ha una qualche capability. In particolare, si può tenere traccia di:

- **Permitted capability:** indicano cosa possiamo fare col real UID che abbiamo.
- **Effective capability:** indicano cosa possiamo fare correntemente con l'effective UID che abbiamo.
- **Inheritable capability:** sono le capability che vengono lasciate a un eventuale thread che viene creato con una exec.
- **Bounding capability:** sono tutte e sole le capability che hanno la possibilità di appartenere all'insieme delle permitted capability e all'insieme delle inheritable capability.
- **Ambient capability:** indicano cosa possiamo fare coi programmi non-SUID (dove i programmi SUID sono quelli che consentono di ottenere temporaneamente i permessi propri di uno user diverso da quello corrente).

#### API per le capability:

- > **int capget (cap\_user\_header\_t hdrp, cap\_user\_data\_t datap):** restituisce le capability di un determinato processo.
- > **int capset (cap\_user\_header\_t hdrp, const cap\_user\_data\_t datap):** imposta le capability per un determinato processo.
- > **cap\_t cap\_get\_file (const char \*path\_p):** restituisce le capability del file di cui viene specificato il pathname.
- > **cap\_t cap\_get\_fd (int fd):** restituisce le capability del file di cui viene specificato il file descriptor.
- > **int cap\_set\_file (const char \*path\_p, cap\_t cap\_p):** imposta le capability per il file di cui viene specificato il pathname.
- > **int cap\_set\_fd (int fd, cap\_t cap\_p):** imposta le capability per il file di cui viene specificato il file descriptor.

NB<sub>1</sub>: le ultime quattro API sono utilizzabili solo se il file system in cui stiamo lavorando è stato montato senza l'opzione NOSUID.

NB<sub>2</sub>: le ultime due API richiedono di avere la capability CAP\_SETFCAP attivata affinché sia possibile impostare le capability per i file.

### Sistemi operativi sicuri (orientati alla sicurezza)

Ci poniamo il seguente problema: come facciamo a evitare che un thread con EUID = 0 esegua delle porzioni di codice per lui di base non legittime? Ad esempio, il fatto che un thread qualsiasi possa fare qualunque cosa all'interno del sistema solo passando in modalità kernel e acquisendo momentaneamente i privilegi di root rappresenta un problema da non trascurare.

La soluzione consiste nel fare in modo che il root venga visto come un utente regolare dal punto di vista del sistema. Ciò implica che deve esistere un ulteriore user diverso da root che si occupi di amministrare la sicurezza all'interno del sistema.

Ora, un sistema operativo sicuro differisce da uno convenzionale poiché permette di specificare le regole di accesso alle risorse con una granularità più fine (dove la granularità più fine è data dai **protection domain**). Così un attaccante ha minori possibilità di far danno, ad esempio, in termini di accesso o manipolazione dei dati.

### Protection domain

È un insieme di tuple di tipo **<risorsa, modalità di accesso>**.

Se una risorsa R non appartiene ad alcuna tupla del protection domain associato a un particolare user o programma (o programma eseguito per conto di uno specifico user), allora non può essere in alcun modo acceduta da quello user o programma.

La filosofia che sta dietro ai protection domain è quella di fornire sempre il **livello di privilegio minimo** possibile.

Qualche volta il protection domain è associato ai singoli processi, piuttosto che agli user e/o ai programmi. In tal caso l'associazione può anche variare dinamicamente: ad esempio, è possibile avere un processo P con un protection domain D iniziale che, quando accede a una risorsa R, vede il suo protection domain passare da D a D' (dove tipicamente viene rimosso il permesso di accesso alla risorsa R), e così via. Di conseguenza, istanze differenti dello stesso programma possono essere associate a protection domain diversi.

Un'ulteriore possibilità è quella di associare un protection domain alla tripletta (**user, programma, orario**), in modo tale che un certo programma che gira per conto di un certo user possa accedere a determinate risorse solo in alcune fasce orarie.

### Security policy

I protection domain da soli non sono sufficienti per rendere sicuro un sistema operativo. In particolare, abbiamo bisogno delle **security policy**, che regolamentano la gestione dei protection domain e possono essere di due tipi:

-> **Discretionary**: sono le security policy tali per cui gli utenti ordinari (incluso il root) hanno la possibilità di definire i protection domain. Thread esegue con targa associata (es: euid) a specifico utente, ci permette di riassociare capabilities.

-> **Mandatory**: sono le security policy tali per cui la definizione dei protection domain è demandata esclusivamente a un amministratore di sicurezza (che non corrisponde all'utente root). Thread con targa associata all'utente non può fare modifiche ai domini di protezione suoi e degli altri. Un thread attaccato non può quindi attuare alcun tipo di notifica. Però, senza domini, non c'è controllo a grana fine, servono entrambi!

Sono necessarie delle politiche di sicurezza mandatory per avere un sistema operativo sicuro.

I sistemi operativi convenzionali offrono delle politiche di sicurezza discretionary e permettono all'utente root di avere il pieno controllo del sistema.

D'altra parte, i sistemi operativi sicuri richiedono la definizione di politiche di sicurezza mandatory e sono tali per cui l'amministratore di sicurezza decide cosa può fare ciascun utente (incluso il root) e cosa no.

Un esempio di sistema operativo sicuro nel mondo Linux è **SELinux** (Secure Linux), composto da domini e mandatorietà, cioè l'amministratore non può lavorare sulla riconfigurazione dei domini. E quindi chi li gestisce? Serve un altro amministratore oltre al root user (col suo dominio), ovvero il **security administrator** (vista come entità esterna, non esistono thread con la targa associata a lui), e questo è alla base del reference monitor. La configurazione è esterna, e comunica col sistema interno in maniera autonoma. Un software di livello kernel che si sincronizza ACL con sistema esterno può essere eliminato? Solo montando un modulo, ma per farlo serve thread rootId, ma se tale thread non può fare questa operazione (poichè non considerata nella ACL), allora non può compiere tale operazione.

Ancora più sicuro? Esiste solo la compilazione a startup delle ACL, poi non si può più fare nulla.

### Reference monitor

È un sistema autonomo che viene definito dall'amministratore di sicurezza e definisce i protection domain (secondo una politica di sicurezza mandatory) mediante un **ACD (Access Control Database)**.

In particolare, quando viene invocata una system call, questa passa per il reference monitor, il quale consulta l'Access Control Database e, in base all'utente / al programma / al processo chiamante, stabilisce se l'esecuzione della system call deve essere accettata oppure rifiutata. C'è layer che intercetta attività thread attivi a livello kernel, possiamo accettare o meno tramite ACL definita con meccanismo mandatorio. La configurazione non è cambiabile, quindi non posso toccare le policy di sicurezza.

