

# APPUNTI DI SISTEMI OPERATIVI AVANZATI

Versione di Simone Festa, basata sulla versione di Arianna Quinci e Matteo Fanfarillo.

Lez.1 (25/09/23)

## Hardware insights

Andiamo nel dettaglio delle features dell'hardware che andremo a usare: sfruttiamo l'hardware per ottenere funzionalità software, si scende nel livello di astrazione.

Generalmente si pensa che lo stato del programma sia legato a quello dei vari componenti software, ma bisogna fare attenzione al fatto che le azioni di alcuni moduli software possono cambiare lo stato dell'hardware, dunque osservando lo stato dell'hardware si può capire cosa sta facendo un'applicazione, infatti ci sono diversi attacchi che sfruttano l'hardware.

Ci sono tre sorgenti di informazioni che tipicamente non vengono prese in considerazione:

1. Compiled decisions → a seconda di come il compilatore prende decisioni si possono verificare scenari differenti.
2. L'hardware introduce decisioni a runtime, ci sono attività di scheduling a runtime non deterministiche, che dipendono dal momento in cui vengono effettuate.
3. L'hardware può avere (o meno) alcune features.

Affidarsi a librerie scritte da altri fa il modo che il software realizzato non risulti efficiente, dal punto di vista ingegneristico non è una buona soluzione.

Vogliamo avere del software corretto, sicuro ed efficiente qualsiasi decisione venga presa a livello hardware e per averlo bisogna conoscere i meccanismi che regolano queste decisioni hardware.

Alcune funzionalità, come il multithreading, erano già presenti a livello kernel, anche se non erano sempre disponibili a livello utente.

L'algoritmo di Lamport (usato per serializzare l'esecuzione delle sezioni critiche) su x86 non funziona, perché la macchina ha un comportamento difforme rispetto al codice che si scrive.

*Esempio:*

*bakery.c* → se si fa il run dell'eseguibile resta in azione per un po', poi andando a leggere il file di output vediamo che alcuni valori sono ripetuti più volte, perché non c'è consistenza globale degli aggiornamenti da parte di tutti i threads, quindi il numero d'ordine può far entrare un thread in CS quando non dovrebbe.

Solitamente si pensa al modello di Von Neumann: una CPU, una memoria, un'astrazione di esecuzione di istruzioni: fetch-execute-store, un intervallo di tempo per processare la singola istruzione e un'immagine di memoria definita allo start up dell'istruzione.

Tutto ciò non è vero in un'architettura moderna. Ciò che succede nell'hardware in un sistema reale non segue il flusso codificato, ma si schedulano una serie di azioni che danno luogo allo stesso risultato che si avrebbe eseguendo il flusso codificato.

Le decisione prese dal processore sono in funzione del suo stato nel momento in cui deve prenderle.

Tipicamente su un processore moderno si lavora in parallelo: ci sono tanti threads attivi, ognuno su una CPU.

## ***Scheduling e parallelismo in un sistema moderno:***

Scheduling:

*Livello hardware* → riguarda tre aspetti: esecuzione di istruzioni appartenenti allo stesso flusso di programma, esecuzione speculativa (sono in esecuzione una serie di attività che potrebbero non essere corrette, come branch prediction) di istruzioni in parallelo e propagazione dei valori attraverso componenti di memoria.

**Livello software** → riguarda tre aspetti: definizione dei tempi di esecuzione dei thread sull'hardware, definizione dei tempi di esecuzione delle attività sull'hardware e supporti di sincronizzazione basati su software (sincronizzazione dei thread/task).

**Parallelismo:**

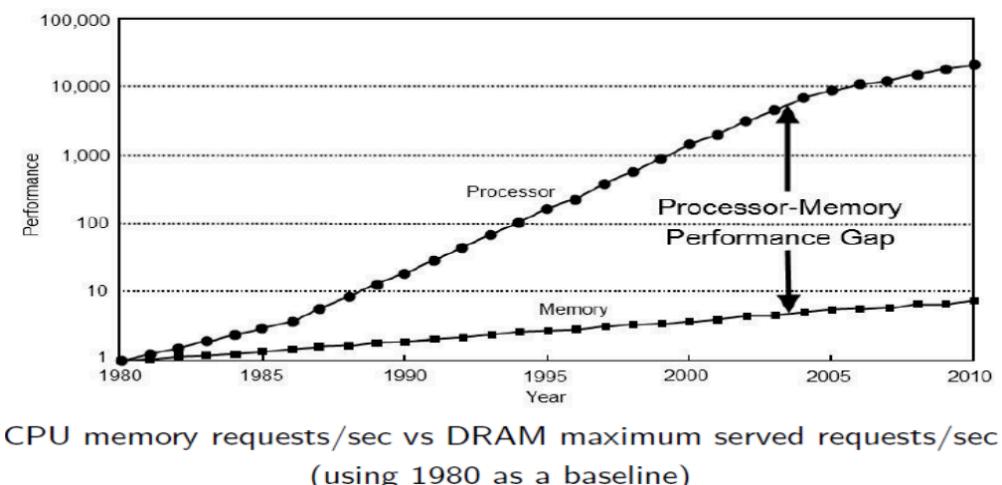
**Livello hardware** → instruction level parallelism (ILP): concerne il fatto che nello stesso istante di tempo il processore può eseguire più istruzioni in parallelo; si può fare anche su flussi multipli.

**Livello software** → thread level parallelism: si combinano flussi multipli a livello hardware, nell'hardware c'è parallelismo anche a livello del singolo flusso.

**Velocità di esecuzione:**

Per eseguire un'istruzione posso aver bisogno di più cicli di CPU, quindi la velocità del processore non ci fa discriminare cosa è veloce e cosa non lo è. La velocità dipende da cosa fa un'istruzione; ciò ci permette di introdurre il concetto di "applicazioni memory bound", la loro velocità dipende dal tempo di accesso alla memoria. Oltre alle applicazioni memory-bound possiamo avere anche singole istruzioni memory-bound. Altri tipi di applicazioni sono quelle CPU-bound e IO-bound.

Qui sotto possiamo vedere il grafico del gap tra la velocità con cui il processore inoltra le richieste alla memoria e quella con cui la memoria le serve.



**Pipeline:**

E' la tecnica hardware con cui si realizzano parallelismo e scheduling.

E' la modalità base con cui l'hardware può eseguire "overlapped processing".

Non c'è separazione temporale tra istruzioni; alcune istruzioni possono essere messe in attesa.

Ciò che segue un certo ordine in un eseguibile, non è detto che sia processato rispettando lo stesso ordine nell'hardware, in ogni caso lo scheduling preserva la causalità.

Questo è un data flow model, ciò che impatta è il flusso dei dati.

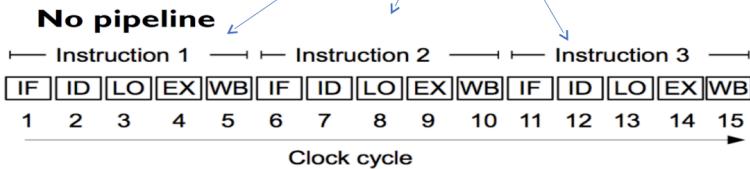
Gli stage di un'istruzione sono: fetch, decode, load, execute e write back.

- 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).

Qui sotto è riportato come sono eseguite queste fasi nel modello di Von Neumann:

- IF – Instruction Fetch
- ID – Instruction Decode
- LO – Load Operands
- EX – Execute
- WB – Write Back

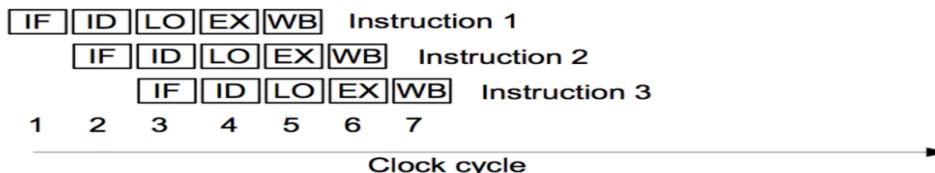
The different phases hopefully need to rely on different hardware components



### Stages nella pipeline:

L'utilizzo della pipeline, invece, permette di andare ad avere una velocità di esecuzione di istruzioni più vicina a quella del processore, perché idealmente si potrebbe processare un'istruzione per ogni ciclo di clock. Il fatto che un'istruzione risulti memory-bound è legato alle fasi di *load* e *write back*.

### **Pipeline**



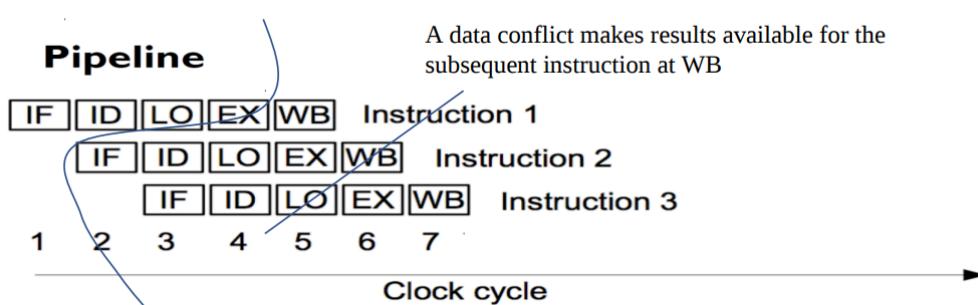
Attenzione: qui consideriamo che per ogni fase si impieghi un ciclo di clock, ma non è sempre così.

Idealmente lo speedup aumenta all'aumentare del numero di stages della pipeline, ma in un processore reale lo speedup a un certo punto è limitato: non posso avere un totale parallelismo se le operazioni hanno interdipendenze.

Dipendenze:

- *data dependency*: un'istruzione aggiorna un registro, che deve, poi, essere letto da un'altra istruzione.
- *control dependency*: l'istruzione successiva al salto viene stabilita dopo la sua esecuzione, non si può stabilire a priori.

A conditional branch leads to identify the subsequent instruction at its EX stage



Come risolvere questi problemi?

A livello software:

- stall (soluzione basica).
- software resequency, che consiste nell'allontanare istruzioni dipendenti nel flusso di esecuzione.  
Lavoriamo sul data flow a livello di compilazione.

A livello hardware:

- Propagazione hardware: mentre si scrivono valori li si propaga.
- Hardware supported hazards: deciso in maniera speculativa.  
Porta con sé delle implicazioni: se faccio qualcosa che non andava fatto lascio tracce che modificano lo stato del processore.
- Out Of Order pipeline: usata da tutti i processori moderni. Le istruzioni attraversano la pipeline in ordine differente, quindi un'istruzione può superarne un'altra. Gli effetti di un'istruzione non toccano i componenti hardware finché non si ha la certezza che l'istruzione dovesse essere eseguita.  
Un'istruzione può sorpassare tutte le istruzioni da cui non dipende.

Attenzione: La differenza tra i software resequency e il riordinamento hardware sta nel fatto che nel primo caso l'eseguibile generato ha un ordine differente delle istruzioni, nel secondo caso invece anche le istruzioni, che sono scritte in un certo ordine, vengono eseguite in ordine differente.

Lez.2 (27/09/23)

**La pipeline del processore x86:**

La pipeline non si vede a livello ISA.

Si è passati nel 1999 da processori con 4 stadi a famiglie di processori a 5 stadi.

**Pipeline e sviluppo del software:**

La pipeline, anche se out of order, attraversa sempre gli stadi di un'istruzione in ordine.

Se abbiamo istruzioni tra loro dipendenti non possono superarsi, ciò ha fatto ragionare in termini di sviluppo del software.

Chi scriveva software di basso livello si aspettava un miglioramento di performance che non sempre c'era.

**Esempio:**

*backward-propagation.c*: se si esegue il test delle performance risulta che il post incremento produce un tempo significativamente inferiore rispetto al pre incremento.

Abbiamo delle istruzioni che generano effetti devastanti sulle performance dei programmi, le istruzioni serializzanti, come *cpuid*.

Questo tipo di istruzioni, quando vanno in esercizio sulla pipeline, mandano in *squash* tutto ciò che è nella pipeline (quindi tutto ciò che è successivo rispetto a loro viene rimosso/flushato), quindi la pipeline si ferma (resta in stall) finché l'operazione non è completata.

Riportano la pipeline a lavorare secondo uno schema sequenziale, perdiamo la potenza computazionale della pipeline. *cpuid* ritorna le informazioni riguardanti la nostra cpu, tra cui il suo identificatore.

Un effetto interessante di un'istruzione serializzante è che garantisce che qualunque modifica apportata a flag, registri e memoria da parte delle istruzioni *precedenti* a lei sia finalizzata prima che una qualsiasi istruzione a lei successiva venga fetchata ed eseguita. E' come mettere un punto.

C'è un'altra classe di istruzioni serializzanti più leggere, che vedremo in seguito.

Il software del kernel si chiede spesso su che cpu sta girando. A livello user non ci interessa, ma a livello kernel, che opera a livello più basso, si può decidere su quale zona di memoria far operare una certa cpu.

Se il kernel usasse davvero cpuid avremmo grandi problemi di performance.  
 Questa informazione viene data a una cpu in maniera differente, vedremo in seguito.

### Pipeline superscalare:

In processori Intel Pentium Pro (1995) si hanno due pipeline che lavorano in parallelo.

Dato che alcune istruzioni sono lente le ALU sono multiple. Si usa out of order basato sull'algoritmo di Tomasulo.

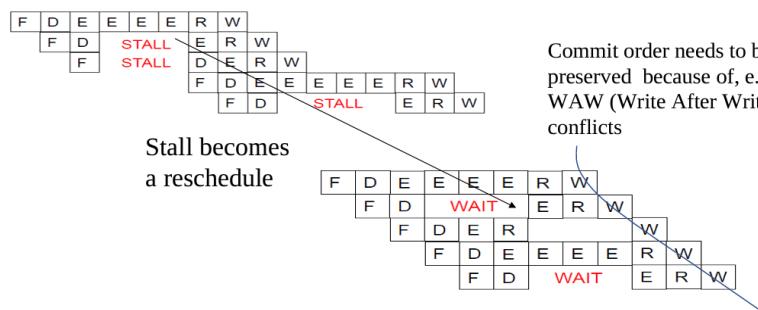
Come funziona un' out of order pipeline?

- Le istruzioni vengono mandate in commit secondo l'ordine vero e proprio del programma, quando un'istruzione termina va "ritirato l'ordine"; questo significa che i *commit* (o *retire*) seguono l'ordine di ingresso delle istruzioni in pipeline, indipendentemente dal verificarsi di eventuali sorpassi in pipeline.
- Si fa il processamento di istruzioni indipendenti "as soon as possible".
- E' efficiente con ridondanza hardware ("ho più corsie").

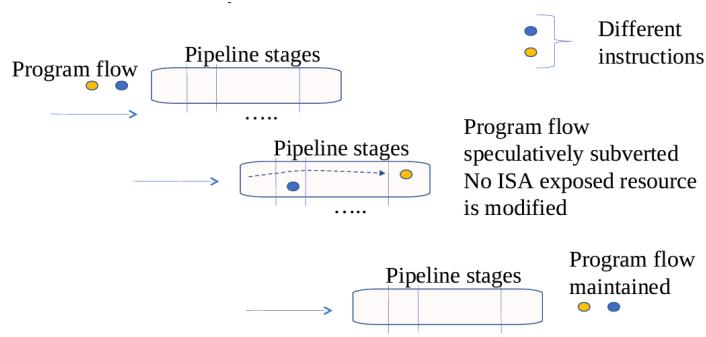
A seconda delle operazioni il numero di cicli macchina da attraversare può risultare maggiore o minore. Latenze elevate possono rallentare chi è più veloce. Nella O.O.O, se posso andare avanti, lo faccio, non mi importa che devo aspettare il commit, se posso eseguire qualcosa lo eseguo. Per questo parliamo 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. Chi impiega meno cicli macchina libera anche più velocemente la pipeline, in una pipeline NON O.O.O. lo stallo impatta tutti, in una O.O.O. non è così. Nel caso in cui ho istruzioni veloci che sorpassano istruzioni lente in cui si decide se eseguire o meno un salto si fanno azioni puramente speculative.

Qui sotto si riporta la differenza tra una pipeline non O.O.O e una O.O.O:

Nel primo schema, la seconda istruzione con stallo eseguirebbe per un solo ciclo "execute", ma deve essere in attesa dei dati dall'istruzione sopra. Con lo stallo "ritardo tutto". Sotto, in ambito OOO, la stessa istruzione aspetta comunque i dati della prima, ma la terza può eseguire senza problemi se non aspetta quei dati, l'unico "vincolo" è che la write la potrà fare solo dopo che le istruzioni precedenti l'hanno fatta.



Esempio di esecuzione in O.O.O Pipeline:



### ***Imprecise exception:***

Un'istruzione attraversa la pipeline, a un certo punto si ha un'esecuzione "offending" (ad esempio si genera un'eccezione), ma qualche altra istruzione può averla superata e può aver generato cambiamenti nell'hardware ("sporcano" lo stato micro-architetturale, il quale non può essere ripristinato, lasciando degli effetti dell'esecuzione speculativa, ma non modificano l'ISA).

Questo è un veicolo per attacchi che vanno a osservare questo tipo di cambiamenti per risalire ad altre informazioni.

Un esempio di attacco di questo tipo è Meltdown.

### ***Algoritmo di Tomasulo:***

Anche se siamo speculativi e O.O.O. si deve garantire che vengano rispettate alcune proprietà: RAW, WAW e WAR.

Se il flusso è A→B allora si ha:

- RAW: B non deve leggere qualcosa che A non ha ancora scritto.
- WAW: B scrive un dato e anche A, il dato può essere scritto da B solo dopo che è stato scritto da A.
- WAR: B scrive il dato solo dopo che A lo ha letto.

Per rispettare questi vincoli entra in gioco l'algoritmo di Tomasulo.

Per RAW bisogna identificare la dipendenza tra B e A e questa identificazione deve comportare un'attesa.

Per WAW e WAR, quando si scrive nella pipeline non si sta in realtà scrivendo in un registro, quindi due scritture avvengono in due locazioni differenti. Queste locazioni come si relazionano con il registro effettivo?

Si utilizza il ***register renaming***: esistono " $n$ " nomi dello stesso registro hardware, dunque " $n$ " locazioni diverse. Nell'ISA non si vedono le diverse stanze del registro, servono all'hardware.

Per generare un nuovo valore si utilizza la prima istanza libera e si permette di leggere dall'ultimo tag, quindi l'hardware deve tener traccia dell'ultimo tag consegnato, esso viene mantenuto tra i metadati di ciascun istanza ***renamed register***.

Il tag viene assegnato rispettando l'ordine con cui le istruzioni entrano in pipeline, quindi se si ha flusso A→B, B non può avere tag inferiore ad A.

***Esempio:*** Se A scrive in TAG 0, e B scrive in TAG 1, leggo da TAG1.

Il limite in questa soluzione sta nel fatto che il numero di registri è limitato, ma comunque si possono riutilizzare perché il numero di istruzioni è limitato.

Tutto ciò che viene fatto in questa esecuzione è speculativo, tranne un unico tag che viene committato.

***Reservation station:*** è una coda associata a una ALU, rappresenta una corsia.

Un'istruzione accodata possiede un tag che dice cosa sta attendendo.

Nell'ambito della RS non c'è il concetto di sorpasso, esso esiste solo tra ALU diverse.

La velocità di un'istruzione non si può sempre determinare a priori, perché dipende dallo stato dell'hardware a un certo istante.

Nelle architetture che utilizzano l'algoritmo di Tomasulo ci sono altre due componenti:

- ***CDB***, common data bus, mette in comunicazione le reservation station col ROB.
- ***ROB***, reordered buffer, serve per riordinare le istruzioni, infatti le istruzioni vanno ritirate nello stesso ordine con cui sono entrate in pipeline. Lo fa mantenendo i metadati delle istruzioni fetchate (come l'ordine per il commit e gli alias). Permette di mandare istruzioni in esecuzione in parallelo.

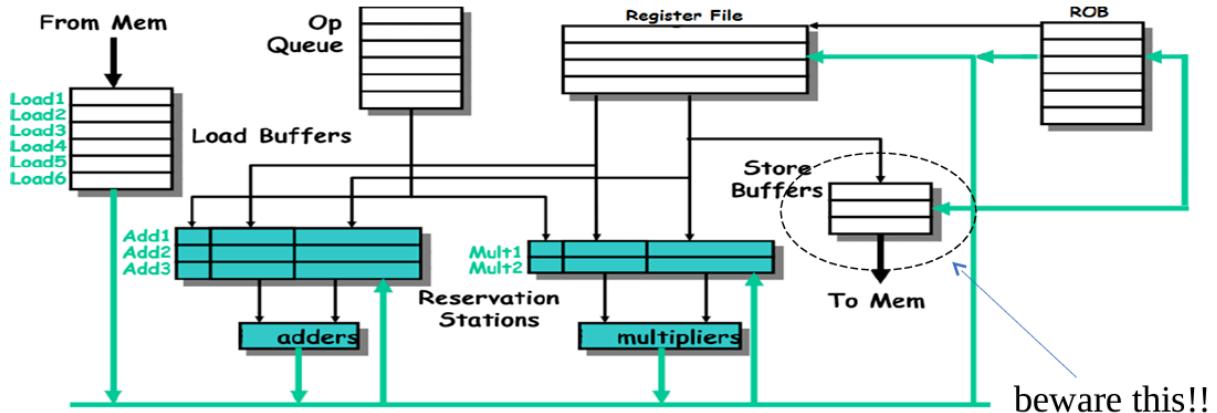
Se l'istruzione committata prevede una scrittura in memoria, il valore di output, prima di essere riportato in memoria, viene inserito nello ***store buffer***.

Quanto tempo il dato resta nello store buffer? Dopo il commit in realtà c'è un altro intervallo temporale durante il quale il dato non viene ancora scritto in memoria. Possiamo avere problemi di consistenza qualora un altro thread voglia leggere il dato, in particolare se gira su un'altra cpu.

Questo è proprio il problema che abbiamo evidenziato nell'algoritmo di Lamport.

Nei processori moderni tutte le istruzioni usano ROB.

Qui sotto è riportata l'architettura di riferimento di Tomasulo:



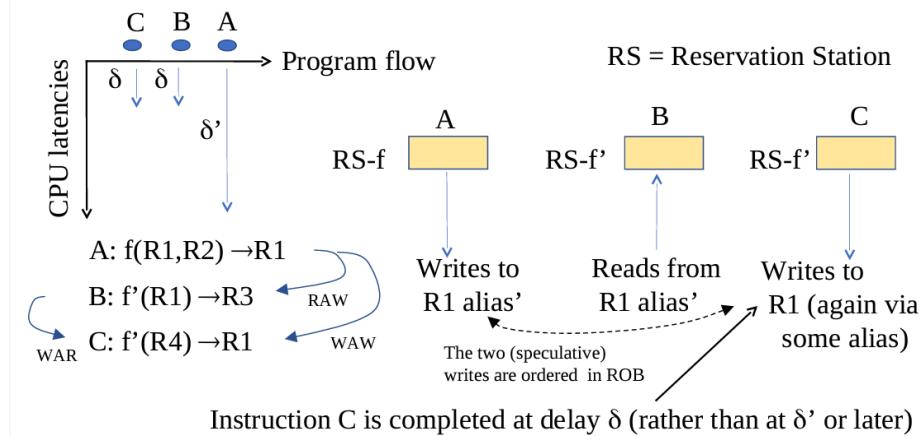
### Problema del memory wall:

Le operazioni che possono eseguire in parallelo devono caricare dati dalla memoria, ma se le attività da svolgere in parallelo superano la velocità con cui prelevo dati non sfrutto tutte le corsie. Il processore non arriva mai a processare in parallelo tutte le istruzioni possibili a causa dei ritardi causati dalla memoria, per cui la potenza e le risorse del processore risultano sprecate.

### Esempio di esecuzione:

$f(\text{registro input}) \rightarrow \text{registro alias output}$ .

B deve leggere dallo stesso alias di A, C può usarne un altro, allora C è processabile in parallelo rispetto A. Anche se C genera risultati prima di A, non viola nulla, basta che B legga dall'alias di A, e C possa mandarlo in commit dopo A e B (ROB lo ordina). Il vantaggio è che ho ridotto il tempo per eseguire C, in quanto non aspetta.



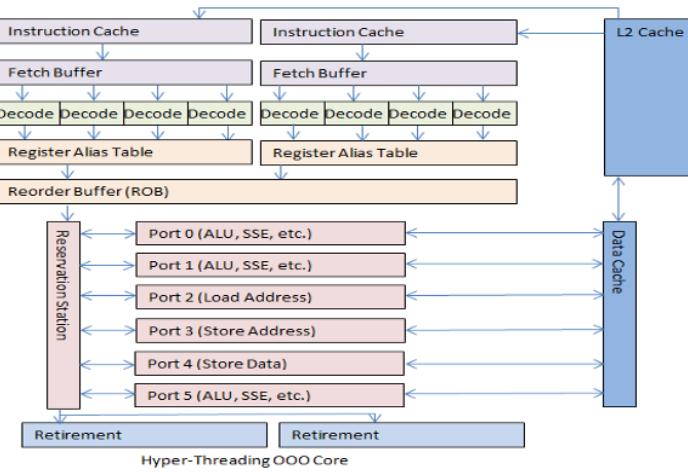
### **Processori hyperthreaded:**

Sfruttano meglio l'architettura, soluzione al *memory wall*. Non si processa un singolo flusso di istruzioni.

Flussi di programmi diversi accedono allo stesso *core fisico* (*hardware engine*, ne è esempio la reservation station), se abbiamo più flussi riusciamo a riempire le corsie. Il core (una CPU può averne vari) è un'unità di esecuzione.

La porzione di processore che comprende fetch, decode, tiene traccia dei registri logici dell'ISA (mediante Register Alias Table) viene replicata.

Ogni cpu ha i suoi register alias, per gestire il proprio flusso. Flussi diversi hanno pochi conflitti.



Il core è tutto ciò che sta sotto il ROB, il thread ciò che sta sopra. Entrambi questi flussi sono speculativi, ciò può portare a problemi di sicurezza.

### Lez.3 (28/09/23)

#### Interrupts:

Quando si fa il “*retire*” di un’istruzione in pipeline si controlla se c’è stato un *interrupt* (proveniente da un dispositivo hardware che indica la necessità di cambiare il flusso di esecuzione) e nel caso si fa il *flush* di tutte le istruzioni nella pipeline (squash, svuoto); ciò implica che gli interrupt portano con sé grandi penalità per le pipeline O.O.O. in termini di performance.

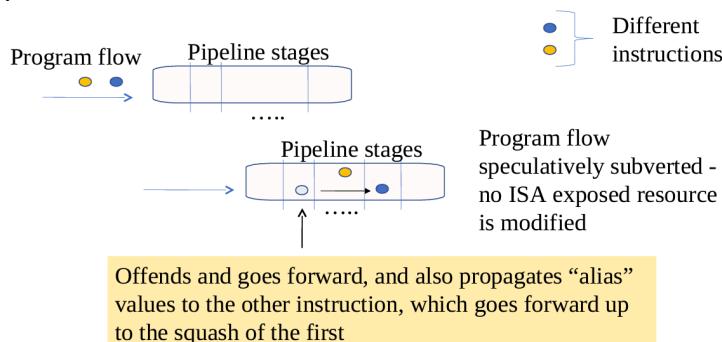
I tipi di eccezioni che possono verificarsi in ciascuno stage della pipeline sono:

- Instruction Fetch & Memory stages:
  - Page fault su instruction/data fetch: accesso a indirizzo logico diverso da quello fisico.
  - Accesso in memoria disallineato.
  - Violazione delle protezioni applicate a una pagina (eg: scrivo su read-only)
- Instruction Decode stage
  - Undefined/illegal opcode
- Execution stage
  - Arithmetic exception
- Write-Back stage
  - No exceptions

Se un’istruzione offending non arriva alla fine della pipeline e arriva un interrupt, è come se essa non fosse mai avvenuta: facendo lo squash non si deve gestire la priorità tra interrupt ed eccezioni provenienti da più istruzioni in pipeline.

Quindi quando viene generata un’eccezione non viene immediatamente gestita, l’interrupt viene generato solo nel retire stage (bit settato ad 1 e si chiama il gestore); l’istruzione quando solleva un’eccezione viene marcata come offending. Le istruzioni successive diventano *phantom*.

Il vantaggio di questo comportamento è che utilizzo meno l’hardware.

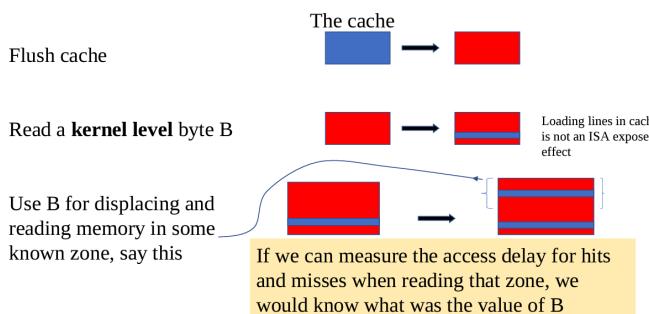


Il problema è che questo comportamento porta con sé dei side effects a livello microarchitetturale. L'istruzione continua a viaggiare nella pipeline per un certo tempo, se scrive su un renamed register possiamo leggere ciò che ha scritto, quindi la pipeline speculativa porta a eseguire sull'hardware azioni illecite. Questo può portare a degli attacchi di tipo **cache side channel**, in cui, partendo dalla cache in uno stato noto, si effettuano operazioni atte a cambiare lo stato della cache (faccio salire in cache un certo dato rilevante in maniera speculativa) che altera lo stato micro-architetturale (non esposto in ISA). Infine, tramite *timing*, misurando i tempi di accesso, riesco a capire qual è la entry acceduta speculativamente. Il tutto mediante un canale laterale. Ciò è alla base dell'attacco *Meltdown*.

### **Meltdown:**

Ciascun processo ha un *address space* suddiviso in una zona per l'esecuzione user mode e una zona di memoria per l'esecuzione kernel mode.

Lo scopo dell'attacco è accedere a un'area di memoria situata nel kernel anche se non si hanno i permessi per leggere informazioni sensibili.



Anzitutto si definisce un array A nella memoria user space e si svuota la cache tramite l'istruzione *clflush*. Dopodiché, mediante una *mov*, si accede a un particolare *byte x* (ad esempio può rappresentare un carattere, supponiamo di conoscerne l'indirizzo) situato nel kernel e si memorizza il byte in un registro; naturalmente questa *mov* è un'istruzione offending. Una volta ottenuto *x*, 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-architetturale da parte dell'istruzione successiva a quella offending che, chiaramente, viene eseguita solo speculativamente. A tal punto, poiché l'array A si trova in user space, è possibile accedere a tutte le sue *entry* e, per ogni *entry*, cronometrare il tempo necessario per l'accesso: la *entry* relativa al tempo di accesso minore sarà quella a spiazzamento *x*, perché si tratta dell'unica *entry* il cui contenuto era stato memorizzato in cache.

Ora è noto lo **spiazzamento x**, che era proprio il byte di memoria prelevato inizialmente dal kernel space. (Quindi è dallo spiazzamento che ottengo l'info, non tanto con il dato all'interno dell'array di indice x).

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 attack** (o covert-channel), ovvero con un attacco che fa uso di un canale laterale per andare a rubare delle informazioni.

Meltdown può essere esteso anche al caso in cui si voglia scoprire un'intera stringa all'interno del kernel space, che può essere una password, una chiave segreta e così via. Comunque sia, poiché ciascun byte che viene letto dal kernel space può assumere 256 valori possibili, poiché i caratteri ASCII sono 256, l'*array A* dovrà essere composto da 256 *entries differenti*.

Scopro un segreto byte a byte ripetendo l'attacco su ogni char.

**Attenzione:** per far sì che l'attacco abbia effetto prima di tutto devo inizializzare l'array A, poiché altrimenti le sue entries non vengono materializzate in memoria.

Si tratta di un **attacco statistico**: in realtà, i tempi di accesso alla cache e alla memoria RAM non sono deterministici; uno stesso accesso talvolta può risultare più lento, talvolta può essere più veloce, dipendentemente dallo scheduling delle istruzioni a livello hardware. Il successo dell'attacco è funzione anche di quanto differisce il tempo di accesso al valore memorizzato in cache da tutti i tempi di accesso agli altri valori che invece devono essere recuperati direttamente dalla RAM.

L'attacco Meltdown si realizza in tre fasi:

- Flush della cache.
- Lettura di byte kerne.
- Usare il byte letto per spiazzarsi in memoria.
- Osservazione dello stato della cache per mezzo delle sue prestazioni; ciò non ha effetto sull'ISA perché la cache non è esposta sull'ISA.

Come si è risolto questo problema?

- Patch per non far accedere a un indirizzo kernel level.
- Se si sta facendo un accesso illecito si va avanti, ma non si passa il valore reale, ma uno di default, questo si realizza tramite una patch hardware nel processore: *early handling* a livello hardware.

### Differenze tra x86 e x86-64:

In x86 si hanno 8 registri general purpose e il program counter è a 32 bit.

In x86-64 si hanno 15 registri general purpose e il program counter è a 64 bit.

x86-64 è retro compatibile rispetto ad x86.

### Instruction architecture:

$$\text{Offset} = \left[ \begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right] + \left[ \begin{array}{c} \text{eax} \\ \text{ebx} \\ \text{ecx} \\ \text{edx} \\ \text{esp} \\ \text{ebp} \\ \text{esi} \\ \text{edi} \end{array} \right] * \left[ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \right] + \left[ \begin{array}{c} \text{None} \\ \text{8-bit} \\ \text{16-bit} \\ \text{32-bit} \end{array} \right]$$

Base                  Index                  scale                  displacement

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

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

*esp* = extended stack pointer (32 bit)

Il default se non si specifica la size è 64 bit.

*Base*, *indice*, *scale* e *spiazzamento* vengono combinati per realizzare un indirizzo complesso con cui accedere in memoria.

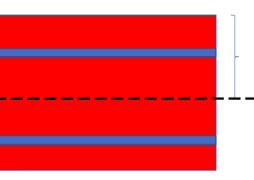
Le parentesi tonde rappresentano un accesso in memoria. Solo *spiazzamento* non le usa, in quanto può essere un indirizzo assoluto già noto, ed è "diretto". "*Base*" è relativo al valore di un pointer, "*index\*scale*" spiazzamento rispetto la "*base*". E' utile per accedere ad array di tipi diversi di dati, a seconda della scala mi spiazzo di un numero di bytes diverso all'interno dell'array.

### Codice Meltdown con sintassi Intel:

```

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



The target cache zone is an array of 256 pages – only the 0-th byte of the B-th page will experience a cache hit (under the assumption that concurrent actions are not using that cache zone)

Note: (riga 4) "al" è l'ultimo byte di "rax", "rax" è shiftato di 12 verso sinistra, ovvero 4096 byte, la dimensione di una pagina di memoria. Ai byte kernel si aggiungono 12 bit a zero per costruire un indice per il mio array probe. (Vedi foto, ad "1" si hanno i bit di "al"). Sto quindi leggendo **solo la prima entry di ogni pagina.** (perchè mi "muovo" di 4096 alla volta!)

Infine, in `rbx` ci metto il valore contenuto nella locazione di memoria identificata da `rbx+rax`, che non è il valore “B”, però verrà memorizzato con offset “B”. Quindi dall’offset capisco qual è il dato di interesse.

Se `rax==0` (riga 6), vuol dire che mi trovo davanti un'area non significativa (es: non valida), allora ritento. Dopo `jz retry` non è detto che in `%rax` ritrovi lo stesso valore, perché la memoria viene acceduta concorrentemente.

Vogliamo che l'array abbia 256 pagine da 4096 byte. Questo ci assicura che gli accessi all'array abbiano una larga distanza spaziale l'uno con l'altro (cambiano i bit significativi che determinano l'indice di set nella cache). Questo previene il prefetcher hardware dal caricare locazioni di memoria adiacenti nella cache. La cache lavora a blocchi. *Per ogni byte letto in cache ne salgono 64 byte. Se leggessi byte a byte leggerei cose rapide e quindi non saprei fare una stima. Potrebbero salire più linee quindi non posso leggere linea a linea.*

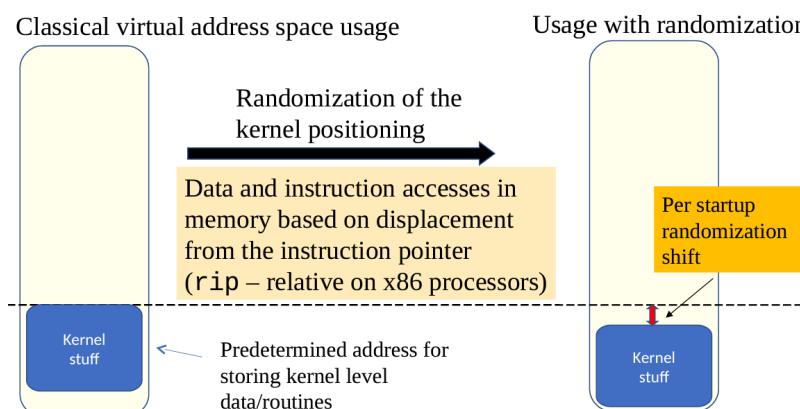
### *Note sull'attacco Meltdown in pratica:*

Riservo una zona di memoria kernel e creo uno pseudofile; chiedendo di leggere dallo pseudofile leggo da questa zona di memoria.

`cat /proc/cpuinfo` è uno pseudofile per vedere a quali bug è soggetta la cpu.

#### **Patch di sicurezza software:**

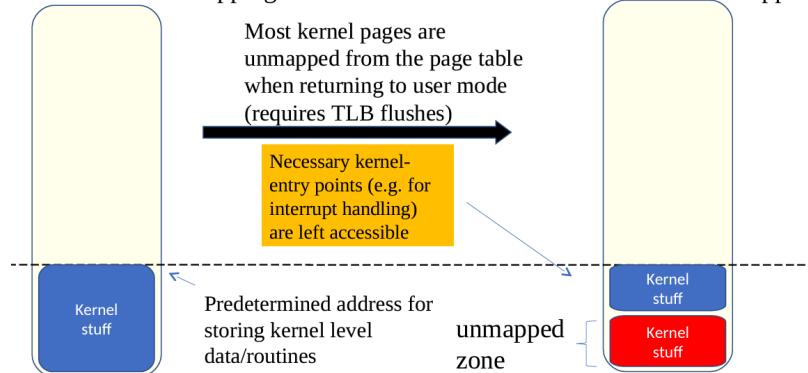
- **KASLR**: ogni volta la tabella delle pagine che si dà all'utente viene randomizzata (offset random rispetto all'indirizzo base del kernel); il kernel viene montato su pagine diverse dell'AS logico. Soffre il brute force. Questa soluzione si può applicare soltanto se il processore è *rip relative*. Il termine "processore RIP relative" si riferisce a un tipo di processore che utilizza l'indirizzamento relativo all'istruzione successiva (RIP, acronimo di "Relative Instruction Pointer") per accedere alla memoria anziché l'indirizzamento assoluto. In questo contesto, "RIP relative" indica che gli indirizzi di memoria sono calcolati rispetto all'indirizzo dell'istruzione successiva nell'assembly del programma in esecuzione.



- **KAISER**: non necessita di randomizzazione. Il kernel è diviso in due zone: una visibile e accessibile a user level, l'altra no. La tabella delle pagine dello user non sa dove queste pagine siano collocate in memoria fisica. Ogni applicazione attiva ha due tabelle delle pagine di primo livello, alcune pagine non sono accessibili in user mode, ma solo in kernel mode; ciò implica che lo stesso processo veda pagine diverse in user e kernel mode. Alcune pagine kernel sono visibili in user mode perché servono in caso di interrupt, bisogna avere alcune informazioni per cambiare la tabella delle pagine e passare in kernel mode, in particolare sono necessarie la parte early delle system call e della gestione degli interrupt. **Si è evoluto in PTI**.

Si può ricompilare il kernel per cambiare il boundary delle pagine non visibili.

Classical kernel mapping

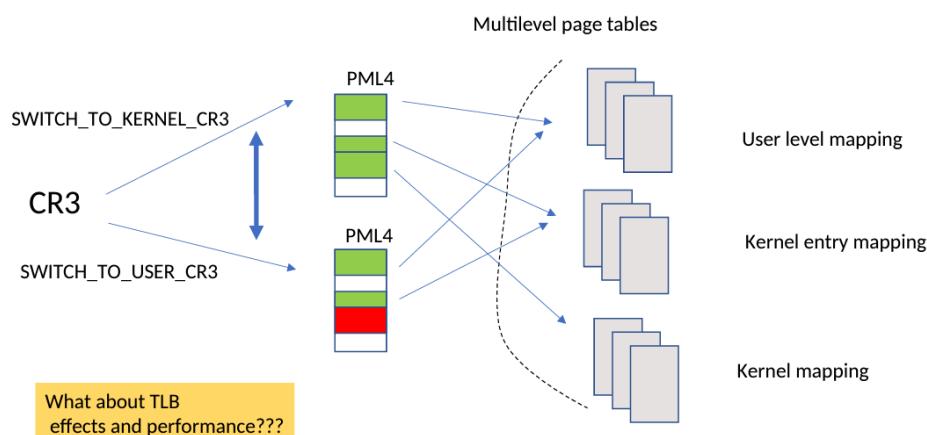


### Attuale implementazione: PTI

Questa tecnica è nota come **page table isolation**; per disabilitarla `pti=off`.

Per ogni pagina che serve dopo il cambio da kernel a user e viceversa si ha un *cache miss*, ogni volta butto il contenuto del TLB, Translation Lookaside Buffer.

Quando si scrive il registro selettore **CR3** si ha il reset del TLB, per semplicità a livello hardware questo avviene in automatico quando si cambia tabella. Meno costoso del *cache flush*, dove flusho a ogni switch.



### Lez.4 (02/10/23)

#### **Salvi sull'architettura pipeline:**

Quando si trova un branch non sappiamo se le istruzioni successive andranno o meno eseguite.

La predizione è funzione della tipologia di branch:

- Conditional branch (`if, else`) → due possibili destinazioni.
- Unconditional branch e call (`jump, call`) → una sola destinazione.
- Return e indirect branch (`ret, jmp eax`) → molteplici destinazioni.

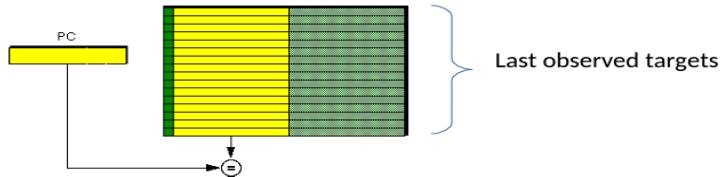
Nel salti incondizionati e nella `call` già nella fase di decode dell'istruzione si può risalire alla destinazione, per quanto riguarda `return` e `indirect branch` sono register dependent, ovvero dipendono dallo snapshot dei registri, e si può risalire alla destinazione in fase di esecuzione.

Solo nei salti condizionati non si sa se il salto dovrà essere fatto o meno, anch'essi sono register dependant e si conosce la destinazione solo in fase di esecuzione.

In un processore pipeline qual è il supporto per i salti?

Branch target buffer: è una cache in cui ciascuna entry riporta un indirizzo e un target da utilizzare quando il flusso di esecuzione passa per il rispettivo indirizzo. Il target indica l'indirizzo da raggiungere quando l'istruzione viene eseguita speculativamente, è un suggeritore.

Nell'hardware si inizia a lavorare in fetch a partire dall'indirizzo target. Non c'è un meccanismo che faccia capire se il target è affidabile, potrebbe essere semplicemente l'ultimo target raggiunto da un `jump`.



Branch prediction buffer: per ogni riga aggiungiamo uno status bit che indica se l'ultima volta che si è eseguito un branch la predizione è stata rispettata. La cache è indicizzata dai bit meno significativi dell'istruzione di salto e un bit di stato. Il recente passato è identificativo dell'imminente futuro.

E' un automa con due stati: Taken - Not Taken.

Tale approccio mal si sposa con cicli annidati:

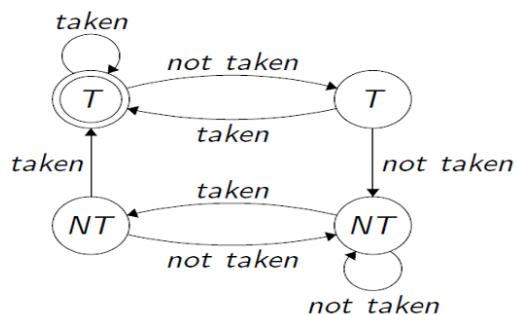
```
#1 for (i = 0; i < 10; i++) {
#2     for (j = 0; j < 10; j++) { ... }
```

Infatti, l'esecuzione passa da #1 a #2 (l'automa parte da Taken ma non saltiamo, 1° errore, allora l'automa cambia stato e dirà che anche dopo non si salta). Quando #2 finisce, c'è un salto per tornare a #1, quindi il predittore ha sbagliato di nuovo.

Preditori per salti condizionali:

Come si può cercare di migliorare l'esercizio nella pipeline considerando le costruzioni classiche del software?

Multiple bit predictor: non è sempre vero che ogni volta che passo per un'istruzione devo fare sempre la stessa predizione. Avere più stati permette di risolvere problemi dovuti alla presenza di cicli annidati: quando passo dal ciclo interno a quello esterno ho un salto al ciclo esterno e devo tornare a eseguire il ciclo interno per un certo numero di iterazioni. Non posso semplicemente fidarmi del comportamento adottato più di recente, ho bisogno di più bit per identificare lo stato.



Con 2 bit ho 4 stati possibili.

Evito un doppio errore: se sbaglio due volte prendo l'altra strada.

E' importante ridurre le predizioni sbagliate perché ogni volta che si verifica un errore nella speculazione la pipeline va in squash, dunque perdo diversi cicli macchina.

C'è bisogno di andare oltre i predittori a due bit?

Sì, perché nei casi sopra esposti non c'è una reale dipendenza dalle istruzioni, ovvero i due cicli lavorano su valori diversi. Potrei avere casi in cui si lavora su stesse variabili.

A motivating example

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

Not branching on these implies  
branching on the subsequent

Idea of correlated prediction: lets' try to predict  
what will happen at the third branch by looking at  
the history of what happened in previous branches

I salti condizionali rappresentano una grossa fetta di istruzioni nel codice, circa il 20%.

Sono stati introdotti due ulteriori tipi di predittori: *correlated 2-levels predictors* e *hybrid local/global predictors* (o tournament).

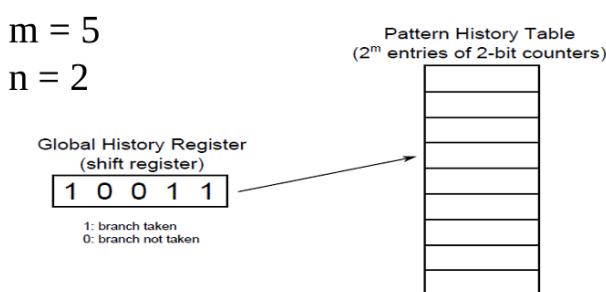
#### Correlated predictors:

Trovano correlazione tra presente e passato; FSM legata a più indirizzi logici.

Si tengono in conto gli ultimi  $m$  salti, dunque abbiamo  $m$  predittori a  $n$  bit.

La predizione attuale si ottiene da una bitmask ottenuta dall'outcome degli ultimi  $m$  salti.

$m$  indica il numero di salti precedenti,  $n$  è il numero di errori da commettere prima di cambiare la predizione.



Nello shift register viene inserito un 1 o uno 0 a seconda che il salto venga fatto o meno, quindi al commit dell'istruzione. Il contenuto dello shift register funge da indice per il PHT, il quale presenta nella singola entry lo "stato" dell'automa a  $n$  stati, assunto in funzione di tale indice (che altri non è che gli ultimi  $m$  salti presi e non).

#### Tournament predictors:

Offrono supporto per predizione locale e globale. Decidiamo se per un certo salto è meglio usare un oggetto locale o globale. Inserisco una FSM che indica quale predittore si vuole usare per un certo salto (se locale o globale). E' a 4 stati, sbaglio al massimo una volta.

Cosa significa globale? E' associato alla presenza di più elementi, non è legato alla presenza di più threads. Come si fa una previsione quando si passa su un'istruzione di salto? L'istruzione di salto è identificata tramite un offset nell'AS. Quando un thread passa su un'istruzione, per tale istruzione è riportato un indirizzo, dove troviamo le informazioni che dicono come ci si deve muovere per prendere una decisione; non si guarda tutto l'indirizzo, ma soltanto i bit meno significativi.

Questi supporti si trovano nel core del processore, ciò implica dire che quando si ha hyperthreading il supporto alla previsione per più threads è lo stesso, quindi se più threads utilizzano gli stessi indirizzi può succedere che siano influenzati dalle decisioni dell'altro.

Nel caso in cui io abbia a che fare con più cloni della stessa applicazione miglioro le prestazioni, però si ha un problema di sicurezza; si può fare in modo che un thread modifichi le informazioni in modo tale che un altro thread faccia esattamente ciò che vuole.

#### Return stack buffer:

E' il supporto per le istruzioni di return. E' una cache al cui interno viene inserito l'indirizzo dell'istruzione successiva alla call chiamata, nei processori moderni si hanno a disposizione 32 entries, in cui il replacement avviene secondo buffer circolare. C'è un return stack buffer per processo.

E' una cache nella CPU, non è esposta a livello dell'ISA, si può modificare solo eseguendo una call o una return.

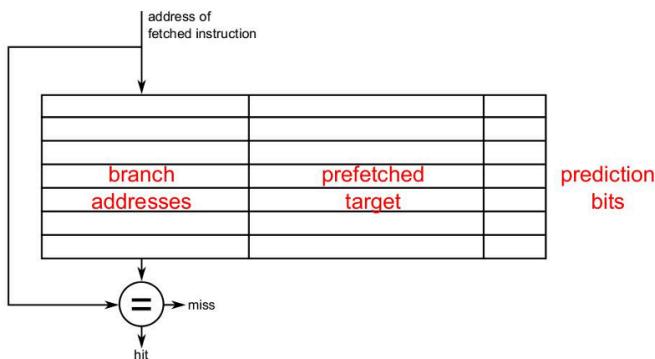
#### Supporto a indirect branches:

Rappresenta il supporto per indirect branches; i salti indiretti hanno diversi target cui possono saltare.

E' una cache in cui mantengo:

- *branch address*: ovvero l'indirizzo dell'istruzione di salto. Ad esempio, l'indirizzo di `jmp eax`. Posso mantenere i bit meno significativi per una questione di località.
- *prefetched target*: dove mi ha portato il salto *branch address* nelle esecuzioni precedenti. Questo è l'indirizzo più "probabile" a cui salterò se ripasso per quel *branch address*.
- *prediction bits*: mi dice se ho indovinato la predizione nelle volte precedenti. Ad esempio, se sbaglio due volte potrei lasciar perdere. Qui dentro posso infatti metterci una macchina a stati.

Nei momenti in cui c'è cache miss non si può prelevare il salto, quindi viene presa l'istruzione successiva per allenarlo.



#### Punto di vista della sicurezza:

Agendo in maniera speculativa è possibile avere dei side effects, ad esempio si può modificare lo stato della cache. *Tutto questo sta nel core*, e se ho hyperthreading, la predizione viene fatta da una comune componente hardware per più thread. Allora un thread1 potrebbe avere un comportamento tale da suggerire al thread2 di fare salti (anche speculativi) che in realtà non dovrebbe fare.

#### Attacchi Spectre:

Si basano sul fatto che i processori agendo speculativamente possono sbagliare.

Un processore può essere indotto a prevedere un salto in un thread in base alle decisioni prese da un altro thread. I processori moderni sono vulnerabili a Spectre.

Quando si fa lo squash della pipeline non si cancellano le informazioni riguardo le previsioni, altrimenti avrei problemi prestazionali. Per realizzare una patch hardware a Spectre dovrei eliminare il supporto hardware alla predizione.

## Spectre v1:

Usa istruzioni di salto condizionali. Il principio è lo stesso di Meltdown.

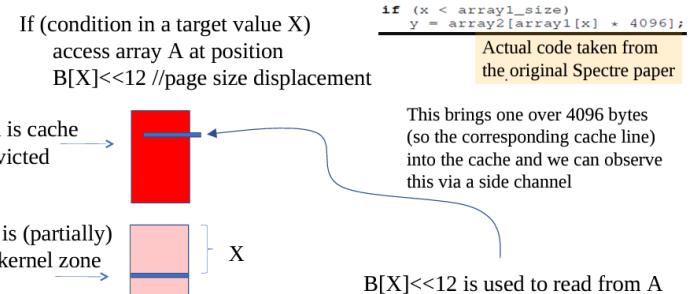
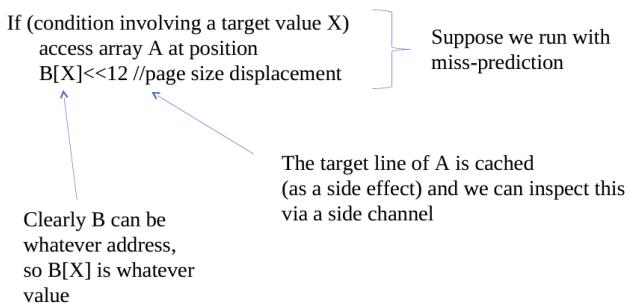
Devo flushare prima di iniziare, in modo da non avere `array1_size` in cache, portandomi a speculare.

B può essere qualsiasi indirizzo, e B[X] qualunque valore. L'idea è che, saltando molto lontano, posso accedere a delle aree kernel.

B[X] è il byte kernel che, shiftato di 12 (una pagina quindi), costituisce un indirizzo a cui posso accedere tramite il probe array A. Poi, confrontando i valori di accesso alle entry dell'array A, posso capire il valore di B[X]. Prima di eseguire l'if è necessario addestrare il predittore di salti a entrare nell'if (nel quale si controlla che x sia un valore "legittimo", ad esempio non eccede la taglia di un vettore). Successivamente, impostando x ad un valore "illegittimo" (eccede la taglia di un vettore), il branch predictor suggerirà comunque di entrarvi in modo *speculativo*, non reale. In questo caso la pipeline viene squashata e non viene nemmeno generata una trap in quanto si tratta di istruzioni a livello if.

(In meltdown faccio qualcosa di illegale in modo diretto, qui sfrutto la *miss-prediction del branch predictor!*) Questo fatto rende più complessa anche la gestione di contromisure in quanto **non si passa mai a kernel mode** e dunque non è possibile ad esempio fare un cambio di tabelle.

### Spectre primer



## Spectre v2:

Se si ha un processore con più hyperthreads si sfruttano gli *indirect branches*.

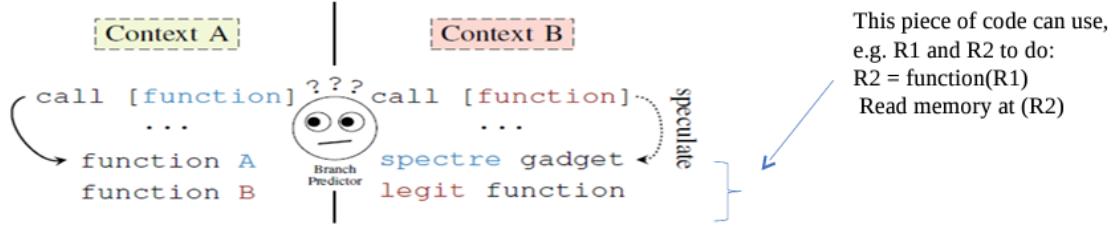
Il thread A può portare avanti delle attività che vanno a cambiare lo stato del branch predictor all'interno del CPU-core (training). Di conseguenza, B può osservare il nuovo stato del branch predictor all'interno del medesimo CPU-core e, quindi, le attività che lui eseguirà in modo speculativo sono decise, e poi osservate mediante side effect, dal thread A. Il tutto diventa più efficiente se si fa uso di *librerie condivise*, poiché le pagine di memoria sono mappate sugli stessi indirizzi fisici. Quindi si forza l'esecuzione speculativa di un *gadget* (come le shared library), per causare side effect sulla cache, direttamente visibili all'attaccante. 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 questo caso è necessario che le istruzioni dei due processi abbiano lo stesso offset dell'address space.

Se ho più core non è un problema, migro sui vari processori se mi accorgo di aver sbagliato predizione.

Solo il training richiede stesso *core*, in quanto poi la cache è condivisa tra i core.

Si può coinvolgere anche un solo registro, lo si usa per muoversi in memoria.

Picture taken from the original Spectre paper



### Mitigazioni per Spectre: Retpoline

Si trasformano i salti indiretti in una serie di altre istruzioni, ad esempio il jump ad un registro viene implementato come una return. L'obiettivo è evitare la *misprediction*.

```

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

Si carica `target_address` in cima allo stack, si chiama `retpline_target` (e quindi in cima allo stack ci va l'indirizzo di ritorno, ovvero l'istruzione subito sotto `call retpoline_target`). In `retpline_target` l'istruzione `lea` cambia il valore nello stack pointer con il valore dello stack pointer +8, quindi punta a `target_address`; poi si esegue la return che porta proprio a questo valore puntato. Ricordiamo che in `x86_64` si ha `rsp` a 64 bit, cioè 8 byte, per questo ci si spiazza di 8. Sostanzialmente, se il core specula "bene", salta a `retpline_target`, se specula "male" (istruzioni sotto "`call retpoline_target`"), esegue operazioni innocue ed è bloccato in un loop per via della `jump`.

Lez.5 (04/10/23)

Altre mitigazioni contro Spectre:

### Spectre v2: indirect branch speculation barrier

Si controlla nella CPU se va o meno effettuata branch prediction. Su `x86` ci sono istruzioni che permettono di disabilitare l'utilizzo di branch prediction su registri `MSR` (model specific registry); si può eseguire per un certo intervallo di tempo in questa modalità.

Abbiamo due modalità:

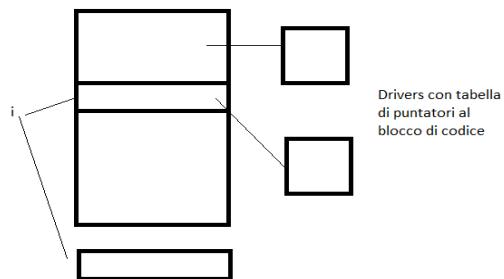
- **IBRS** (indirect branch restricted speculation): Si crea una sorta di guscio per cui ciò che avviene all'esterno non ha impatto. Possiamo quindi, grazie a un flag, differenziare la predizione *user* da quella *kernel*. IBRS si può utilizzare quando il software è composto da più componenti con livelli di criticità diversi.
- **IBPB** (indirect branch prediction barrier): Si può resettare il branch prediction, tutto ciò che è successo prima di questo istante non mi interessa.

Systemcall Linux: `prctl`.

### Spectre v1: conditional branch countermeasure - Sanitizzazione

Quando si accede qualcosa in modalità kernel avviene lo switch della tabella delle pagine.

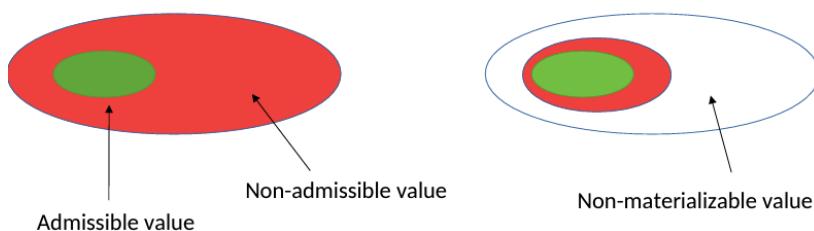
Quando si passa in modalità kernel si possono avere errori dovuti a ciò che è avvenuto in modalità user prima dello switch.



Prendiamo l'esempio di un *driver*, ossia una tabella che nelle entry possiede un pointer ai servizi del kernel. Lo user passa al kernel un certo indice "i" per poter utilizzare un determinato servizio che è fuori al driver, il kernel confronta tale indice con la taglia dell'array. Tuttavia potrebbe speculativamente accedere a tale zona di memoria kernel (ad esempio istruendo il predittore di salti passando tutti indici corretti in precedenza).

Per risolvere questo problema si adotta la **sanitizzazione**: quando eseguiamo realmente un certo predicato a livello software ciò che passiamo non deve far parte dei valori ammissibili o NON ammissibili, ovvero il predicato non deve lavorare su valori potenzialmente pericolosi.

Il driver risulta costituito dalle entries valide (verde) e da valori *null* (bianco) associabili a poche entries non valide (rosse, devono essere il minor numero possibile per essere gestite), così cadiamo in una zona rossa che non dà problemi poiché controllata. Realizzata con maschera di bit e poi facendo il controllo.



### **Loop unrolling:**

Un altro aspetto interessante dei salti è che un'istruzione di salto passa attraverso una probabilità che il salto sia preso o meno. C'è una tecnica che riduce il volume dei salti data una serie di istruzioni macchina, si utilizza in scenari di salti frequenti, ovvero i cicli; questa tecnica è nota come *loop unrolling*: faccio "unrolling" per diminuire il numero di salti (quindi azzardo di meno), vado a ciclare su porzioni di codice di dimensioni maggiori.

A livello prestazionale incide non solo cosa il processore deve eseguire a valle del salto, ovvero le istruzioni vere e proprie, ma anche le istruzioni di controllo di flusso per eseguire il salto. L'obiettivo è ridurre l'overhead di queste ultime.

Come si automatizza questo processo? Si marcano delle istruzioni come "loop unroll", ciò viene fatto in fase di compilazione e si può indicare anche la profondità dell'unroll.

Per far uso di questa ottimizzazione aggiungere "-o" da riga di comando.

Il loop unrolling porta con sé dei side effects: nel ciclo eseguo più attività, quindi ho bisogno di usare più registri, inoltre dato che il numero di istruzioni è maggiore non potrò mantenerle tutte in cache, dunque avrò dei cache miss.

Potrebbe essere meglio ridurre il numero di iterazioni dei cicli utilizzando tecniche di vettorizzazione, supportata dai processori moderni.

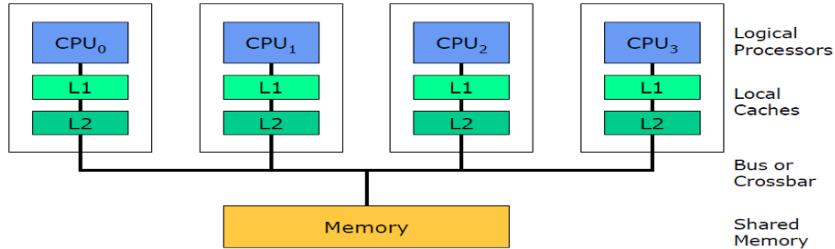
### **Clock frequency and power wall:**

Il power wall è un problema importantissimo. 130 W è la potenza massima sopportabile da un processore prima che si bruci.

Il consumo di potenza è funzione di voltaggio e frequenza del processore secondo la relazione:  $V^*V^*F$ .

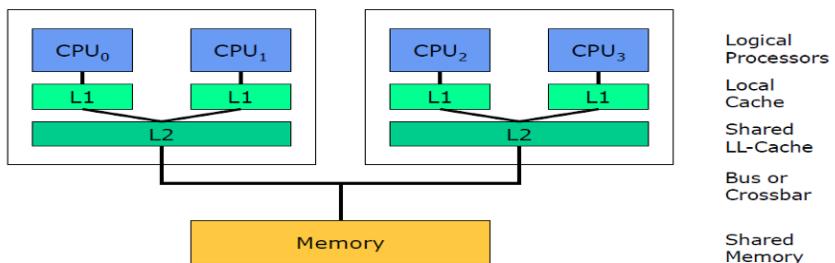
Non si può abbassare il voltaggio V più di tanto, la frequenza F non si può aumentare più di tanto, quindi la velocità dei processori non si può aumentare ulteriormente. Per ottenere macchine più potenti si ricorre al parallelismo nel chipset, concetto già introdotto con l'hyperthreading.

### Symmetric multiprocessor:



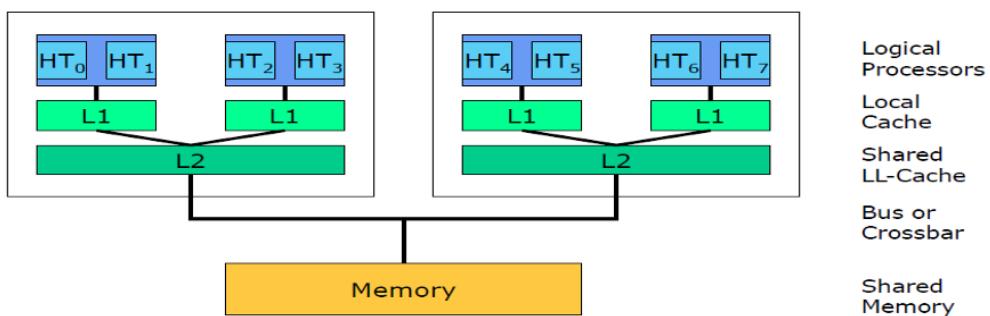
Ogni processore è single core e single thread. Gli accessi in memoria impiegano tutti lo stesso tempo.

### Chip multiprocessor (multicore):



Processore con più cores, ogni core è single thread.

### Symmetric multi-threading:



Chip multi processore con più hyperthreads. Dal punto di vista del SO è come se il numero di processori fosse pari al numero di hyperthreads.

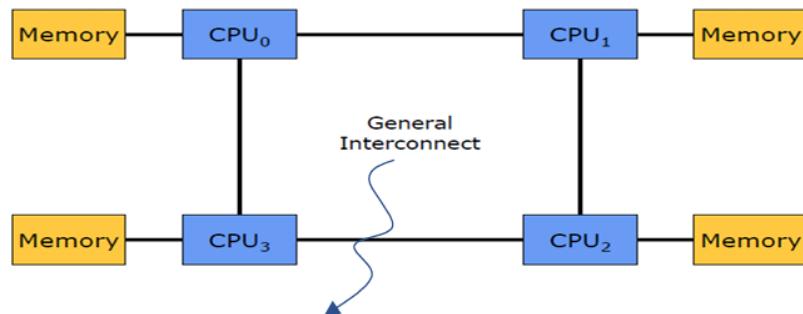
La memoria rappresenta il collo di bottiglia perché tutti gli hyperthreads vi accedono in concorrenza.  
Memoria esposta in ISA, le cache no!

### **Uniform memory access UMA:**

Parallelizzare la memoria e la possibilità di accedervi, tutti i cores accedono alla stessa memoria in maniera uniforme utilizzando le stesse componenti hardware.

### **NUMA**

Sta per “non uniform memory access”. La coppia Memory-CPU è un nodo NUMA.



This may have different shapes  
depending on chipsets

La memoria è divisa in banchi, ciascuno direttamente collegato a uno o più cores.

Ogni CPU accede “asimmetricamente” alla memoria vicina a lui o lontana, in termini di tempo gli accessi alla memoria più vicina richiedono un numero di cicli di clock inferiore.

Oltre al problema del tempo superiore per accedere a memoria remota, c’è anche il problema del traffico, se c’è traffico non stiamo usando l’architettura in maniera efficiente.

Senza traffico si ha:

Tempo per un accesso locale: 50 cicli di clock per cache hit, 200 cicli di clock per cache miss

Tempo per accesso remoto: 200 cicli di clock per cache hit, 300 cicli di clock per cache miss.

In SMT vediamo la memoria a livello dell’ISA, in NUMA no.

### **Coerenza della cache:**

L’architettura di cache è un’architettura di replicazione di informazioni, questa replicazione va gestita in maniera coerente. Questo problema riguarda le operazioni in memoria realmente richiamate dal processore, se un dato è disponibile in buffer cache non viene richiesto alcun accesso in memoria. Inoltre, si passa sempre per *store buffer* prima dell’ISA.

Quali proprietà deve avere la cache per garantire coerenza?

- *Causal consistency along program order*: Un thread che scrive su una certa locazione di memoria ram X deve poter rileggere l’ultimo valore scritto se nessun altro thread ha eseguito scritture su quella stessa locazione di memoria. Non può leggere il valore da lui scritto su un’altra locazione di memoria ma deve farlo dalla stessa su cui ha eseguito la scrittura.
- *Avoidance of staleness*: Se la scrittura e la lettura non sono concorrenti, ma sono abbastanza distanti nel tempo, la lettura di un processore da una determinata locazione di memoria X, che avviene dopo che un altro processore ha scritto nella stessa locazione, deve restituire il valore scritto

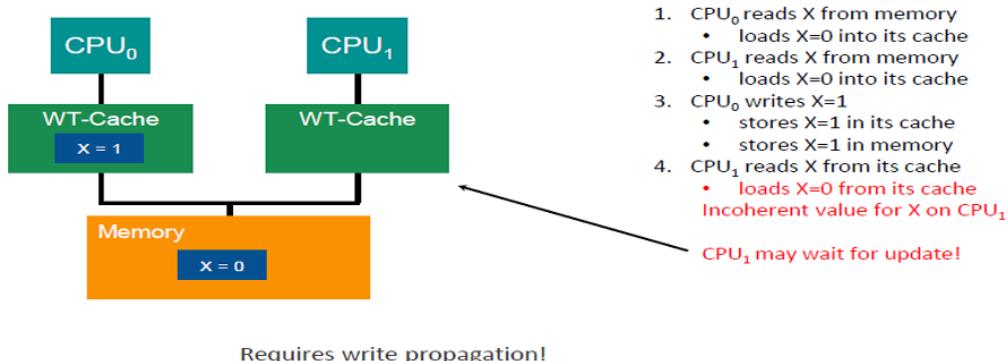
da quest'ultimo processore, a meno che qualcun altro non abbia effettuato ulteriori scritture nel frattempo.

- *Avoidance of inversion of memory updates:* Se ho una sequenza di scrittura dalla cache verso la RAM, l'ordine di scrittura in cache deve essere riproposto anche nella RAM, e tutti i processori vedono lo stesso ordine.

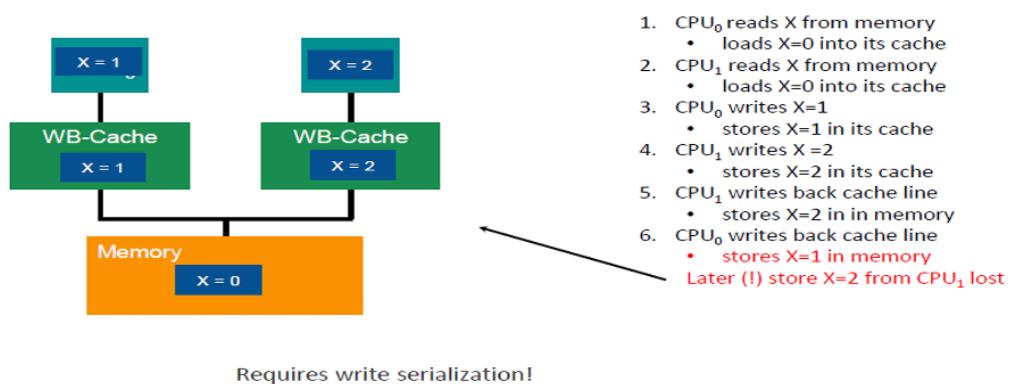
Osservazione: queste proprietà NON parlano di locazioni di memoria multiple.

Vediamo alcune *cache policy*:

Write-through-cache: le informazioni scritte in cache vengono subito riportate in RAM; però per la lettura mi affido alla "cache". Quindi, una CPU potrebbe scrivere più volte in cache (e quindi in memoria), ma una seconda CPU che legge solo (e lo fa dalla cache) non vede il nuovo valore in memoria RAM.



Write-back-cache: le informazioni non vengono subito riportate in memoria; ogni cpu scrive nella propria cache, quando la scrittura viene propagata in memoria richiede serializzazione degli aggiornamenti, altrimenti non rispetta l'ordine delle scritture in memoria (Ognuno scriverebbe quando vuole!)



### Cache coherence protocols:

Sono i protocolli di coerenza delle cache, che permettono di rispettare le tre proprietà precedentemente enunciate. Bisogna stabilire:

- Le transazioni supportate dal sistema di cache distribuite.
- I possibili stati; tipicamente l'oggetto che può cambiare stato è la linea di cache.
- Eventi gestiti da cache controller.
- Transizioni tra stati.

Il design è influenzato da diversi fattori:

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

Le diverse implementazioni di CC protocols differiscono per: throughput, latenza e overhead in termini di spazio (metadati per collocare una linea di cache in un certo stato).

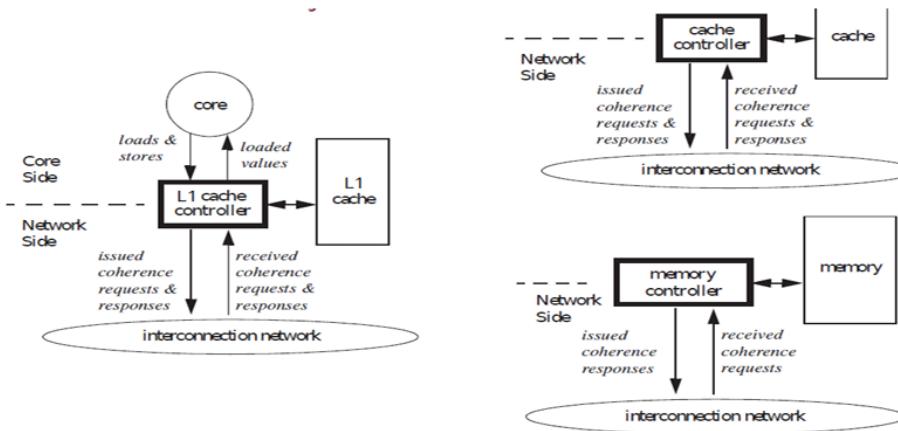
### **Famiglie di CC protocols:**

- *Invalidate*: quando aggiorno una replica le altre devono essere invalidate. Soltanto chi ha scritto l'ultima informazione ha la versione aggiornata del blocco, tutti leggeranno da lui.
- *Update*: si scrive una linea di cache, questa nuova informazione va propagata a tutte le repliche presenti; dato che una linea di cache è grande per trasmettere l'informazione usiamo molta banda.

Nei protocolli di tipo update come fanno le repliche a parlare e come conoscono l'una lo stato dell'altra?

### Snooping cache (tipologia update):

Ogni componente dell'architettura di cache osserva tutto; c'è un mezzo broadcast tramite cui vengono richieste le transazioni. Ogni elemento della memoria è agganciato ad un *controller*. Chi acquisisce il *broadcast medium* trasmette le informazioni, gli altri non possono trasmettere nulla. Ciò comporta un problema di scalabilità; questo problema sussiste anche nel caso in cui la frequenza di transazioni sia alta.



Su architetture moderne si utilizza la famiglia invalidate; ha poco senso sfruttare la località nell'ambito di threads diversi, tipicamente funziona su singolo thread. Inoltre può capitare di dover aggiornare una replica non necessaria, o ricevere più volte lo stesso input per invalidare una replica.

Vediamo la famiglia invalidate.

### MSI:

Una scrittura porta a richiesta di invalidazione sul mezzo broadcast per tutte le altre repliche.

I tre stati coinvolti sono *modified*, *shared* e *invalid*. Si tiene traccia dello stato di ciascun blocco:

Modified: sono stati modificati dati condivisi.

Shared: più componenti condividono la stessa linea, i cpu-core leggono altre copie del medesimo blocco, esistono più copie valide. Non posso scrivere.

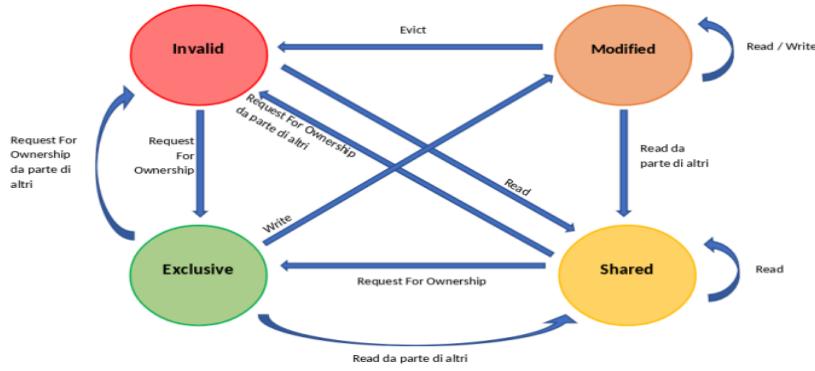
Invalid: la linea non è valida perché sovrascritta. Non posso né leggere né scrivere.

Per quanto riguarda le operazioni di *lettura*:

- Se la policy della cache è write-through, recuperiamo 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.

Perchè questo modello non funziona? Nel caso di aggiornamenti multipli su una linea di cache ogni update chiede a tutti gli altri di invalidare, ma gli altri hanno già linea non valida a causa degli update precedenti, quindi si sprecano cicli per operazioni inutili.

### MESI:



E' simile al modello MSI, ma introduce lo stato Exclusive; esso consiste nel dire a tutti che la linea di cache è esclusivamente mia, quindi tutti devono invalidarla. Questo vuol dire che gli altri automi MESI andranno sullo stato "Invalid", mentre il proprietario per leggere/scrivere andrà in "Modified", restandovi finchè nessun altro chiede di leggere i dati. Il vantaggio risiede nel fatto che aggiornamenti multipli non richiedono molteplici richieste di invalidazione.

Gli altri possono continuare a svolgere altre attività, il mezzo broadcast non è in uso.

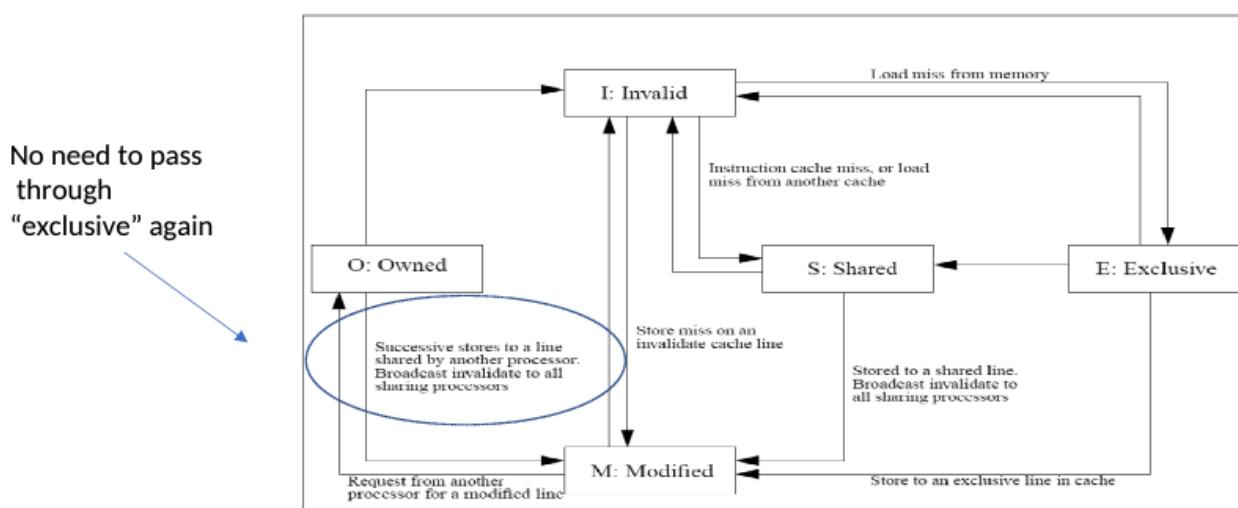
In MESI quando un componente vuole leggere la linea c'è la transizione da modified a shared.

#### Transizioni in MESI (e quando richiedere il broadcast medium):

1. Read locale (fatta dal processore stesso):
  - Stato M, stato E e stato S: no transazioni distribuite, il processore ci sta già lavorando.
  - Stato I: la cache non possiede la copia che garantisce CC, bisogna generare una richiesta sul bus, che può forzare operazioni dalle altre cache.
2. Write locale:
  - Stato M: nessuna transazione sul bus.
  - Stato E: nessuna transazione sul bus, c'è solo la transizione verso lo stato M per tenere traccia di aver modificato la copia.
  - Stato S: già ho la linea localmente, devo fare la richiesta sul bus per ottenere l'esclusività tramite *broadcast medium*.
  - Stato I: bisogna generare richiesta sul bus per ottenere l'esclusività.

Lez.6 (05/10/23)

#### MOESI:



Viene introdotto un nuovo stato: Owned, il quale indica che il dato è condiviso ma si possiede la copia master e dunque è possibile modificarla (andando in *Modified*) senza dover passare nello stato Exclusive. Questo evita una serie di transizioni di stato da S a E per scrivere la copia. Se il proprietario scrive, partendo dallo stato Owned, non si deve preoccupare di chiedere nuovamente l'esclusività del blocco, bensì passa

direttamente allo stato *Modified*, invalidando tutte le altre copie.

In MESI avrei dovuto fare una transizione in più: Shared → Exclusive → Modified).

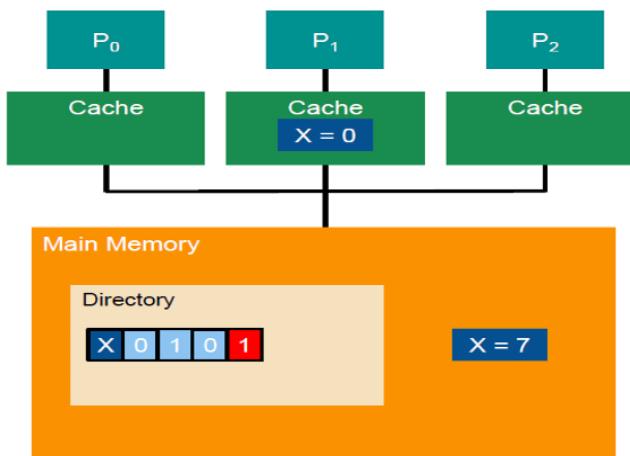
Stati stabili di MOESI:

1. Modified → il blocco è valido, esclusivo, di proprietà e potenzialmente sporco.
2. Owned → non è l'unica copia valida, ma è tuttora in fase di aggiornamento.
3. Exclusive → è l'unica copia presente, ma non è sporca, per poterla modificare si deve fare transizione nello stato Modified.
4. Shared → il contenuto è condiviso, le operazioni di lettura possono essere fatte da tutti gli altri cores.
5. Invalid → non si può leggere o scrivere il blocco.

#### Directory based protocols:

E' uno schema secondo il quale si riesce ad agire in parallelo: abbiamo interazioni soltanto tra componenti interessati al cambiamento di stato.

Non si utilizza mezzo condiviso (non scala bene), ma comunicazione punto a punto.



**Directory:** Per ogni blocco di memoria (grande quanto una linea di cache) si mantiene una entry (quindi una riga) della directory. Ci sono *n bit di presenza* (in figura quelli celesti), che indicano se il blocco è o meno nell'i-esima cache; l'ultimo bit della directory è il *dirty bit*, che indica se la linea è stata modificata senza aver riportato in memoria tale aggiornamento (in figura quello rosso).

Utilizzando queste informazioni si sa quale componente contattare.

Se ci sono due bit pari a 1 significa che su quella linea di cache siamo in condivisione con altri componenti.

Read miss, ovvero voglio leggere un blocco ma ho cache miss in lettura:

1. Dirty bit=0:
  - Leggi dalla memoria principale
  - Imposta presence[i]
  - Fornisci dati al lettore
2. Dirty bit=1:
  - Richiama la linea di cache da P<sub>j</sub> (determinata da presence[j] = 1)
  - Disattiva il dirty bit
  - Imposta presence[i] = 1
  - Fornisci dati al lettore e propagali alla memoria.

Write miss:

1. Dirty bit=0
  - Invia invalidazioni a tutti i processori P<sub>j</sub> con presence[j] = 1
  - Disattiva il bit di presenza per tutti i processori
  - Imposta il bit dirty = 1 (ora lo sto usando io)
  - Imposta presence[i] = 1, proprietario P<sub>i</sub>
2. Dirty bit=1:
  - Richiama la linea di cache dal proprietario P<sub>j</sub> (lo riconosco da presence[j] = 1)

- Disattiva presence[j] = 0
- Imposta presence[i] = 1, il dirty bit rimane impostato
- Ack al writer

### **Aspetti di performance della cache esposti al software:**

Quando scriviamo in cache vengono svolte una serie di attività a livello hardware.

Vediamo quali approcci basici permettono di utilizzare consapevolmente l'architettura hardware. In particolare la cache è impattata quando abbiamo una struct.

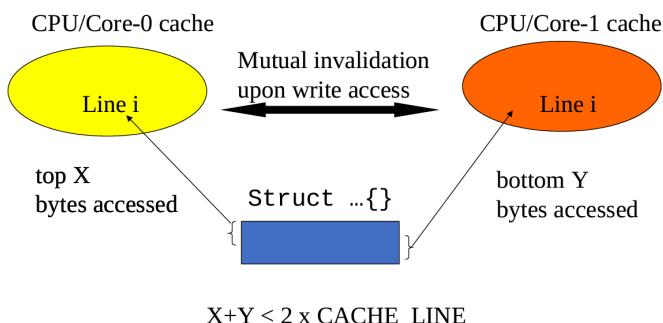
Come avere l'incidenza minore possibile sulle linee di cache?

Loosely related fields: quando tocco un campo raramente ne tocco un altro.

Metto i campi su linee di cache differenti:

1. Altrimenti porterei in cache un campo che non mi serve.
2. posso eseguire in parallelo operazioni loosely coupled, che operano su loosely related fields; se fossero sulla stessa linea a uso esclusivo altri dovrebbero attendere. Questo porterebbe al problema della false cache sharing.

Secondo lo stesso principio strongly related fields andrebbero sulla stessa linea di cache.



### **Tecniche di inspection delle cache lines:**

Nell'ISA l'unica operazione che posso fare sulla cache è il *flush* dei dati.

Misurando i side effects legati al tempo posso capire se una linea di cache è stata o meno acceduta.

#### Attacco Flush+reload:

Si utilizza quando si hanno dati condivisi.

Si flusha il contenuto di una linea di cache, poi passato un po' di tempo riaccoedo a quei dati e se il tempo risulta elevato significa che non sono stati acceduti da nessuno, altrimenti significa che sono stati acceduti, in quanto sono tornati disponibili in cache.

Cosa serve per implementarlo?

1. High resolution timer.  
RDTSC: è un timer ad alta risoluzione; è una singola istruzione macchina (64 bit di timestamp counter, i 32 più significativi nel registro %edx, i meno significativi in %eax).
2. Operazioni di cache flush senza richiesta di privilegi.  
CLFLUSH: cache flush con indirizzo di memoria, che corrisponde a una singola linea. Problema: la linea prima di andare in cache sta nello store buffer, se vogliamo che avvenga il flush dobbiamo eseguire prima mfence, l'istruzione per portare i dati dallo store buffer in cache.

### **ASM inline:**

Vediamo come utilizzare istruzioni macchina in un blocco di codice C.

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

Va marcato il codice come *asm*.

`AssemblerTemplate` rappresenta una serie di istruzioni dell'ISA, si lavora con memoria e registri. Se ho un intero non posso esprimere nell'ISA, quindi bisogna raccordare il codice assembler con qualcosa che è al di fuori, perciò si usa `OutputOperands` e `InputOperands`, per portare valori nei registri.

`Clobbers`: quando si genera l'eseguibile il compilatore deve ripristinare il valore di alcuni registri (`push`, `pop`) perché potrei sporcarli con le istruzioni.

`GoToLabels`: posso specificare labels per parti di codice assembler, target di salti.

`Volatile`: il compilatore non deve apportare ottimizzazioni rispetto a ciò che ho scritto io.

Il simbolo “=” significa che l'operando a esso assegnato va usato come output.

Altrimenti è input.

Come si indicano i registri? Con simboli che indicano registri, il simbolo “*r*” indica un registro generico, scelto dal compilatore, esso ha un indice che può essere usato per richiamarlo, quindi posso risalire al registro usato.

```
unsigned long probe(char* adrs) {
    volatile unsigned long cycles;
    asm(
        "mfence \n"
        "lfence \n"
        "rdtsc \n"
        "lfence \n"
        "movl %%eax, %%esi\n"
        "movl (%1), %%eax\n"
        "lfence \n"
        "rdtsc \n"
        "subl %%esi, %%eax\n"
        "clflush 0(%1) \n"
        : "=a" (cycles)
        : "c" (adrs)
        : "%esi" , "%edx" );
    return cycles;
}
```

A barrier on all memory accesses

Barriers on loads

Prima di `clflush` ho una serie di istruzioni per portare il dato in lettura e, dunque, in cache.

“%1” corrisponde a “c”.

Devo eseguire in ordine le istruzioni, per questo ci sono le `fence`, altrimenti potrei avere misure sbagliate dei tempi.

## Lez.7 (09/10/23)

Ci si può proteggere dal fatto che l'attaccante possa vedere cosa è stato toccato e risalire al valore? Nei processori moderni l'istruzione di accesso al `timestamp` può essere eseguita solo in kernel mode.

Ciò non permette di proteggersi da Meltdown e Spectre, perché per misurare il tempo posso usare altri metodi: metto in piedi un thread che incrementa un contatore in memoria, esso di dà il numero di tick logici attraversati da questo flusso.

Un altro thread può leggere più volte questo valore.

Misurare il tempo in maniera relativa mi dà comunque un'indicazione di quanto tempo sia passato.

### Esempio:

[flush-and-reload/far-emulated-timer.c](#)

Usiamo `probe` e `probe_noflush` per fare il confronto tra i tempi di accesso in memoria e in cache.

Per un cache miss il tempo risulta tre volte maggiore di quello per un cache hit.

Attenzione: flush and reload senza cache inclusiva non funziona, perché qualcosa fatto ad un certo livello della cache può non essere visibile da qualcun altro a livelli di cache inferiori, quindi le latenze possono non essere raccordabili rispetto a ciò che abbiamo osservato.

Tuttavia, gli attacchi possono ancora avere successo quando eseguiti tra processi in esecuzione sullo stesso core della CPU, indipendentemente dal tipo di sistema di cache utilizzato. Questo perché i core della CPU condividono spesso la cache, e ciò permette agli attacchi di sfruttare tale condivisione per recuperare informazioni.

### ***Memory consistency:***

Non ha nulla a che vedere con la CC, bensì ha a che fare con la relazione tra ciò che viene eseguito in memoria dal flusso sulla cpu e ciò che effettivamente si vede in memoria.

Quando leggiamo in memoria dovremmo leggere l'ultimo valore scritto in memoria, il problema è che la store potrebbe essere già stata chiamata, ma la scrittura effettiva non essere avvenuta.

Bisogna definire l'ordine ammissibile delle operazioni.

Il program order è l'ordine, per processore, degli accessi in memoria da parte del programma in esecuzione.

Il visibility order è l'ordine secondo cui gli accessi risultano visibili dall'architettura di memoria agli altri processori.

Attenzione: il problema esiste perché l'architettura è multiprocessore.

### ***Sequential consistency:***

Il risultato di ogni operazione è lo stesso che si avrebbe se le operazioni di tutti i processori fossero sequenziali e le operazioni di ciascun processore avvenissero secondo l'ordine specificato nel suo programma.

Esempio: Nell'immagine, il program-order di CPU1 è (a1,b1), quello di CPU2 è (a2,b2).

La prima sequenza proposta è consistente, la seconda no perché non rispetta i vincoli appena descritti.

<i>CPU1</i>	<i>a1,b1,a2,b2</i>
<i>[A] = 1;(a1)</i>	<b>Sequentially consistent</b>
<i>[B] = 1;(b1)</i>	<b>Visibility order does not violate program order</b>
<i>CPU2</i>	
<i>u = [B];(a2)</i>	<i>b1,a2,b2,a1</i>
<i>v = [A];(b2)</i>	<b>Not sequentially consistent</b>
<i>[A],[B] ... Memory</i>	<b>Visibility order violates program order</b>
<i>u,v ... Registers</i>	

Le architetture moderne non sono sequentially consistent.

Gli algoritmi pensati per eseguire su un'architettura sequentially consistent, come bakery, saltano; anche l'algoritmo di Dekker (segnalazione per entrare in CS) salta.

Non conviene usare *sequential consistency* in termini di performance della memoria; ad esempio i cache miss vanno serviti in sequenza anche se sono operazioni di scrittura senza istruzioni correntemente dipendenti.

### ***Total store order TSO:***

Le operazioni che eseguiamo passano per lo store buffer (sia letture dalla memoria che scritture in memoria). Perché si usa lo store buffer?

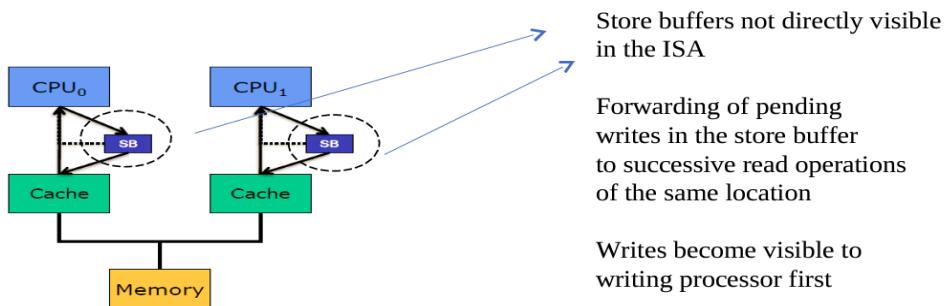
Se bisogna portare un dato in memoria la linea di cache deve diventare a uso esclusivo, ciò implica cambiare lo stato degli altri: bisogna attendere che il bus si liberi, richiedere l'invalidazione della cache degli altri e infine cambiare lo stato dell'oggetto.

Lo Store Buffer 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.

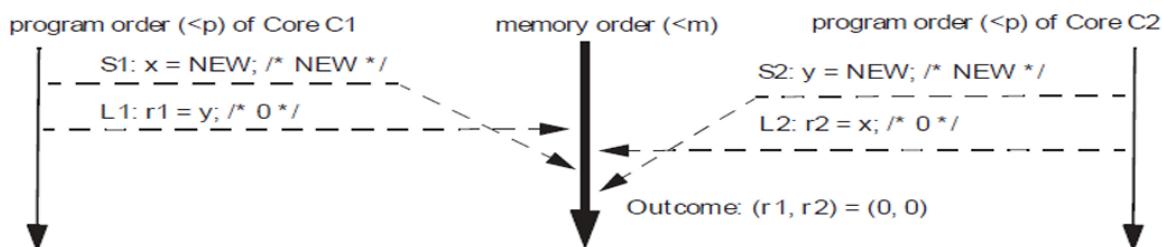
Con lo store buffer il valore viene effettivamente riportato in memoria solo quando i protocolli di consistenza della cache danno l'ok. Su x86 lo store buffer è FIFO, ma non necessariamente su tutte le architetture.

Quindi non è detto che quello che scrivo prima nello SB vada prima in memoria.

TSO viene utilizzato su diverse architetture moderne.



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 (*load-store bypass*). Un esempio di algoritmo che non funziona senza sequential consistency è Algoritmo di Dekker, dove ognuno prima effettua la *store* su una variabile (che passerà per lo store buffer) e fa la *load* dell'altro (mutua esclusione).



### Istruzioni di fence:

Come si scrivono programmi concorrenti che lavorano in maniera corretta su informazioni condivise?

Si utilizzano le istruzioni di *fence* della memoria, che realizzano una sorta di barriera a valle della quale l'operazione è resa visibile a chiunque esegue; serializza le operazioni.

Le istruzioni sono:

- **sfence:** serializza e operazioni di store in memoria precedenti a tale comando.
- **lfence:** serializza le operazioni di load dalla memoria precedenti a tale comando.
- **mfence:** serializza tutte le operazioni, un'unione delle prime due istruzioni.

Attenzione: queste operazioni sono diverse da quelle serializzanti della pipeline, non flushano la cache.

Ci sono istruzioni che possono diventare serializzanti usando "lock" (rimane in attesa di lock) o "trylock" (ritorna subito se non prende lock); quando rilasciamo il lock creiamo una barriera:

Istruzioni di questo tipo sono: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XAND.

Alcune di queste istruzioni sono di per sé **atomiche** in lettura e scrittura (eseguono senza interruzioni), come CMPXCHG, e tali operazioni atomiche fanno parte della tipologia **read-modify-write RMW**. Se precedute da LOCK diventano **serializzanti** (tutto quello che è successo prima viene scritto in memoria e anch'esse sono globalmente visibili). Ciò vuol dire che il mezzo di interconnessione viene bloccato a favore

dell'unità processante che le sta eseguendo (o gli viene affidata l'esclusività di una linea di cache se stiamo usando protocolli di coerenza cache). MESI le supporta.

Si possono usare le fences anche con linguaggio di alto livello:

```
void __mm_sfence(void)
void __mm_lfence(void)
void __mm_mfence(void)
bool __sync_bool_compare_and_swap (type *ptr, type
oldval, type newval)
```

#### Implementazione di una barriera:

```
long control_counter = THREADS;
long era_counter = THREADS;

void barrier(void){
    int ret;

    while(era_counter != THREADS && control_counter == THREADS);

    ret = __sync_bool_compare_and_swap(&control_counter,THREADS,0);
    if(ret) era_counter = 0;

    __sync_fetch_and_add(&control_counter,1);
    while(control_counter!=THREADS);
    __sync_fetch_and_add(&era_counter,1);
}

era_counter initial update already committed when performing this
```

`control_counter` è il numero di thread entrati ed `era_counter` il numero di thread usciti.

Chi entra a chiamare questa funzione non può uscire finché non è stata chiamata da tutti gli altri threads; mettiamo i threads in uno stato di sincronia; tutti i threads devono essere usciti dalla barriera.

Si può entrare in un'altra barriera solo se tutti i threads sono usciti da quella precedente.

`sync_fetch_and_add` aggiunge un'unità al contatore.

#### Esempio:

[active\\_sync\\_barrier/barrier\\_synchronization.c](#)

Entrambi i contatori vengono allineati a 64 bytes, così vanno su linee di cache differenti, si potrebbe lavorare sulle due variabili in contemporanea in questo modo, ma in questo caso non si può fare per via dell'attesa. In cicli diversi l'iniziatore può essere un thread diverso.

#### Lock vs coordinazione più scalabile:

Abbiamo visto come le RWM possono essere utilizzate insieme ai lock. Tuttavia possiamo usare approcci più scalabili e avanti un impatto minore sulle performance del sistema:

- **algoritmi lock-free e wait-free:** Nel caso lock-free, posso ottenere degli abort, e quindi ritento (alg. Linearizzabile). Nel caso wait-free, non ho mai abort, ed è consistente. C'è concorrenza tra lettura e scrittura, ma non tra due scritture. Nella lock free se ho due operazioni incompatibili una deve essere rifiutata e rieseguita con try.
- **Read copy update:** 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.

Osservazione: i lock servono per rendere le attività linearizzabili.

#### Linearizzabilità:

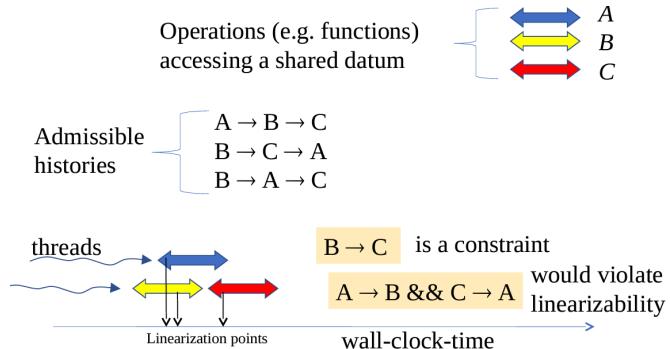
La linearizzabilità è una restrizione della serializzabilità perché concerne il singolo dato. Ciò è vero se gli accessi concorrenti al dato, anche se durabili diversi cicli di clock, si possono vedere come materializzati in un singolo punto, e tutte le operazioni che si sovrappongono possono essere ordinate in base al loro istante di materializzazione.

Vorremmo ottenere qualcosa di simile senza lock. Ciò sarebbe utile nei casi come questo:

Prendiamo un sistema moderno con un thread in esecuzione su una VM. Questo thread può essere deschedulato e altri threads possono operare su una struttura dati condivisa se crasha. Si evita il problema del cpu stealing.

Le operazioni concorrenti usano accessi atomici, e la sequenzializzazione in CS avviene tramite operazioni atomiche.

Attenzione: un thread che sta eseguendo un'istruzione atomica può fallire, quindi deve riprovare a eseguirla. In base al tipo di struttura su cui vogliamo lavorare può essere meglio lock free o wait free.



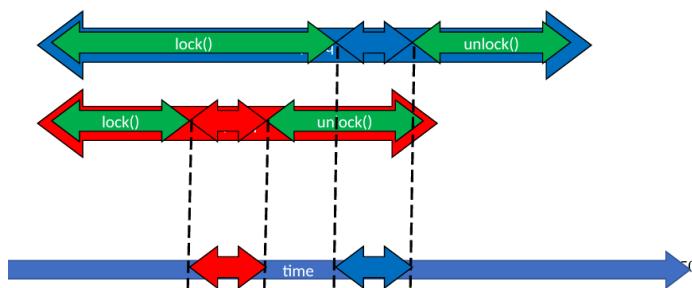
Una struttura atomica garantisce linearizzazione quando si agisce sulla stessa porzione della struttura dati. Un thread che esegue una funzione e poi ne chiama un'altra è sempre vero che vede tutto ciò che è avvenuto prima sulla struttura dati condivisa?

Se non eseguiamo sulla stessa cpu può darsi che lo store buffer non sia ancora stato flushato, quindi le modifiche potrebbero non ancora essere effettive sulla struttura in memoria, questo anche su singolo thread. **Operazione fallita = no operazione.**

Lez.8 (11/10/23)

### RMW vs locks vs linearizability

Le operazioni RMW, nonostante implementino accessi in memoria non banali, appaiono in modo atomico sull'intera architettura hardware. Sono sfruttabili per definire dei punti di linearizzazione delle operazioni, dando la possibilità di ordinarle. Le operazioni RMW possono fallire e il loro esito può influenzare l'esecuzione o meno di una particolare istruzione e l'istante in cui essa viene eventualmente materializzata. **Con 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.



### Non-blocking coordination

Invece di utilizzare le RMW per implementare dei lock facciamo sì che nelle API che invochiamo siano usate delle RMW per notificare l'aggiornamento sulla struttura.

Ad esempio se usiamo CMPXCHG è possibile che il valore rispetto cui si fa il confronto non sia più presente, se quando si esegue l'aggiornamento il valore non è più quello che aveva letto l'operazione fallisce. Abbiamo i seguenti approcci:

- 1) Lock-freedom: almeno un'istanza di una chiamata a funzione termina con successo in tempo finito e tutte le istanze di chiamata a funzione terminano in tempo finito (o con successo o no). Se fallisco,

posso ricominciare da zero, logica abort-retry.

Se due ordini di operazioni sono incompatibili uno solo è accettato. I thread sono indipendenti, se crashano devo solo riprovare.

- 2) Wait-freedom: tutte le istanze di una chiamata a funzione terminano con successo in tempo finito.  
Tutto quello che faccio va “sempre bene” (abort impossibile).  
Dipende però con che struttura lavoro.

### Vantaggi della sincronizzazione non bloccante:

Ogni thread termina le operazioni in un tempo finito, indipendentemente dal suo comportamento. Il tempo non dipende dal comportamento del thread e dalla sequenza di acquisizione dei lock come nella sincronizzazione bloccante.

### Esempio: lista concatenata (lock free):

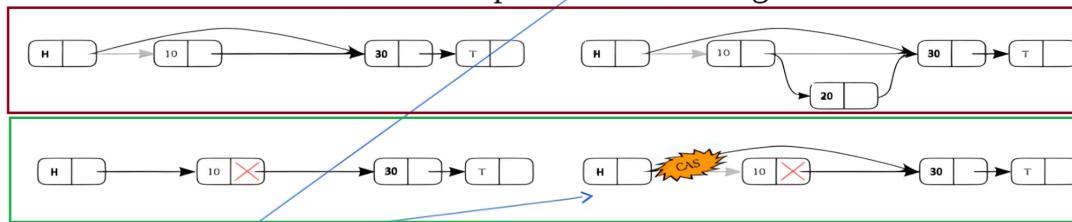
In questo schema si fa uso dell’istruzione atomica *compare and swap*.

Tale istruzione richiede tre elementi: un puntatore ad un’area contenente un valore “val”, un valore “old” e un valore “new”. Aggiorno “val” al valore “new” solo se “val == old”, ovvero se non ci sono state modifiche nel mentre.

- 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

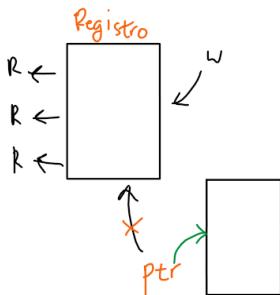
- Insert: Trovo la posizione per il nuovo elemento, lo collego al successore e poi con una *compare and swap* collego a esso il predecessore. L’unico caso in cui si ha un fallimento (e quindi un abort) è quello in cui c’è un altro inserimento concorrente nello stesso punto della lista collegata (i valori “val” e “old” non coincidono più!).
- Remove: Per la rimozione non posso semplicemente effettuare una CAS sul nodo successivo (30 in figura) al nodo in rimozione (10) e poi rimuovere il nodo, perché nel mentre potrei avere un nuovo nodo (20) che si collega a 10 prima che venga rimosso, generando un path invalido. La soluzione (in verde) è marcare il nodo 10 come “da eliminare”, tramite bit di controllo in modo atomico. Poi, tramite CAS, aggiorno il puntatore all’elemento successivo a quello da eliminare.

Se un thread A elimina un certo elemento e intanto un thread B ha preso il riferimento a tale elemento prima di essere deschedulato, nel momento in cui A riutilizza il nodo per inserire nuove informazioni, B può fare “traversing”, ovvero può leggere le informazioni anche se non era previsto.

### Sincronizzazione wait-free:

Non abbiamo più la logica abort-retry, tutti svolgono sempre un lavoro utile. Vediamo un esempio di tale approccio:

Prendiamo il *registro atomico* ( $1, N$ ) *wait free*, avente uno scrittore e  $N$  lettori. Le operazioni read e write sono atomiche e possono richiedere più cicli. Non usiamo i lock, non sono ottimali, bensì si usano molteplici istanze del registro atomico e un pointer all'istanza più recente. Lo scrittore aloca una nuova istanza e tramite CAS aggiorna il pointer alla nuova istanza, che sarà usata per la lettura, la quale non genera problemi in quanto non ha interferenze. Per far funzionare bene il tutto, si usano  $N+2$  slot (istanze multiple dello stesso registro). Essendo  $N+2$  gli slot,  $N$  i lettori e uno scrittore, lo scrittore avrà sempre uno slot in cui scrivere, quindi non va mai in blocco. Il "problema" è che qualche lettore potrebbe puntare a buffer molto vecchi, e questi non sono infiniti.



Se il writer ( $w$ ) vuole fare update di più bytes in posizioni distanti va fatto atomicamente.

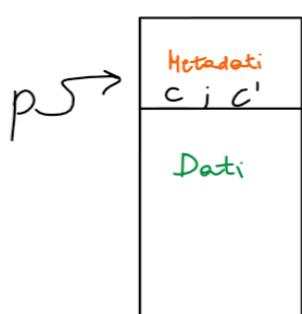
Aggiorno il pointer ad un'altra area con le informazioni aggiornate; quindi prima si aggiornano le informazioni nella nuova area di memoria e poi il puntatore.

*Il problema è che fine fa la vecchia struttura?* Qualche thread può leggere il vecchio registro.

Si usa un'unica variabile di sincronizzazione con due campi: **ultimo slot scritto e numero di lettori che leggono lo slot**. Questo secondo campo viene aggiornato dai lettori mediante una chiamata `fetch_and_add_atomic`, un'istruzione esegue lettura dati e incremento.

Lo scrittore può, mediante una `atomic_exchange`:

- Aggiornare l'indice, associato a uno slot in cui non ci sono più letture pendenti) e impostare 0 lettori sul nuovo slot.
- Le vecchie informazioni appena sostituite vengono memorizzate nei metadati dell'ultimo slot utilizzato. Così, quando i lettori terminano la lettura, rilasceranno un token tale per cui, se nei metadati associati al registro si avrà **#lettura iniziata = #lettura terminata**, porteranno lo slot a essere riusabile). Contiamo tutti coloro che hanno preso il riferimento, questo funziona se le operazioni eseguite sono in numero finito.



Cosa fa Linux?

- wait-free su strutture semplici.
- RCU su strutture più complesse; RCU funziona per strutture accedute frequentemente e con più letture che scritture. Ad esempio si usa nelle hash tables dei process control blocks.

### **Read Copy Update RCU:**

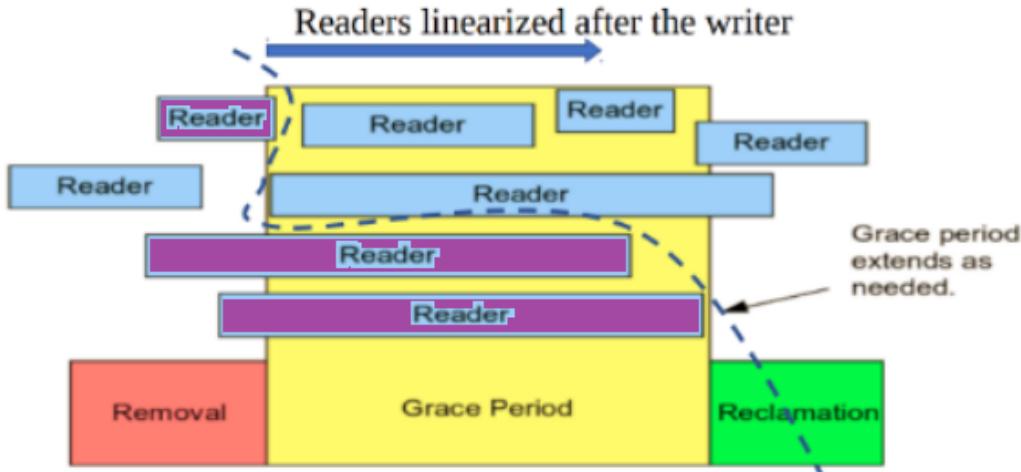
E' una sincronizzazione che realizza un trade-off tra l'utilizzo dei lock e la lock-freedom.

Serializzando le scritture si ha *un solo writer per volta*, quindi quando scrive non attende nessuno, mentre gli  $n$  readers non devono attendersi né tra di loro né con il writer. Ottimo con read-intensive.

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 deallocated, si ha il cosiddetto *grace period*.

Un'area di memoria si può liberare solo dopo aver completato il grace period.

Coloro che accedono dopo l'update del writer non devono essere attesi.



Nella figura qui sopra si hanno tre reader (quelli che iniziano prima o dentro la *removal* e terminano *oltre la removal*) che eseguono una lettura concorrente alla removal e che quindi devono essere attesi prima della rimozione vera e propria (reclamation): il grace period dura fin tanto che tutti e tre questi reader non hanno concluso la loro lettura. Il Grace period è un periodo di grazia in cui, anche se sono pronto a cambiare indirizzo, lascio "attivo" il vecchio indirizzo per far concludere le letture.

Come si implementa?

Ad alto livello abbiamo:

- *Reader*: segnala che vuole leggere, legge e segnala che ha terminato la lettura.
- *Writer*: prende il lock, aggiorna la struttura (è come se fosse una lista di due elementi, si identifica un nuovo blocco e "taglia" i link del vecchio nodo), aspetta i lettori standing (sulla versione non aggiornata), poi rilascia il buffer (riusabile) e il lock.

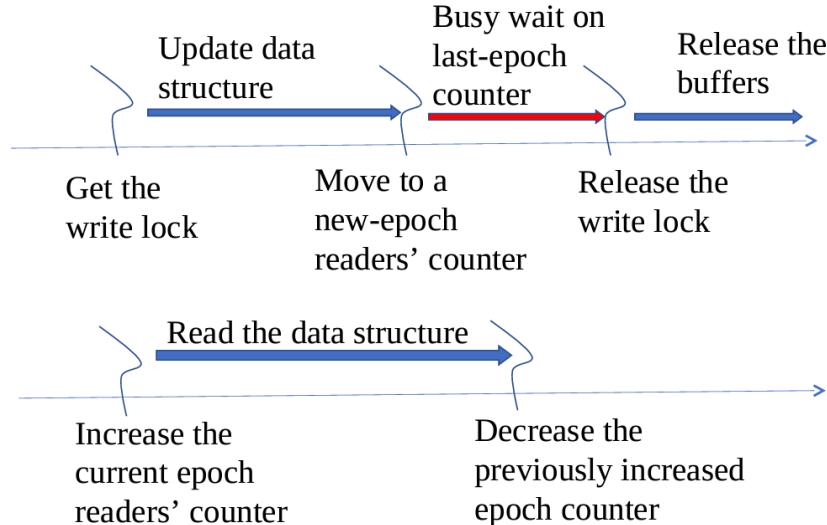
#### RCU preemptable e non preemptable:

- Nel caso ***NON-preemptable RCU***, il reader disattiva la possibilità di essere interrotto (non rilasciano la cpu su richiesta), per il writer c'è un thread di livello kernel che cerca di passare sulle altre cpu, se ci riesce non ci sono più readers concorrenti. lo scrittore, subito dopo aver aggiornato la struttura dati, invoca: `for_each_online_cpu(cpu) run_on(cpu)`.

Quando lo scrittore verrà schedulato su tutte quante le CPU, sarà possibile concludere il grace period. Lo svantaggio 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.

- Nel caso ***preemptable RCU*** potrei subire un interrupt, ma stando al "ragionamento" di prima, lasciare la CPU vuol dire aver finito. Si usa il concetto di *epoch*: Si ricorre all'utilizzo di un *presence-counter* atomico che va a indicare quanti lettori stanno correntemente insistendo su una determinata versione (epoch) della struttura dati; è atomico perché, come al solito, vengono acceduti in lettura e in scrittura con un'unica istruzione atomica (i.e. `fetch_and_add`). Nel momento in cui lo scrittore aggiorna la struttura dati, redireziona il puntatore al presence-counter verso una nuova istanza di presence-counter (quello relativo alla nuova epoch), 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 epoch (last-epoch), in modo tale che sia tutto

consistente. Chiaramente, il grace period termina quando il last-epoch counter diviene uguale a zero.



#### API:

`rcu_read_lock()`

- This signals that the reader has started operating (e.g. it may block preemptability, or do nothing in a kernel with `CONFIG_PREEMPT = n`)

`rcu_read_unlock()`

- This signals the reader has ended its activity

`synchronize_rcu()`

- This API implements the wait of the end of the grace period

`call_rcu(...)`

- This API enables the usage of a callback to be executed after the grace period has ended, but does not lock the updater thread

In `call_rcu` passo il puntatore alla funzione da eseguire al termine del grace period (callback).

#### API in contesto bloccante (sleepable RCU):

```
int init_srcu_struct(struct srcu_struct *sp);
void cleanup_srcu_struct(struct srcu_struct *sp)

int srcu_read_lock(struct srcu_struct *sp);
void srcu_read_unlock(struct srcu_struct *sp, int idx);

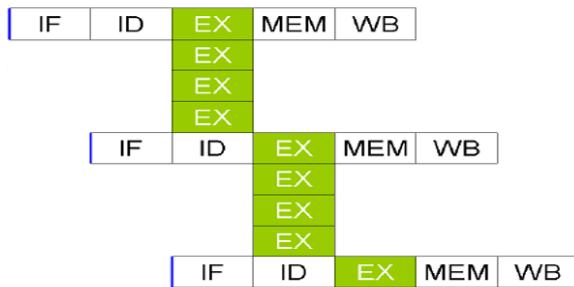
void synchronize_srcu(struct srcu_struct *sp);
```

Servono quando il lettore ad un certo punto chiama un servizio kernel bloccante durante l'accesso alla struttura.

### Aspetti relativi alla parallelizzazione (o vettorizzazione):

Meccanismo tale per cui una singola istruzione esegue più task in parallelo, in particolare è utile per il calcolo vettoriale, coinvolgendo molteplici registri in contemporanea.

- **SIMD (Single Instruction Multiple Data)**: La vettorizzazione è una forma di SIMD. Istruzioni con fase exe con sorgenti e destinazioni multiple; ci sono più valori in input a una ALU.



- **MIMD (Multiple instructions Multiple Data)**: Multiprocessore o chip con hyperthreading (MI); è supportato anche da singolo processore con supporto alla speculazione (MD)
- **MISD (Multiple Instruction Single Data)**: Diverse istruzioni lavorano sullo stesso dato, supportato tramite renamed registers, un dato ha molteplici locazioni.
- **SISD (Single Instruction Single Data)**: Single core e non c'è speculazione. È Von Neumann.

### Vettorizzazione su x86

Su x86 SSE (*Streaming SIMD Extension*) è il supporto alla vettorizzazione, ci sono 8 registri a 128 bit per supportare le operazioni vettoriali (registri XMM). In x86-64 SSE2, ci sono 16 registri.

Ci sono alcune API che utilizzano questi registri; alcune istruzioni macchina usate intrinsecamente da queste API richiedono che la memoria sia allineata, se i dati non sono allineati: *general protection error*. La vettorizzazione può essere implicita o esplicita.

- **Vettorizzazione implicita**: Possiamo attivare la vettorizzazione automatica quando possibile usando il flag -O in fase di compilazione.
- **Vettorizzazione esplicita**: si usano esplicitamente istruzioni che coinvolgono i registri vettorizzati.

## Kernel programming basics:

Vediamo le basi con cui si può costruire tutto a livello kernel.

Andiamo a studiare la superficie che delimita lo spazio user level rispetto a quello kernel.

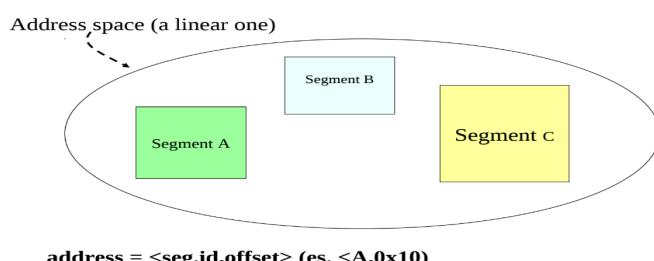
Un thread vede tutta la memoria, sia la parte user che quella kernel, ma può accedervi o meno a seconda che sia o no in kernel mode; capiremo come questo viene gestito in maniera sicura.

### **Indirizzamento lineare:**

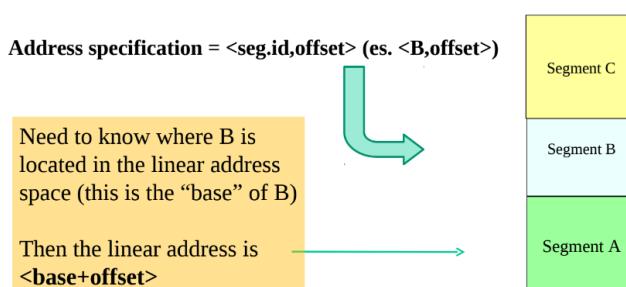
Il thread che esegue è associato ad un contenitore di memoria. Si può andare in qualsiasi punto del contenitore utilizzando un offset (indirizzo lineare).

### **Segmentazione:**

Quando lavoriamo su un sistema moderno c'è anche un altro approccio per accedere alla memoria: la segmentazione.

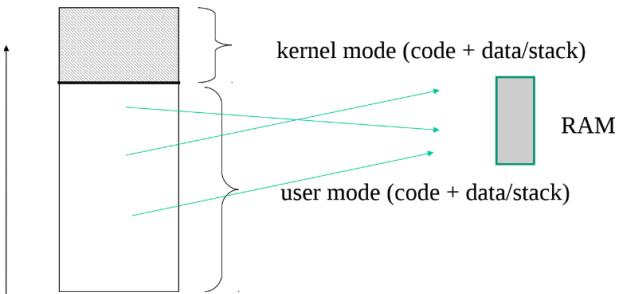


L'address space è suddiviso in segmenti. Un indirizzo è dato dall'identificatore del segmento e un offset per spiazzarsi all'interno dello specifico segmento. La somma è un indirizzo lineare. Anche dal punto di vista del processo abbiamo questa visione, ma non risulta distante dalla linearizzazione, in quanto i segmenti formano un contenitore di memoria lineare.



Il contenitore è logico, a esso corrisponderà uno spazio di memoria fisica (indirizzo fisico), tramite paginazione.

Si dispone di una page table che mappa ciascuna pagina logica dell'address space in una pagina di memoria fisica all'interno della RAM.



Linear addressing + mapping to actual storage (if existing)

Nei processori moderni si supporta in maniera efficiente la segmentazione insieme alla *paginazione*; 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.

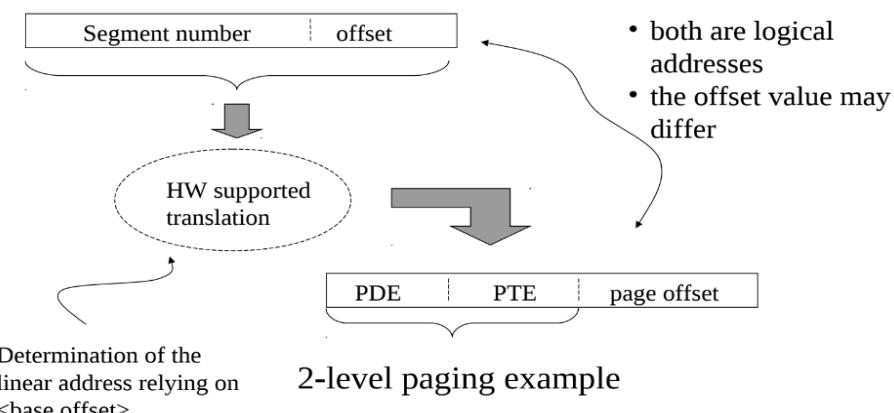
La segmentazione mi aiuta a organizzare la memoria logica in aree dedicate a scopi diversi, ma questo serve maggiormente al programmatore, e riproporre questa suddivisione sulla memoria potrebbe comportare degli sprechi, e qui interviene la paginazione, in cui dividiamo i segmenti in pagine (più piccole) e le organizziamo meglio in memoria fisica.

Se non specifico il segmento, se ne usa uno di default per facilitare la programmazione.

Lo schema del mapping prevede:

*Segmentazione → Linearizzazione → Paginazione → Indirizzo fisico*

La paginazione può essere anche a più livelli, qui sotto vediamo un esempio a due livelli:



L'address space viene quindi diviso in *sezioni*, ciascuna delle quali è suddivisa in *pagine*.

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      // usiamo un segmento per accedere in memoria
push %rbx              // usiamo un segmento per sfruttare lo stack
```

Usiamo anche il segmento associato a *text*, dove si muove il program counter. Il tutto è *implicito*.

### **Supporti alla segmentazione:**

I processori di sistema, ossia quelli che sono orientati a ospitare un sistema operativo, fanno affidamento alle componenti hardware per un accesso veloce e trasparente alla segmentazione.

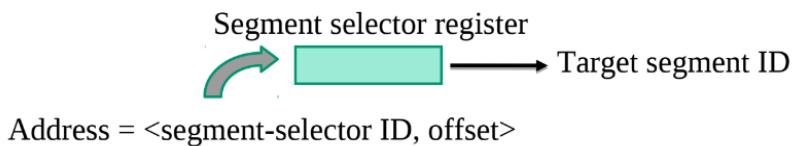
In particolare si usano *registri CPU* e *tabelle di memoria* puntate dai registri. Possono puntare sia in memoria fisica sia in memoria logica, in quest'ultimo caso serve la paginazione per mappare in RAM.

### **Selettor di segmento/registro di segmento:**

Quando esprimo un indirizzo in realtà non esprimo l'ID del segmento da accedere ma il *nome del registro*. Tale registro è detto selettor di segmento, a cui si applica l'offset. In x86 si hanno sei registri di segmenti per CPU. (Quindi, nel selettor di segmento, ci metto il nome del registro associato al segmento di interesse). E' tipicamente il kernel che lo popola.

Riprogrammando il selettor di segmento ho molta flessibilità perché è sufficiente reimpostarlo per toccare un'altra zona di memoria.

Accedendo in momenti diversi lo stesso selettor possiamo accedere a diversi identificatori di segmenti.



### **Supporto alla segmentazione in x86:**

- Real mode:

Si hanno registri di segmento in cui si possono scrivere gli *id dei segmenti*.

*Non c'è paginazione*, indirizzo lineare e fisico coincidono, ciò vuol dire che l'indirizzo fisico è direttamente accessibile: `id_segmento*16+offset`. Proprio perché non c'è paginazione, i segmenti devono essere aree contigue in memoria fisica. I registri di segmento e general-purpose (per mantenere l'offset) sono a 16 bit. A seconda di come si scrive il registro di segmento si ha una visione differente della memoria. Non ci sono bit per proteggere gli accessi al registro di segmento. Non c'è separazione tra user e kernel mode.

- Protected mode:

I registri selettori sono a 16 bit: 13 per l'id del segmento, 3 sono bit di controllo. Registri general-purpose a 32 bit per l'offset, quindi si possono usare 4 Gigabyte di memoria.

Non solo si identifica la zona, ma si controlla anche che un'azione su una zona sia o meno lecita. C'è anche una *tabella di segmento "table"* che tiene traccia dell'indirizzo base di ciascun segmento. Se non uso paginazione, tale indirizzo è quello in memoria fisica, se la uso, questo è un indirizzo logico.

`<segment, offset> → table[segment].base+offset`

- Long mode:

Per ottenere l'indirizzo lineare si utilizza la stessa tecnica del protected mode, con registri selettori a 16 bit (13 per id, 3 per controllo) e registri general-purpose a 64 bit (per mantenere l'offset, però se ne usano 48). Se il nostro processore non è impostato in long mode opera in protected mode.

Sono permessi fino a  $2^{48}$  byte di memoria lineare.

### **Tabella dei descrittori:**

Sono tabelle che tengono traccia degli indirizzi base dei segmenti.

Ci sono due tipologie di tabelle: **GDT** (global descriptor table) e **LDT** (local descriptor table).

La CPU può puntare al massimo *due* di queste tabelle, ma non sono due totali, ne posso avere molteplici in memoria.

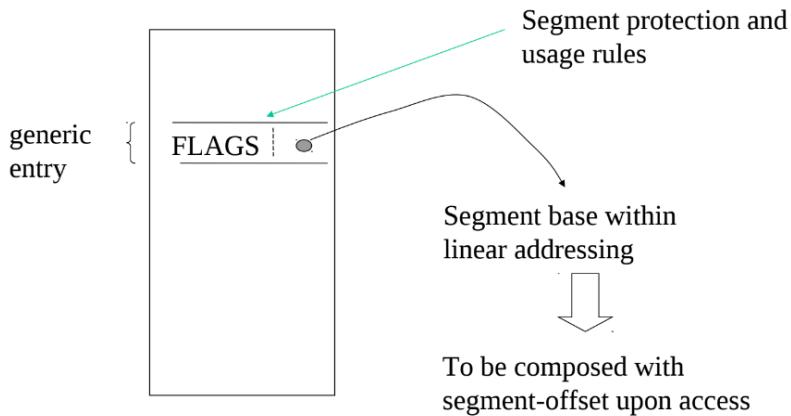
La GDT determina dove, all'interno dello spazio di indirizzamento lineare, sono collocati (almeno) i *segmenti* che afferiscono alle locazioni di livello kernel. E' una per CPU.

La LDT determina dove, all'interno dello spazio di indirizzamento lineare, sono collocati i segmenti relativi alla parte user dell'applicazione, se ciò non viene fatto dalla GDT.

Prima, col fatto che l'indirizzo lineare mappava direttamente in memoria fisica, la LDT era utile per cambiare vista della memoria e flusso di esecuzione per i thread, facendoli puntare a LDT diverse.

In sistemi moderni, si considera solo la GDT, che a ogni istante di tempo dice, per un thread in esercizio sulla CPU, esattamente ogni segmento toccabile dai selettori dei registri usati dal thread, quindi cambiano le entry in base al thread. Avendo tipicamente indirizzi logici, posso mappare tali indirizzi in memoria fisica usando *page table*, senza usare LDT. Quindi ogni thread vede segmenti diversi, i quali vengono mappati da una page table univoca (perché thread hanno stesso address space), facendoli accedere ad aree diverse.

Per una entry la GDT mantiene la base del segmento e dei flag di protezione per quest'ultimo. Quindi il segment ID contenuto nei registri di segmento viene usato come indice per identificare la entry della tabella dove è contenuta la base del segmento a cui si somma offset.



Il flag rappresenta le informazioni di controllo.

#### **Segmentazione e paginazione:**

La segmentazione è nata per regolamentare i permessi di accesso al codice, mentre la paginazione è nata per migliorare l'utilizzo della memoria fisica andando a risolvere le problematiche legate alla frammentazione esterna.

Lavorando con segmentazione possiamo avere segmenti molto ampi in memoria, mentre lavorando con paginazione si può avere grana più fine, non devo portare tutto il segmento in memoria!

Avere una copia privata di una pagina anziché di un segmento abbattere i costi.

La segmentazione mi permette di realizzare la seguente cosa:

Se ho due thread che lavorano sulla stessa funzione e voglio che vadano in zone di memoria differenti, ciò non si può fare con la paginazione, perché la tabella delle pagine è unica per tutti i thread di un processo (condividono stesso address space), ma si può fare con la segmentazione, associando un puntatore a diversi registri selettori a seconda del contesto di esecuzione.

Oss: La segmentazione è stata introdotta dopo che il supporto alla paginazione era già presente.

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.

#### **Modello di protezione basato su segmentazione:**

Ogni segmento è associato ad un livello di protezione, segnato un numero intero  $h$  che indica il suo livello di protezione (di privilegio). 0 è il livello di protezione massimo e se  $h$  aumenta, il livello di protezione decresce. Ciascuna routine (e ciascuna istruzione) assume il livello di protezione del segmento a cui appartiene.

Ciascuna routine  $r1$  avente un livello di protezione pari ad  $h$  può invocare una qualsiasi altra routine  $r2$  col medesimo livello di protezione  $h$ , sia se  $r1$  e  $r2$  appartengono allo stesso segmento (*intra-segment jump*), sia se non vi appartengono (*cross-segment jump*).

Si può sempre saltare dal livello di privilegio  $h$  a un livello di privilegio inferiore  $h + i$  (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**. I gates rappresentano delle porte di accesso ad un segmento, espresso come `<seg.id, offset>`; se si viene da un livello di protezione ammissibile (identificato da un massimo livello  $h+i$  a partire dal quale è possibile accedere) da un gate si può entrare. Non tutti i gates sono uguali.

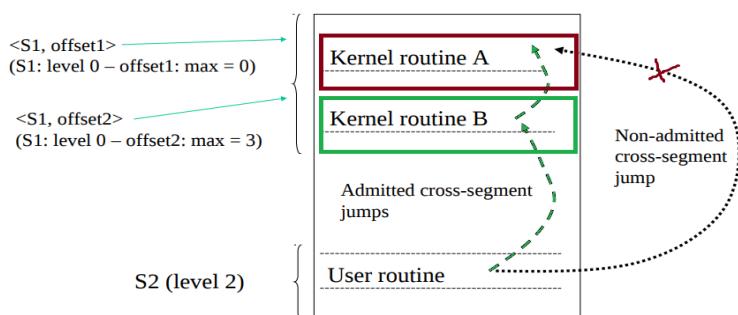
Il livello di protezione della routine corrisponde al livello di protezione del segmento su cui si trova. All'entrata del gate posso avere operazioni preliminari diverse.

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

- Gestori degli interrupt (asincrone): Ad esempio da livello user a livello Kernel uso un gate.
- Trap software (sincrone): Possono essere delle eccezioni (e.g. page fault) o delle system call.

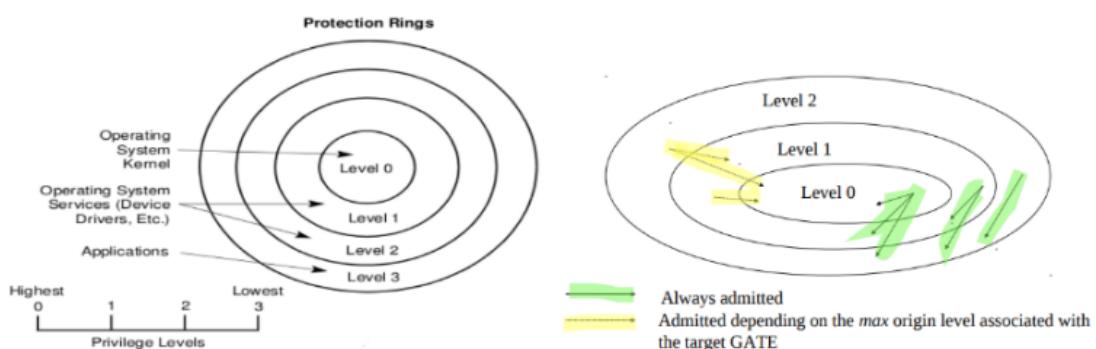
Tutto dipende da ciò che c'è nella GDT. Per codificare questi livelli ci servono due bit, che sono proprio i due bit dei tre bit di protezione visti nelle pagine prima.

### Esempio:



- Il kernel è su un segmento diverso rispetto ad user, la kernel routine A ha segmento 0 e un certo offset, il massimo privilegio richiesto è 0. User non può andarci, essendo livello 2.
- Il gate per kernel routine B ha invece come massimo livello di privilegio 3, user può usarlo, essendo livello 2 (cross-segmento da s2 a s1). Salendo in `<s1, offset2>`, user può andare anche in kernel routine A (intra-segmento, da s1 a s1).

### Implementazione su x86:



Su x86 si hanno quattro livelli di protezione (2 bit), sono gli ultimi due bit di controllo del selettori di segmento.

Abbiamo 6 registri selettori di segmento:

- CS code segment: indica il segmento di memoria contenente il codice che stiamo correntemente utilizzando in CPU.
- SS stack segment: indica qual è il segmento di memoria che ospita lo stack su cui stiamo correntemente lavorando.

- *DS data segment register*: indica qual è il segmento dati corrente, cioè quello da 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
- *FS e GS data segment register*: usato solo se previsto dal programmatore. FS usato per TLS, GS è per-cpu-memory.

La loro struttura include:

- 13 bit per l'indice dell'entry GDT/LDT associata al segmento.
- Bit successivo *Table Indicator*, per dire se usiamo GDT o LDT.
- Ultimi due bit per *Requestor Privilege Level – RPL*, livello protezione corrente.

Per CS l'RPL si chiama CPL, ovvero Current Privilege Level.

In x86 CS,DS,ES e FS puntano tutti allo stesso base address, 0x00

## Lez.10 (16/10/23)

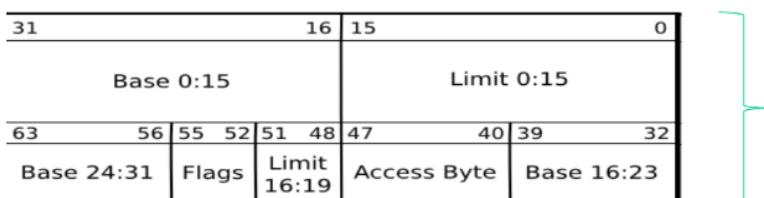
### GDT Entries in x86:

x86 può lavorare in modalità protected o long.

#### Protected mode

Ogni elemento della GDT ci dà informazioni sul segmento associato al rispettivo indice.

In protected:

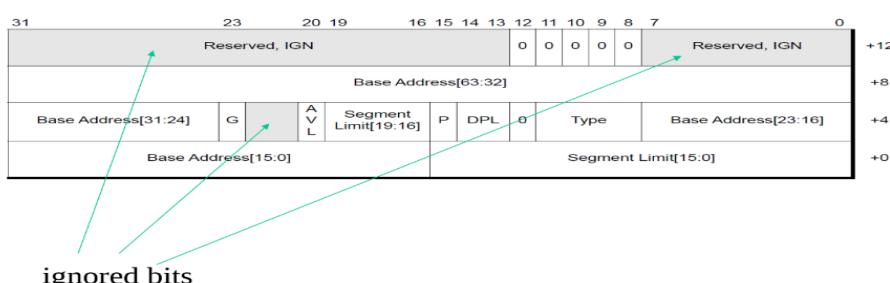


This directly supports protected mode

- Base: indica l'indirizzo base del segmento puntato dalla entry. Abbiamo 16 bit in una zona e altri 16 in un'altra.
- 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.
- Access byte: qui si hanno i bit che descrivono la protezione del segmento. Tra questi rientrano i due *privilege bit*, 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

In long mode è stata aggiunta un'altra zona, per raddoppiare la taglia, è marcata come riservata, può essere usata solo a livello firmware. Sebbene siano 64 bit, se ne usano 48.



## Accesso alle GDT entries

È possibile accedere direttamente alla GDT via software sfruttando il gdtr register, che è un registro "packed" (registro che contiene più dati combinati al suo interno) in memoria, composto da due parti:

- La parte bassa tiene traccia dell'indirizzo lineare in cui si trova la tabella.
- La parte alta indica la taglia della tabella, cioè numero di elementi.

### Esempio:

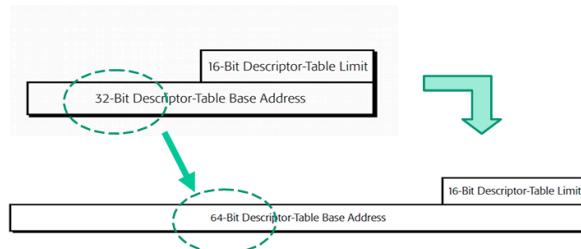
GDT-[AND SEGMENTS/gdt.c](#)

Dato che è un registro packed in memoria bisogna farne lo store in una struttura packed.

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:

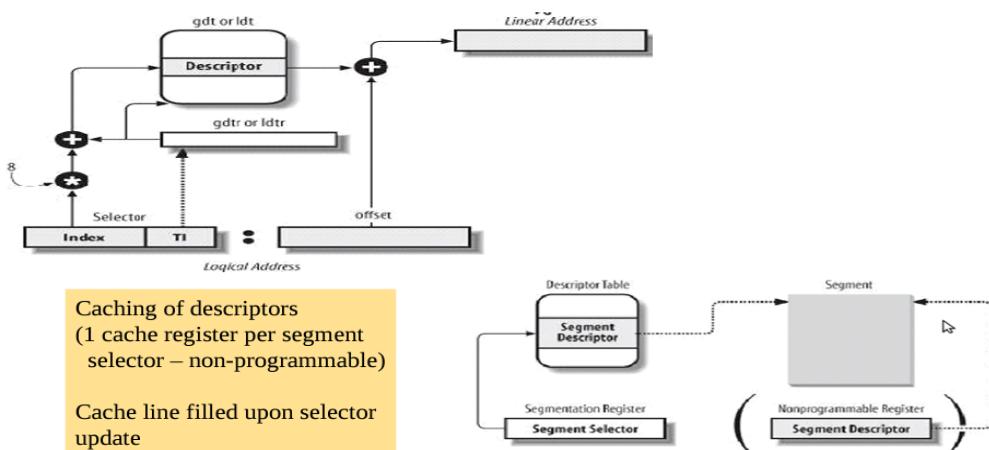
Oss: in un SO moderno c'è una GDT per ciascuna CPU, alcuni elementi possono contenere valori diversi, ciò mi permette di avere viste diverse.

### Estensione del GDTR in long mode:



### Schema di accesso alle GDT entries, livello hardware:

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



Ciò però non ci piace perché introduce parecchio overhead di esecuzione. Quello che si 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.

Attenzione: questa architettura non è programmabile.

## Segmenti in Linux

Nella tabella oltre alle basi abbiamo anche informazioni di protezione e bit che ci dice se un'entry è o meno valida. Molti partono dallo stesso indirizzo, rendendo più facile la programmazione sia per macchine che supportano la segmentazione sia non.

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

FS e GS sono descritti in alcune entries particolari: TLS#1, TLS#2 e TLS#3.

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.

Tipicamente è legato al contesto di esecuzione. Si sfrutta la segmentazione quando necessaria.

Quando carichiamo kernel code nel selettor di segmento cambia il livello di protezione.

## Aggiornamento del segment selector

CS mantiene il Current Privilege Level, ed è aggiornabile mediante dei *control flow variations* (realizzando dei salti tra segmenti) ma non è aggiornabile in modo esplicito. Gli altri *segment selector* sono aggiornabili esplicitamente se il nuovo Requestor Privilege Level RPL è peggiore del CPL corrente.

## GDT entries in Linux x86

<i>Linux's GDT</i>	<i>Segment Selectors</i>	<i>Linux's GDT</i>	<i>Segment Selectors</i>
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	APMBIOS 32-bit code	0xb0
TLS #2	0x3b	APMBIOS 16-bit code	0xb8
TLS #3	0x43	APMBIOS data	0xc0
reserved		not used	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

Lez.11 (18/10/23)

### **TSS (task state segment):**

È il segmento in cui viene descritto lo stato di un task, dove il **task è il thread correntemente in CPU**. Qui sono riportati gli snapshot del CPU-core in cui il task viene eseguito e le informazioni necessarie per supportare correttamente il modello ring. Questa zona è identificabile dal processore leggendo in uno specifico registro R.

Il base field del registro TSS dell'n-esimo processore punta all'n-esima entry del TSS tramite il segmento TSS in maniera trasparente. Ci sono più aree TSS, una per ogni CPU. Può essere acceduto solo in kernel mode.

#### *Cosa c'è nel TSS?*

Idealmente, nel TSS si può fare uno snapshot completo della CPU.

Si salva tutto ciò che è attualmente in CPU, ciò permette di fare uno swap di immagini di CPU (nelle CPU moderne context switch).

L'immagine da prendere si specificava aggiornando il riferimento R ad un altro TSS.

*Oggi questa possibilità non è considerata, quindi a cosa serve?*

Per mantenere lo stato del task corrente. Per evitare che thread diversi cadano nella stessa area, il segmento TSS non viene fissato a base 0.

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.

SS0-ESP0; SS1-ESP1; SS2-ESP2 sono i valori per aggiornare lo stack pointer e il selettore di segmento in base ai ring 0, 1 e 2.

E' un supporto adeguato per il modello a ring, a livelli diversi ho bisogno di stack area diverse.

Ciò avviene quando si fa l'aggiornamento del Context Switch.

Per il ring 3 non serve perché lo stack pointer del ring 3 viene salvato quando si cambia livello.

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

Although it could be ideally used for hardware based context switches, it is not in Linux/x86

It is essentially used for privilege level switches (e.g. access to kernel mode), based on stack differentiation

### Variante x86-64:

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

room for 64-bit  
stack pointers has been created  
sacrificing general registers  
snapshots



Su x86-64 dato che i registri sono a 64 bit alcune informazioni sono stateificate.

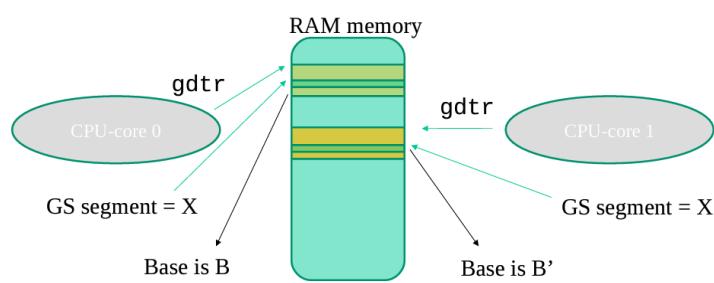
Il registro *packed TSS register* mantiene l'indirizzo lineare del TSS (senza accedere spesso alla GDT in caso di cambio priorità), e l'indice della entry del TSS nella GDT per identificarlo in RAM; Questo registro si può caricare con LTR- Load Task Register; la sorgente della load può essere un registro general purpose o un indirizzo di memoria.

### Replicazione della GDT:

Nei SO moderni ogni CPU ha la propria GDT in memoria. La replicazione permette di avere entries diverse nelle diverse GDT, ciò permette a threads che girano lo stesso codice di vedere zone diverse di memoria. Le GDT possono essere in zone diverse della RAM, soprattutto su un'architettura NUMA si può avere accesso efficiente in RAM.

Ogni *systemcall* risulta in un accesso alla GDT.

In questo modo non c'è bisogno di chiamare CPUID per capire quale porzione di memoria è dedicata alla CPU.



Same displacement within segment X seamlessly leads the two CPU-cores to access different linear addresses

### Per-cpu memory in Linux:

Ci sono zone riservate per-cpu nello spazio di indirizzamento lineare.

Per spiazzarsi in queste zone ci si basa sul segmento GS.

Ci sono macro per selezionare lo spiazzamento in GS e macro per implementare accessi in memoria sulla base dello spiazzamento selezionato.

Quando sviluppiamo moduli kernel possiamo definire variabili per-CPU.

Ne vediamo un esempio qui sotto:

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

Equivale a `mov ax, gs:[x]`

### **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, abbiamo un TLS per ciascun thread.

È basato sulla configurazione di diversi segmenti associati ai selettori FS e GS.

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 il thread 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 del thread, 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**.

Attenzione: non c'è garanzia che un thread continui a girare sulla stessa cpu, quindi non è corretto chiamare `this_cpu_read(x)`; ciò si verifica in caso di preemption; dovremo garantire che si continui a eseguire sulla stessa CPU.

GS ha un ruolo fondamentale quando operiamo in kernel mode, FS, invece, si usa user mode.

### **System calls per la gestione dei segmenti:**

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

- `int arch_prctl (int code, unsigned long addr) //set, passo indirizzo`
- `int arch_prctl (int code, unsigned long *addr) //get, ptr indirizzo`

Sottofunkzioni sono GET\_GS e GET\_FS, che ritornano dove si trovano FS e GS in memoria; per l'aggiornamento abbiamo SET\_FS e SET\_GS.

Con l'aggiornamento si cambiano informazioni in TLS, di conseguenza vengono modificati anche i registri. Con SET si passa l'indirizzo, con GET il puntatore all'indirizzo, per questo si hanno due segnature per la systemcall.

### **Registri di controllo su x86:**

Abbiamo 4 registri di controllo CR0-CR3.

- CR0: contiene informazioni basiche su come lavora il processore. Il 31-esimo bit indica se includere o escludere la paginazione. Gli altri bit danno informazioni diverse a seconda di come lavora il processore.
- CR1: riservato all'operatività interna del processore.
- CR2: se un accesso in memoria virtuale non è riuscito, qui viene scritto l'indirizzo che ha causato l'eccezione (e.g. in caso di page fault).

Lo utilizza il kernel: dopo la trap prende il controllo e legge questo registro.

- CR3: Se CR0 supporta memoria virtuale e paginazione, allora ho puntatore che referenzia la page table in memoria fisica. Quindi ho l'indirizzo fisico della page table.

Attenzione: un page fault può essere causato anche da codice kernel, sempre a causa di richieste user.

## Interrupt e trap

NB: L'uso di GATE, INT, fa tutto parte del concetto di **slow system call**.

L'esecuzione Context Switch può avvenire a causa di un interrupt o di una trap.

Il concetto di interruzione (*interrupt*) a seguito di interazione con dispositivi è asincrono, mentre il concetto di eccezione (*trap*) è sincrono. La **trap** è generabile da:

- Operazione illegale: istruzione non committed
- Chiamata interrupt in modo sincrono: istruzione "INT", quando si passa controllo al kernel è committed. Questa istruzione deve utilizzare necessariamente un gate (ring3 → ring0).

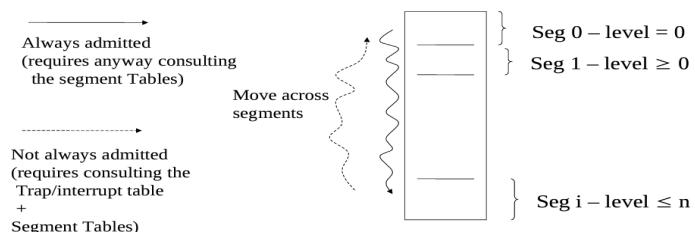
Le trap sono state usate storicamente come modo per accedere al kernel mode (tramite system calls).

Il kernel mantiene una tabella per trap e interrupt, Trap Interrupt Table TIT. Ogni entry ha un GATE descriptor (<segment.id, offset>) che indica le informazioni associate al GATE e il livello di protezione.

L'ID del segmento deve essere associato in modo che il processore aggiorni il descrittore. Lo stesso GATE può apparire in più entry, di solito con livelli di protezione diversi.

In x86 la variazione del flusso è implementata in tre modi:

- **Intra-segment:** quando c'è un jump tradizionale, la `call` è sullo stesso segmento, ad un altro offset del segmento corrente.
- **Cross-segment:** quando c'è un long jump, il processore controlla che la GDT lavori in un livello di protezione raggiungibile da quello corrente.
- **Cross-segment via gate:** quando c'è un'istruzione di **trap** (INT) e specifico anche quale gate si vuole usare. *Ogni system call passa inizialmente attraverso la porta 0x80*. Richiede di consultare la TIT e la tabella dei segmenti e aggiornare i registri di segmentazione.



## **Dettagli sui gate in x86:**

Su x86 la TIT si chiama *Interrupt Descriptor Table (IDT)*.

La posizione in memoria di questa tabella sta nel registro *idtr*.

Il puntatore rappresenta un indirizzo logico, poi si va sulla tabella delle pagine per conoscere quello fisico, in *idtr* c'è anche scritta la taglia della tabella.

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, perché, 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.

## **Strutture dati per il dispatching delle systemcalls:**

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. Viene impegnata un'unica istanza della IDT per tutte le system call che il kernel mette a disposizione.

Quando si chiama una systemcall si passa attraverso un **gate** per entrare in *kernel mode*.

Si mantiene la **system call table**, che è una struttura dati software che, per ogni entry, mantiene l'indirizzo di memoria di una specifica system call. Mostra solo le system call chiamabili.

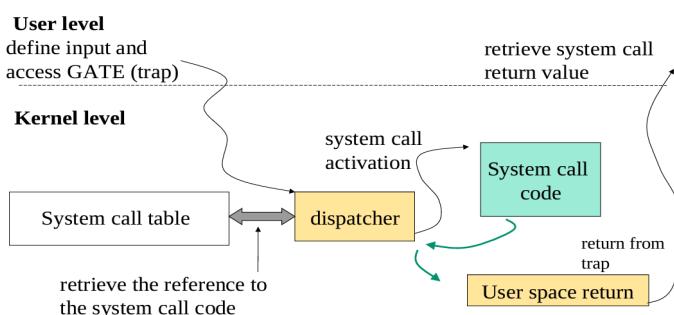
Il **dispatcher** delle system calls decide quale entry utilizzare e, di conseguenza, quale blocco di codice kernel chiamare a seconda dell'applicazione richiesta.

Prima di chiamare questa funzione scriviamo l'indice della systemcall in uno dei registri ABI (application binary interface, corrispondente ad API a livello di codice macchina).

Il supporto al meccanismo di indicizzazione dei servizi è standardizzato.

Il dispatcher esegue delle operazioni preliminari prima di chiamare la system call tra cui operazioni di sicurezza (in questo modo si disaccoppia quello che il kernel può vedere del livello user)

Dopo che la systemcall è stata eseguita inserirà il valore di ritorno in un registro di processore; restituirà il controllo al dispatcher che attiverà il blocco di codice che torna il controllo alla modalità user: viene invocata l'istruzione **ReTurn from Interrupt (RTI)** e si fornisce al chiamante il valore di ritorno della system call.



Attenzione: su x86\_64 non c'è più il concetto di trap.

## **Atomicità dell'esecuzione del dispatcher:**

Il dispatcher di default è interrompibile.

Per proteggersi da eventuali interrupt, è possibile fare uso di due istruzioni:

- CLI (clear interrupt bit), la quale elimina la possibilità per gli interrupt di interrompere l'esecuzione del dispatcher.
- STI (set interrupt bit), la quale ripristina l'opzione precedente. Ciò potrebbe non essere sufficiente per l'atomicità dell'esecuzione del dispatcher: nelle architetture multiprocessore, si è soggetti a un accesso concorrente ai dati. Per ovviare a questo problema, si può ricorrere alle soluzioni classiche di *spinlock* o *Compare And Swap*.

Prima dell'attivazione del dispatcher si è in modalità user, dunque non c'è alcuna necessità di non-interrompibilità.

## **Componenti del software per le systemcalls:**

Cosa succede quando si chiama una system call?

Dal punto di vista utente:

- si forniscono i parametri di input al GATE e alla systemcall corrente,
- Si attiva il GATE, questo task è machine dependent perché il trigger del GATE avviene tramite codice ASM.
- si ottiene il valore di ritorno.

Dal punto di vista del kernel abbiamo:

- dispatcher
- system call table

- codice della systemcall corrente

## Aggiunta di una nuova system call

Posso far riferimento a tre approcci:

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

In Linux i due schemi di indicizzazione sono: `unistd32.h` e `unistd64.h`; si utilizza l'uno piuttosto che l'altro a seconda del dispatcher attivato.

A ogni servizio del kernel posso arrivare attraverso entrambe le vie, tramite entrambe le indicizzazioni. Averle entrambe è utile per retro compatibilità.

Per lo schema di indicizzazione `unistd64.h`, si fa uso di un'altra system call table e di un altro dispatcher, il quale però non è accessibile tramite la IDT e, quindi, tramite l'istruzione INT.

Il software nuovo deve usare `unistd64.h`, con nuove modalità per accedere ai servizi, mentre quello vecchio usa le trap.

### **Formati delle systemcalls prestabiliti:**

Come vengono specificati i parametri di input alle systemcalls? Si passano tramite registri di CPU, non memoria, perché il livello user e kernel possono condividere soltanto l'immagine di CPU.

Abbiamo headers con delle macro che permettono di passarle come parametri e quando il compilatore invoca la macro c'è dentro tutto ciò che serve per costruire una funzione con dentro del codice ASM (in `int0x80` su Unix) che utilizza l'indicizzazione corretta.

La macro costruisce automaticamente l'oggetto, che poi può essere compilato, linkato e invocato.

Per le systemcalls in Unix al massimo sono supportati sei parametri di input e uno per il dispatcher su un processore in modalità protected, questo perché ci sono solo sei registri general purpose, questo limite è allineato anche ai limiti dell'hardware.

### **syscall:**

Genera una systemcall con nome "name" e con valore di ritorno di tipo "type".

Permette di creare systemcalls che non accettano parametri di input.

Prima di invocare `int 0x80` si carica nel registro `%eax` il parametro `__NR_##name` per indicare l'indice della system call da chiamare. Il valore di ritorno viene scritto in `__res`.

`__syscall_return` è un'attività one shot; se `__res`, ovvero il valore di ritorno, è compreso in un certo range abbiamo un errore, impostiamo `ERRNO` con valori negativi, mentre systemcall `__res` è pari a -1. Un esempio classico di questo tipo di systemcall è la `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

Dopo il codice assembly abbiamo l'esecuzione del task `__syscall_return(type, __res)` in cui si gestisce il valore di ritorno per capire come trattare la variabile ERRNO.

Ci sono anche tasks che devono essere eseguiti prima del codice assembly.

```

#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]

Perché c'è un `do...while(0)`? Si svolge un'attività di tipo all or nothing., passo forzatamente per il `return`. Permette di definire in una macro un'attività multi-statement, garantisce che il contenuto della macro sia svolto come un'unica attività e permette di usare degli `if` all'interno della macro.

### `_syscall1:`

Accetta due parametri, il primo è l'indice della systemcall che il dispatcher deve invocare, l'altro il parametro da passare alla systemcall.

`type1` è il tipo del parametro da passare al kernel, `arg1` il valore.

`__NR_##name` permette di specificare il valore di un'altra macro.

```

#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

Come faccio a passare 7 parametri (considerando l'indice della systemcall per il dispatcher) se ho soltanto sei registri general purpose?

```

__asm__ volatile ("push %%ebp ; movl %%eax, %%ebp ; movl %1,%%eax ; \
                  int $0x80 ; pop %%ebp")

```

"push %%ebp" per salvare il settimo parametro sullo stack.

In %%eax devo aver scritto il valore da passare al kernel, quello da inserire in %%ebp, alla fine rimetto a posto %%ebp.

In %%eax ho caricato arg6, lo vedo perché "0" in questa notazione è come se ci fosse scritto "a".

Poi lo muovo in ebp.

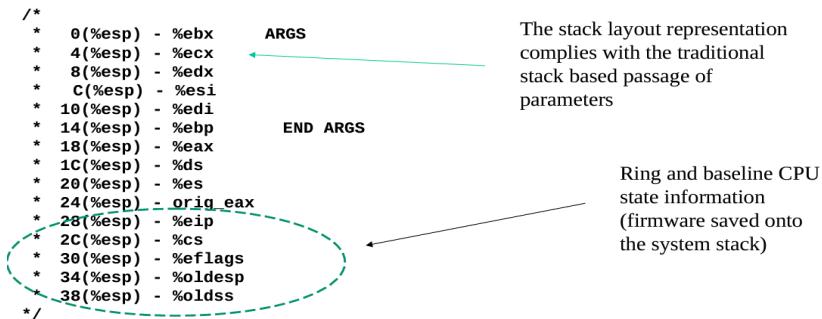
Quando si fa "movl %1, eax" identifico una locazione di memoria intera scelta dal compilatore su cui deve fluire il numero di systemcall da chiamare prima del blocco di codice.

Possiamo lavorare in maniera safe usando %ebp, questo registro a livello user non serve a molto.

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

### **Convenzioni di chiamata per systemcalls in UNISTD\_32:**

Quando lavoriamo su uno schema unistd\_32 la systemcall dove trova i parametri? Dipende dall'ABI, da come si è deciso di comunicare questa informazione al sottosistema che implementa questa systemcall nel kernel. L'ABI prevede che le informazioni siano allineate allo stack come segue (quindi troverò così lo stack del kernel):



A mettere le informazioni nella stack area è il dispatcher, perché queste informazioni devono essere nell'ABI quando parte il sottosistema, che è dispatchato dal dispatcher.

Attenzione: il dispatcher è uno solo, tutti i servizi sono dispatchati da questo, quindi queste cose devono essere messe nello stack esattamente così a prescindere dal sottoservizio che verrà eseguito.

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.

Questo discorso dell'allineamento dello stack visibile a ogni sottosistema del kernel permette di dire che a livello user si ha la possibilità di compattare all'interno dei registri informazioni che saranno rese visibili a questo software per:

1. falsare l'esecuzione di questo software.
2. far sì che chi sta falsificando l'esecuzione possa rubare queste informazioni.

Un altro aspetto interessante è che ci sono informazioni in più rispetto ai registri scritti a livello applicativo, essi sono stati pushati sullo stack dal dispatcher, chi ha scritto le altre informazioni sulla stack area?

Quando abbiamo codice user che chiama codice kernel e usa CS bisogna:

1. Capire che se stiamo cambiando la regola di segmentazione poi andrà ripristinata quella originale.
2. Per cambiare la regola ci sono le tabelle GDT e tutte le altre strutture già introdotte, tutto ciò il dispatcher non lo guarda, se chiamiamo 0x80 vengono salvate qui dentro dal firmware del processore.

Quindi il processore quando cambia regola di segmentazione si salva la regola da ripristinare, lo stack segment da rimettere a posto e così via. Il lavoro è fatto in maniera combinata dal dispatcher e dal firmware.

## **Convenzioni di chiamata per systemcalls in UNISTD\_64:**

Quando usiamo la regola di indicizzazione a 64 bit abbiamo che il modulo del kernel che implementa la systemcall deve conoscere le informazioni per eseguire seguendo tale struttura:

```

/*
 * 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.
 */

```

*Il software trova le informazioni nei registri di processore e non sulla stack area. Ciò evita tanti accessi in memoria.* In x86-64 lavoriamo primariamente con un'architettura di dispatching completamente diversa che non utilizza i GATE.

E' possibile avere un unico sottosistema del kernel che è in grado di trovare i parametri nella stack area o nei registri a seconda dell'indicizzazione? Evidentemente no, il sottosistema è uno e prenderà i parametri sempre allo stesso modo. Ciò che è stato fatto nell'architettura è quanto segue:

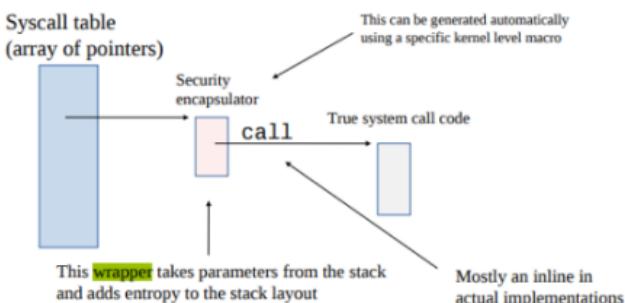
- Come detto, per chiamate UNISTD\_64 il dispatcher mette le informazioni nei registri di processore.
- Se arriva una chiamata con **trap**, INT 0x80, il dispatcher mette le informazioni nella stack area e fa partire un oggetto intermedio (**wrapper**) che prende i parametri dalla stack area e li mette nei registri di processore. Il sottosistema va a prendere i parametri di registri.

Il wrapper può introdurre del rumore (informazioni dummy) sullo stack per impedire che il software poi osservi la stack area e possa risalire allo snapshot di cpu.

### **Dettagli sul passaggio di parametri**

Una volta preso il controllo il dispatcher fa uno snapshot completo della cpu e lo va a inserire nello stack a 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 (stack o, in alcuni casi, registri). 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.

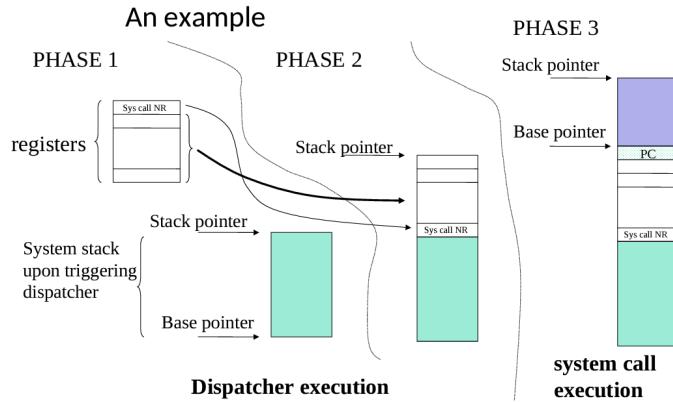
Ci sono implementazioni in cui il dispatcher funge anche da wrapper. *Dispatcher = gate di accesso al kernel.*



### Esempio:

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.



### Allineamento:

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`.

### Come si introducono nuove systemcalls?

Fornire un file C che contenga la definizione numerica della nuova system call e la macro per creare il modulo associato alla nuova system call.

```
#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);
```

### Lez.13 (23/10/23)

#### Implicazioni sulle performance dell'uso di 0x80 (slow system call):

Usare la trap ha un costo notevole, in quanto ho i seguenti accessi alle tabelle in memoria:

1. Accesso IDT che fornisce informazioni sul gate da utilizzare.
2. Accesso alla GDT per ottenere il segmento CS kernel.
3. Accesso alla GDT per reperire il segmento TSS del thread corrente, il quale serve per recuperare lo stack di livello kernel (Nel TSS ho stack livello 0 e 3).
4. Attesa di molti cicli di clock per ottenere i dati.
5. Ritardo generato dall'architettura NUMA.

Ciò non ci piace, soprattutto per quanto riguarda alcuni servizi, ad esempio il servizio "gettimeofday" è mappato su una system call; per prendere il cronometro passa un certo numero di cicli di clock e dipende da chi sta chiamando questa system call, quindi qualcuno otterrà un tempo più preciso, qualcuno meno.

### Fast System call path

Nei sistemi x86 è stata data la possibilità di passare il controllo al kernel senza effettuare alcun accesso in memoria e, quindi, senza accedere mai alla IDT e alla GDT, mantenendo nel processore le informazioni per fare questa transizione.

Vengono introdotti dei registri addizionali, **MSR (Model Specific Register)**, in cui inserire CS kernel e il suo entry point, il segmento DS e lo stack kernel. Le informazioni sono in cpu quindi non si deve fare una trap.

Questi registri sono preconfigurati per contenere queste informazioni, poi quando servono queste vengono installate su registri operativi di processore. Non paghiamo ritardi. Ci sono due supporti “fast” per entrare in kernel mode: *sysenter* a 32 bit, *syscall* a 64 bit.

Solo il salvataggio della stack area richiede accesso in memoria, ma in *x86\_64* lo switch della stack area viene fatto solo se necessario.

In entrambi i modi CS, EIP e SS si caricano da MSR, con ESP/RSP notiamo che nel caso di 64 bit lo switch stack non viene eseguito a livello hardware ma software se serve.

- CS register set to
  - the value of SYSENTER\_CS\_MSR for 32 bits
  - another bitmask taken from IA32\_STAR\_MSR for 64 bits
- EIP register set to
  - the value of SYSENTER\_EIP\_MSR for 32 bits
  - IA32\_LSTAR\_MSR for 64 bits
- SS register set to
  - the sum of 8 plus the value in SYSENTER\_CS\_MSR for 32 bits
  - another bitmask taken from IA32\_STAR\_MSR for 64 bits
- ESP/RSP register set to
  - the value of SYSENTER\_ESP\_MSR for 32 bits
  - nothing is done for 64 bits

Ci sono due supporti “fast” per uscire da kernel mode: *sysexit* a 32 bit, *sysret* a 64 bit.

- CS register set to
  - the sum of 16 plus the value in SYSENTER\_CS\_MSR for 32 bits
  - a bitmask from IA32\_STAR for 64 bits
- EIP register set to
  - the value contained in the EDX register for 32 bits
  - RCX for 64 bits
- SS register set to
  - the sum of 24 plus the value in SYSENTER\_CS\_MSR for 32 bits
  - a bitmask from IA32\_STAR for 64 bits
- ESP register set to
  - the value contained in the ECX register for 32 bits
  - nothing for 64 bits

RIP è l’indirizzo iniziale dove eseguiamo in kernel mode. Il registro viene prepopolato a livello kernel. È come un gate, il kernel prende il controllo esattamente a questo indirizzo.

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

Due percorsi implicano due gate diversi e due indicizzazioni diverse.

### **Aggiornamento dei registri MSR**

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

### **Costrutto *syscall()***

```

#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);
}

```

In questo esempio vediamo come tutto ciò di cui abbiamo parlato a livello kernel viene visto a livello user. Viene introdotta una nuova funzione: `syscall()`, che fa parte della libreria `stdlib.h`.

Bisogna indicare quale system call si vuole chiamare, mediante il suo codice.

L'implementazione effettiva dipende se si usa *fast* o *slow system call path*, ovvero chiamata a `int 0x80` o `syscall`.

### **Virtual Dynamic Shared Object:**

Si ha un VDSO per ogni processo. Trattasi di libreria condivisa.

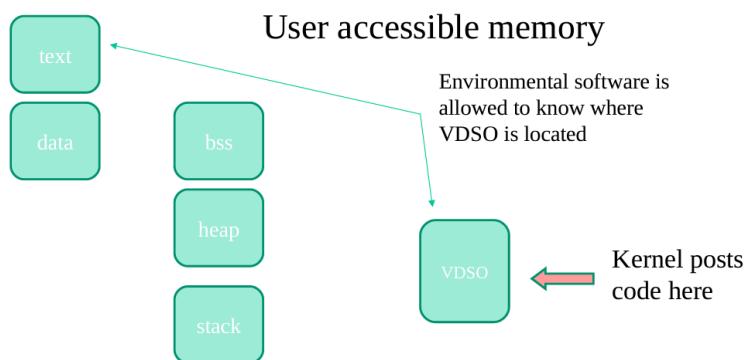
E' una zona di memoria virtuale (una o più pagine create dal kernel) condivisa con il kernel, che contiene il codice che può essere eseguito a seguito di richieste utente.

Si può sapere dove si trova questa zona mediante delle funzioni particolari:

`getauxval(AT_SYSINFO_EHDR)`, dove `AT_SYSINFO_EHDR` è il comando per sapere dove si trova il VDSO.

Il VDSO non si trova sempre nello stesso punto; all'attivazione di *ogni processo* (uno per ogni processo) viene randomizzato per complicare la vita degli attaccanti.

Usando questa zona implicitamente si usa fast system call path.



Il VDSO è nato per le system calls per ottenere i clocks.

### **Effetti sulle performance:**

- Sfruttamento dell'indirizzamento lineare: VDSO sfrutta l'indirizzamento lineare, tipico dei gestori di memoria del sistema operativo, *per bypassare la segmentazione* e le operazioni correlate. Ciò semplifica gli accessi alla memoria e riduce l'overhead associato alla segmentazione.
- Riduzione degli accessi alla memoria: Grazie all'utilizzo dell'indirizzamento lineare e al bypass della segmentazione, VDSO riduce il numero di accessi alla memoria necessari per eseguire le chiamate di sistema.
- Riduzione dei cicli di clock: Studi dimostrano che l'utilizzo di VDSO può ridurre i cicli di clock necessari per le chiamate di sistema fino al 75%.

### **La system call table:**

A livello kernel per introdurre un nuovo servizio bisogna operare sulla system call table.

Essa mantiene una serie di entries, ognuna delle quali costituisce un puntatore nel kernel. Non ci si accede direttamente, ma è il dispatcher ad accedervi altrimenti dovrei avere centinaia di servizi indirizzati in due modi.

Non possiamo modificare la tabella per avere più spazio quando compiliamo il kernel perchè ci sono altre strutture allineate in memoria e ci potrebbe essere sovrapposizione su un certo indirizzo.

Queste informazioni ci sono perchè a tempo di compilazione ho necessità di informazioni su indirizzi logici, ad esempio per gli MSR. Ci sono altri approcci:

Durante lo sviluppo nel kernel Linux si mantenevano delle entries libere per i nuovi servizi; attualmente non ci sono più; ogni zona blank è utilizzata da chi ne ha bisogno. Devo sovrascrivere qualche entry, è più tracciabile perchè si riutilizzano entries sparse. Questo è possibile perché alcune entries sono mantenute per servizi progettati ma non implementati. Restano dei "buchi", vanno identificati e va trovata la tabella in memoria.

Trovare la tabelle in memoria non è un problema banale: gli indirizzi sono randomizzati ed alcuni servizi non utilizzati non sono trovabili, vengono nascosti per motivi di sicurezza.

Le entries inutilizzate puntano al codice "nessun servizio" (che è diverso da null!): `sys_ni_syscall`.

Il kernel 2.4 mette le system call in `entry.s`. Il kernel 2.6 in `syscall-table32.s`.

Per le versioni 4.15 del kernel e UNISTD 64, la tabella è definita in: `syscall_64.c`

Attualmente, la table è prodotta a *compile time*, ed è un array in C 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. Inizialmente questi file avevano un formato .S, erano cioè scritti con delle direttive ASM preprocessore. 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)

## asmLinkage

A seconda dell'ABI i parametri sono o meno allineati alla stack area, ci sono delle direttive che permettono di specificare questo, in modo che il passaggio dei parametri avvenga in maniera compliant rispetto a ciò che si aspetta il dispatcher.

Il costrutto "`asmLinkage`" è una direttiva che sta ad indicare che la system call troverà i suoi parametri allineati sulla stack area: la sua invocazione viene fatta attraverso il dispatcher che allinea gli elementi sullo stack.

Questo va bene su un'architettura software in cui il dispatcher ha il solo compito del dispatching.

## Dispatcher kernel 2.4 unistd-32:

```

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

```

`SAVE_ALL` per allineare lo stack. Prende `%eax`, che contiene il numero del servizio, lo confronta con le system call, se ne matcha una la chiama, essa dovrà essere marcata come "asmLinkage", perchè i parametri sono sullo stack.

Il valore di ritorno si trova su %eax. Questo approccio non va più bene:  
Se passiamo un indice più grande della size della system call table, la zona di codice in giallo è vulnerabile a mis-predizione per salti condizionali se non è prevista sanitizzazione.

### **Dispatcher kernel 2.4 unistd-64 e swapgs:**

```

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

```

#define PDAREF(field) %gs:field

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

**Beware this!!!**

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

Come prima, ma l'indice della system call è su %rax. Prima di fare queste operazioni occorre farne altre preliminari. In particolare, usiamo swapgs per sostituire al GS corrente (quello di livello user) il GS alternativo (quello di livello kernel). In questo modo stiamo cambiando la visione della memoria.

Questo swap deve avvenire in diversi punti del sistema operativo, ad esempio in seguito ad una trap conseguente ad un memory fault: anche il kernel può avere dei faults, in questo caso si deve fare o no lo swap? Se si decide speculativamente si può sbagliare ad eseguire.

Sulla per-cpu memory ci sono le informazioni che sono necessarie per eseguire il thread, come ad esempio la posizione della stack area per quel thread. Qui serve lo switch perché la chiamata è sincrona.

Troviamo poi attività per allineare lo stack, quindi attività sugli argomenti.

Le due istruzioni successive invece, aggiornano la stack pointer (come abbiamo visto deve essere fatto a livello software se serve). Come prima, la parte in giallo può causare problemi.

### **Dispatcher kernel 4 e successivi:**

```

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.
 */
swapgs
/*
 * 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_CS           /* pt_regs->ss */
pushq    PER_CPU_VAR(rsp_scratch)  /* pt_regs->sp */
pushq    %r11              /* pt_regs->flags */
pushq    $USER_CS           /* pt_regs->cs */
pushq    %rcx              /* pt_regs->ip */
pushq    %rax              /* pt_regs->orig_ax */
GLOBAL(entry_SYSCALL_64_after_hwframe)
pushq    %rax
PUSH_AND_CLEAR_REGS rax=$ENOSYS
TRACE_IRQS_OFF
/* IRQs are off. */
movq    %rax, %rdi
movq    %rsp, %rsi
call    do_syscall_64
/* returns with IRQs disabled */
TRACE_IRQS_IRETQ
/* Try to use SYSENTER 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.
*/
#endif CONFIG_X86_64
_visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
{
    struct thread_info *ti;

    enter_from_user_mode();
    local_irq_enable();
    ti = current_thread_info();
    if (READ_ONCE(ti->flags) & _TIF_WORK_SYSCALL_ENTRY)
        nr = syscall_trace_enter(regs);

    /*
     * NB: Native and x32 syscalls are dispatched from the same
     * table. The only functional difference is the x32 bit in
     * regs->orig_ax, which changes the behavior of some syscalls.
     */
    nr &= SYSCALL_MASK;
    if (likely(nr < NR_syscalls)) {
        nr = array_index_nospec(nr, NR_syscalls);
        regs->ax = sys_call_table[nr](regs);
    }
    syscall_return_slowpath(regs);
}
#endif

```

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

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

Abbiamo due dispatchers, quello di primo livello non chiama direttamente la system call, ma va a chiamare il dispatcher di secondo livello o **stub**, “do\_syscall\_64”.

Cosa fa “do\_syscall\_64”? Mediante una maschera normalizza l’indice passato dall’utente, questo serve per evitare problemi inerenti alla mis-speculation, infatti se l’utente passa un indice fuori dalla system call table speculativamente si potrebbe accedere a qualsiasi punto, causando side-effects sfruttabili da un eventuale attaccante. Ciò viene fatto nella parte cerchiata.

Con la normalizzazione si evita ciò perché riporta all’entry sys\_ni\_syscall.

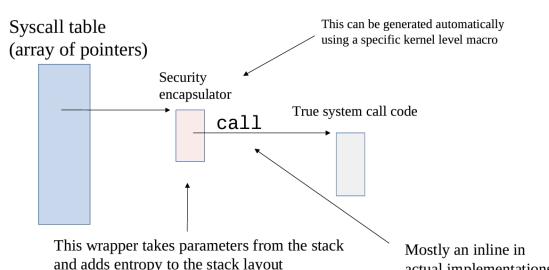
### Stack masking:

L’indirizzo che chiamiamo nella system call table chiama un intermediario, il quale prima applica offuscamento e poi invoca il vero e proprio codice della system call.

Questo oggetto intermedio (wrapper) può causare entropia nella stack area, ad esempio randomizzazione. L’oggetto “security encapsulator” è asmlinkage, il codice vero e proprio della system call no.

Il kernel Linux genera automaticamente questo oggetto intermedio quando sviluppiamo una nuova system call; si utilizzano delle macro di livello kernel per fare ciò.

In questo modo possiamo invocare anche system call senza parametri di input.



Le seguenti macro non solo generano la syscall ma anche il relativo wrapper: SYSCALL\_DEFINE0, SYSCALL\_DEFINE1, SYSCALL\_DEFINE2, SYSCALL\_DEFINE3...

Abbiamo, poi, SYSCALL\_DEFINEx che serve per system calls con numero di parametri arbitrario.

Quando si processa una macro vengono generati due blocchi di codice, quello per il wrapper e quello per la system call vera e propria; essi si chiamano rispettivamente \_\_x64\_sys\_name e \_\_do\_sys\_name o \_\_se\_sys\_name.

Il wrapper fa anche in modo che si possa usare anche unistd\_32, utilizzando la sanitizzazione (o normalizzazione) per pointers a 32 bit laddove sono a 64.

## Lez.14 (25/10/23)

### **Page table isolation (PTI):**

Nei dispatchers è stata introdotta una modifica legata all'attacco Meltdown.

E' stata introdotta "SWITCH\_TO\_KERNEL\_CR3", dove passiamo dalla page table user a kernel.

```

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

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

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

PUSH_AND_CLEAR_REGS rax=$-ENOSYS

TRACE_IRQS_OFF

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

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

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

```

Notiamo come il preambolo, rispetto alla funzione `syscall()`, sia più costoso rispetto alle implementazioni nelle vecchie versioni del kernel.

### **Attacco swaps:**

`swaps` è necessario in ogni entry point per lavorare sulla per-cpu memory usando il GS corretto.

Può tuttavia causare problemi di sicurezza.

Ciò che fa è alterare i registri: swap di 2 MSR di processore; può fare attività speculative, in particolare legate al fatto che ci può essere un *if* in base al quale fare o meno lo swap.

Sul processore abbiamo il GS corrente e quello alternativo.

Il GS corrente è contenuto nel registro `MSR IA32_GS_BASE`, dove la macro `IA32_GS_BASE` identifica la base del segmento GS, esso è accessibile dall'utente. Portando valori in cache poi possiamo facilmente recuperarne il valore. Sotto, un esempio:

```

if (coming from user space) ← e.g. is this a user
    swapgs   space fault?
mov %gs:<percpu_offset>, %reg
mov (%reg), %reg1

```

Può verificarsi 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 in realtà 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). 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 possono essere recuperate mediante un cache side channel.

Come si risolve?

- Quando si esegue in kernel mode si sovrascrive swapgs. Oneroso.
- Il codice successivo deve essere eseguito solo dopo che swapgs è andata in commit. Non c'è possibilità di speculazione.
- Sui processori moderni è presente una facility hardware, SMAP (Supervisor Mode Access Prevention), per cui con dei bit di controllo si bloccano gli accessi alle pagine user-level quando si è in modalità kernel, così non ci sono side effects legati a ciò che scrive lo user come base per GS.

### ***Organizzazione del software kernel in Linux:***

L'80-90% del codice per le system calls sta in directories specifiche.

Qui configuro cose che posso usare su startup Linux, non facciamo mai lo startup di queste cose!

Sto solo creando contenuto file system.

Come si ricompila il sorgente del codice del kernel Linux?

Si può usare un makefile.

Makefile:

- azione: ha un body in tab dalla linea successiva; tipicamente le azioni possono essere comandi shell
- dipendenze: azioni da svolgere prima dell'azione.
- variabili: \$var\_name per richiamare il valore; per definirla: "var\_name=val"
- ci sono linee che possono essere eseguite comunque: make+action.

Il sorgente contiene un *makefile*, poi possiamo chiamare:

- *make config* (o menuconfig): per configurare i sottosistemi del kernel e le loro API.
- *make* compila il target di default.

### **Step compilazione**

Questo non basta per far sì che il kernel sia utilizzabile, bisogna installarlo e far sì che possa utilizzare dei moduli.

- 1) **make config (o manuconfig)**: configura il kernel che sto compilando e i sottosistemi da includere.  
Ci sono varie possibilità:  
  - **allyesconfig**: tutto ciò che è nel sorgente kernel deve essere compilato. Questo molto spesso porta a moduli conflittanti visto che ho più versioni di sottosistemi per le diverse architetture.
  - **allnoconfig**: si includono solo i moduli core. Questa opzione compila sempre.
    - rispondere per ciascun sottosistema se includerlo o no.
    - utilizzare un file di configurazione del web.
    - riutilizzare file di configurazione della directory /boot (se il kernel da compilare è lo stesso di quello installato).
- 2) **make**: compila il kernel, generandone un'immagine.

- 3) **make modules**: specifica i moduli utilizzabili, non specificati nell'immagine precedente.  
I moduli servono ad estendere le capability del kernel.  
Un modulo è costituito da una serie di routines e strutture che può usare il kernel per estendere le proprie capabilities.
- 4) **makemodules\_install (ROOT)**: installa moduli nel file system (/lib/modules). Solo il root può.
- 5) **make install (ROOT)**: installa l'immagine del kernel. In particolare scrive in /boot l'immagine kernel, la system map e il file di configurazione.
- 6) **mkinitrd (o mkinitramfs)-o initrd.img**: crea un RAM disk, un file system montato temporaneamente dal kernel durante il boot avente alcuni moduli compilati. Essi vengono agganciati a runtime al kernel, e solo dopo si smonta questo file system. Opera su una finestra temporale finita.
- 7) **update-grub (o grub(2)-mkconfig-o /boot/grub/grub.cfg (ROOT))**: per avere un'entry point a ciò che ho installato.

### initrd - initial ram disk:

E' un RAM disk inizialmente utilizzato come filesystem, poi vengono montati i moduli e *initrd* viene smontato. Permette di godere di grande flessibilità, potendo montare qualcosa on-demand e qualcosa in automatico. Con initrd, l'avvio del sistema può avvenire in due fasi: inizialmente, il kernel si avvia con un insieme minimo di driver compilati direttamente nel sistema; successivamente, vengono caricati moduli aggiuntivi da initrd.

*L'installazione è una fase diversa rispetto allo startup, in quanto durante l'installazione andiamo a configurare ciò che poi potrà essere usato durante lo startup.*

È possibile far partire un sistema Linux senza usare un initrd, ma questo normalmente richiede che la partizione radice deve poter essere montata utilizzando solo moduli che sono compilati nel kernel. Quindi un kernel generico dovrebbe contenere il supporto necessario a montare qualsiasi tipo di partizione radice, e finirebbe così per contenere moltissimi moduli non necessari.

### **Anatomia del kernel (system map):**

Prevede una tabella con gli indirizzi logici di tutte le funzioni e le variabili del kernel.

Le mappe ci permettono di analizzare ciò che abbiamo.

System map è utile per: debug kernel, kernel run-time hacking. Non serve per il boot!

La mappa di sistema contiene i simboli ed i rispettivi riferimenti in memoria virtuale.

Per ogni riga si ha: nome del simbolo-posizione-tipo di simbolo.

Ogni simbolo ha un tag associato che definisce una classe di storage.

- *T* = funzione globale
- *t* = funzione locale, ovvero presente in un punto del kernel compilato come static.
- Lo stesso vale per i dati (*D,d*) e per i dati read-only (*R,r*).

Anche se poi il kernel randomizza la collocazione effettiva degli oggetti, la system map torna utile per capire la distanza tra gli oggetti. La mappa dà sempre la posizione relativa tra gli oggetti.

Nel filesystem tutto questo è visibile nello pseudo file **proc**, che riporta tutti i simboli del kernel.

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 mentre il software del kernel viene caricato in memoria sono:

- Il **firmware** è un programma su un ROM disk, rappresenta la logica per i moduli presenti *sull'hardware*. Allo startup il firmware viene eseguito (attività a livello di processore), viene caricato il bootloader che carica il kernel del SO, che fa le proprie azioni di setup (setup di strutture dati, di software e attivazione di processi). Nel kernel viene attivato almeno un processo, l'idle process, che può prendere il controllo di una o più cpu.

- Il bootsector è un settore predeterminato di un dispositivo (i.e. ssd) che mantiene il codice dello startup del sistema. Possiamo configurare il processore per dire che la prima cosa che deve essere fatta è prendere le informazioni da questo dispositivo, metterle in memoria e passare il controllo al bootloader, che è il vero e proprio codice eseguibile.

### Operazioni boot (1-3) e startup (3-4)

1. Il firmware viene eseguito, carica in memoria e lancia il contenuto del bootsector.
2. Il codice del bootsector potrebbe a sua volta caricare altre porzioni del bootloader.
3. Il bootloader carica il kernel del sistema operativo e gli affida il controllo.
4. Il kernel esegue le proprie operazioni di startup, come attivazione del codice software e strutture.  
Almeno un processo inizia la sua esecuzione (IDLE PROCESS).

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

### Firmware tradizionale in x86:

Il firmware è embedded nel BIOS, un sottosistema hardware read-only. Ci si interfaccia con i tasti F1, F2... Tramite il bios impostiamo i parametri riconfigurabili; essi vengono passati tramite semiconduttori alimentati a batteria; se si tiene la macchina non alimentata da batteria per diverso tempo le informazioni vengono perse e si torna alla configurazione di default.

I parametri possono ad esempio determinare l'ordine di ricerca dei boot sector su diversi dispositivi.

Il settore del primo dispositivo mantiene il master boot record (MBR), contenente il codice eseguibile e la partition table. Il codice usa la tabella per capire, ad esempio, se il dispositivo è suddiviso in partizioni.

La tabella ha quattro entries, quindi al massimo quattro partizioni.

Il primo settore di ciascuna partizione si comporta da boot sector.

Il codice permette di leggere la tabella e passare il controllo a qualcosa che si trova in tale tabella.

Una partizione può essere estesa: ogni partizione può essere a sua volta suddivisa in partizioni, ciascuna di esse può avere il proprio boot sector (BS). La sua evoluzione è UEFI.

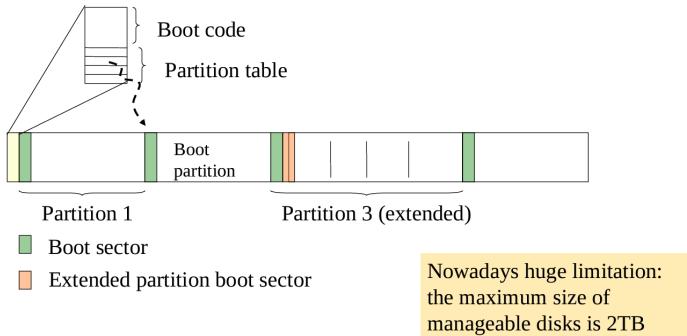
### Immagine RAM dell'MBR:

Offset	Size (bytes)	Description
0	436 (to 446, if you need a little extra)	MBR <b>Bootstrap</b> (flat binary executable code)
0x1b4	10	Optional "unique" disk ID <sup>1</sup>
0x1be	64	MBR <b>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

### Schema esemplificativo di Bios:

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



In tutto questo schema si possono gestire fino a 2T di dati, oggi non basta come taglia di un dispositivo, per questo è stato introdotto [UEFI](#).

## UEFI

UEFI (unified extended firmware interface) serve per le attività iniziali. Si occupa della gestione di dispositivi dell'ordine degli zettabyte.

L'hardware è un interprete, riesce ad eseguire codice eseguibile EFI, dunque risulta più flessibile; va oltre il caricare qualcosa e passargli il controllo.

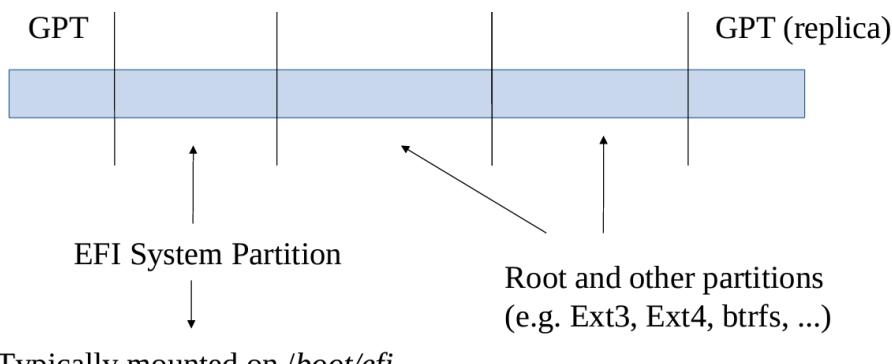
C'è anche il supporto al *secure boot*; l'hardware controlla che il boot rispetti alcune regole.

UEFI realizza il partizionamento dei dispositivi basandosi sul concetto di GPT (Globally unique identifier partition table GUID); essa è una tabella senza una taglia predeterminata.

In teoria può contenere qualsiasi numero di partizioni.

La GPT è replicata, così se la tabella originale è corrotta una delle copie continuerà a funzionare e permettere di identificare le partizioni.

C'è una partizione particolare, la EFI system partition, da cui si leggono i programmi EFI da eseguire allo startup.



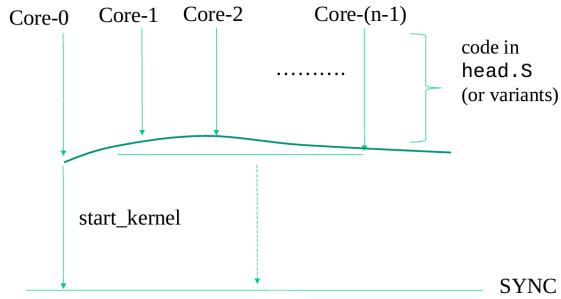
[blkid](#) è il comando che permette di capire cosa c'è o meno sul nostro hard drive.

## Boot del kernel del SO

- 1) Il bootloader / EFI-loader **carica** in memoria l'*immagine iniziale del kernel* del SO.  
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`. 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).

Nel caso **multi-core**, la funzione `start_kernel()` è eseguita da una sola CPU detta master. Tutte le altre aspettano che finisca tramite sincronizzazione hardware. Per recuperare l'ID del core corrente si può usare la funzione kernel `smp_processor_id()` che a sua volta chiama CPUID.

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.



Quindi prima si esegue il 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.

Lez.16 (30/10/23)

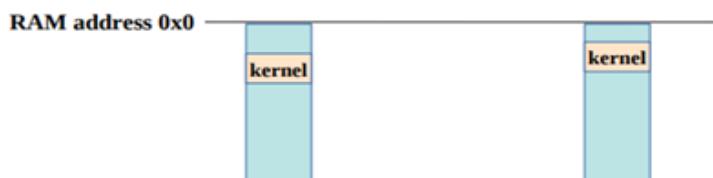
## Kernel level memory management:

L'immagine kernel allo startup kernel non è la stessa di quella allo steady state. Devo valutare:

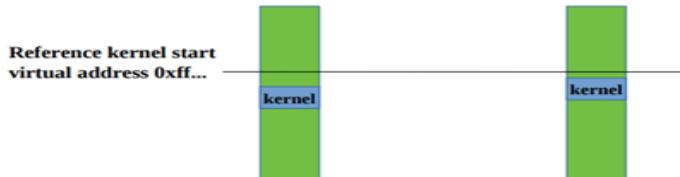
- Dove carichiamo l'immagine kernel attuale? C'è randomizzazione anche in RAM?
  - Come rendiamo la memoria fisica raggiungibile dagli indirizzi virtuali? Anche nella memoria fisica introduco la randomizzazione?
  - Le informazioni kernel allo startup devono sparire?
  - Quali sono i meccanismi kernel per manipolare la memoria allo steady state

### **Caricamento del kernel in memoria fisica:**

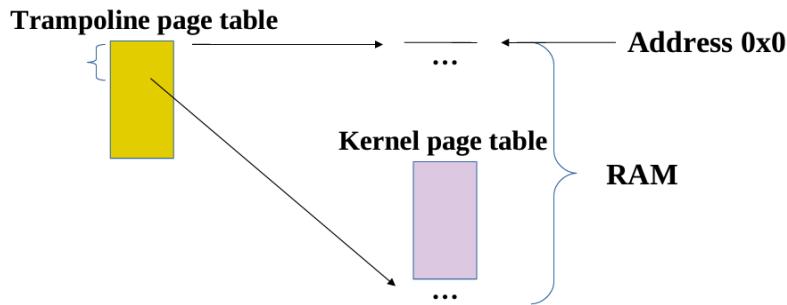
La regione fisica in cui viene caricato il kernel viene individuata dal **bootloader**. Se vogliamo randomizzare tale posizione occorre che l'immagine del kernel supporti un bootloader che si occupa della randomizzazione. La randomizzazione può causare che la parte iniziale della memoria non venga usata. Dal momento che il kernel utilizza indirizzi logici per accedere alla memoria è necessario definire un **page table** allo startup.



Anche la memoria virtuale in cui viene localizzato il kernel viene identificata dal bootloader. Anche qui avviene la randomizzazione che causa l'inutilizzo di una parte iniziale di memoria.



Il kernel caricato in memoria all'indirizzo fisico P\_ADDR ha nella sua immagine una page table che viene configurata allo startup in modo da fare la conversione tra indirizzi fisici e logici. Il kernel deve conoscere la collocazione fisica della tabella che è data da P\_ADDR + **offset**. La presenza di questo offset impone limiti nella compilazione del kernel (ad esempio non possiamo espandere strutture dati). Questa tabella è nota come **identity page table**. Il bootloader a sua volta utilizza un'altra tabella delle pagine per raggiungere la memoria fisica e collocare il kernel detta **trampoline page table**.



### **Randomizzazione:**

Sia la randomizzazione logica che quella fisica sono necessarie. La randomizzazione logica è pensata per proteggere il kernel dagli attacchi. Tuttavia, se l'attaccante scopre la collocazione in memoria fisica del kernel, può manipolare l'identity page table per raggiungere tale area. Per questo serve anche la randomizzazione fisica.

### **Se Kernel non randomizzato:**

In questo scenario l'indirizzo fisico e logico del kernel e della tabella delle pagine è noto a compile time. Dunque visto che il bootloader sa dove è la collocazione fisica del kernel non ha bisogno di utilizzare la trampoline page table. 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, senza aspettare lo startup del kernel.

### Supporto alla randomizzazione in Linux:

Si usa la macro CONFIG\_RANDOMIZE\_BASE.

Per escludere la randomizzazione: nokaslr; questa opzione è utile in fase di debugging.

Se si randomizza nell'AS fisico la posizione delle immagini del kernel può stare in qualsiasi punto della RAM; per quanto riguarda la posizione in memoria logica l'immagine del kernel sta tra 16 Mb e 1Gb.

### Avvio paginazione Linux IA32:

```
/* * Enable paging */ 3:  
movl $swapper_pg_dir-__PAGE_OFFSET,%eax  
movl %eax,%cr3 /* set the page table pointer.. */  
movl %cr0,%eax  
orl $0x80000000,%eax  
movl %eax,%cr0 /* ..and set paging (PG) bit */
```

Notiamo che nella prima movl swapper\_pg\_dir è l'indirizzo logico della tabella delle pagine che sommato al suo offset dà l'indirizzo fisico. L'indirizzo della tabella viene memorizzato in %cr3. Dopodichè si attiva la paginazione in %cr0 (si sfrutta %eax come registro intermedio)

### Avvio paginazione Linux x86-64

Su x86\_64 la startup ha istruzioni diverse; viene utilizzato lo stesso schema di prima, ma la "early\_top\_pgt" non è la page table definitiva, ma quella trampolino.

"init\_top\_pgt" è la page table kernel, il suo indirizzo logico viene definito a startup time.

Si usano i seguenti simboli:

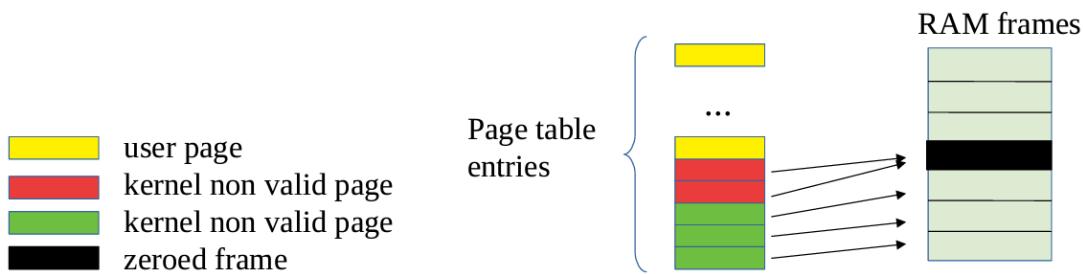
- Kernel 2: swapper\_pg\_dir
- Kernel 3: init\_level4\_pgt

- Kernel 4/5(randomizzazione): `init_top_pgt` (definito a start up time), `early_top_pgt` non è indirizzo effettivo ma è a compile time

### Rompere la randomizzazione tramite la patch per Meltdown:

La patch a Meltdown non mette la cpu in stallo quando si tenta di accedere ad una pagina kernel.

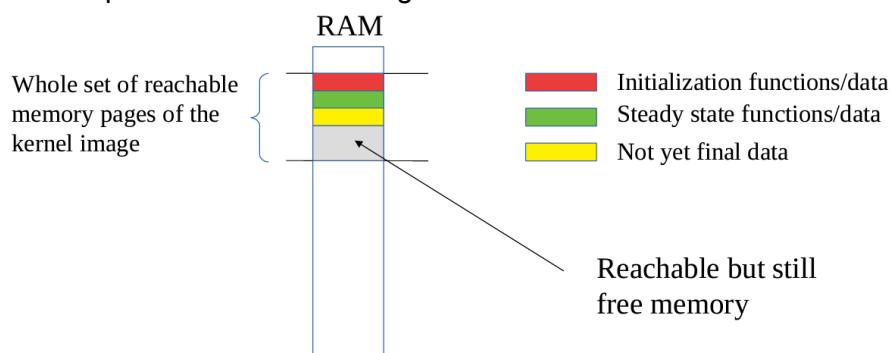
Semplicemente passa il valore 0 in modo che l'attaccante possa leggere solo lo 0-esimo byte dell'array probe. Tuttavia, se tentiamo di accedere a zone kernel non mappate (quindi il "buco" di cui parlavamo all'inizio), la cpu va in stallo perché la tabella delle pagine non risponde. Dunque attaccante agendo per tentativi può capire qual è l'indirizzo logico del kernel, rendendo nulla la randomizzazione. . E' necessario che tutte le pagine della memoria fisica abbiano la stessa taglia altrimenti possiamo estrarre informazioni studiando le latenze.



### Cosa carichiamo in RAM quando carichiamo l'immagine del kernel?

All'inizio si carica un oggetto compatto fatto di funzioni e dati di inizializzazione, funzioni e dati steady state per la gestione delle attività e "not yet final", ovvero la page table che via via viene modificata dal kernel; a steady state il kernel viene esteso nelle sue funzionalità.

Alla fine del caricamento c'è una zona raggiungibile, ma libera, raggiungibile significa che il software può leggervi e scrivervi. Le strutture per la gestione della memoria non sono ancora presenti perchè non si sa ancora quanta memoria dovrà gestire.



Le funzioni ed i dati di inizializzazione possono essere un set di gadget, a steady state non servono, quindi ciò che non serve deve essere rimosso dalla memoria; questo per:

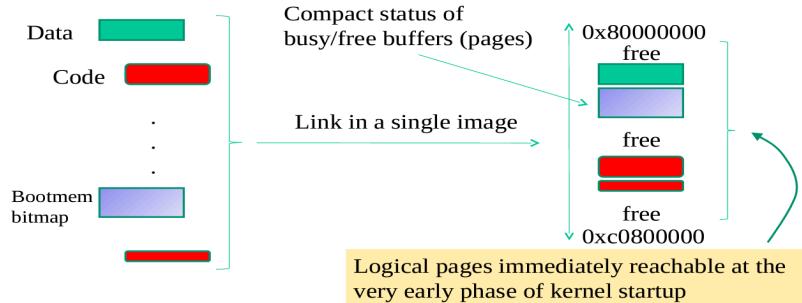
- ottimizzare la gestione delle risorse
- per gestire meglio la complessità allo startup
- per gestire meglio aspetti di sicurezza (e.g. eliminare potenziali gadget)

Dobbiamo identificare ciò che non serve e poter recuperare le pagine di memoria.

### Oggetti di inizializzazione

Una volta che il kernel ha raggiunto lo steady state gli oggetti di inizializzazione diventano inutili. Occorre dunque rimuoverli per liberare spazio ed eliminare gadget utilizzabili per attacchi. L'eliminazione necessita di supporto a run-time (quando avviene eliminazione effettiva) e compile-time (dobbiamo sapere cosa è eliminabile). Si usa la marcatura `_init` per indicare che è una funzione da eliminare. Il compilatore inserisce queste funzioni nella stessa pagina logica in modo da poterle eliminare. Un esempio è la funzione `_init_start_kernel(void)`. Queste pagine sono identificate dal sottosistema `bootmem` che viene usato per gestire la memoria quando il kernel non è steady. Il bootmem mantiene una bitmask che identifica

pagine libere dalle pagine da buttare. Il bootmem può anche allocare memoria tenendo traccia delle pagine libere (è usato nello startup)



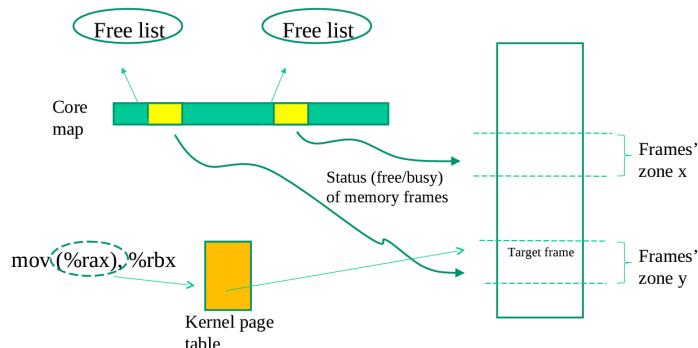
Le macchine moderne hanno un'architettura di tipo NUMA: ogni CPU core ha dei banchi RAM vicini e altri lontani. Ciascun banco di memoria viene detto nodo NUMA e dispone di un proprio allocatore. Nei sistemi operativi moderni si utilizza una versione più evoluta di bootmem detta **memblock** che tiene traccia dei frame liberi con una granularità per nodo NUMA. Nonostante le API per gestire la memoria nella bootmem e nel membblock 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()`).
- La possibilità di ottenere sia l'indirizzo virtuale (mediante le API `memblock_alloc*`(*green*) sia l'indirizzo fisico (mediante le API `memblock_phys_alloc*`(*green*)) delle pagine allocate nel membblock.

### **Strutture dati kernel per la gestione della memoria:**

- **Kernel (identity) 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:** mantiene le informazioni di stato per ogni pagina della memoria fisica e per ogni nodo NUMA
- **Free list:** Lista dei frame fisici liberi per ogni nodo NUMA. Permette di allocare e deallocare memoria. Consulta la core map.

Nessuna di queste è finalizzata allo startup del kernel. 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 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 accessibili da tutte le CPU. Ci sono poi *quick list* per CPU (non c'è bisogno di sincronizzazione).



### Obiettivi del setup dell'identity page table:

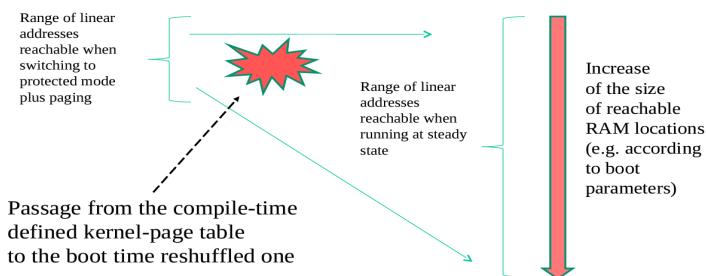
- Permette al software del kernel e applicativo di utilizzare indirizzi logici.

- Permette al software kernel che usa indirizzi logici di lavorare su tutta la RAM ammissibile. La forma definitiva della tabella delle pagine del kernel di conseguenza di solito non viene configurata nell'immagine originale del kernel caricata in memoria dato che:

- la RAM disponibile può essere parametrizzata, nel senso che la quantità massima di RAM disponibile dipende dalla specifica macchina.
- potremmo avere randomizzazione.

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.

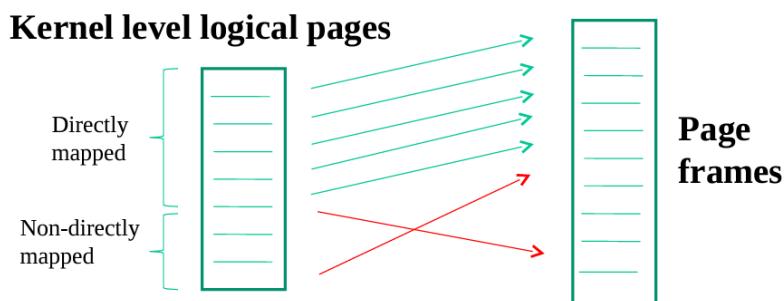
#### Schema x86 protected mode:



Le tabelle delle pagine possono contenere pagine di dimensioni diverse; su x86 si può lavorare solo su pagine di 4K.

#### **Direct & non-direct map:**

- Direct mapping: si risale alla posizione in RAM di una pagina logica applicando un offset all'indirizzo logico. Si risale facilmente da indirizzo logico a fisico e viceversa, grazie alla relazione  $PA = \psi(VA)$ .
- Non-directly mapped: non si applica l'offset:  $PA \neq \psi(VA)$ . Per conoscere la posizione fisica nel caso *non-directly mapped* bisogna scandire l'identity page table, facendo attenzione a gestire opportunamente la concorrenza.



Alcuni allocator forniscono memoria directly mapped, altri non-directly mapped.

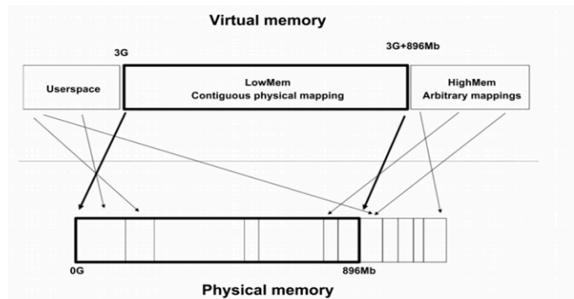
#### Zone nel mapping delle pagine:

La memoria fisica viene organizzata dal sistema operativo in "zone", esse determinano il tipo di utilizzo per queste pagine.

Le zone più comuni sono:

- DMA: riserva memoria ad operazioni specifiche di dispositivo.
- NORMAL: pagine kernel directly mapped.
- HIGH: mappa in modo non-directly mapped le pagine kernel e non kernel (dà maggiore flessibilità perché posso raggiungere più indirizzi).

## Schema x86 protected mode



La memoria è contigua a livello logico ma non a livello fisico. Gli indirizzi non directly mapped sono usati quando serve tanta memoria (non importa che sia contigua) o uso la frammentazione.

### x86 long mode:

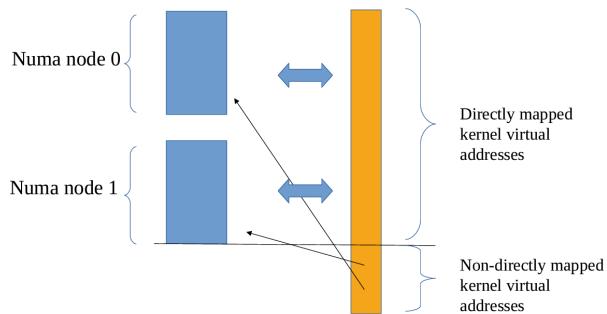
Su x86-64 abbiamo  $2^{48}$  indirizzi logici possibili; il kernel arriva su qualsiasi pagina di memoria mappata direttamente.

*La stessa memoria si può usare quando si utilizza indirizzamento non directly mapped, utile per un buffer logico grande, dove la memoria fisica non è importante.*

Si bypassa il limite dei 32 bit: il kernel Linux ha 1Gb (da 3Gb a 4Gb), come faccio a mappare direttamente tutta la memoria?

Per la sicurezza è importante disporre di un'area non directly mapped, utilizzata per le informazioni più critiche, in quanto con memoria directly mapped se ho l'indirizzo logico posso risalire a quello fisico.

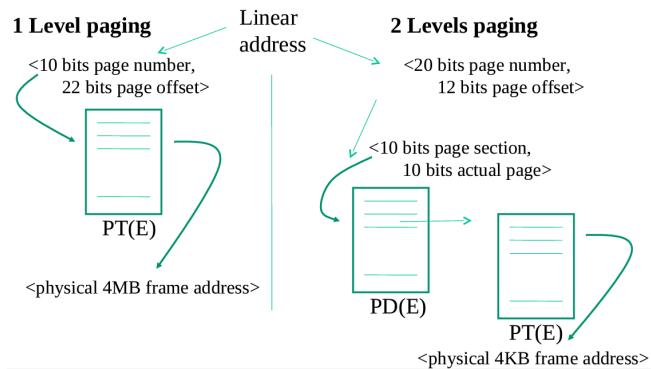
Le stesse zone sono raggiungibili sia con indirizzamento directly mapped che non-directly mapped, ciò vuol dire che ho un *doppio mapping*. Le zone non sono più rilevanti, ma in architettura ancora ci sono. La memoria è contigua a livello logico (arancione) ma non a livello fisico.



Lez.17 (02/11/23)

### **Struttura delle page table**

Consideriamo *x86 protected mode*. La tabella può essere composta fino a 4 livelli: la prima tabella referenzia la seconda che referenzia la terza che referenzia la quarta. Nell'ultima tabella c'è il riferimento alle pagine. Garantisce una maggiore modularità in quanto se replica la prima tabella la seconda può rimanere uguale.



### Paginazione ad un livello:

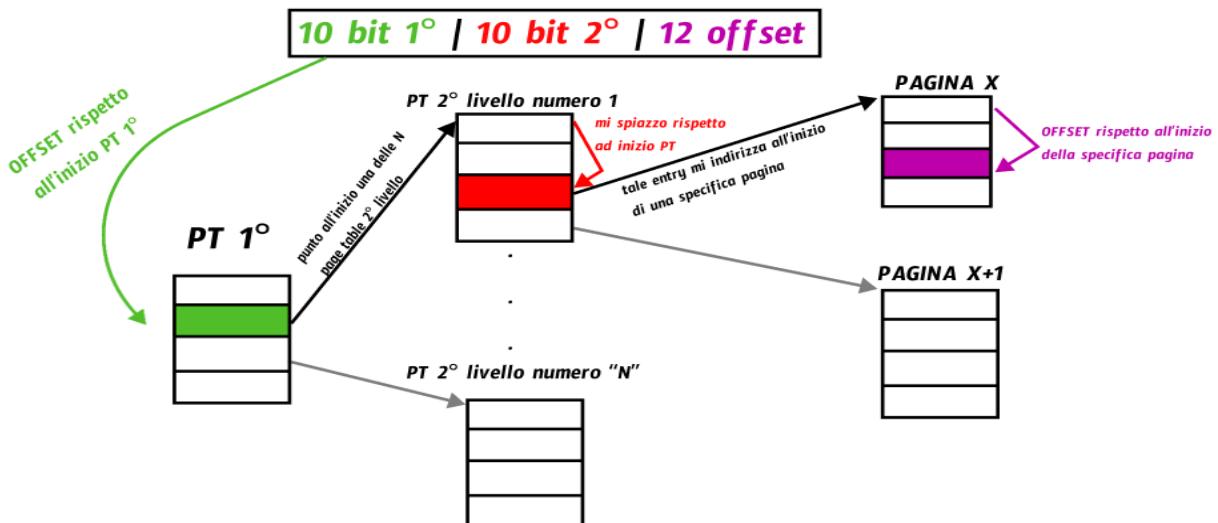
Qui gli indirizzi lineari, che sappiamo essere a 32 bit, sono suddivisi in questo modo:

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

### Paginazione a due livelli:

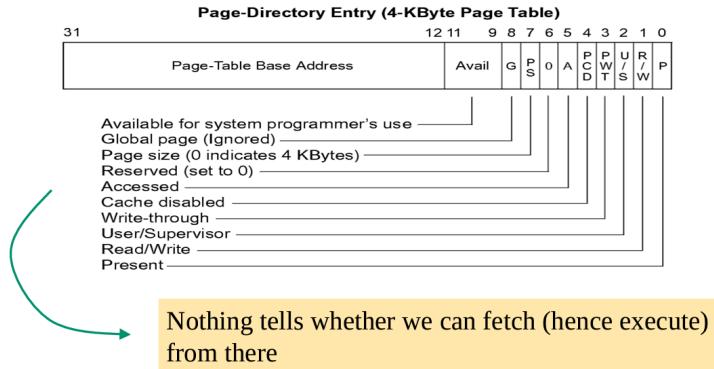
Qui gli indirizzi sono suddivisi in quest'altro modo:

- i primi 10 bit (**PDE** – page directory entry) identificano l'offset della page table di primo livello che punta ad una delle 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. Ciò vuol dire che in una pagina posso indirizzare  $2^{12}$  byte = 4 KB. Se in una pagina posso spiazzarmi fino a 4KB, allora questa è la sua dimensione.



Se agire in un modo o nell'altro viene scelto dal processore sulla base di bit di controllo contenuti nella tabella di primo livello. Le page entries della tabella sono composte da 32 bit e sono 1024. Ogni parte della tabella deve essere allocata in memoria allineata a indirizzi 4K (ecco perché questa è la dimensione minima della pagina)

### Struttura delle entries della PDE (tabella di primo livello):

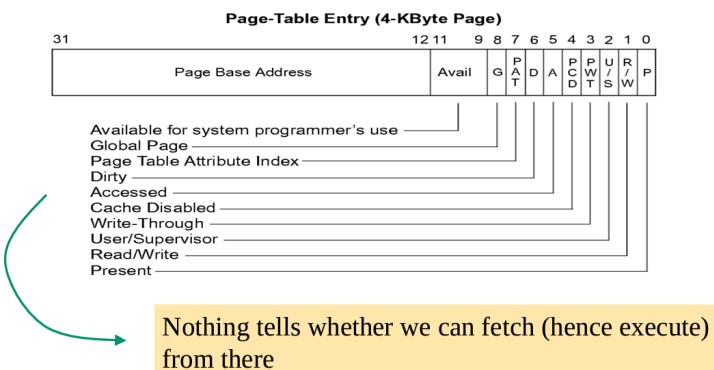


- 12 bit di controllo, ovvero come viene rappresentata un'entry.
- I 20 bit più significativi rappresentano l'indirizzo base; ciò implica che le tabelle siano allineate a 4K, dato che se i 12 lsb sono posti a 0 riportano l'indirizzo della tabella, sono necessariamente allineati. L'allocatore della page table ci deve dare memoria allineata.
- Il bit "page size" ci dice se le pagine sono da 4 Kb o da 4 Mb.
- Il bit "supervisor" indica la modalità operativa del processore, se privilege 0 siamo supervisor.
- il bit "accessed" dice se il firmware ha consultato questa entry, è uno sticky bit settato a 1 dal firmware qualora esegua un accesso. I clock algorithms usano questo bit per sapere se una zona è stata o meno acceduta.

### Struttura delle entries della PTE (tabella di secondo livello):

Nella PTE non c'è la size, perchè l'oggetto è di default di 4 Kb.

- Il bit *present* indica se l'indirizzo è valido e può essere utilizzato dal firmware per accedere in memoria fisica, non è settato dal firmware. Se invalido, l'accesso viene fatto da parte di un'istruzione in pipeline, cosa succede se il bit è 0? Nulla finchè l'istruzione non va in commit, quindi parte dell'informazione deve sempre essere processata, questo problema si chiama **L1TF** ed è un bug drammatico che non presenta alcuna patch software. *Se usiamo speculativamente un indirizzo fisico in pipeline stiamo già bypassando il software*. Questo bug permette ad una VM attiva su un sistema tradizionale di rubare la memoria di qualsiasi altra macchina attiva.



### Setup dell'identity page table:

#### Caso kernel 2.4 (no rand disk):

Allo startup del kernel l'indirizzamento usa la paginazione ad un livello che mappa solo due pagine: quindi il kernel dispone solo di 8MB. *L'indirizzo fisico della tabella delle pagine è tenuto nel registro CR3*. La tabella viene setuppata per poter permettere al kernel di raggiungere gli indirizzi.

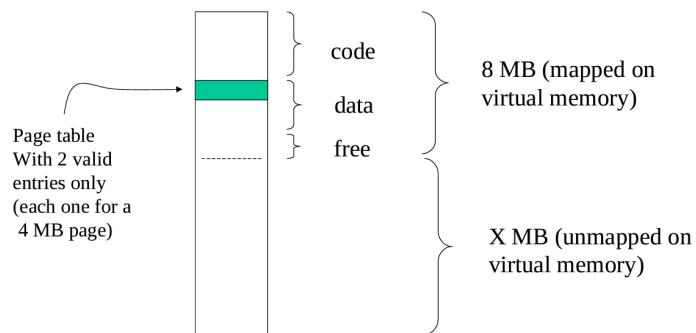
Per quanto riguarda l'indirizzamento lineare dei sistemi i386, all'interno di un address space è possibile esprimere al più 4 GB ( $2^{32}$ ) di indirizzi logici.

- 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.

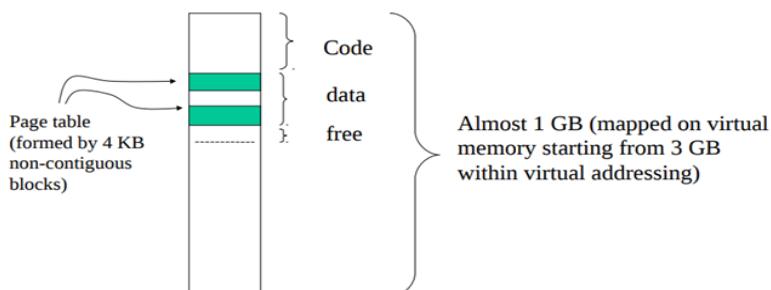
Il layout di memoria allo startup kernel è il seguente



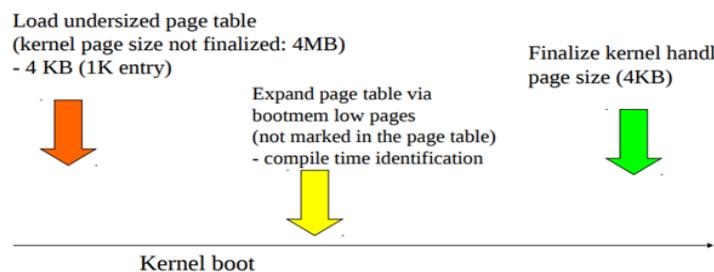
Per passare dalla fase di startup del kernel alla fase steady state, è necessario far fronte alle seguenti problematiche, in quanto si parte con una certa struttura, ma non la si mantiene alla fine:

- 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:



Riassunto degli stadi:

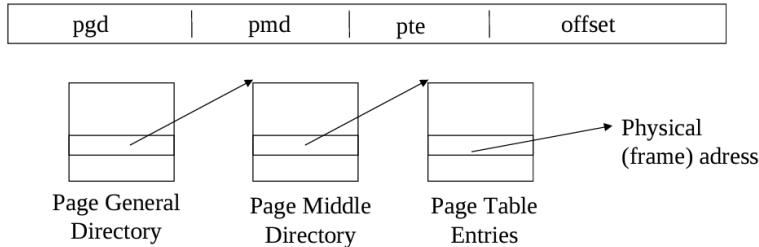


Linux 2.4 era progettato per lavorare con indirizzi di questo tipo (ossia con tre livelli di indirezione):

- pgd = offset in page table di primo livello
- pmd = offset in page table di secondo livello

- pte = offset in page table di terzo livello

In realtà le macchine i386 ne usano solo 2: **pgd Linux** mappa su **i386 pde** e **pte Linux** mappa su **i386 pte**



### Macro per mapping:

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

### Strutture dati:

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

La definizione 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 `PGD_t`, `PMD_t` e `PTE_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).

### Bitmask:

In `include/asm-i386/pgtable.h` ci sono delle macro che definiscono la posizione dei bit di controllo nelle tabelle. Ci sono anche le seguenti macro per settare i bit.

```
>#define _PAGE_PRESENT      0x001
>#define _PAGE_RW           0x002
>#define _PAGE_USER          0x004
...
>#define _PAGE_ACCESSED     0x020
>#define _PAGE_DIRTY         0x040 /* proper of PTE */
```

### Algoritmo Inizializzazione della page table:

Step eseguiti ciclicamente

1. Determiniamo, per ogni iterazione, l'indirizzo virtuale da mappare in memoria (questo viene mantenuto nella variabile `vaddr`)
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 ossia quella che mappava i 4MB (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 4MB più avanti rispetto a quello appena mappato.
5. Saltare allo step 1 fintanto che esistono ancora indirizzi virtuali da mappare (ovvero fintanto 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

Codice:

```
for ( ; i < PTRS_PER_PGD; pgd++, i++) {  
  
    vaddr = i*PGDIR_SIZE; /* i is set to map from 3 GB */  
    if (end && (vaddr >= end)) break;  
    pmd = (pmd_t *)pgd; /* pgd initialized to (swapper_pg_dir+i) */  
  
    .....  
  
    for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {  
        .....  
  
        pte_base = pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);  
  
        for (k = 0; k < PTRS_PER_PTE; pte++, k++) {  
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;  
            if (end && (vaddr >= end)) break;  
            .....  
            *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);  
        }  
        set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte_base)));  
        .....  
    }  
}
```

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()`.

### ***Relazione con eventi di trap e interrupt:***

Il primo bit che viene consultato nell'ambito di una entry della tabella delle pagine è il *bit “present”*. Se è 0 si verifica una trap, il sistema operativo prende il controllo e determina se l'accesso è legale, ovvero se la pagina è legalmente accessibile ma non presente in memoria fisica; a livello OS questa informazione viene messa in memoria fisica, si aggiorna la tabella e si ritorna il controllo all'istruzione; nel fare questo si possono fare altri controlli tra questi bit, attenzione perché ciò può essere gestito iterativamente dal kernel. Questi bit non vengono guardati tutti insieme, ma in ordine.

### ***Esempio:***

```
#include <kernel.h>  
  
#define MASK 1<<7  
  
unsigned long addr = 3<<30; // fixing a reference on the  
// kernel boundary  
  
asmlinkage int sys_page_size(){  
  
    //addr = (unsigned long)sys_page_size; // moving the reference  
    return(swapper_pg_dir[(int)((unsigned long)addr>>22)]&MASK?  
        4<<20:4<<10);  
}
```

Il primo `addr` è un indirizzo che pensiamo sia nella tabella (sbagliato). Il secondo indirizzo (che è il nome della funzione `asmlinkage` da noi definita) è nella tabella. Con lo shift di 22 bit levo quelli meno significativi e applicando la maschera estraggo il bit che indica la taglia.

### **Physical address extension (PAE):**

Si estendono gli *indirizzi fisici* (non logici!) a 36 bits facendo sì che si abbia  $2^{36}$  ossia 64GB di memoria RAM. La paginazione opera a 3 livelli e non più 2, in quanto con due livelli dovrei gestire tabelle molto grandi. Ciò che si fa è quindi usare più tabelle con entry di taglia doppia ma con numero delle entry nelle tabelle dimezzate a 512. Poiché, nella trattazione svoltasi finora, si hanno due livelli di paginazione, si va incontro a un supporto di 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.

### **Architettura x86-64:**

Si estende lo schema PAE attraverso il "**long addressing mode**".

Gli indirizzi logici sono a 64 bit.

Viene introdotto il concetto di "canonizzazione degli indirizzi", sebbene abbia a disposizione 64 bit se ne possono usare solo i 48 meno significativi.

256 Tb raggiungibili per ora sono sufficienti, non tutti sono utilizzabili, in Linux 128Tb.

Come funziona la canonizzazione?

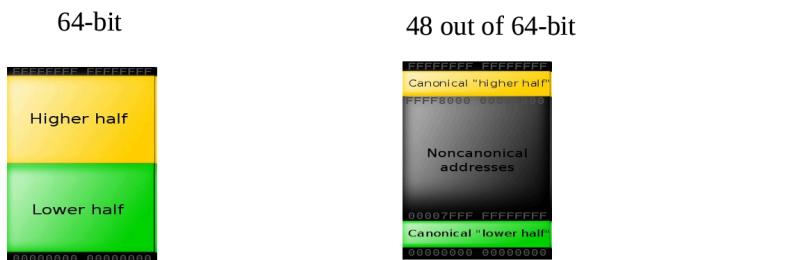
L'address space è suddiviso in **high half** e **low half**, con la canonizzazione è come se li separassi, tramite il primo bit si capisce se si sta accedendo alla parte low o alla parte high.

La lower half viene usata dallo user, la higher usata dal kernel, che ci fa utilizzare soltanto le VDSO.

E' solamente una suddivisione logica.

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.

Lo scopo è ottimizzare la circuiteria del processore, per supportare  $2^{64}$  dovrebbe cambiare notevolmente tutto il design interno.

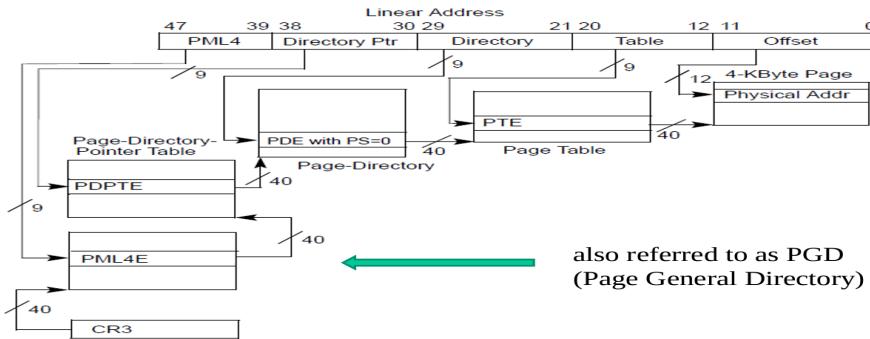


Tipicamente nella higher half viene posizionata roba utente, nella lower half roba kernel di cui l'utente può accedere direttamente al VDSO.

### **Page table addressing a 48 bit**

Abbiamo la page table con 4 livelli: PML4, PDPT, PDE, PTE.

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 dell'indirizzo a 48 bit è la seguente:



#### Struttura delle entries delle tabelle (uguali):

Se il settimo bit è pari a 1: zona riservata al processore + indirizzo di pagine da 1 Gb; non ho bisogno di grana più fine, sono le cosiddette **“huge pages”**, che sono zone di memoria contigua usate dal processore. Anche nel terzo livello la struttura è simile.

Lez.18 (06/11/23)

Nelle entries della tabella di primo livello non ci sono bit di controllo riguardo il fatto che le informazioni in relazione a questa pagine sono fetchabili o no, per questo motivo in passato sono stati realizzati attacchi di code injection, perchè injettare codice binario fa sì che esso possa essere eseguito.

Inserire queste informazioni di controllo non risolve il problema di usare istruzioni già presenti in memoria per realizzare un attacco.

### *Campi della PTE:*

La forma canonica vale solo per gli indirizzi logici, nella gestione della RAM non è così.

L'indirizzo a 48 bit è diviso in zone; ha 12 bit di offset (pagina di 4k è la grana più fine raggiungibile), il resto è diviso in zone da 9 bit, ciascuna zona indica una tabella di un certo livello.

Il 63esimo bit è “XD”, per proteggere la possibilità di eseguire fetch; possiamo disabilitare l'esecuzione di una certa zona di memoria logica. L'accesso in fetch in memoria prevede il controllo di questo bit; ad esempio permette di proteggere la stack area: le pagine della stack area sono marcate come non eseguibili quando un programma viene caricato in memoria.

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)

### Huge pages:

Il bit “size” indica se quando scendiamo attraverso una catena di page tables dobbiamo arrivare all’ultimo livello o dobbiamo fermarci prima; ovvero se le pagine sono “regioni” di memoria, ovvero pagine più grandi che possono essere gestire al secondo (1Gb) o al terzo livello (2 Mb).

x86\_64 offre questa funzionalità ed anche Linux; in particolare Linux offre la possibilità di gestire 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()`. 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. *Le huge pages sono utili perché se due pagine sono contigue in memoria fisica la probabilità che siano in una linea di cache conflittante è molto bassa.*

Il motivo è che se lavori con pagine sparse, la probabilità che ci siano due linee di cache conflittanti è non nulla, mentre è nulla se pagine contigue.

Da un’entry PML4 si possono raggiungere  $2^{27}$  frames, ovvero 512Gb ( $2^{27} * 4k$ ), difficile vederli su un’architettura standard. Sulla memoria fisica il mapping può essere diretto o indiretto.

Se configuriamo molte huge pages riduciamo la possibilità di utilizzare la memoria, dopo che il kernel dà le pagine all’utente non sono più sotto il suo controllo, quindi non può decidere di swapparne alcune.

Ci sono delle huge pages riservate per le attività del kernel che necessitano di huge pages.

### Virtualizzazione della memoria in ambito VMs:

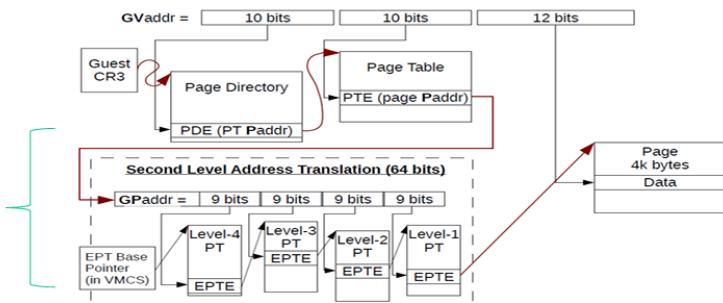
La VM vorrebbe utilizzare la memoria fisica a partire dall’indirizzo 0, ma l’host parte da 0.

Sulle architetture moderne con supporto alla virtualizzazione quando gira una VM l’hypervisor usa registri del processore virtuale, questo guest register ci porta alle tabelle delle pagine della VM.

L’indirizzo fisico riportato non è davvero quello fisico, ma è un indirizzo logico. Si usano *shadow copies*, ad esempio CR3 è del processore virtuale. Il problema è che si compiono attività non consone alla sicurezza.

- A scheme:

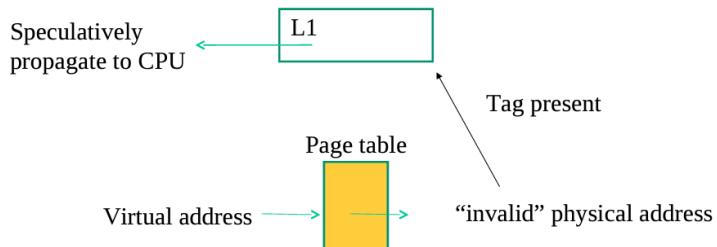
Keeps track of the physical memory location of the page frames used for activated VM



### L1TF:

C'è un problema relativo allo svolgimento di attività non consone dal punto di vista della sicurezza; ad esempio se si accede al presence bit e l'entry non è valida cosa accade (presence **bit** = 0)?

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, 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 considera il seguente comportamento: se abbiamo una entry della page table con presence bit pari a 0, verrà generato un page fault. Tuttavia, i processori vulnerabili a questo attacco utilizzano le entry della page table come un indirizzo fisico per leggere dati nella cache L1. Se tale indirizzo è presente nella cache, i dati possono essere propagati alla pipeline che li utilizzerà prima che l'istruzione vada in retirement. Vediamo che utilizzando una logica simile a Meltdown possiamo capire il contenuto di tale indirizzo.



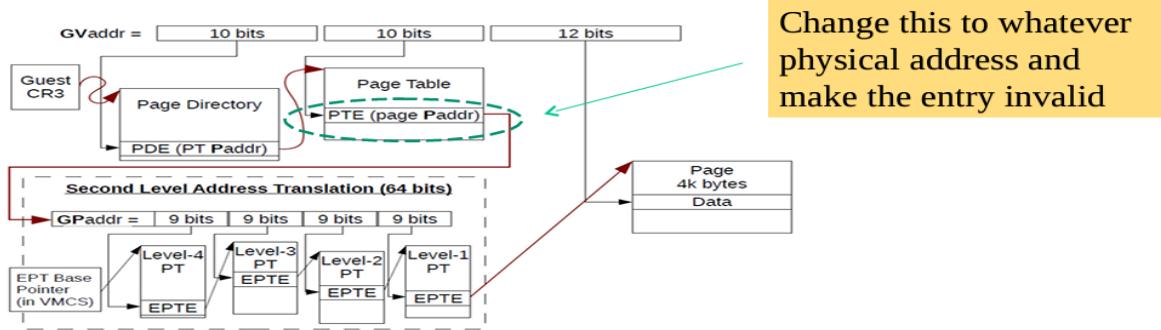
L'attacco va a buon fine nel momento in cui il contenuto della entry invalida della page table mappa dei dati attualmente salvati in cache.

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.

Nel caso di VM per accedere alla memoria si usa la seguente procedura:

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



Per l'attacco, supponiamo che in memoria fisica, all'indirizzo X, ci sia un dato in cache L1, e consideriamo la entry *e1* della PTE guest. Allora è possibile fare in modo che *e1* sia non valida e abbia un contenuto che, senza sfruttare la “second level address translation”, punti proprio all'indirizzo fisico X.

Se una entry è non valida si cerca *direttamente* di accedere in *memoria fisica* per avere il mapping corretto, perché significa che *l'associazione tra pagina logica e fisica non è valida, non che la pagina fisica in sé non è valida*. Se il presence **bit** = 1, allora sfrutterà il mapping delle page table.

Così, quando un'applicazione che gira all'interno della VM guest tenta di effettuare un accesso in memoria sfruttando proprio la entry *e1* 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 estrarre 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 (salto tra domini diversi). Una condizione necessaria per cui questo attacco vada a buon fine è che l'attaccante e la vittima girino in due hyperthread associati al medesimo CPU-core, in modo tale che condividano la medesima cache L1. Con questo attacco è possibile leggere dati di altre VM. Alla fine avremo un page fault e il ritiro dell'istruzione, ma ho esposto dati.

## Lez.19 (08/11/23)

### Organizzazione memoria

Nelle architetture NUMA la memoria è organizzata in banchi. Ogni banco ha un costo di accesso. Un banco è chiamato anche *nodo* e il concetto è rappresentato in Linux da una `pglist_data`. Ogni nodo è diviso in zone della memoria descritte da `struct zone_struct`.

Ogni pagina fisica è poi rappresentata da una `struct page`.

### Core map:

È una struttura dati che tiene traccia dello stato (e.g. libero o 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. La core map è una struttura base utilizzata poi da strutture superiori come la **free map**.

In C, è definita da `mem_map_t`:

```
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;
```

- Il primo campo della struct "list" serve per creare collegamenti tra strutture dati, il collegamento serve per supportare il concetto di free list.
- Il campo count è un contatore di riferimenti al frame associato a questa entry.
- Il campo flags riporta informazioni sullo stato dell'oggetto.

L'array non è presente allo start up ma creato al boot (viene utilizzata anche la *bootmem* per capire quali zone sono già in uso).

### **Free list - nodi o Zone di memoria**

Ogni elemento ha al suo interno un array di puntatori alle "heads" delle free lists. Interroga e aggiorna core map. La core map mi dice se c'è un frame libero, ma in memoria potrei avere una coppia di frames liberi, quindi ho molte free lists. *Sintetizza* le informazioni riguardanti la core map, individuando "zone libere". Se si hanno più freelist della stessa taglia si uniscono a formarne una di taglia maggiore, questo meccanismo si chiama "core hashing".

```
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;
```

- node\_zones: zone che sono presenti nel nodo.
- node\_mem\_map: puntiamo ad un'entry della core map.
- nr\_zones: numero di zone gestite dalla 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: bitmap delle free area usato dall'allocatore buddy.
- lock: lock per proteggere da accessi concorrenti.
- zone\_mem\_map: la prima pagina nella mem\_map a cui si riferisce questa zona.

### **Buddy allocator:**

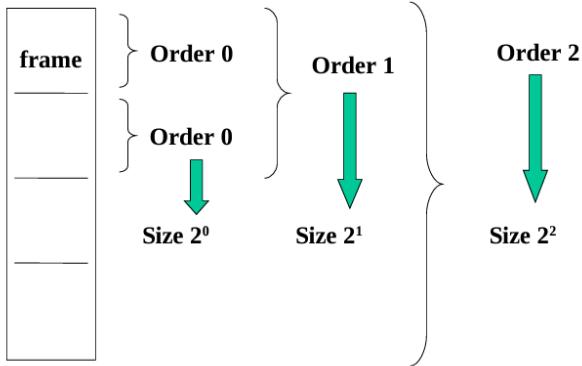
Gli allocatori sono "buddy" ("accoppatore"), e seguono tale schema:

Due frame vengono uniti se:

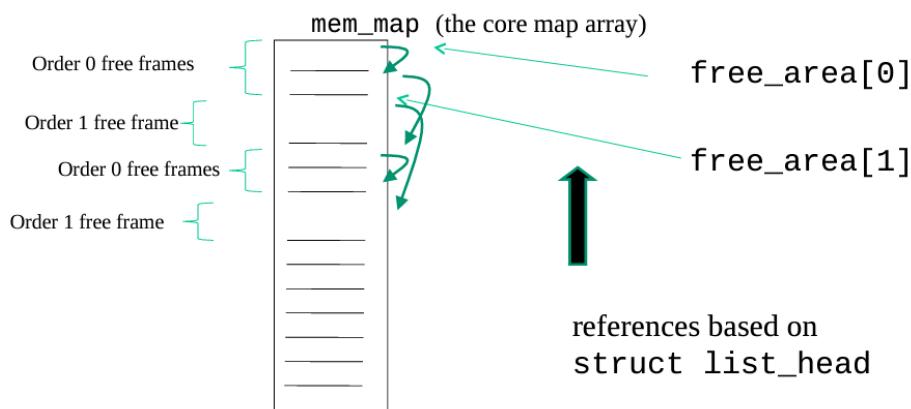
*liberi; contigui;* la taglia è allineata con l'ordine successivo. (es: se ho 3 frame, arrivo solo a ordine 1).

Due blocchi di ordine 0 diventano un blocco di ordine 1, poi questo blocco si può unire ad un altro blocco di ordine 1 creandone uno di ordine 2, sempre secondo le stesse regole.

Il buddy system ci permette di lavorare con granularità minima del frame in memoria fisica.

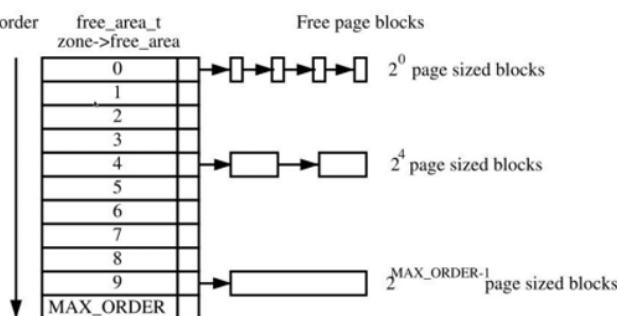


- Per quanto riguarda la **core map**, è organizzata in più liste collegate, dove ciascun elemento di ogni lista è relativo a una entry della core map stessa. In particolare, si ha una lista collegata per gli elementi di ordine 0, una lista collegata per gli elementi di ordine 1, e così via.



Recall that spinlocks are used to manage this data structure

- 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** avente size del blocco pari a  $2^i$ , porzione della core map associata a una specifica ZONE di uno specifico nodo NUMA).



Nel momento in cui si richiede un'area di memoria di un certo livello e non è disponibile si passa al livello superiore e si fa spitting della memoria.

## Esempio:

ho bisogno di un'area di ordine 0, ma ho solo aree di ordine 1, prende un'area di ordine 1 e la suddivido nelle due di ordine 0.

A volte le system calls possono richiedere operazioni su uno specifico oggetto, essendo operazioni write intensive (prendo o rilascio memoria); bisogna fare attenzione alla concorrenza, si usa spinlock, dunque comprometto la scalabilità.

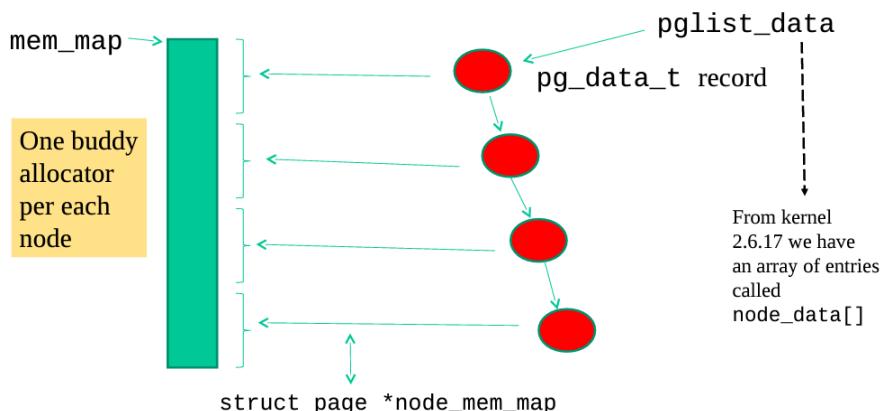
Qual è la taglia massima che si può prendere con questo allocatore? Prima  $2^5$  ora  $2^{11}$ .

Se voglio una zona di memoria contigua più grande dovrò rivolgermi ad un altro allocatore, perché con il buddy allocator per avere una zona più grande dovrei fare più chiamate, ma più chiamate mi danno zone non contigue.

## Architettura NUMA

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:



## Allocazione:

Cosa succede quando un'API tocca queste strutture?

Un thread user è in esecuzione, avviene un interrupt e l'handler prende il controllo; l'handler potrebbe dover interagire con queste strutture, rimarrebbe bloccato e, di conseguenza anche il thread rimarrebbe bloccato e la cpu andrebbe a qualcun altro.

Se si operasse in questo modo la gestione del kernel impatterebbe sul thread.

Le API non possono essere chiamate anonimamente, ma vanno marcate *per indicare se sono bloccanti o meno*. Vanno gestiti due scenari diversi in ambito di contesti di esecuzione:

- **Process context** → l'allocazione è causata da una system call o trap. In questo caso il processo ha chiamato una API. Se la richiesta non può essere soddisfatta il processo viene messo in attesa. C'è uno schema a priorità dei processi in attesa.
- **Interrupt context** → l'allocazione è effettuata 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. API non sono interrompibili e non ci sono schemi di priorità, si svolge un'attività all-or-nothing.

## API di allocazione:

Dopo il booting è possibile chiamare le seguenti API per interagire con il memory management system.  
Sono definite in `malloc.h`

- `unsigned long get_zeroed_page(int flags)`  
removes a frame from the free list, sets the content to zero and returns the virtual address
  - `unsigned long __get_free_page(int flags)`  
removes a frame from the free list and returns the virtual address
  - `unsigned long __get_free_pages(int flags, unsigned long order)`  
removes a block of contiguous frames with given `order` from the free list and returns the virtual address of the first frame
  - `void free_page(unsigned long addr)`  
puts a frame into the free list again, having a given initial virtual address
  - `void free_pages(unsigned long addr, unsigned long order)`  
puts a block of frames of given order into the free list again
- Note!!!!!! Wrong order may give rise to kernel corruption in several kernel configurations**

- `get_zeroed_page` prende pagina, un oggetto nella free list, che verrà marcato come “user”. La pagina restituita è anche azzerata.
- `get_free_page` per prendere una pagina, non necessariamente verrà azzerata. Va specificato l'ordine del blocco.
- `get_free_pages` per lavorare su ordine superiore allo 0.
- `free_page` per rilasciare un elemento di ordine 0.
- `free_pages` per rilasciare un elemento di ordine superiore allo 0. Va specificato l'ordine del blocco.

“addr” è quello logico, anche se viene allocata memoria fisica ciò che viene tornato è l'indirizzo logico.

### flags - used contexts

`GFP_ATOMIC` the call cannot lead to sleep (this is for interrupt contexts)

`GFP_USER - GFP_BUFFER - GFP_KERNEL` the call can lead to sleep

I flags `GFP_USER - GFP_BUFFER - GFP_KERNEL` indicano la priorità in caso di process context.

`GFP_ATOMIC` è per interrupt context.

Ci sono anche altri valori per questi flags, ad esempio per chiedere zona DM piuttosto che di altro tipo. L'allocatore buddy permette di prendere DMA, permette di risalire da indirizzo logico a fisico con offset, questo è importante per costruire le tabelle delle pagine.

Si può prendere high memory?

Il flag esiste, ma l'**high memory** non è supportata da tutte le releases Linux; questo perché con l'indirizzamento logico attuale tutta la memoria è *directly mapped*, quindi va contattato un altro allocatore.

**L'high memory sta scomparendo perché tutta la memoria è raggiungibile tramite direct mapping.**

Il buddy allocator ha il vincolo di dare memoria allineata; avere la possibilità di ottenere memoria non allineata è utile, perché per un attaccante risulta più complesso scoprire la posizione di un oggetto in memoria fisica.

Di solito, si lavora con normal memory (frame directly mapped, allineata) forniti dal **buddy allocator**, ma se volessi una high memory (non directly mapped, non allineata), 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. Avere questo allocatore sopra a quello buddy permette di allocare zone di memoria di dimensioni maggiori.

Il buddy allocator usa cache, senza far riferimento a *free list* e meccanismi di *lock*. La sua cache è un concetto di "**quicklist**": insieme di pagine cached, prelevate e messe in più *quicklist*, ogni cpu ha le proprie, così non ci sono più problemi di gestione della concorrenza. Tali liste sono per-cpu (no lock né sincronizzazione, appunto). Dato un allocatore buddy, e una macchina NUMA con più allocator, quale uso? Soluzione: **mem-policy**.

### **Mem-policy:**

La memoria presa tramite le API da quale allocatore buddy proviene?

Su OS moderni esistono le mem-policies: metadati associati a threads attivi, quindi associati al **thread control block**, e alla memoria logica per capire dove deve avvenire la materializzazione della memoria in RAM. Il software del kernel deve capire quale allocatore considerare.

Il thread controlla le mem-policies per capire dove prendere memoria.

E' possibile allocare memoria in uno specifico nodo NUMA con l'API "alloc\_pages\_node", Prima di allocare il frame si consultano i mem-policy data ossia metadati associati a thread e memoria virtuale per capire dove la memoria deve essere materializzata.

Le mem-policies potrebbero essere ridirezionate tutte sullo stesso buddy allocator, la quicklist si esaurisce presto, non va bene questo approccio.

Possiamo avere mem-policies:

- per zona: l'AS della parte user non è necessariamente già materializzata, per una zona di pagine si può decidere su quale nodo andranno materializzate.
- per thread.

La system call per settare le mem-policies per il thread chiamante è: `set_mempolicy`.

Si può specificare la modalità:

- **default**: la memoria viene presa sul nodo NUMA corrente, porta a problemi di località in caso in cui il thread venga migrato
- **bind**: passo una bitmask che dice su quali nodi NUMA si può prendere memoria. Si passa da un nodo all'altro finché non si trova memoria, se nessuno dei nodi specificati la possiede fallisce anche se la memoria è disponibile su altri nodi
- **interleave**: a turno gli elementi vengono mappati su tutti i nodi. Questa modalità viene usata dall'**idle process** nell'allocare memoria per il kernel a boot time, non deve stare tutto su un unico nodo, avrei problemi di accesso.

La stessa operazione si può fare con `mbind`: permette di specificare dove materializzare memoria mmapped.

Esiste anche `move_pages`, per spostare pagine. Se un processo fa parte di un certo gruppo può fare questa operazione anche su altri processi se ne ha le capabilities. Questo servizio è sincrono, ritorna quando sono state fatte tutte le move.

*Mentre migriamo una pagina il thread corrente può provare ad accedervi, andrà in page fault finché non sarà ri-materializzata.*

### **Quicklist:**

Il buddy allocator usa lock; ciò porta problemi di scalabilità, per risolverli si fa pre-reserving della memoria.

Andiamo su un allocatore che ha già preso memoria dal buddy allocator.

Sono dei buffer pre-reserved per il caso specifico delle page table. Non c'è impatto su attività concorrenti.

Come avere quicklists per-cpu? Il puntatore alla quicklist è una variabile per-cpu.

Com'è fatta una quicklist? Viene allocata mediante l'api "quicklist\_alloc".

```

static inline void *quicklist_alloc(int nr, gfp_t flags, ...) {
    struct quicklist *q;
    void **p = NULL;

    q = &get_cpu_var(quicklist)[nr];
    p = q->page;
    if (likely(p)) {
        q->page = p[0];
        p[0] = NULL;
        q->nr_pages--;
    }
    put_cpu_var(quicklist);
    if (likely(p))
        return p;

    p = (void *)__get_free_page(flags | __GFP_ZERO);
    return p;
}

```

Beware these!!

Abbiamo un array di quicklists, ogni entry è un puntatore ad una quicklist.

Devo leggere una variabile per-cpu, come posso essere sicura di lavorare sulla stessa variabile dopo l'esecuzione di un blocco di codice? Devo marcare per il thread corrente la cpu non-preentable (nel suo TCB).

`put_cpu_var` lascia la cpu, marca nuovamente il thread come preemptable. Nel blocco in mezzo si prende un puntatore, si vede se l'oggetto esiste, se esiste lo si scollega dalla lista. Non ci sono operazioni atomiche, nessun flush dello store buffer, nessuna perdita di cicli di clock, si va semplicemente a marcare il thread. Se la quicklist è vuota si fa “`get_free_page`”, si chiama l'allocatore di basso livello, poi se il buddy allocator ha la propria free-list la usa.

Prima l'allocazione cached veniva fatta in modo diverso: la cache era condivisa tra più cpu, c'era bisogno di task per memory management condiviso.

Attenzione: non-preemptable non significa non interrompibile; si può essere interrombili, ovvero ricevere interrupt, ma non preemptable, dunque non lasciare la 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?

**Lez.20 (09/11/23)**

Quicklist oggi:

Nelle versioni più recenti del kernel, la quicklist è uno dei componenti del sistema buddy. Viene utilizzata internamente in questo sistema, in particolare per gestire operazioni di allocazione di dimensioni di 4KB. Ciò consente di utilizzare le quicklist in modo trasparente affidandosi alla stessa API utilizzata per il buddy. Le quicklist operative ad altri livelli sono comunque disponibili per la programmazione del kernel.

**SLUB o SLAB Allocator:**

Tutto ciò che abbiamo visto finora gestisce il discorso dell'allocazione della singola pagina o di multipli della singola pagina, cosa accade se vogliamo allocare una quantità di memoria al di sotto della pagina?

Si usa il buddy system sottostante per questo scopo, in particolare abbiamo *cache con oggetti (chunks) di taglie diverse*.

C'è un'organizzazione in cache diverse che gestiscono tutti gli oggetti della stessa taglia.

Ci sono due liste: *free list* e *busy list*. Come è implementato questo in Linux?

C'è una cache di memoria lato kernel: **k-mem-cache**.

Ci sono due tipi di API per operare su questi oggetti: generali (voglio memoria) o che operano su specifiche cache.

In realtà è più complicato di così: una cache ha più aree pre-riservate per consegnare chunks per due motivi:

- *Numa-awareness*: si possono consegnare aree su un nodo piuttosto che su un altro
- Quando creiamo una cache creiamo un oggetto su un nodo, ci sono metadati associati. A causa della “*lazyness*” posso avere scenari in cui il metadata esiste ma l’area di memoria ancora no, in quanto viene presa quando serve. Si possono avere cache diverse che operano sulla stessa area di memoria, cache associate.

Attenzione: la cache non è hardware, è semplicemente un layer sotto il buddy system per avere informazioni immediatamente disponibili.

Tutto questo è nascosto dietro API che permettono di eseguire allocazioni senza specificare la cache da cui prendere memoria.

API:

**kmalloc**: restituisce l’indirizzo logico del chunk allocato.

**kfree**: per liberare un chunk.

Anche questa memoria è DMA, ma rispetto al buddy allocator permette di avere maggiore scalabilità, dato che ci sono cache diverse con aree pre-riservate.

L’indirizzo logico restituito è cache-aligned, ciò porta a vantaggi nell’accesso in cache.

**kmalloc\_node**: si può interrogare la cache per acquisire memoria specificando il nodo NUMA.

Questa architettura si chiama SLUB (prima SLAB) ed è visionabile consultando lo pseudo-file: [/proc/slabinfo](#).

#### Virtualizzazione del cached allocator:

Si possono creare SLUB allocator differenti per dimensioni diverse di allocazione, se per una certa dimensione già esiste uno SLUB allocator allora quello nuovo sarà virtuale e mapperà sull’istanza esistente.

Uno SLUB allocator può essere distrutto solo quando i chunk allocati sono stati tutti rilasciati.

Possibilità di far merge di cache con una sola preallocazione di aree utilizzabili.

API:

- **kmem\_cache\_create**: creo l’oggetto cache ma non la cache effettiva. Se il puntatore alla funzione è NULL allora verrà assegnata una cache preesistente (ma nuovi metadati). Altrimenti la funzione *ctl* verrà chiamata per ogni chunk al momento dell’allocazione della cache.
- **kmem\_cache\_alloc**: per prelevare memoria dalla kmem-cache
- **kmem\_cache\_free**: per deallocare memoria
- **kmem\_cache\_destroy**: per distruggere la kmem-cache.

```
struct kmem_cache *kmem_cache_create(char *name,
                                     size_t size,
                                     size_t align,
                                     unsigned long flags,
                                     void (*ctl)(void *))
```

This is the memory initialization function

```
int kmem_cache_destroy(struct kmem_cache *cache)
```

```
void *kmem_cache_alloc(struct kmem_cache_t *cache, int prio)
```

```
void kmem_cache_free(struct kmem_cache_t *cache, void *ptr)
```

Le API viste prima sono di livello più alto rispetto a queste.

Ogni cache ha una free-list e una release-list.

Il meccanismo della mem-cache è per CPU (c'è area pre riservata per ciascuna CPU). La free-list di una mem-cache è quindi per CPU mentre la release-list è acceduta concorrentemente dalle CPU (in maniera non bloccante grazie a CAS).

#### SLUB 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 libera da consegnare quando viene richiesta un'allocazione. Se tutti utilizzano pochi chunk si ha che tutti andranno ad utilizzare lo stesso offset per i propri chunk e ci saranno dei conflitti su linee di cache. 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 allocator. Allora l'offset del primo buffer di memoria libera da consegnare è pari  $DSIZE + ALN * COL$ . Dove DSIZE è la taglia dei metadati per un cache allocator e COL è il colore.

#### Large size allocations:

Si utilizza l'allocatore `vmalloc` per allocare memoria di ampia taglia, avendo memoria fisica frammentata.

Si prende memoria logica ampia e la si mappa su zone diverse di memoria fisica: logicamente ho memoria contigua, fisicamente no, per cui non si tratta necessariamente di memoria directly mapped.

La `vmalloc` può essere bloccante, in interrupt context non viene mai chiamata.

Tipicamente la `vmalloc` viene usata quando si monta un modulo kernel. Qualunque struttura dati nel kernel non sta su memoria directly mapped, siamo noi a dover chiamare il buddy allocator per avere DMA.

Per deallocare memoria si usa la `vfree`. Sopra questo oggetto si possono creare ulteriori allocator per avere memoria non directly mapped. Gli stack allocators prendono memoria pre-riservata secondo questo schema.

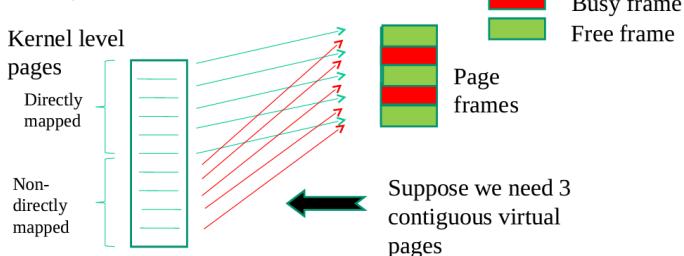
#### kmalloc vs vmalloc:

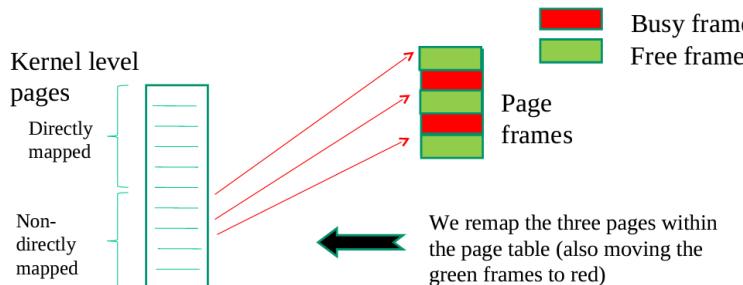
- Allocation size:
  - 128 KB for `kmalloc` (cache aligned)
  - 64/128 MB for `vmalloc`
- Physical contiguousness
  - Yes for `kmalloc`
  - No for `vmalloc`
- Effects on TLB
  - None for `kmalloc`
  - Global for `vmalloc` (transparent to `vmalloc` users)

La mappatura di pagine kernel ha una natura globale. Quando si esegue una `vmalloc` o una `vfree` su un core tutti gli altri core devono visualizzare l'aggiornamento. Dunque i TLB devono essere aggiornati.

#### Operazioni della vmalloc:

Based in remapping a range of contiguous pages in (non contiguous) physical memory





Clearly with `vmalloc` we typically remap much larger blocks of pages

### ***Re-mapping delle pagine kernel vs stato dell'hardware:***

Questi allocatori come interagiscono con l'hardware?

La componente hardware interessata è il TLB Translation Lookaside Buffer, struttura dati che cacha le associazioni indirizzo logico – indirizzo fisico.

Se si fa una `vfree` si indica che l'associazione indirizzo logico-fisico non deve più essere presente, quindi il TLB viene invalidato.

Su x86\_64 non c'è un'istruzione macchina che permette di invalidare il TLB; questo è un aspetto importante da esaminare.

L'update del TLB su architetture moderne è implicito o esplicito?

### Eventi interessanti per reset/ripopolamento del TLB:

L'aggiornamento del TLB può essere implicito o esplicito. Per processori x86 l'automazione è solo parziale. Solitamente avviene in modo automatico all'update del contenuto del registro CR3 (cambiamenti della page table). Invece i cambiamenti all'interno della page table corrente non si riflettono in modo automatico sul TLB. Per operare in modo esplicito sul TLB si fa uso di **kernel hooks** (nel caso di gestione totalmente implicita sono mappati su operazioni nulle). A livello software i threads devono eseguire operazioni esplicite per riallineare il TLB. Gli eventi sul TLB possono essere classificati in base alla scala (globali o locali) e alla topologia:

- Globali: indirizzi virtuali accedibili da tutti i core in modo concorrente.
- Locali: indirizzi virtuali accedibili in concorrenza time-sharing (cioè nello stesso address space).
- remapping degli indirizzi virtuali in indirizzi fisici
- modifica regole di accesso a indirizzo virtuale

I costi del flush del TLB sono vari. 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

### API:

- `flush_tlb_all`: per flushare tutti i TLB. Qualcuno modifica CR3 per flushare il suo TLB, poi con un interrupt notifica che va riscritto CR3 a tutte le altre cpu. Poi devo accorgermi che operazioni sono state svolte, questo si fa mediante software, che diventa una barriera. `vmalloc` e `vfree` possono usare questa api. Sostanzialmente metto delle barriere, perché tutti devono completare l'operazione. Il che è molto dispendioso in termini di tempo.
- `flush_tlb_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). Mandiamo avvertimento se sta girando su un thread di questo processo. Quindi non aspettiamo che altri completino l'aggiornamento, avvertiamo solamente.
- `flush_tlb_range`: si utilizza quando va flushata una parte del TLB di livello user, ad esempio quando si chiama `mprotect`.
- `flush_tlb_page`: si utilizza per flushare una singola entry del TLB
- `invlpg`: si usa per invalidare un'entry del TLB.
- `flush_tlb_pgtables`: 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
- `update_mmu_cache`: Viene chiamata dopo un page fault. Indica che per l'indirizzo virtuale addr esiste una nuova traduzione a pte. Le architetture poi decidono cosa fare di questa informazione. Alcune fanno un preloading delle entries TLB

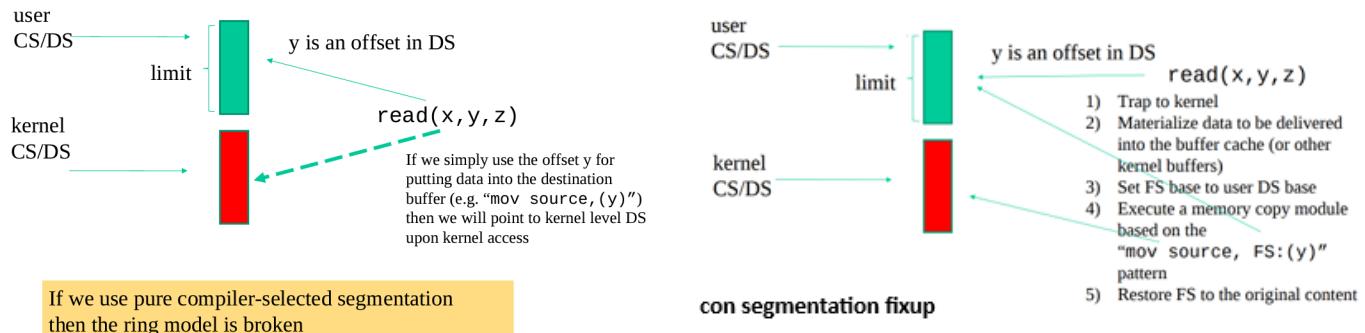
## Cross ring data move:

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.

### Segmentazione flessibile:

E' il caso di **x86 protected mode**. I segmenti possono avere offset diversi da zero e posso quindi avere una separazione piena tra i segmenti. Tuttavia è necessario rendere trasparente a livello applicativo questa segmentazione.

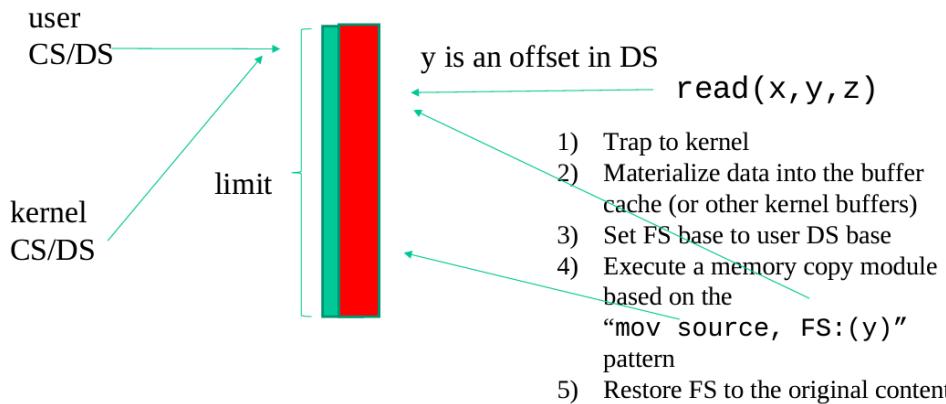
Alla fine, questa soluzione consiste nell'avere all'interno di uno spazio di indirizzamento di un'applicazione un certo numero di segmenti user level che ricoprono gli indirizzi di memoria più bassi e un certo numero di segmenti kernel level che ricoprono gli indirizzi di memoria più alti. Consideriamo 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 in quanto leggiamo dati di livello kernel e non user.



La soluzione al problema appena esposto consiste nel rendere "*hand crafted*" (= scritti in maniera machine dependent dal programmatore del software del kernel) i pezzi di codice del kernel per lo spostamento cross ring dei dati e non più generati dal compilatore. Più specificatamente, è possibile sfruttare un selettore di segmento programmabile (come FS) per mappare momentaneamente il segmento FS sul segmento DS *user level* e poi applicare il displacement y a FS per il movimento dei dati: in tal modo, per la `read()`, viene utilizzata un'area di memoria user space e non più kernel space. L'operazione appena descritta è chiamata **segmentation fixup**. Chiaramente, comporta dei costi relativi al cambio di stato dei registri del processore (i.e. dei selettori di segmento), che di fatto richiede degli accessi aggiuntivi alla memoria.

### 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 questo caso far puntare FS a DS non funziona dal momento che user DS e kernel DS sono sovrapposti, dunque verrà letto il segmento kernel.



Per risolvere ciò i sistemi operativi attuano un "taglio" software a ciascun address space delimitando parte user e parte kernel. Per ogni thread ho un metadata nel TCB chiamato `addr_limit` che indica fino a dove possono puntare i suoi pointers. 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). E' chiaro che alcuni thread kernel devono avere accesso all'intero address space. Per recuperare la struct contenente questo metadata si usa l'API `get_fs()` e per modificarlo (se kernel < 5.9) `set_fs()`. Attualmente non è più modificabile ma è definito a compile time

### Soluzione odierna:

Tutto questo nel kernel Linux non c'è più. "`addr_limit`" viene settato a compile time alla lower half dell'AS. Esistono patch per lavorare oltre il limite, usano un nuovo sottosistema nel kernel.

### API:

```
unsigned long copy_from_user(void *to, const void *from,
                             unsigned long n)
```

Copies n bytes from the user address(from) to the kernel address space(to).

```
unsigned long copy_to_user(void *to, const void *from, unsigned
                           long n)
```

Copies n bytes from the kernel address(from) to the user address space(to).

```
void get_user(void *to, void *from)
```

Copies an integer value from userspace (from) to kernel space (to).

```
void put_user(void *from, void *to)
```

Copies an integer value from kernel space (from) to userspace (to).

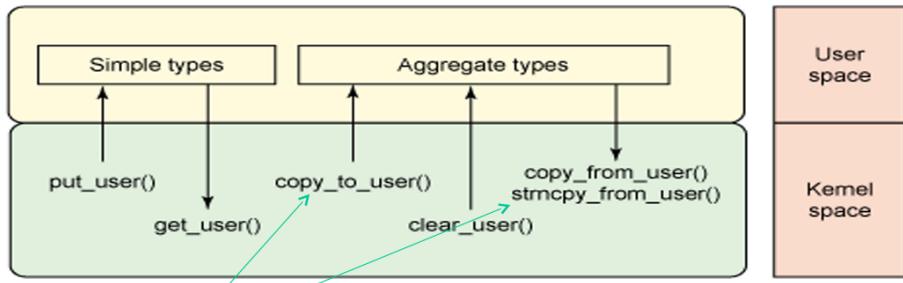
```
long strncpy_from_user(char *dst, const char *src, long count)
```

Copies a null terminated string of at most count bytes long from userspace (src) to kernel space (dst)

```
int access_ok(int type, unsigned long addr, unsigned long size)
```

Returns nonzero if the userspace block of memory is valid and zero otherwise

`access_ok()` è un'API su cui poggiano anche le altre. Restituisce un numero diverso da zero se il blocco di codice passato è un blocco user valido. La validità non dipende solo dal fatto se l'indirizzo rispetta l'`addr_limit` ma anche se quelle pagine sono materializzate. Se così non fosse il kernel tenterebbe di accedervi causando un SEGFAULT a livello kernel.



These functions return the residuals  
(bytes not managed)

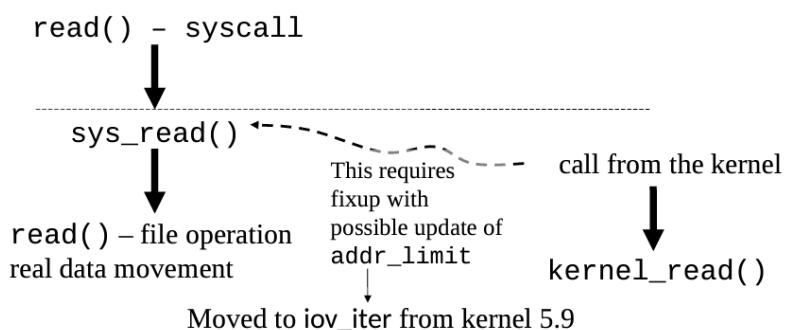
Most of them ground on  
`access_ok()`

The actual copy operation may lead the thread to sleep  
(we will be back to this issue when talking of contexts)

### Ridondanza delle api:

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. 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:

Esaminiamo un aspetto interessante relativo al lavoro su copie di memoria.

Quando si esegue in kernel mode si può necessitare di attività di `memcpy()` con indirizzo sorgente arbitrario (ad esempio relativo a parte user). In questo caso andremmo a copiare una parte user in una zona kernel.

Con la *constrained supervisor mode*, andiamo ad inserire dei vincoli per mezzo di bit di controllo addizionali.

Il supporto attuale su x86 è dato da:

- **SMAP**: blocca gli accessi alle pagine user quando si lavora a CPL0 (massimo, kernel mode)
- **SMEP**: blocca il fetch di istruzioni dalle pagine user quando si lavora a CPL0.

Vengono attivati mediante il 20esimo e 21esimo bit di CR4.

Possono essere disabilitati temporaneamente, ad esempio nei servizi `copy_to_user` e `copy_from_user`.

### Timeline di copy\_to\_user:

Se volessi effettivamente copiare in memoria utente.

- Viene effettuato il check su `addr_limit` per verificare se l'operazione è fattibile.
- Viene determinata la quantità di dati che può essere effettivamente copiata. Lo fa `access_ok`, tempistica  $O(n)$ , perchè prende Thread Control Block, poi Tabella management per vedere se c'è la quantità richiesta.
- Viene disabilitato lo SMAP.
- Viene effettuata la vera e propria copia dei dati.
- Viene riabilitato lo SMAP.

Tutto ciò serve a fixare il problema nato dallo user, il quale passa parametri non corretti. Ma quante volte capita? Secondo alcune statistiche, quasi mai.

### Segmentation fault di livello kernel:

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 SEGVFAULT**. 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). Alla fine:

- 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.

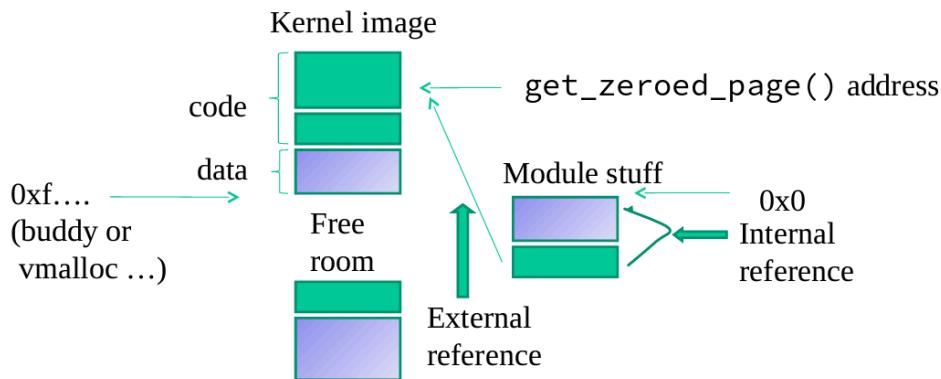
## Linux modules:

I moduli sono parti in più aggiunte al kernel, essi possono utilizzare il kernel e viceversa.

Si possono montare quando il kernel è già running. Si possono usare per avanzare nello sviluppo, possono essere aggiunti alla mainline e possono essere utilizzati anche per lo startup del kernel.

### **Step per l'inserimento di 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 collocata questa memoria in modo tale da risolvere i riferimenti interni del modulo (che siano essi relativi ai dati o relativi al codice). Ricordiamo che il modulo utilizza *indirizzi virtuali*.
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.



### **Cosa accadeva nel kernel 2.4:**

Il modulo “.o” è un oggetto non eseguibile, ma con tutte le relazioni tra oggetti del modulo ed esterno.

Per embeddare un modulo si usano comandi shell: il “.o” diventa un eseguibile, che poi viene caricato nel kernel. Gli shell commands fanno questo a livello user, ciò significa che lo user viene a conoscenza dell’immagine del kernel. Si hanno le seguenti system call:

- `create_module`: riserva un buffer logico kernel e assegna un nome al modulo
- `init_module`: carica l’immagine finalizzata del modulo nel buffer. Chiama la funzione di setup del modulo. La funzione di setup prende come parametri valori di variabili globali che possono essere settati come parametri della `init_module`
- `delete_module`: chiama la funzione di shutdown del modulo e rilascia il buffer. E’ impossibile rimuovere moduli che non abbiano la funzione di shutdown. Se ce l’ha ma è lockato non è smontabile a meno che non sia forzata l’operazione (rischioso perché i thread potrebbero avere riferimenti a quella memoria). Per tenere conto di chi ha il riferimento dell’area di memoria c’è un contatore tra i metadati del modulo.

### **Cosa accade a partire da kernel 2.6:**

Il modulo è un “.ko”, ovvero un kernel object. Esso non è ancora eseguibile, ma ha più metadati rispetto ad un “.o”, necessari al kernel per finalizzare il file.

Il lavoro di raccordo è fatto dal kernel, ovvero associa ogni simbolo alla sua posizione nell’AS logico.

Parlando di system call:

Non c'è più la `create_module`, inglobata dalla più complessa `init_module` (tra i parametri si passa il file che rappresenta il modulo, tutto il resto viene fatto in maniera trasparente all'utente) e dalla `delete_module`.

#### Parti in comune:

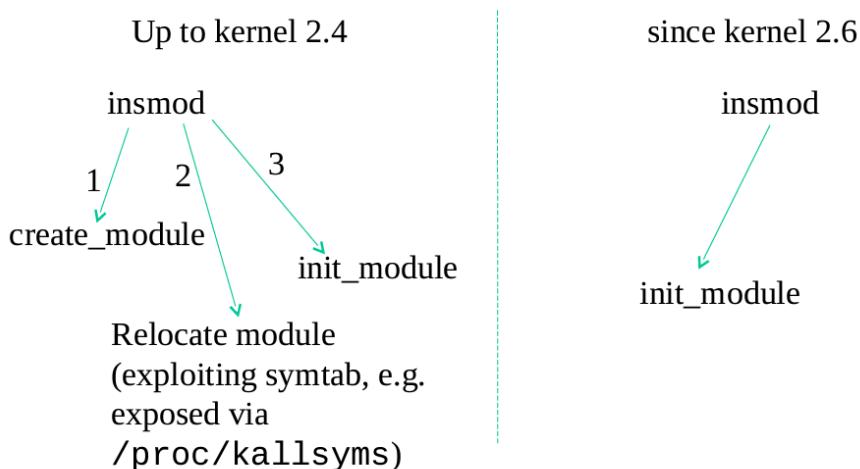
La funzione di startup deve chiamarsi `init_module` e la funzione di uscita `cleanup_module`.

#### Passaggio dei parametri

E' possibile passare parametri al modulo non come parametri di funzione. Essi si passano come valore di variabili globali definite all'interno del modulo. Bisogna marcare esplicitamente queste variabili come "module parameters". Per far ciò si usano le seguenti macro in cui si specifica il nome della variabile e il tipo del dato: `MODULE_PARM(variable, type)` (`old style`) e `module_param(variable, type, perm)`; dove `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. 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. Nella directory `/sys` ci sono anche altri pseudo-file riguardanti lo stato del modulo. Per passare degli array come parametri si usa `module_param_array()`. Prende 4 parametri: il nome array, il tipo, l'indirizzo di una variabile che specifica la taglia dell'array, i permessi.

#### Montaggio e smontaggio modulo

Un modulo può essere caricato dall'amministratore tramite lo shell command `insmod`. E' possibile passare parametri nella forma `variable=value`. Lo smontaggio può essere fatto con il comando `rmmod`. Possiamo anche caricare un modulo tramite `modprobe` che identifica e monta anche tutti i moduli da cui dipende il mio. Come possiamo vedere, fino al kernel 2.4 la `insmod` si occupava di finalizzare il modulo andando a consultare la tabella dei simboli. Dal 2.6 in poi insmod triggerà solo l'esecuzione di `init_module`



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. Questa è una lista di API aggiuntive:

- `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`, la quale contiene i metadati).
- `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.
- `int init_module (void *module_image, unsigned long len, const char *param_values)`: (2.6), 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`. I flag sono usati per indicare se si deve eseguire il check sulle signature dei simboli (cioè se c'è match tra i tipi).

### **Struttura dei moduli:**

#### Header:

```

#define __KERNEL__           For inclusion of header file parts
                           with pre-processor
                           directive ifdef __KERNEL__

#define MODULE               For inclusion of header file parts with
                           Pre-processor directive ifdef MODULE

#include <linux/module.h>
#include <linux/kernel.h>

.....
#include <linux/smp.h>    ← SMP specific stuff

```

Possiamo includere degli headers.

Define: `__KERNEL` e `MODULE`, includo degli `ifdef`.

Tipicamente negli headers ho già le inclusioni necessarie, quindi queste macro non si usano più molto.

#### Contatore modulo:

##### *Kernel 2.4:*

In versioni ancestrali il contatore si toccava con `MOD_INC_USE_COUNT` e `MOD_DEC_USE_COUNT`. Sono macro che modificano negli headers il valore del contatore.

`MOD_IN_USE` è una query sul valore del contatore, dice se un modulo è lockato per l'utilizzo o no.

La macro per resettare il contatore è: `CONFIG_MODULE_FORCE_UNLOAD`.

##### *Kernel 2.6:*

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. 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.
- `module_put` (`struct module *module`): decrementa di un'unità lo usage count del modulo specificato come parametro.
- `CONFIG_MODULE_UNLOAD`: per controllare se il counter è strettamente maggiore di zero.

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)`

### ***Esportazione simboli:***

Nei moduli non si possono utilizzare tutti i simboli del kernel.

Nel kernel c'è una tabella con tutti i simboli, solo gli elementi esportati possono essere utilizzati dal modulo montato. I simboli possono essere esportati con la macro “`EXPORT_SYMBOL`”.

I simboli del kernel sono divisi in: `static` (non sono nella kernel table), `available` (sono nella kernel table) ed `exported` (sono nella kernel table e sono utilizzabili dai moduli).

In kernel 2.4 tutti i simboli erano in `kallsyms`.

Oggi in questo pseudofile non c'è la lista dei simboli, ma non la loro posizione in memoria; in kernel 2.6 per avere che le macro funzionino sul nome delle variabili, oltre che su quello delle funzioni, il kernel deve essere compilato con `CONFIG_ALLSYMS`.

Su una certa versione del kernel per sapere quali simboli sono utilizzabili bisogna consultare `"/lib/modules/<kernel version>/build/module_symvers"`.

Tuttora il kernel Linux offre api `kallsyms_lookup_name`, che prende il nome in input per sapere dove si trova in memoria. Questa api era esportata fino al kernel 5.7, tramite essa si può sapere tutto.

### ***Sottosistema kprobe:***

Linux è self-patching; il kernel può intercettare e modificare se stesso. Intercetta un'istruzione e fa altro.

Il sottosistema che permette di farlo è **kprobe**, e le API sono:

- `int kprobes_register_kprobe(struct kprobe *)`: passo una tabella che descrive la kprobe
- `void unregister_kprobe(struct kprobe *)`
- `int disable_kprobe(struct kprobe *kp)`: disabilita la probe
- `int enable_kprobe(struct kprobe *kp)`: abilita la probe.

### **Struct kprobe:**

Contiene campi nome e addr.

Metto nome null, scrivo io addr, non voglio applicare probe ad un sottosistema in memoria, ma voglio intercettare l'indirizzo che passo io.

Si può abilitare e disabilitare la probe.

Se la disabilito posso contare mediante il campo “`missed`” quante volte sono passato su addr senza modificarne il comportamento.

Si passano due fuction pointers: pre-handler prima di passare su un'istruzione macchina cui ho applicato la probe e post-handler viene eseguito dopo. Questi vengono eseguiti single step subito prima e/o subito dopo.

```

<linux/kprobes.h>

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

}};


```

These are pre/post  
the single stepped  
execution of the  
instrumented  
instruction/opcode

## Lez.23 (16/11/23)

### kprobe vs kretprobe:

Posso mettere un wrapper all'accesso e all'uscita a qualsiasi sottosistema del kernel.

Con **kretprobe** siamo interessati all'intero sottosistema, mentre la **kprobe** si può applicare a singole istruzioni.

C'è una tabella kretprobe in cui specificare le informazioni del sottosistema cui si è interessati.

Se gli handlers sono nulli sto solo intercettando l'accesso al sottosistema.

Quando usiamo kretprobe al ritorno della funzione non si deve ritornare al chiamante effettivo, ma al post handler.

Il salvataggio degli indirizzi di ritorno impatta su "maxactive"; il sottosistema di kernel probe può mantenere un numero massimo di return addresses, se qualche altro thread cerca di entrare viene impedito l'accesso ed il numero di istanze perse viene salvato in "nmissed".

I pre-handlers possono sempre essere attraversati, nmissed è inherente ai post-handlers.

In caso di preemptability la cpu può andare a qualcun altro, quindi il valore di default per max-active è pari a  $\max(10, 2 * NR\_CPUS)$ , nel caso non preemptable maxactive è pari al numero delle cpu.

Ecco le API:

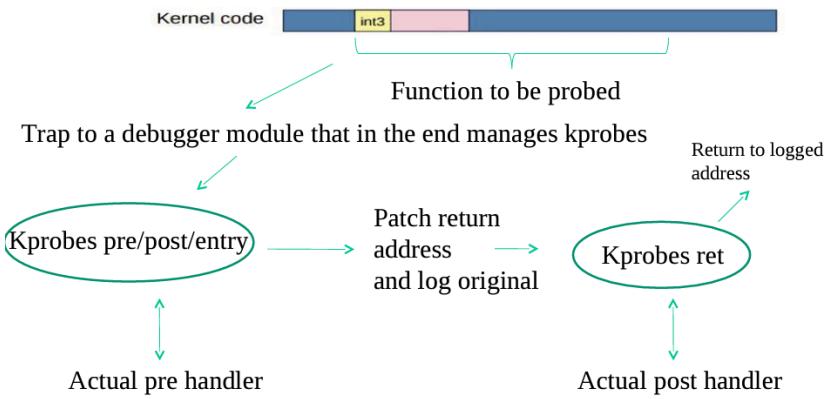
- int register\_kretprobe(struct kretprobe \*)
- int unregister\_kretprobe(struct kretprobe \*)
- static inline int disable\_kretprobe(struct kretprobe \*rp)  
return disable\_kprobe(&rp->rp);
- static inline int enable\_kretprobe(struct kretprobe \*rp)  
return enable\_kprobe(&rp->rp);

### Come funzionano le probes:

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, ovvero un'istruzione che solleva una trap per attivare un modulo di debug che, in fin dei conti, gestisce le kprobe. Quando il breakpoint viene raggiunto c'è una trap che passa il controllo ad un software del sottosistema **kprobe**. Viene eseguito il **pre-handler**.

Successivamente si salva il return address e lo si aggiorna per eseguire l'handler di uscita.

Infine si esegue l'handler di uscita e si torna al return address:



### Handlers KPROBE:

Prendono una serie di parametri; due sono quelli di base: puntatore allo snapshot di cpu e puntatore alla tabella che descrive la probe.

Il puntatore allo snapshot di cpu è modificabile da questi handlers, è un oggetto read-write, se lo aggiorniamo possiamo intrometterci:

Ecco gli handlers:

- `typedef int (*kprobe_pre_handler_t) (struct kprobe*, struct pt_regs*)`
- `typedef void (*kprobe_post_handler_t) (struct kprobe*, struct pt_regs*, unsigned long flags)`
- `static int (*handler) (struct kretprobe_instance *ri, struct pt_regs *regs)`
- `static int (*entry_handler) (struct kretprobe_instance *ri, struct pt_regs *regs)`

Come vediamo queste funzioni prendono in input lo snapshot della CPU. Per recuperare il valore di ritorno di una kprobe si usa la macro `regs_return_value(regs)`.

### Probing deny:

C'è una blacklist, uno pseudofile che contiene tutti i sottosistemi non kprobing, le chiamate kprobe su questi sottosistemi falliscono. Attualmente le kprobe sono non-preemptable cioè gli handler non possono contenere servizi bloccanti. Questo perché l'indirizzo della kprobe corrente è salvato all'interno della variabile per-CPU `current_kprobe`. Se ci fosse la preemption e c'è un cambio di contesto tale valore potrebbe essere sovrascritto.

### jprobe:

Una variante delle kprobe è la jprobe.

Prima di entrare in un sottosistema entro in un wrapper che ha la stessa segnatura della funzione che chiamo; in questo modo non devo cercare in cpu i parametri che gli voglio passare. Internamente ho la tabella kprobe, che è un unico sottosistema.

### Esempio:

```

int my_handler (....)

.....
static struct jprobe my_probe

my_probe.kp.addr = (kprobe_opcode_t *)target_address;
my_probe.entry = (kprobe_opcode_t *)my_handler;
register_jprobe(&my_probe);

```

### Linux kernel versioning:

Abbiamo delle macro utilizzabili per risalire alla versione corrente del kernel:

- 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.
- 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`.

Normalmente il check si fa in questo modo:

```
#if LINUX_VERSION_CODE > KERNEL_VERSION (x,y,z)
    <esegui in un certo modo>
#else <esegui in un altro modo>
#endif
```

### Rinominare startup e shutdown function:

- `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`.

### **Messaggistica col kernel:**

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()` (sono disponibili solo allo startup del virtual file system). Si utilizzano dei sottosistemi già presenti nell'immagine kernel (in particolare `printk()`). I messaggi possono essere scritti in un buffer circolare proprio del kernel (operazione che viene eseguita sempre) oppure in console (operazione che, essendo costosa, viene eseguita in base al livello di criticità).

Esempio di uso: `printk(KERN_INFO "Message: %s\n", arg);`

I livelli di criticità 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 */
```

Ci sono api che applicano ad un messaggio il livello di priorità; gli passiamo il contenuto del messaggio. Normalmente, il passaggio dei pointer avviene in maniera cifrata. La si bypassa con `%px` invece che `%p`.

Per la `printk()` esistono 4 parametri configurabili che determinano quale deve essere l'effettivo trattamento dei messaggi di output. Sono associati alle seguenti variabili:

- `console_loglevel`: livello di priorità che ci dice che i messaggi devono andare anche in console; si può riconfigurare mediante modifica di uno pseudofile.
- `default_message_loglevel`: viene associato ai messaggi che non specificano esplicitamente un livello di priorità
- `minimum_console_loglevel`: è il livello minimo che permette di fare il log dei messaggi nella console
- `default_console_loglevel`: è il livello di default per messaggi destinati alla console.

Si trovano nello pseudofile /proc/sys/kernel/printk.

Alzando il valore di console\_loglevel aumentiamo il traffico in console; la printk è non bloccante, l'attesa è attiva, quindi non va bene.

#### Come lavorare sul buffer circolare?

Tramite: `int syslog(int type, char *bufp, int len)`

Passiamo il codice numerico che indica l'operazione che si vuole eseguire sul buffer circolare (esistono dei valori predefiniti), un puntatore ad un'area di memoria su cui eseguire e il numero di byte coinvolti. Questi ultimi due parametri servono per scambiare informazioni.

Questo flusso non si può chiudere e riaprire, è sempre aperto.

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

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

#### Caratteristiche del buffer circolare:

La sua taglia continua ad aumentare nelle releases recenti.

La gestione effettiva dei messaggi avviene con la `printk()`, tale che la delivery debba essere “at most once”; la scrittura in console è sincrona.

Correntemente si sta andando verso implementazioni lockless del buffer circolare.

#### La funzione “panic”:

E' un'altra funzione che produce messaggi.

Riporta su console i messaggi che ha in input, internamente chiama la `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, l'esecuzione del kernel viene bloccata.

## Kernel level task management:

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

- User mode
- Kernel mode
- Esecuzione del gestore di interrupt

Posso avere un thread che può eseguire user o kernel, su cui ad un certo punto c'è la richiesta di gestione di un interrupt. Il punto di esecuzione del thread in cui ciò avviene è assolutamente **non deterministico**.

Possiamo avere due thread che si contendono una risorsa, ma se colui che la prende per prima poi viene colpito da una interruzione, che porta all'esecuzione del gestore, il secondo thread subirà dei ritardi.

I SO moderni risolvono il problema del non determinismo con la tecnica del “*time reconciliation*”.

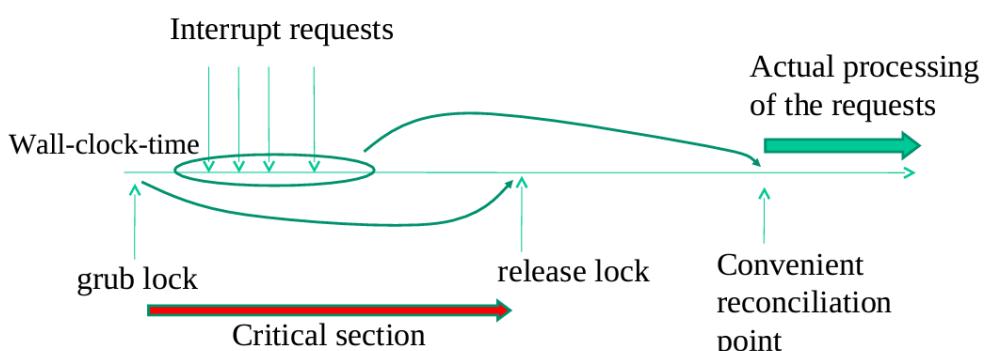
### Riconciliazione temporale:

Decido i momenti in cui devono avvenire le attività: le richieste di interrupt vengono annidate secondo uno schema “*deferring*” della richiesta, ovvero la richiesta può essere eseguita in seguito, ma non subito.

Il momento in cui il thread rilascerà la cpu non è impattato dall'esecuzione della gestione degli interrupt.

**Many-to-one**: tutte le richieste ad un certo punto vengono processate in un'unica attività, dunque bisogna gestire la priorità.

Quando accetto l'interrupt, è perchè c'è il 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.



Vedremo cosa accade dal momento in cui il dispositivo fa la richiesta di interrupt ed il momento in cui viene eseguita.

### Punti di riconciliazione o Safe Places:

Ci deve essere garanzia che la richiesta venga processata.

Il supporto convenzionale prevede che essi siano:

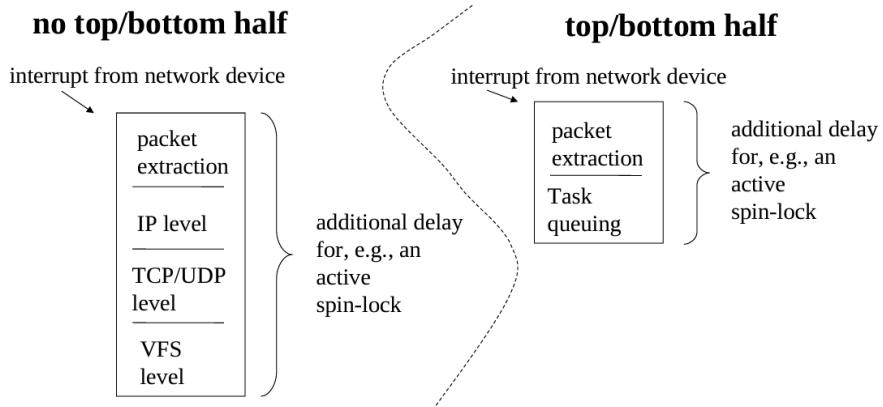
- Ritorno da una system call: abbandoniamo il codice kernel senza problemi. Il supporto tecnologico è di livello applicativo. Servono delle app di livello user, non scontato.
- Context-switch: si rilascia la cpu in favore di qualcun altro, il supporto tecnologico è l'idle process.
- Riconciliazione in process context: 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. Solitamente coinvolge demoni di sistema

## **Top-bottom half programming:**

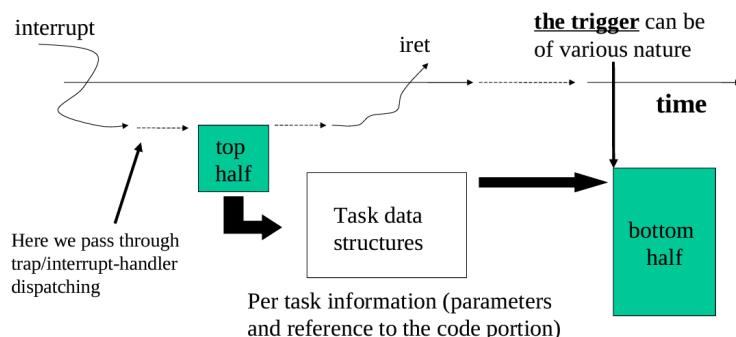
Quando avviene una richiesta di interrupt significa che qualcosa nello stato del dispositivo va modificato.

Se l'interrupt non viene subito eseguito e non si modifica lo stato si potrebbe creare un blocco.

La tecnica top-bottom half programming fa sì che la **top half** venga eseguita immediatamente quando arriva un interrupt, questa è un'attività minimale per mettere tutto a posto, anche se il thread è in CS. E' un'attività serializzata, non interrompibile.

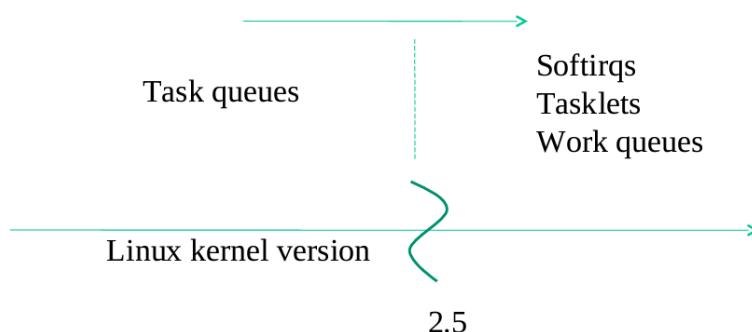


La **top half** esegue delle operazioni minimali di gestione dell'interrupt tra cui schedulare il task *bottom half* in una struttura dati. La **bottom half** serve a finalizzare la gestione dell'interrupt nel punto di riconciliazione. La TH registra la BH in una struttura dati task queue. Registra pointer alla funzione e dati di input della BH. Permette di gestire eventi in modo “timely”, faccio attività minimali, senza mantenere bloccate le risorse fino all'accadimento dell'evento. Questo schema è utile per evitare scenari di **deadlock**, infatti nel momento in cui più threads si trovano a dover gestire degli interrupt aumenta la concorrenza, per questo si svolgono solo attività minimali e la gestione vera e propria viene accodata e rimandata.



## Supporto Linux:

Tradizionalmente avevamo che la top-half scriveva in “task queue” e la bottom half vi leggeva; ad un certo punto questo procedimento non andava più bene per motivi legati allo sviluppo hardware.



Dobbiamo sapere cosa andare ad utilizzare in situazioni diverse.

## Task queues:

Tre diverse:

- **tq\_immediate**: coda contenente i task che devono essere eseguiti con la cadenza degli interrupt da timer (non alla ricezione di un interrupt da timer) oppure quando si ritorna da una system call.
- **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).

Facilities che le sfruttano:

- **tq\_struct**: è un elemento della task queue; contiene un puntatore alla routine ed un puntatore al parametro della routine.
- **DECLARE\_TASK\_QUEUE**: per creare una task queue nuova,

```
struct tq_struct {
    struct tq_struct *next; /*linked list of active bh's*/
    int sync; /* must be initialized to zero */
    void (*routine)(void *); /* function to call */
    void *data; /* argument to function */
}
```

These are the task specific fields

## API:

- **int queue\_task(struct tq\_struct \*task, task\_queue \*list)**: aggiunge un task.
- **run\_task\_queue(task\_queue \*list)**: esegue tutti i task.
- **int schedule\_task(struct tq\_struct \*task)**: inserisce il task nella coda di default **tq\_schedule**.

## Limitazioni task queues:

Il kernel Linux invoca "do\_bottom\_half" in due punti: ritorno da una system call e durante **schedule()**.

Bisogna però gestire i seguenti aspetti:

- le Bottom Half non devono contenere servizi bloccanti in quanto, essendo eseguibili da un qualsiasi thread, potrebbero essere eseguite da un thread di natura non bloccante.
- Se viene schedulato un thread ad alta priorità questo deve prima eseguire i task e poi avviare la sua esecuzione, ritardandola.

## Esempio:

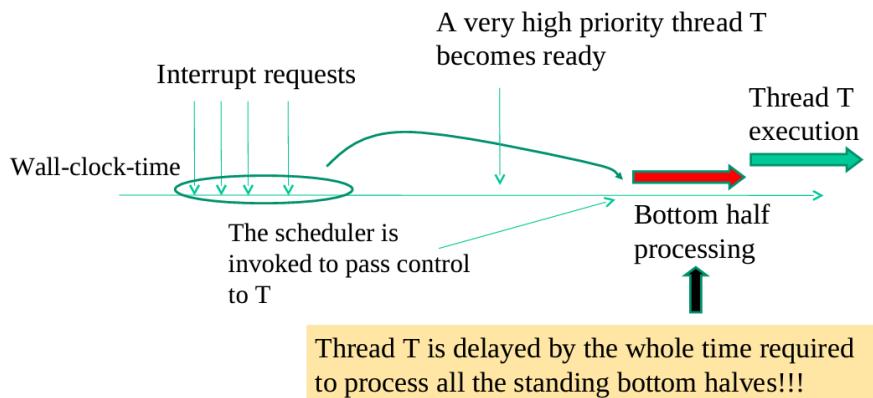
Supponiamo di avviare l'esecuzione dei bottom-half  $f_1, f_2, \dots, f_n$  contestualmente alla funzione **schedule()**. Questo vuol dire che  $f_1, f_2, \dots, f_n$  devono essere eseguiti mentre un thread A sta rilasciando la CPU a vantaggio di un altro thread B e, quindi, devono essere presi in carico 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.

Il problema è legato al fatto che l'esecuzione di tutti i *bottom-half* (e quindi, la gestione di tutti gli interrupt) è demandata allo stesso thread. 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). Di conseguenza, abbiamo bisogno di:

- Una maggiore scalabilità e località.
- Una maggiore flessibilità (ad esempio specificare il nodo NUMA).
- 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 3 tipi di **deferred** interrupts: le **softIRQ**, le **tasklet** e **work queue**), togliendo quindi il concetto delle task queue.

Deferred work: Quando arriva un interrupt non viene immediatamente eseguita tutta l'attività richiesta per gestirlo, ma viene eseguita un'attività minimale, che va poi a schedulare un lavoro differito da eseguire in seguito, che è il vero e proprio gestore dell'interrupt, questo schema si chiama *top-bottom half*, la top half è l'attività minimale svolta subito all'arrivo dell'interrupt, la bottom half è quella svolta in deferred work.



### SoftIRQ:

Già si sa qual è la bottom half, non avviene nessun inserimento in coda, si conosce già il codice da invocare nel punto di riconciliazione. Sapendo già dove sono le bottom half, e con il top half identificato dalla *interrupt table*, il top half risulta più ridotto, basta settare il flag di esecuzione della bottom half.

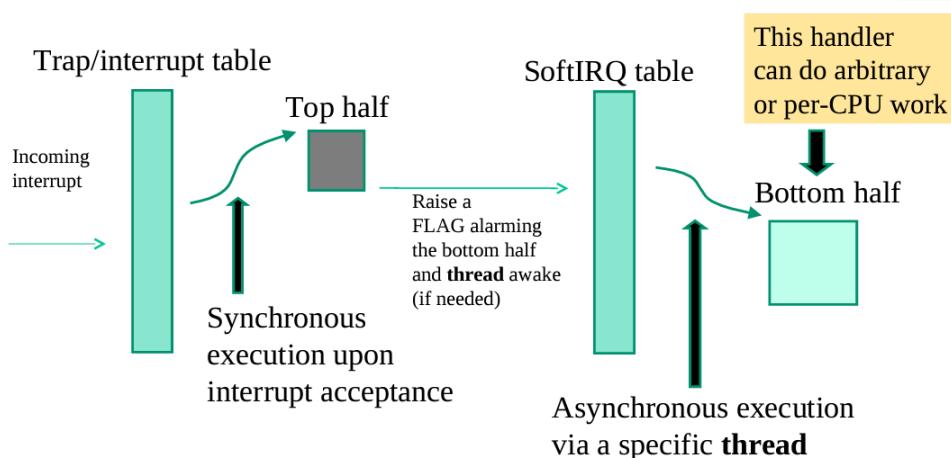
Il concetto di accodamento viene esteso: il bottom half è un oggetto che va a guardare nella coda se ci sono azioni da svolgere; questa coda è la coda di **tasklets** su Linux. (*La coda di attività che farò deferred*).

Non accodiamo i bottom half, ma le informazioni che gli servono per processare.

Per processare le tasklet c'è un bottom half apposito.

La top half su x86 è identificata dalla IDT, o Trap Interrupt Table, come visibile nella foto a seguire.

La top half va nella SoftIRQ table e setta il bit a 1 così da risvegliare il demone della gestione delle attività della bottom half.



### SoftIRQ table:

C'è un **softIRQ daemon per ciascuna CPU**. Scorre la softIRQ table ed esegue la bottom half relativa alle entries con il bit di esecuzione pari a 1.

Le applicazioni critiche possono avere priorità più alta di questo demone e ciò garantisce maggiore flessibilità. La tabella ha delle entries associate ad un livello di priorità, a ciascun livello si associa la gestione di specifici interrupt.

```

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
}

```

High priority queued stuff

Stuff to do on timers or reschedules

Normal priority queued stuff

**TIMER\_SOFTIRQ** e **HRTIMER\_SOFTIRQ** sono relative ai timer, su architetture moderne ci sono almeno due timers da cui possono arrivare gli interrupt.

Su **HRTIMER** si può operare in maniera semplice, tipicamente è usato per l'assegnazione delle cpu.

Si possono listare i demoni softIRQ con **ksoftirq[n]**, dove n è il numero della cpu affine.

Si può configurare l'arrivo degli interrupt soltanto su alcune cpu, colpendo così, ad esempio, solo i threads su una certa cpu. Questo genera affinità tra interrupt e demone che lo processerà.

Questo meccanismo permette di: processare gli interrupt di applicazioni critiche e avere località nel placement delle informazioni sull'architettura RAM.

Linux dice l'affinità corrente tramite lo pseudo file **/proc/irq/\$IRQ\_NUMBER/smp\_affinity**.

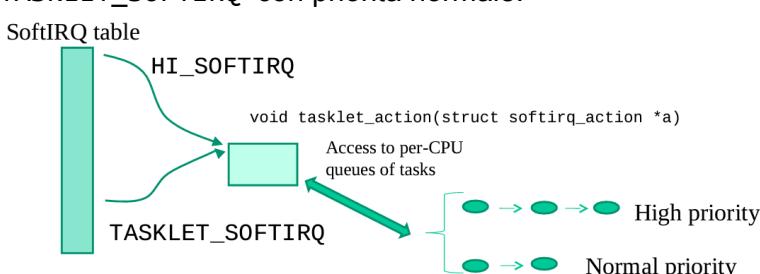
Se settato, il valore **f** può essere inoltrato l'interrupt su tutte le cpu, solitamente la scelta avviene tramite round robin.

### Vantaggi delle softIRQ:

- Esecuzione multithread delle attività nella bottom half.
- 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.
- Associazione dell'esecuzione delle attività ai core della CPU se necessario (ad esempio, località su macchine NUMA)
- Il programmatore può accodare task per gestire gli interrupt, sfruttando alcuni dei bottom-half presenti nell'architettura, (vedi i tipi **HI\_SOFTIRQ** e **TASKLET\_SOFTIRQ**)

### Tasklet:

Le tasklet sono più utilizzate delle SoftIRQ anche se sono costruite su due softirq: **TASKLET\_SOFTIRQ** e **HI\_SOFTIRQ**. In poche parole le tasklet sono softirq che possono essere *allocated a runtime* e, a differenza delle softirq, *tasklet dello stesso tipo non possono essere eseguite su processori diversi*. Le tasklet sono strutture dati che mantengono 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. Le tasklet sono variabili per-CPU e possono avere due tipi di priorità: **HI\_SOFTIRQ** con priorità alta e **TASKLET\_SOFTIRQ** con priorità normale.



### Tasklet API:

- `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)`: il task diventa non processabile. Ne anticipo l'inserimento.
- `tasklet_enable(struct tasklet_struct *tasklet)`
- `tasklet_disable(struct tasklet_struct *tasklet)`
- `tasklet_disable_nosynch(struct tasklet_struct *tasklet)`: potrei voler disabilitare una task che è in esercizio. Se chiamo con questa funzione il thread non viene posto in attesa.
- `void tasklet_schedule(struct tasklet_struct *tasklet)`: inserimento nella coda meno prioritaria.
- `void tasklet_hi_schedule(struct tasklet_struct *tasklet)`
- `void tasklet_hi_schedule_first(struct tasklet_struct *tasklet)`: aggiunta in testa alla coda.

### Inizializzare un tasklet & limiti:

```
void tasklet_init(struct tasklet_struct *t,  
                  void (*func) (unsigned long), unsigned long data) {  
    t->next = NULL;  
    t->state = 0;  
    atomic_set(&t->count, 0);           ← This enables/disables  
    t->func = func;  
    t->data = data;  
}
```

Ho più di un counter. Bisogna fare attenzione se utilizzo la funzione di un modulo che ho montato (c'è possibilità che questo possa essere smontato). 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. Il demone non deve essere bloccato. Se la funzione usa un servizio bloccante il demone potrebbe lasciare la CPU

### Recap sui tasklet:

- I task correlati ai tasklet vengono eseguiti tramite specifici thread del kernel (l'affinità della CPU si usa qui quando si registra il tasklet)
- Se il tasklet è già stato pianificato su un core CPU diverso, non verrà spostato su un altro core CPU. se è ancora in sospeso (i softirq generici possono invece essere elaborati da core CPU diversi)
- I tasklet hanno un livello di pianificazione simile a quello di `tq_schedule`; la differenza principale è che il contesto effettivo del thread dovrebbe essere un "contesto interrupt", quindi senza fasi di "no-sleep" all'interno del tasklet (un problema già evidenziato)

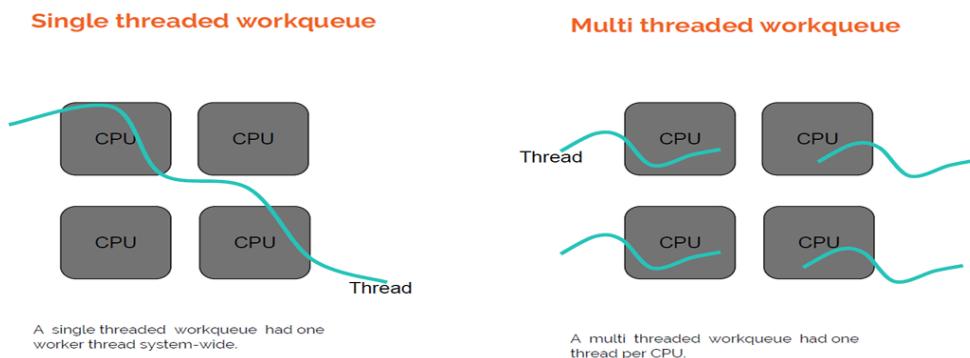
### Work queue:

Nascono perché gli utenti `tq_immediate` normalmente devono usare i tasklet, gli utenti `tq_timer` direttamente i timer; se ciò non basta si usano le work queues. In particolare, accoda il lavoro da svolgere (i task) verso un apposito demone multi-thread (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 sono interrompibili; inoltre, le funzioni chiamate da una work queue possono invocare dei servizi bloccanti (cosa assolutamente vietata per le tasklet), ma ciò rimane comunque sconsigliato. Il punto fondamentale è che il demone non è uno solo, altrimenti andrebbe in blocco e torneremmo al problema visto con i tasklet. Non c'è quindi vincolo sulla CPU. (ogni work queue ha un thread per ogni CPU).

## API:

- `schedule_work(struct work_struct *work)` permette di inserire il lavoro in una work\_queue di default di sistema, gli viene passato come parametro solo il lavoro.
- `schedule_work_on(int cpu, struct work_struct *work)`: in realtà le work queue sono delle multicode, una per ciascuna CPU. Con questa API accodo il task ad una specifica CPU.
- `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)`
- `struct workqueue_struct *create_singlethread_workqueue(const char *name)`: usa un solo thread, il demone gira tra le CPU. No concorrenza.



- `void destroy_workqueue(struct workqueue_struct *queue)`
- `int queue_work(struct workqueue_struct *queue, struct work_struct *work)`
- `int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay)`: marco l'oggetto con un delay. Finché non è trascorso non può essere eseguito.
- `int cancel_delayed_work(struct work_struct *work)`
- `void flush_workqueue(struct workqueue_struct *queue)`: esegue tutti i job.

## Lez.25 (22/11/23)

### Problemi delle work queues:

Quando un thread pesca un elemento dalla coda esiste una probabilità diversa da 0 che esegua un'attività bloccante, essendo il numero di demoni fisso c'è una probabilità che vadano tutti in blocco, ad esempio su una *memcache*.

C'è un altro problema: dobbiamo gestire un numero elevato di tipologie di deferred work, quindi il numero di threads attivi diventa molto elevato e ciò comporta il dover salvare diverse strutture associate ai threads (stack, TCB, eccetera) e questo può portare a dei deadlock.

Un ulteriore problema è relativo al fatto che thread diversi si occupano di work queues diverse, quindi si è soggetti ad una serie di context switch.

### Concurrency managed work queues:

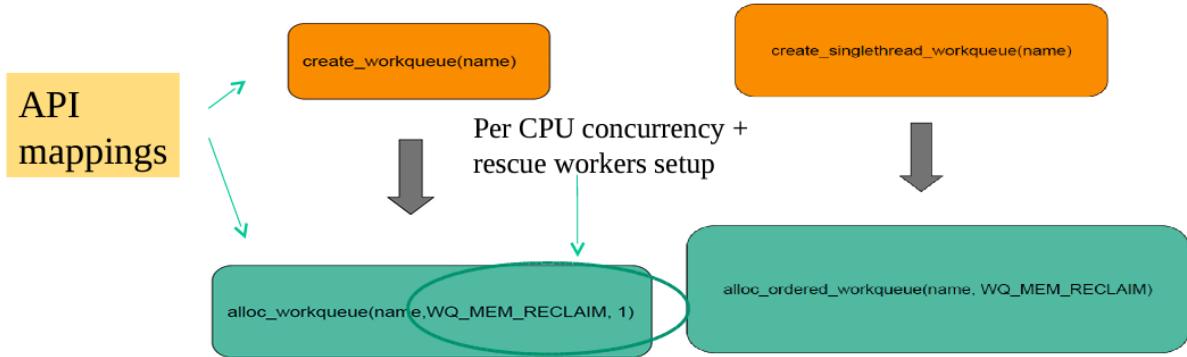
Per risolvere i problemi di cui abbiamo parlato prima sono state introdotte le concurrency managed work queues. Si creano più code, ma non si creano i threads associati alla loro gestione, nel sistema c'è un pool di threads per gestire tutte le code, il numero di threads nel pool è controllato dinamicamente dal kernel.

Permette di:

- Evitare il problema della proliferazione di threads per distinguere deferred work da processare, quindi di ridurre la probabilità di deadlock.
- Eliminare i context switch, un thread può lavorare su più code.

- Tirare su altri threads, se si va in blocco su un certo numero di thread, per processare le attività deferred che permettono di sbloccare quelli in blocco, questa è la gestione dinamica della concorrenza.

API:



Prima, i demoni responsabili delle work queue venivano creati mediante API, per creare una nuova work queue. Attualmente si utilizzano due tipi di funzioni per la creazione di work queue:

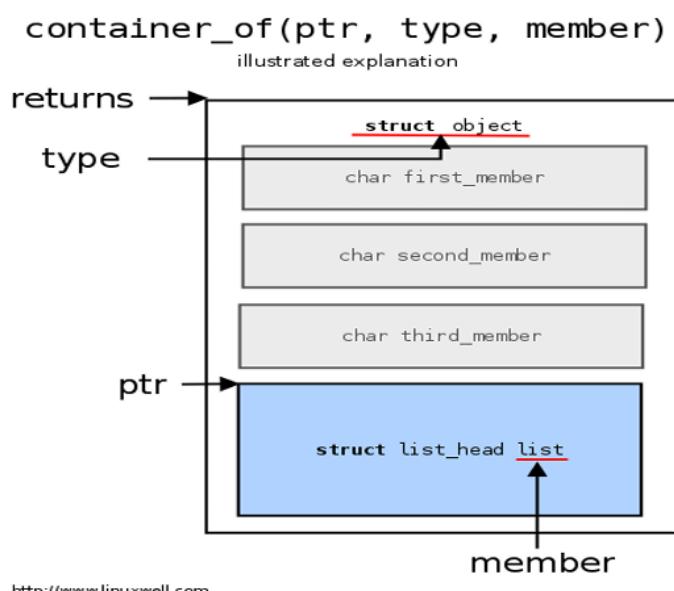
- OLD: `create[_singlethread|_freezable]_workqueue()`
- NEWER: `alloc[_ordered]_workqueue()`: La parola ordered sta ad indicare che la work queue viene eseguita in maniera sequenziale (quindi se un thread ci sta lavorando un altro non può farlo, come nel caso single threaded).

#### Macro container\_of:

Nel kernel possiamo costruire strutture più complesse che gestiscono puntatori.

Sono liste di oggetti articolate, con la macro “`container_of` (`ptr, type, member`)”.

Il parametro `ptr` è l’indirizzo del membro che conosciamo, `member` è il nome del membro a cui si riferisce il puntatore e `type` è il tipo del container.

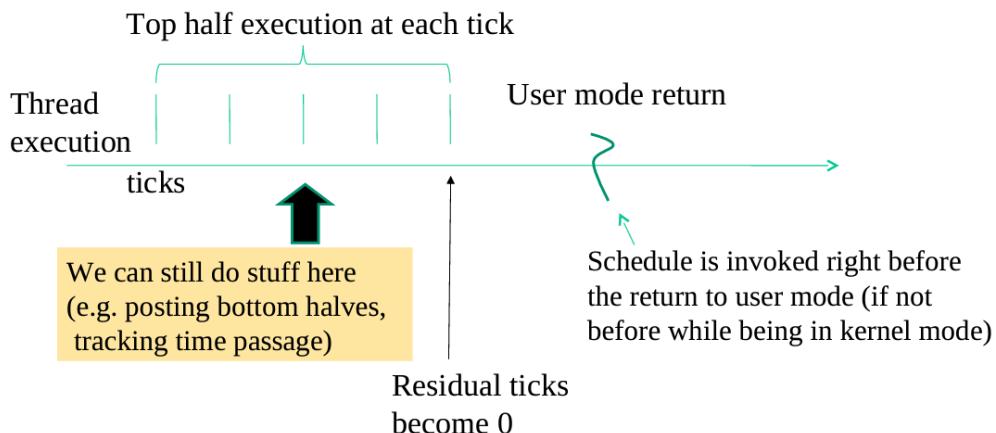


Attenzione: su release recenti c’è randomized layout, quindi non possiamo risalire “a mano” ad un campo usando il displacement.

### Interrupts vs passage of time vs CPU-scheduling:

Vediamo come accorgersi del tempo che sta passando, questo ci serve per riassegnare la cpu in *time sharing*, per fare alcune azioni in alcuni momenti o per monitorare quanto tempo è stata usata una risorsa. Tipicamente il tempo è misurato in termini di interrupt mandati dal timer, non si cambia thread appena arriva un interrupt, altrimenti avremmo problemi, quindi quando la cpu deve essere riassegnata asincronamente ci si accorge che deve avvenire context switch, ma non lo si fa avvenire.

In particolare, a ogni tick, il **top-half** controlla il valore di “**residual ticks**” (= quanto di tempo residuo) e lo decremente, per cui va a segnalare la necessità di deschedulare il thread solo se la condizione “**residual ticks == 0**” è verificata. Lo scheduling verrà effettuato quando viene raggiunto un **safe place** all'interno del codice kernel.



### Supporti per accorgersi del passaggio del tempo:

Un **safe place** è quello in cui un **thread** deve tornare ad eseguire user mode.

I tick residui indicano quanto tempo resta per usare la cpu.

Quando si è in *Context Switch* si potrebbe bloccare l'interrupt?

Questo si faceva su macchine single core e single hyperthread, in cui si era sicuri che sulla cpu tutto avveniva in maniera atomica, non poteva succedere nient'altro.

Su un sistema moderno non si può più fare per diverse ragioni:

- Se blocchiamo l'interrupt blocchiamo tutta l'architettura top-bottom half.
- Gli interrupt sono fondamentali, bisogna eseguire almeno l'attività minimale per mettere a posto lo stato.
- Nel kernel sono da evitare codici del tipo: `while (!condition)`. Infatti il thread rimane nel ciclo while finché la condizione non è verificata, impedendo di raggiungere un safe place e lasciare la CPU. Peggio ancora se la condizione viene modificata da un altro thread che però non riesce a prendere la CPU

### Hardware Timers:

Ci sono tre sorgenti:

- Time Stamp Counter: conta il numero di clock della CPU, al passare del tempo viene incrementato il registro.
- LAPIC: è il timer usato per indicare il passaggio dei tick; usato principalmente per time-sharing.
- High Precision Event Timer: offre una grana più fine di LAPIC, utile ad esempio per real time.

Gli interrupt che arrivano da questi sono gestiti con top-half.

### Top-half LAPIC “old style”:

Old style significa in contesto di task queues. La parte Top Half compie le seguenti operazioni:

1. Si marca la task queue relativa alla gestione degli interrupt come attiva (pronta per essere flushata).

2. Si incrementa la `global volatile unsigned long jiffies`. Essa tiene traccia del numero di tick che sono trascorsi da quando gli interrupt sono abilitati.
3. Si fanno attività minimali riguardo il passaggio del tempo.
4. 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 *safe place* e, se il controllo ha esito positivo, viene invocata la funzione `schedule()` per effettuare il cambio di contesto.

```

void do_timer(struct pt_regs *regs)
{
    (*(unsigned long *)&jiffies)++;
    #ifndef CONFIG_SMP
    /* SMP process accounting uses
       the local APIC timer */
    update_process_times(user_mode(regs)); void timer_bh(void)
    #endif
    mark_bh(TIMER_BH);
    if (TQ_ACTIVE(tq_timer))
        mark_bh(TQUEUE_BH);
}
    {
        update_times();
        run_timer_list();
    }

```

This runs any time-related action

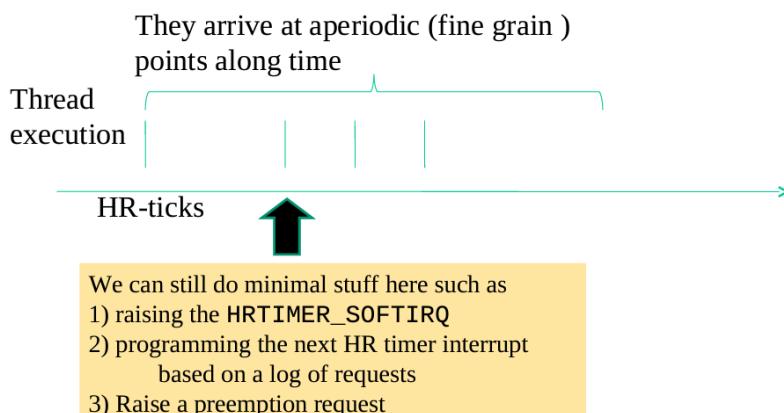
Il modulo scheduler viene attivato dal kernel Linux chiamando la funzione `schedule()` in un *safe place*. Oggi si lavora su *softIRQ*.

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.

#### Top-half High Resolution 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. Se il thread si trova in un contesto *safe place*, può essere deschedulato dalla CPU, la quale viene data a un qualcosa avente priorità più alta, come un demone.



Quando si utilizza la system call `usleep()` (grana più fine di `sleep()`) si usa HR timer:

1. Il thread chiamante invia una trap al kernel.
2. Il kernel inserisce una richiesta di HR-timer nel log (e possibilmente riprogramma il componente HR-timer).
3. Lo scheduler viene chiamato per passare il controllo a qualcun altro. Qui inizia la sleep.
4. All'atto dell'expiration dell'HR-timer per questa richiesta durante l'esecuzione di un altro thread, questo verrà possibilmente deschedulato (il prima possibile) per riprendere quello in sospensione.

Cosa succede quando un thread da user chiama kernel per eseguire `usleep()`?

Scendiamo a livello kernel e mettiamo una richiesta per ricevere interrupt da HR timer, questa richiesta viene messa nella struttura per gestire questo oggetto. Ovviamente la struttura gestisce queste richieste, ogni richiesta ha anche un tempo preciso in cui deve essere eseguita, in particolare questo tempo serve per programmare il prossimo interrupt.

Scriviamo l'entry nella struttura e l'oggetto viene riprogrammato per mandare l'interrupt quando ci interessa. Poi chiamiamo lo scheduler, quando finisce l'intervallo veniamo rimessi schedulabili in cpu dal demone; in particolare il top half sveglia il demone che sveglia noi.

Questo ci fa capire che non posso risvegliarmi esattamente al tempo in cui vorrei, se fossi rischedulato immediatamente all'arrivo dell'interrupt non avrei più processing top-bottom half.

Come interagire con hrtimer:

Abbiamo una struct hrtimer, per collegarla e far capire al top half dove si trova essa deve essere pre-allocata.

Alcune API per HR timer:

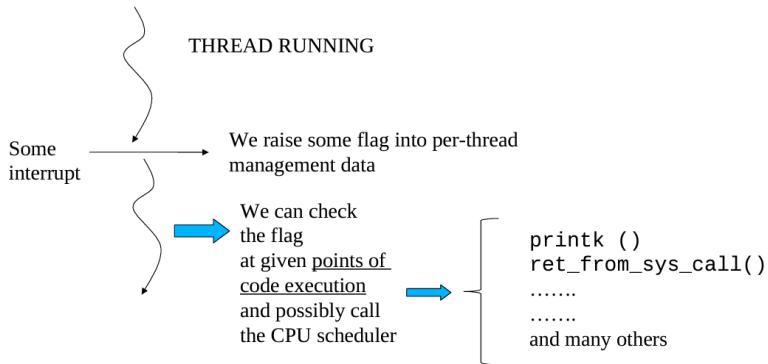
- `ktime_t ktime_set (leng secs, long nanosecs)`: imposta il timer in cui specifico secondi e nanosecondi.
- `void hrtimer_init (struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode)`: inizializza un HR Timer, dove: `struct hrtimer *timer` è una struttura che specifica la funzione che dovrà essere eseguita dopo la scadenza del timer con il relativo argomento; `clockid_t which_clock` specifica il meccanismo di clock da utilizzare:
  - `CLOCK_MONOTONIC`: tempo si muove solo in avanti, visione locale rispetto ai tick.
  - `CLOCK_REALTIME`: matcha il real-time corrente del mondo, visione globale.
- `enum hrtimer_mode mode` specifica la modalità con cui si vogliono specificare i tempi (modo relativo o assoluto).
- `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 timer in maniera bloccante (se il timer è attivo).
- `int hrtimer_try_to_cancel (struct hrtimer *timer)`: elimina un timer in maniera non bloccante.

Richiesta di preemption:

Quando eseguiamo codice ci sono molti *safe places*. Quando si passa per un safe place ci deve essere consapevolezza del fatto che potrei o meno rilasciare la cpu in quel punto.

Nel TCB c'è un'informazione che dice se si è preemptable o meno. E' variabile per thread.

Supponiamo che arrivi un interrupt, che flagga il thread per lasciare la cpu, ma se passa per un safe place ed è marcato come non-preemptable la cpu resta a lui.



Come lavorare sull'atomic counter che dice se si è o meno preemptable?

A seconda del fatto che un sottosistema abbia bisogno o meno che il thread sia preemptable il contatore venga o meno incrementato, in questo modo quando si esce da un sottosistema se c'è ancora bisogno di essere preemtible si rimane tali.

*API:*

Il check in safe place è fatto su questo atomic counter, ci sono API per farlo:

- `preempt_enable()`: si decrementa il counter, inoltre la parte finale di preemptable è un safe place; dunque se il counter è 0 si lascia la cpu passando per il safe place.
- `preempt_disable()`: si incrementa il counter.
- `preempt_enable_no_resched()`: si decrementa il counter ma non c'è un safe place; non si fa il controllo sul fatto che il contatore sia pari a 0.
- `preempt_check_resched()`: si controlla se bisogna rischedulare in seguito a richieste di preemptability ricevute.
- `preempt_count()`: per leggere il valore del contatore.

In `kprobe_read_interceptor` abbiamo usato queste cose.

#### Preemption vs per-CPU variables:

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 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.

### Osservazioni:

Quando si usa `HRtimer` tramite la `struct hrtimer` si passano le informazioni che servono al demone per processare l'interrupt.

`hrtimer_init` prende come parametro il pointer alla struttura, questo è un aspetto interessante se lo associamo al concetto di "`container of`", perché possiamo inserire questa struct in una struttura più complessa.

Se un thread per cambiare cpu vuole andare in blocco non possiamo usare "schedule"; lo sblocco è causato da altri thread o da un thread in interrupt context.

Ci sono API per mettere threads in blocco e risveglierli, esse si basano tutte su un'api di più basso livello.

### ***Il ruolo del TCB nei SO moderni:***

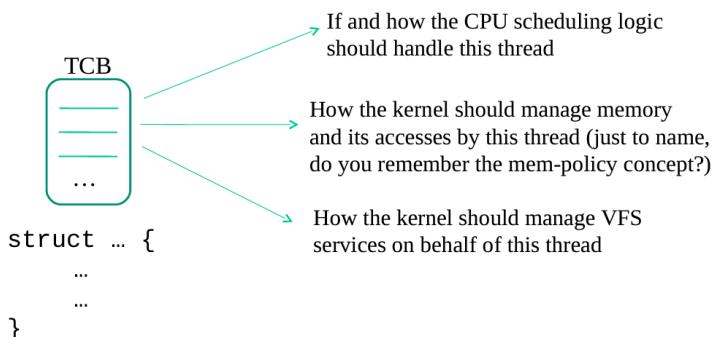
Un Thread Control Block mantiene informazioni relative al fatto che c'è un thread attivo in cpu.

Un TCB è una struttura dati che mantiene informazioni relative a:

- 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 (si usano massivamente i pointers): 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.

### Esempio:



### **TCB dispatchabili**

Abbiamo metadati che ci dicono come prendere la memoria durante l'esecuzione del thread.

Nel TCB c'è un'informazione che dice se il thread è dispatchabile, dunque la funzione di scheduling usa questa informazione per considerare il thread tra quelli che possono andare in cpu.

La logica con cui avviene lo scheduling non ha nulla a che vedere con la dispatchabilità, essa influenza solo il fatto che il TCB sia o meno visibile alla logica di scheduling.

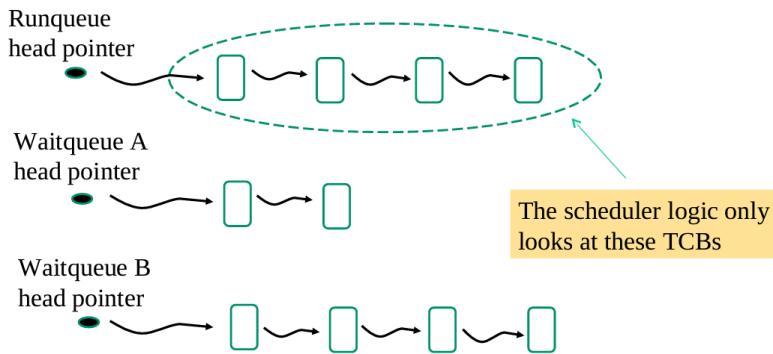
La logica di scheduling è di bassissimo livello, sopra di essa si possono costruire altre logiche.  
(implementano sleep e wakeup services, servizi kernel).

### Sleep wakeup services:

Il TCB è **dispatchabile se è legato ad una run queue**, che può essere una multicoda.

Se non sta in questa coda può trovarsi in una "**wait queue**".

Questa è la struttura per sleep e risveglio dei threads.



### Logica di scheduling vs servizi bloccanti:

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*.

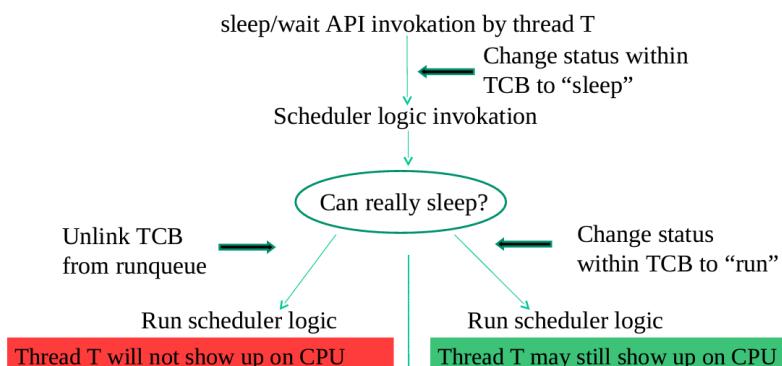
Inizialmente, lo scheduler logic è in esecuzione su un CPU-core all'interno del contesto del thread A. Nel momento in cui A chiama un servizio bloccante, invoca in modo sincrono lo scheduler logic per essere deschedulato, per cui il suo TCB deve ancora trovarsi sulla runqueue. Tuttavia, lo scheduler 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 sta ancora nella runqueue quando la selezione del prossimo thread viene fatta. Potrebbe quindi rimanere in CPU.

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"; (questo può verificarsi se il thread viene colpito da un evento da gestire).
- 4) Quando la sleep termina, scollegare il TCB del thread chiamante dalla waitqueue.

### Timeline:



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, come servizi sincroni (e.g. sys\_kill), Top half o Bottom half.

### **Context switch:**

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 (`%sp`) del thread entrante, ovvero se si cambia stack area di sistema. Se cambiamo `%sp` senza istruzioni convenzionali cambiamo thread, quando cambiamo thread cambiamo TCB, perché stiamo cambiando l'area di memoria in cui avviene ogni cosa. Per anni il design dei SO si è basato su questo. Se cambiamo thread in Linux cambiamo valore di “`current`”, simbolo che punta a TCB corrente. Oggi c'è maggiore flessibilità per identificare una stack area diversa rispetto a `current` (macro legata al valore dello stack pointer), questa flessibilità è necessaria perché in caso di multi-threading è necessario saper gestire più stack areas.

### **“TCB in LINUX 2.6: task\_struct”:**

include/linux/sched.h as struct task\_struct

The main fields (ref 2.6 kernel) are

```
> 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*/
```

synchronous and  
asynchronous  
modifications

Quando parliamo di TCB in Linux parliamo di task\_struct.

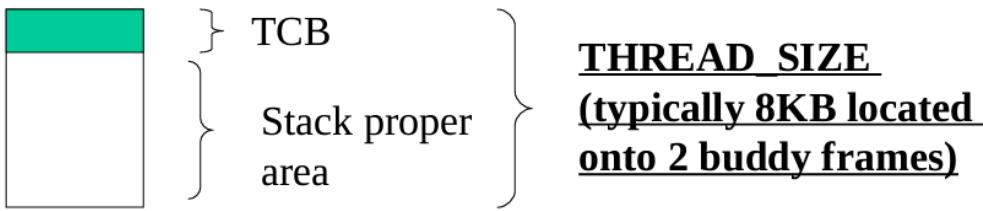
- `volatile state` indica se si è o meno dispatchabili.
- `struct mm_struct *mm`, per la gestione della memoria del thread T, esterna al TCB.
- `pid` è il process id che si aveva nelle releases originali in Linux, nelle releases moderne ci sono due identificatori, ad esempio questo è legato alla tecnologia dei containers (AS diversi).
- `pgrp` è l'identificatore del gruppo.
- `need_resched` è il flag per indicare la possibilità di essere rischedulati, quando counter va a 0.
- `long counter` è il numero di tick da spendere prima di lasciare la cpu.
- `nice` e `policy` servono per capire se un thread è “amichevole” e con quale politica di scheduling riconsiderarlo. La policy ci fa capire che ci sono scheduler diversi in Linux, da usare a seconda del thread.

In versioni più recenti non abbiamo grandi modifiche:

- Alcune informazioni sono compattate in bitmasks, ad esempio `need_resched` è diventato il bit `TIF_NEED_RESCHED` all'interno di una bitmask.
- Le informazioni compattate possono essere facilmente accessibili tramite specifiche macro/API.
- Più campi sono stati aggiunti per riflettere nuove capacità, ad esempio, nella specifica Posix o negli interni di Linux .
- I principali campi sono ancora presenti (stato, pid, tgid,...).

### **Allocazione del TCB prima del kernel 2.6:**

Prima l'allocazione della stack area e del TCB era unica, una sola richiesta di due pagine dal buddy allocator. Ciò implica avere allineamenti della stack area alle pagine, inoltre se la taglia del TCB aumenta con buddy allocator devo raddoppiare la dimensione. Con context switch si cambia lo stack di sistema. Avere questo approccio implica che, se con nuove versioni di Linux dovesse aumentare la dimensione del TCB, si ridurrebbe lo spazio a disposizione per lo stack, aumentando anche il rischio di buffer overflow. La soluzione sarebbe chiedere più pagine, ma il buddy allocator lavora con pagine in potenze di 2.

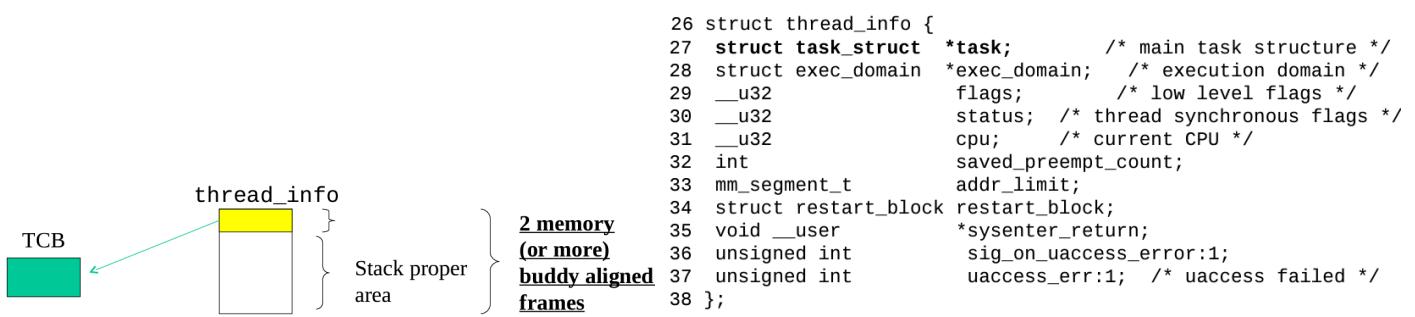


Questo avviene anche per l'idle process, il cui TCB è init\_task.

#### **Allocazione del TCB da kernel 2.6 a 4.8:**

Si è deciso di scollegare stack area e TCB, ma avendo un puntatore al TCB nella stack area, in particolare esso è “`thread_info`”. Per arrivare al TCB devo accedere all’header, dunque devo trovarmi nella stack area corretta. Questa struttura è tuttora valida.

Perciò, rimane comunque semplice accedere al TCB ma, al contempo, qualunque espansione del TCB non inficia sull’area di memoria dedicata allo stack. Se il TCB si riempie è comunque possibile scrivere pochi byte in `thread_info`.



La dimensione dello stack del thread di livello kernel è pari a due o più frame dati dal buddy allocator. Dal kernel 4 è possibile specificare l’ordine di allocazione tramite macro. 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.

#### 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 quale fosse 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 seguenti sottocasi:

- *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).
- Attualmente, l’indirizzo del TCB è salvato all’interno di una **variabile per CPU**. Essa viene aggiornata ogni volta si fa il rescheduling (dunque cambiare il kernel level stack non implica necessariamente un context-switch come accadeva prima). Per recuperare il suo valore si usa

```

get_current():
    struct task_struct;
    DECLARE_PER_CPU(struct task_struct *, current_task);

    Static __always_inline struct task_struct
    *get_current (void) {
        return this_cpu_read_stable (current_task);
    }
#define current get_current()

```

## 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 la **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 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).

### runqueues kernel 2.4:

Originariamente in Linux la runqueue era una sola.

Se guardiamo la definizione di strutture dati presenti nel kernel è vero che era un array, con un elemento per ciascuna cpu, ma tutti questi oggetti puntavano allo stesso elemento: l'idle process, che è sempre schedulabile. 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.

### waitqueues kernel 2.4:

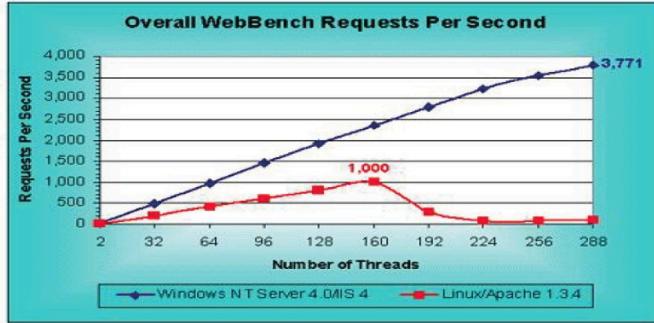
Possiamo creare una wait queue con la macro **DECLARE\_WAIT\_QUEUE\_HEAD(queue)**.

Possiamo utilizzare le seguenti API:

- **void interruptible\_sleep\_on(wait\_queue\_head\_t \*q)**: quando ci si mette su questa waitqueue si marca che si può essere interrotti da segnalazioni.
- **void sleep\_on(wait\_queue\_head\_t \*q)**: è la versione non interrompibile di quella sopra.
- **void interruptible\_sleep\_on\_timeout(wait\_queue\_head\_t \*q, long timeout)**: dopo lo scadere del timeout risveglia il thread.
- **void sleep\_on\_timeout(wait\_queue\_head\_t \*q, long timeout)**: è la versione non interrompibile di quella sopra.
- **void wake\_up(wait\_queue\_head\_t \*q)**: risveglia tutti i threads nella wait queue.
- **void wake\_up\_interruptible(wait\_queue\_head\_t \*q)**: riporta nella runqueue tutti i thread appartenenti alla wait queue passata come parametro che erano accodati come "interruptible".
- **wake\_up\_process(struct task\_struct \* p)**: riporta nella runqueue solo il thread specificato dal TCB passato come parametro ( selettiva).

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 wait queue potrebbero esserci anche centinaia di thread, e non è ideale poter 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**:



Taken from 1999 Mindcraft study on Web and File Server Comparison

### Stato di un thread:

I thread possono trovarsi in uno di questi stati fondamentali, definiti in `/linux/sched.h`

- `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.

Le API wait queue si occupano anche dell'unlink dalla wait queue stessa dopo essere ritornate dall'operazione di scheduling. Andiamo ad osservare per esempio `interruptible_sleep_on()`:

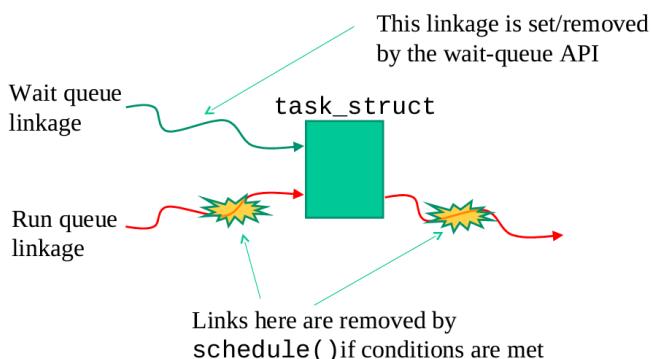
```
#define SLEEP_ON_HEAD \
    wq_write_lock_irqsave(&q->lock,flags); \
    __add_wait_queue(q, &wait); \
    wq_write_unlock(&q->lock);

#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
}
```

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.

### TCB linkage dynamics:



Mi collego a `waitqueue`, e poi eventualmente mi sgancio, tramite `schedule()`, da `runqueue`. Le `task_struct` (ovvero TCB) sono oggetti parametri che ci permettono, potenzialmente, di rimanere in più di una lista.

### **Wait event queues:**

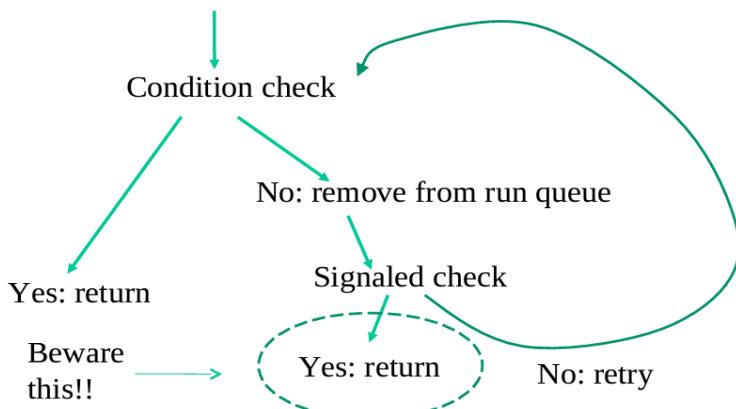
Visti i limiti della parametricità in merito al risveglio dei thread, vengono introdotte le *wait event queues*, in cui si fa polling automatico delle condizioni. Permettono di risvegliare i thread tramite condizioni. Le condizioni per una stessa coda possono essere diverse per thread diversi, ciò consente risvegli selettivi a seconda di quale condizione viene effettivamente attivata. Lo schema si basa sul polling delle condizioni al risveglio e sul successivo re-sleep.

### Nuove API:

- `wait_event_interruptible(wq, condition)`: inserisce TCB nella wait queue.
- `wait_event(wq, condition)`: come prima, ma ignora le segnalazioni.
- `wait_event_timeout(wq, condition, timeout)`: interagisce anche con sottosistemi che guardano il passaggio del tempo.
- `wait_event_command(wq, condition, pre-command, post-command)`: per collegare attività da fare in ingresso ed in uscita. Pre e post command sono statement C.

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:



Ciò avviene quando chiamo le API per dormire:

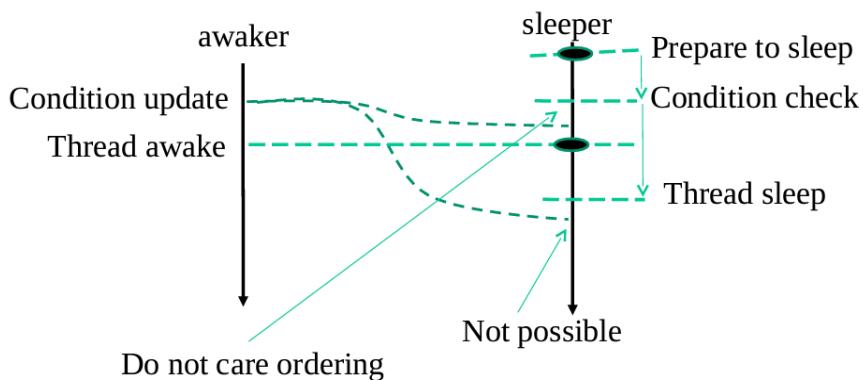
Quando vengo riportato in CPU, c'è *condition check*, se è vera, ritorno e non dormo più.

Altrimenti, mi rimuovo dalla *runqueue*. In questo sottoramo, se c'è *anche* una segnalazione (yes), vengo rimesso in CPU (e *runqueue*), riprendendo il controllo. Se non c'è, faccio il *retry*.

Può capitare che, cambiando la *condition check*, e senza segnali, non si riesca ad andare più nel ramo sinistro? La risposta è *no*, vediamo perchè.

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()`) così come le API per svegliare il thread quando cambiano lo status del thread), per cui può essere acceduto e modificato da un solo thread per volta (anche se, ovviamente, TCB diversi possono essere modificati anche concorrentemente). La memoria si sincronizza e bypassa il problema del Total Store Order. Prima vengono fatti gli update sulla condizione e poi i check. Il locking permette di linearizzare le operazioni.

vm\_area\_struct:

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 mantiene informazioni riguardanti un'area di memoria virtuale valida per il thread. Ecco 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;
    .....
};
```

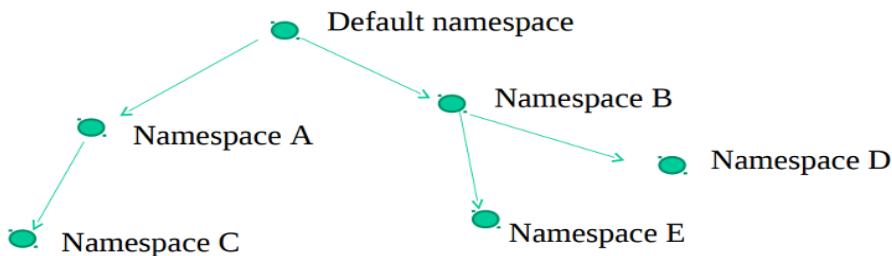
- `unsigned long vm_start`: indica dove inizia la zona di memoria (è il primo indirizzo all'interno della zona).
- `unsigned long vm_end`: indica dove termina la zona di memoria (è il primo indirizzo fuori dalla zona).
- `pgprot_t vm_page_prot`: indica i permessi di accesso alla zona di memoria.
- `struct 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.

## Namespaces ed identificazione 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 (`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. 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`.

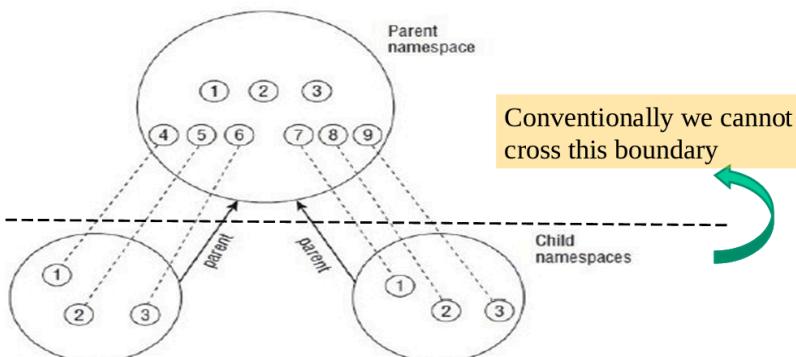


Il thread T, la cui creazione porta all'istanziazione di un nuovo namespace X, avrà il PID virtuale pari a 1 nel contesto del namespace X e, sempre all'interno di X, il suo parent avrà il PID virtuale pari a 0.

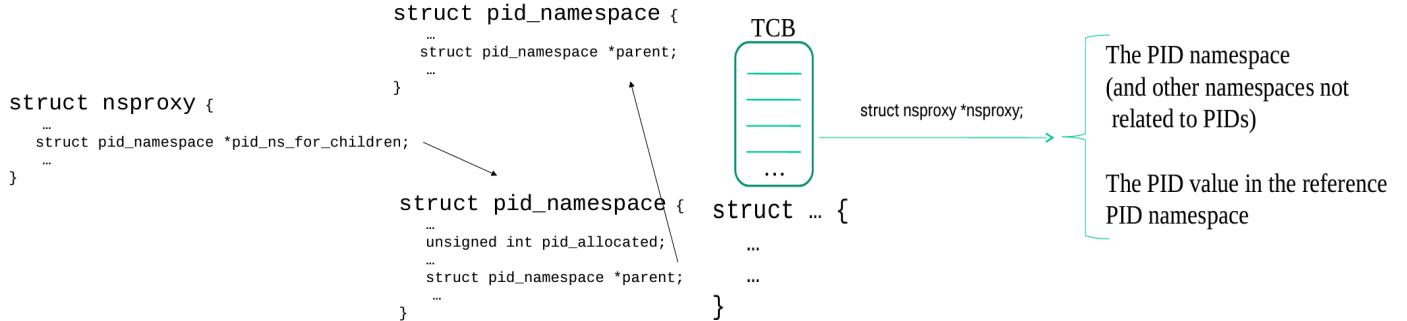
In ogni caso, tutti i thread, indipendentemente da quale sia il namespace a cui appartengono, avranno un PID definito all'interno del namespace di default (che corrisponde appunto al PID reale).

## Visibilità dei namespaces

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. I thread dello stesso namespace possono lavorare in maniera congiunta. La creazione di un namespace dipende dalla creazione dei thread, che per definizione sono sempre child, non posso quindi creare un nuovo nodo radice.



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.



`pid_allocated` è il PID virtuale. Si hanno anche informazioni sul parent., per riconoscerlo.

### 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 wait queue), 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 basate su accesso hash. Linux offre delle macro per attraversare tali strutture.

#### API kernel:

Se voglio rimettere un thread nella RQ devo avere un puntatore al suo TCB.

Ci sono servizi che a partire dal PID permettono di risalire al TCB, noi lavoriamo sul PID.

Da una parte possiamo usare il PID nel namespace originale, di default, dall'altra abbiamo la possibilità di usare il PID nel namespace corrente.

Tutto ciò si usa anche nei servizi di sleep-wakeup.

In Linux si utilizza una hash table in cui si inseriscono i PID ed essi portano ad un puntatore al TCB.

```

/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct *pidhash_pprev;

#define PIDHASH_SZ (4096 >> 2)
extern struct task_struct *pidhash[PIDHASH_SZ];
#define pid_hashfn(x) (((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1)
  
```

Abbiamo l'api "[find\\_task\\_by\\_pid](#)", che prende un elemento nella hash table e verifica anche se sta in una lista di trabocco. Il pid è quello originale, nel TCB.

```

static inline struct task_struct *find_task_by_pid(int pid) {
    struct task_struct *p,
        **htable = &pidhash[pid_hashfn(pid)];

    for(p = *htable; p && p->pid != pid;
        p = p->pidhash_next) ;
    return p;
}
  
```

Altra api è "[find\\_task\\_by\\_vpid](#)", in cui il vpid è quello nel namespace in cui stiamo lavorando quando facciamo la search. Controlliamo se chi fa la richiesta può lavorare su questo vpid.

```
struct task_struct *find_task_by_vpid(pid_t vpid)
```

Si utilizza questa api nella “`kill()`”, in releases recenti.

La kill all’interno del kernel prende il thread e lo riporta sulla RQ marcando un bit in una maschera di segnali, in caso poi è lui che aggiusta le cose per eseguire un gestore.

#### Virtual pid implementation:

Per gestire questi **virtual pid** abbiamo un’altra tabella di hashing basata su “struct pid”.

Questa struttura può essere collegata ad altre strutture per creare una mappa di hashing con liste di trabocco. Usando questa struttura possiamo fare il retrieve di informazioni, ad esempio per capire se il pid appartiene al nostro stesso namespace.

```
struct pid {
    atomic_t count;
    unsigned int level;
    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX];
    struct rcu_head rcu;
    struct upid numbers[1];
};
```

This keeps the number and the reference to the namespace

When accessing the target PID records we can match with the namespace of the caller

We can query for individuals or groups (see TGID)

La funzione `access_pidfd_pidns` si usa per verificare se due thread hanno lo stesso namespace. Essa viene chiamata ad esempio quando un processo fa una kill di un altro processo):

Il retrieve è fatto con pointer a strutture; poi si vede se ciò che fa l’attore rispetta il fatto che può usare soltanto PID inferiori nella catena dei namespaces.

```
static bool access_pidfd_pidns(struct pid *pid) {
    struct pid_namespace *active = task_active_pid_ns(current);
    struct pid_namespace *p = ns_of_pid(pid);
    for (;;) {
        if (!p)
            return false;
        if (p == active)
            break;
        p = p->parent;
    }
    return true;
}
```

This is called, e.g., when a kill from the current thread is issued towards another threads

#### API nei moduli:

Vediamo API che permettono di gestire tutto ciò anche nei moduli, perchè non tutte quelle viste sono esposte, alcune possono essere critiche dal punto di vista dell’isolamento.

“`pid_task`” come funziona? Se ho un pid posso cercare il **vpid** ad esso associato, andando tramite hash table a partire dal pid a trovare la task struct, dunque trovo il namespace proxy.

Posso usare il vpid per passare il parametro che identifica la task struct.

Questo è utilizzabile nei moduli, ciò significa che nei moduli si può usare il vpid.

```
struct task_struct *pid_task(struct pid *pid, enum
    pid_type);
```

PIDTYPE\_PID or other

find\_vpid(pid)



pid\_task(find\_vpid(pid), PIDTYPE\_PID);

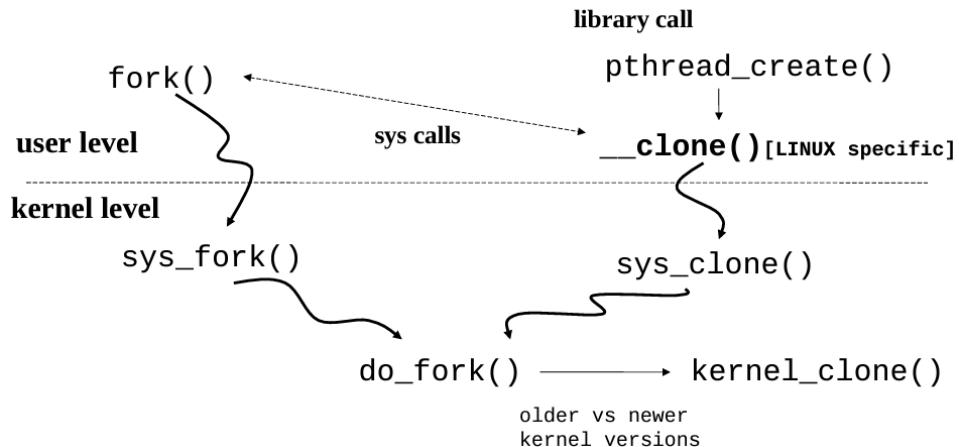
Querying the TCB address by the default PID

Come funziona ciò dal punto di vista reale?

Non esiste un nuovo namespace se non viene creato un nuovo thread che si muove al suo interno.

### **Creazione di threads e processi:**

Segue tale schema:



Come si tira su un nuovo thread in Linux?

Partendo dal kernel c'è un solo percorso per tirare su un nuovo thread, originariamente si chiamava "`do_fork()`", ora si chiama "`kernel_clone()`".

Su questo percorso possiamo arrivare da due direttivi, sono due perché permettono di passare parametri diversi per creare il nuovo sottosistema.

La "`fork()`" user è mappata su frontend kernel "`sys_fork()`" e non accetta parametri; l'unica cosa che fa è indicare ai registri di cpu quale servizio va chiamato.

Quando tiriamo su un nuovo thread usiamo la "`pthread_create()`", la quale sfrutta "`clone()`" (che è di più basso livello), che internamente chiama "`sys_clone()`".

Come vediamo, 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 della provenienza:

- da una `sys_clone()`, la `do_fork()` si limiterà ad eseguire le operazioni necessarie per avviare un flusso di esecuzione appartenente al processo chiamante.
- 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).

Nella `pthread_create()` passiamo diversi parametri, la creazione di un nuovo namespace viene fatta così: indichiamo che quando viene clonato un nuovo thread va anche creato un nuovo namespace.

In `sys_clone()` ci sono una serie di flag in base ai quali si danno risposte diverse alle domande che la `do_fork()` si fa.

### Funzione di libreria clone:

Return value mapped to thread exit code

```
NAME      top
        clone, __clone2 - create a child process

SYNOPSIS  top
        /* Prototype for the glibc wrapper function */
#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 */

DESCRIPTION top
        clone() creates a new process, in a manner similar to fork(2).
```

Parameters can vary in number and order

`child_stack` indica l'indirizzo del nuovo stack. Infatti non è il kernel che definisce dove deve stare la stack area. La funzione `fn` da eseguire all'avvio del thread non deve essere passata al kernel. `ptid` e `ctid` sono i pointer ad aree in cui è scritto ID del parent e del child. Al ritorno dalla funzione, il thread controlla se è parent o child (solo se è child esegue `fn`). I flag sono i seguenti:

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

Tra questi abbiamo “`CLONE_VM`” che indica che il nuovo thread dovrà condividere la tabella di memory management. Lavoriamo sullo stesso AS del thread originale.

Le implementazioni specifiche per le varie architetture sono:

### Newer pthreadXX( ) services

The raw system call interface on x86-64 and some other architectures (including sh, tile, and alpha) is:

```
long clone(unsigned long flags, void *child_stack,
           int *ptid, int *ctid,
           unsigned long newtls);
```

On x86-32, and several other common architectures (including score, ARM, ARM 64, PA-RISC, arc, Power PC, xtensa, and MIPS), the order of the last two arguments is reversed:

```
long clone(unsigned long flags, void *child_stack,
           int *ptid, int *ctid,
           unsigned long newtls);
```

On the cris and s390 architectures, the order of the first two arguments is reversed:

```
long clone(void *child_stack, unsigned long flags,
           int *ptid, int *ctid,
           unsigned long newtls);
```

### Esempio:

/namespaces/`new_namespace.c`

Quando si crea un nuovo Namespace si genera una nuova istanza di servizio Linux, quindi il primo thread avrà PID pari a 1. L'id del thread, visto dal padre e ritornato dalla `clone()`, è nel namespace originale.

Facendo “ps” da shell del nuovo thread vedo tutti, perché lavoro nel filesystem, sto usando uno pseudofile, non una system call.

Se proviamo a fare una `kill()` su un altro NS non dovrei esserne in grado, infatti ricevo “no such process”.

### Azioni della `do_fork`:

- 1) Alloca un nuovo TCB.
- 2) Alloca una nuova area di stack.
- 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 in ottica 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.

## Sincronizzazione:

Tra le primitive di sincronizzazione abbiamo:

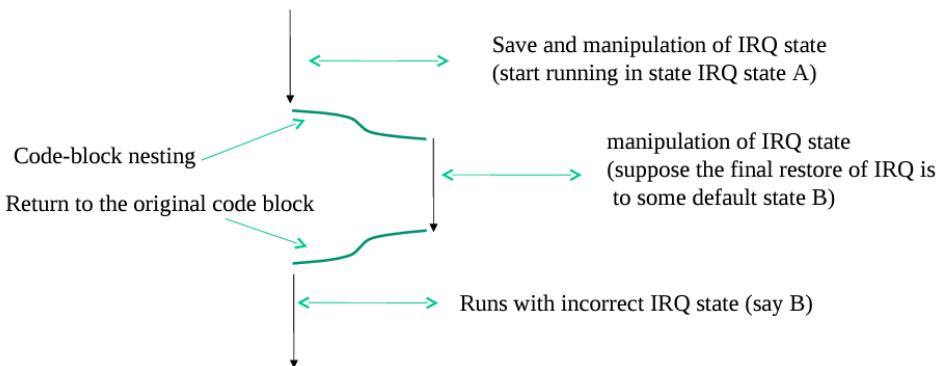
- `DECLARE_MUTEX (name)`: dichiara un nuovo mutex, restituendo un ptr a una struct semaphore.
- `void sema_init (struct semaphore *sem, int val)`: inizializza uno specifico mutex.
- `void down (struct semaphore *sem)`: acquisisce un token dal mutex specificato in maniera bloccante. Se il token non è disponibile potrebbe andare a dormire.
- `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.

## Spinlock API:

Com'è fatto lo spin-locking realmente nel kernel?

Lo **spinlock** può essere eseguito in un blocco in cui non vogliamo essere interrotti; per fare questo chiamiamo un API che prende il lock e mette il processore non interrompibile:

Tuttavia, non so quale fosse lo stato del processore prima, magari già era interrompibile, portando a ciò:



- `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, ci informa se il processore era o meno interrompibile, aiuta a ripristinarne lo stato.

```
#include <linux/spinlock.h>
```

```
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);
```

### Read-write lock:

Per discriminare readers e writers abbiamo anche:

```

rwlock_t xxx_lock = __RW_LOCK_UNLOCKED(xxx_lock);
unsigned long flags;

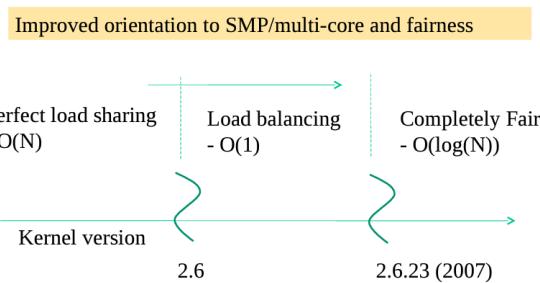
read_lock_irqsave(&xxx_lock, flags);
.. critical section that only reads the info ...
read_unlock_irqrestore(&xxx_lock, flags);

write_lock_irqsave(&xxx_lock, flags);
.. read and write exclusive access to the info ...
write_unlock_irqrestore(&xxx_lock, flags);

```

Lez.29 (04/12/23)

### **Evoluzione logica dello scheduler Linux:**



- **Perfect load sharing:** il carico dei CPU-core è perfettamente bilanciato ma il costo per lo scheduling è  $O(N)$  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  $O(\log(N))$ .

Ad un certo istante prendo decisioni su come assegnare la CPU ai thread.

Indipendentemente dal momento nella storia dello sviluppo Linux ci sono concetti basici usati comunque.

Ad esempio, nello sviluppo dello scheduling di cpu c'è il concetto di **epoca**:

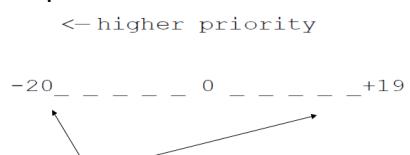
*Ad un certo istante prendo decisioni su come assegnare la CPU ai thread. In particolare assegno ad ogni thread un numero di tick. Un'epoca finisce quando tutti i thread sulla runqueue hanno terminato i loro tick. (altrimenti, se considerassi quelli con ancora dei tick ma non in runqueue, potrei prolungare di troppo l'epoca. Restano comunque tick residui).*

*Quando un'epoca finisce si riassegnano i tick per l'epoca successiva. Assegno in base alla priorità.*

### **Posix classic:**

Questo concetto come è stato implementato inizialmente in Linux?

Le specifiche Posix tradizionale dicevano che lo schema di priorità doveva seguire il seguente schema:



We can move across priority values by exploiting thread niceness

Inizialmente si attribuiva un numero di tick a ciascun thread a seconda di dove il thread era posizionato in questo schema.

La priorità era esclusivamente funzione della **niceness** del thread.

Questo ha dato luogo allo sviluppo della funzione di scheduling originale in Linux.

### **Perfect load sharing:**

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).

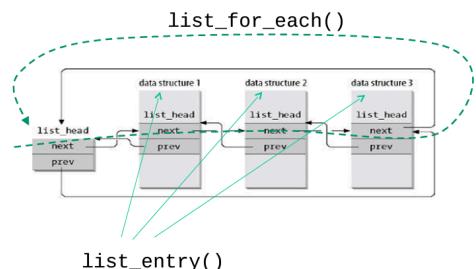
L'esecuzione di `schedule()`, in Linux 2.4, comprende:

- 1) Verifica se il thread corrente (da deschedulare) deve essere rimosso dalla *runqueue* prima di andare a scandire la *runqueue* stessa. Ciò lo si fa controllando lo stato del TCB:
  - 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 non ha ricevuto alcuna segnalazione, allora viene rimosso dalla runqueue.

```
.....
prev = current;
.....
switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:;
}
prev->need_resched = 0;
```

- 2) Analisi della runqueue; qui chiaramente la runqueue deve essere unica, altrimenti non potremmo parlare di *perfect load sharing scheduler*.

La runqueue sfrutta una lista doppiamente collegata. Ogni TCB ha un *list\_head*. Si scandisce tale lista, partendo da *idle\_task* sempre schedulabile tramite una MACRO, per girare sulla RQ, e con un'altra MACRO si fa il retrieve del TCB.



Per ogni TCB vediamo l'**affinità**, e se compatibile si calcola la *goodness*; che ritorna un certo peso, essa viene aggiornata mano a mano che si trova un peso migliore. Questa funzione, ad esempio, predilige threads che condividono lo stesso AS, così non è necessario fare lo swap della tabella delle pagine. Se adeguata, allora il thread è schedulabile in CPU, altrimenti continua la ricerca.

```
repeat_schedule:
    /* Default process to select...*/
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p, this_cpu)) {
            int weight = goodness(p, this_cpu, prev->active_mm);
            if (weight > c)
                c = weight, next = p;
        }
    }
```

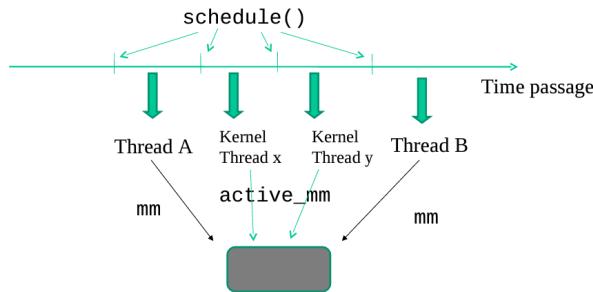
Se la *goodness* è per tutti 0, vuol dire che abbiamo finito i tick. E' necessario riassegnarli a tutti.

- 3) Context switch vero e proprio, in cui il thread selezionato viene posto in CPU.

### Strutture di memory management:

Per quanto riguarda il memory mapping, la `mm_struct` ha due campi:

- `struct mm_struct *mm`: identifica le informazioni di memory management che caratterizzano l'address space riguardante la parte applicativa del thread. Se il thread è di livello kernel è NULL.
- `struct mm_struct *active_mm`: identifica le informazioni di memory management di livello user dell'ultimo thread eseguito in quella CPU. Per un thread applicativo coincide con mm. Per un thread kernel (demone) identifica l'address space dell'ultimo thread eseguito. Viene usato quando si calcola la goodness perché se la page table è la stessa del nuovo processo non devo ricalcolare (ottimizzo).



### Calcolo della goodness:

Ci sono bonus a seconda della soddisfazione di alcune condizioni.

- La goodness è forzata a 0 se non ci sono più ticks da spendere.
- Un punto di bonus si prende se la PT è condivisa.
- Il bonus di +15 è dato se si è stati gli ultimi a girare in cpu, perché se siamo esattamente lo stesso thread possiamo favorire ancora di più l'efficacia dell'architettura di cache e la località.

Se il thread ha ancora dei ticks può ancora usare la cpu, questo bonus ci serve perché la funzione di scheduling è chiamata ogni volta che si passa in safe places (quindi si vuole rimanere in cpu), non solo quando scade il lapic timer per la fine dei ticks.

Fondamentalmente ciò che stiamo facendo è far usare ad un thread tutti i ticks a sua disposizione insieme, quindi è un'utilizzazione batch dei ticks.

$$\begin{aligned}
 \text{goodness}(p) = & 20 - p->\text{nice} && (\text{base time quantum}) \\
 & + p->\text{counter} && (\text{ticks left in time quantum}) \\
 & + 1 && (\text{if page table is shared with the previous process}) \\
 & + 15 && (\text{in SMP, if } p \text{ was last running on the same CPU})
 \end{aligned}$$

NOTE: **goodness is forced to the value 0 in case `p->counter` is zero**

### Gestione dell'epoca:

Un'epoca è completata quando tutte le goodness sono a 0.

Che facciamo quando viene chiusa un'epoca?

Nello scheduler Linux tradizionale la riassegnazione veniva fatta a tutti i threads, sia quelli in RQ che quelli in WQ. Le funzioni di hashing permettono di passare su tutti i TCB a prescindere se fossero o meno nella RQ. Ecco come avviene la riassegnazione dei ticks:

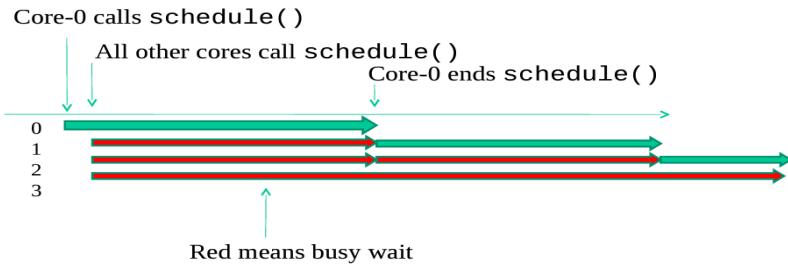
$$p->\text{counter} = p->\text{counter} / 2 + 6 - p->\text{nice}/4$$

### Problema di performance:

I task che hanno già terminato i tick rimangono nella runqueue e viene sempre ricalcolata la loro goodness.

Per scandire la lista con n TCB si ha complessità O(n). Inoltre, visto che la lista è una, se più CPU vogliono

scandirla devono prendere un lock:



### Load balancing:

Nelle releases più recenti c'è probabilità di collisione quasi nulla di accesso alla stessa RQ da parte di cpu diverse. Abbiamo possibilità di mantenere workload bilanciato tra più cpu.

Ci si allinea ad uno standard più nuovo dal punto di vista della priorità di scheduling, abbiamo più livelli di priorità: da 0 a 139.

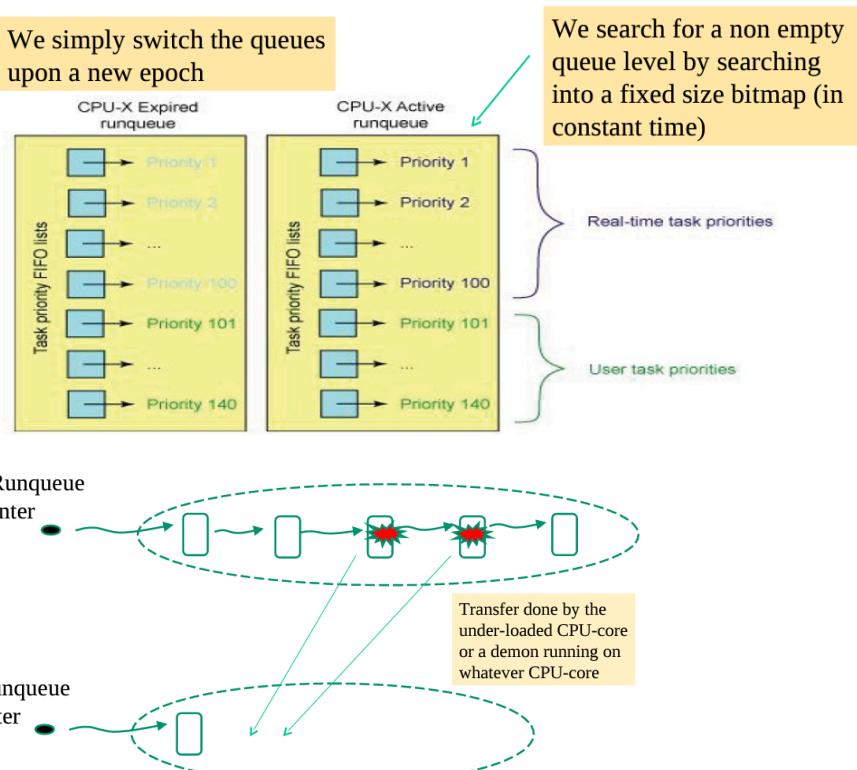
- Fino a 99 queste priorità sono definite **real-time** (`SCHED_RR` o `SCHED_FIFO`)  
Con `SCHED_RR` si è preemptabili dallo scheduler, ad un certo punto si può dover lasciare la cpu in un safe place;  
Con `SCHED_FIFO`, è lui che decide di chiamare un servizio bloccante ed andare in una WQ.
- Da 100 in su si è **non real-time** (`SCHED_OTHER`), quest'ultima era la vecchia gestione delle priorità.

Si può cambiare priorità con la systemcall `sched_setscheduler()`.

Lo scheduler O(1) inizialmente era pensato per gestire sia thread **real time** che `SCHED_OTHER`.

Capiamo quali caratteristiche portano a O(1):

- La RQ vista quando chiamo lo scheduler contiene solo threads runnable aventi ticks, si chiama **active queue**. Abbiamo **expired queue** che contiene thread che hanno esaurito i tick.  
Quando si arriva in una nuova epoca, si effettua uno swap tra i puntatori.
- Abbiamo una RQ per ciascuna CPU, ogni CPU guarda la propria quando deve schedulare, può guardare le altre se la propria è vuota. Parliamo quindi di **multi-queue**; ciò significa che un TCB può essere collegato ad **una sola** specifica coda, esse sono *organizzate per priorità*. Ciò consente la ricerca del thread a più alta priorità tramite una **bitmap**, non devo scandire tutta la coda.



## Attenzione:

In release recenti di Linux, un'applicazione cpu-intensive single thread, senza affinità settata, potrebbe non avere le prestazioni attese. Questo perché, soprattutto in presenza di demoni di sistema, tale applicazione potrebbe essere migrata molte volte. Risolvo settando l'affinità.

## **Funzionamento dell'API thread wakeup()**

```
wake_up_process(...)
```

```

graph TD
    A[Can the thread run on this CPU?  
If YES put on the local runqueue] --> B[If NO, get affinity info from TCB and put in some  
remote runqueue via the below API]
    B --> C[void ttwu_queue(struct task_struct *p,  
int cpu, int wake_flags)]
  
```

“`wake_up_process`” è l'unica api del kernel Linux che permette di ricollegare un TCB ad una RQ, in particolare su quella della cpu da cui chiamiamo `wakeup`, se l'affinità lo permette, altrimenti se ne sceglie una presente nello schema di affinità secondo schemi randomici.

Con “`ttwu_queue`”, un thread ad altissima priorità su `cpu0` può risvegliare altro thread importante su altra `cpu1` e mandargli un interrupt.

## Attribuzione dei ticks:

In **load sharing** l'assegnazione dei tick da spendere è basata sulla nozione di carico che è un'informazione mantenuta nel TCB come nuovo campo:

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

Dove:

```
struct load_weight {
    unsigned long weight; //determina il carico di ogni thread, basato su nice ness.
    u32 inv_weight;
}
```

## Dettagli aggiuntivi sulle priorità SCHED\_OTHER:

Si hanno due tipi di priorità per i non-real-time:

- priorità statica: assegnata dal programmatore
- priorità dinamica: varia a seconda dell'interattività del thread. Si parte da quella statica, ma esiste il concetto di penalità e premio, basata sull'interattività del thread. Genera inversione di priorità, ovvero un thread non real-time potrebbe spodestare un altro thread che in principio era più importante.

## **Scheduler attuale:**

Il problema è stato risolto eliminando la priorità dinamica.

I livelli di priorità non sono più gestiti nello stesso modo, ma se si è non real time, cioè `SCHED_OTHER`, non si va in code diverse a seconda del livello di priorità, ma si va tutti nella stessa coda di priorità.

`SCHED_OTHER` non è una lista, altrimenti si sarebbe  $O(N)$ , è un albero, per questo  $O(\log N)$ .

### Funzioni per lo scheduling:

Prima c'era solo `schedule()`, ora possiamo fare più operazioni.

Possiamo anche settare la priorità dinamica per le releases che la offrono.

`"load_balance()"` è una macro usata da demoni per bilanciare il carico.

<code>p-&gt;time_slice</code>	The residual ticks in the current epoch
<code>schedule</code>	The main scheduler function. Schedules the highest priority task for execution.
<code>load_balance</code>	Checks the CPU to see whether an imbalance exists, and attempts to move tasks if not balanced.
<code>effective_prio</code>	Returns the effective priority of a task (based on the static priority, but includes any rewards or penalties).
<code>recalc_task_prio</code>	Determines a task's bonus or penalty based on its idle time.
<code>source_load</code>	Calculates the load of the source CPU (from which a task could be migrated).
<code>target_load</code>	Calculates the load of a target CPU (where a task has the potential to be migrated).

### Stack refresh esplicito:

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-coreA`. Se poi viene rischedulato su un `CPU-coreB` diverso, abbiamo una 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 qual volta che si riprende il controllo della CPU; tale meccanismo è noto come *explicit stack refresh*.

### Struttura run queue:

```

struct rq {
    /* runqueue lock: */
    spinlock_t lock;

    /* nr_running and cpu_load should be in the same cacheline because remote CPUs
use both these fields when doing load calculation. */
    unsigned long nr_running;
#define CPU_LOAD_IDX_MAX 5
    unsigned long cpu_load[CPU_LOAD_IDX_MAX];
    unsigned char idle_at_tick;

    .....
    /* capture load from *all* tasks on this cpu: */
    struct load_weight load;

    .....
    struct task_struct *curr, *idle;
    .....
    struct mm_struct *prev_mm;
    .....
};


```

L'address space precedente passato qui serve per capire se mandare in flush tutta l'architettura TLB.

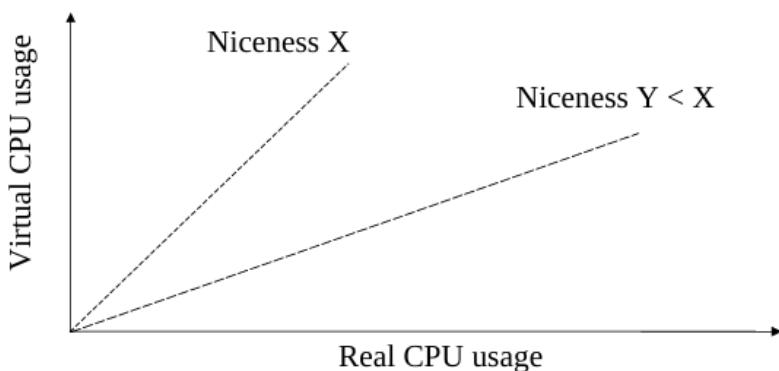
## Completely fair scheduling

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 (inserimento e rimozione nodo). 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, migliore è il posizionamento del thread nell'albero. Il thread che va in CPU è quello più a sinistra*.

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 in 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.



## Thread kernel:

Come tirare su un thread a livello kernel?

Prima si chiamava la **`sys_clone()`**, oggi si usa l'API "**`kernel_thread()`**" basata sull'ASM "**`arch_kernel_thread()`**", che internamente compie del preambolo prima di invocare **`sys_clone()`**, se il child è stato creato correttamente, si chiama la **`exit()`**.

```
long kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
    struct task_struct *task = current;
    unsigned old_task_dumpable;
    long ret;

    /* lock out any potential ptracer */
    task_lock(task);
    if (task->ptrace) {
        task_unlock(task);
        return -EPERM;
    }

    old_task_dumpable = task->task_dumpable;
    task->task_dumpable = 0;
    task_unlock(task);

    ret = arch_kernel_thread(fn, arg, flags);
    /* never reached in child process, only in parent */
    current->task_dumpable = old_task_dumpable;

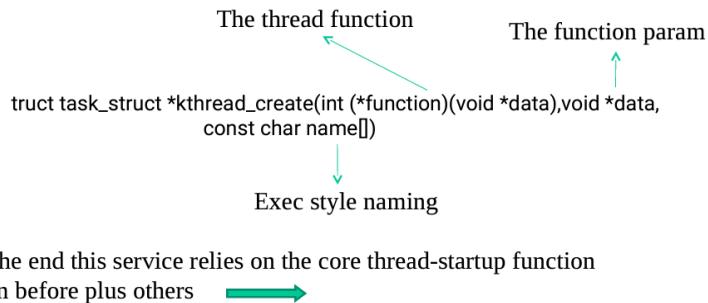
    return ret;
}

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 */
        "cmpl %%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\t"
        "1:\n\t"
        "=a" (retval), "=S" (d0)
        :"=r" (__NR_clone), "i" (__NR_exit),
        "r" (arg), "r" (fn),
        "b" (flags | CLONE_VM)
        : "memory");
    return retval;
}
```

Su releases moderne abbiamo l'API:

```
struct task_struct *kthread_create (int (*function)(void *data), void *data, const char name[])
```



Che realizza i seguenti step:

1. Il thread creato dorme su *wait queue*, ovvero esiste ma non è realmente attivo.
2. Dobbiamo “esplicitamente” svegliarlo.
3. Per quanto riguarda i segnali, abbiamo quanto segue:
  - 3.1) Possiamo killarlo, se il thread (o il creatore) lo consente, essendo un thread kernel (demone) ha delle informazioni di controllo che esplicitano o meno la possibilità di kill.  
Ciò sveglia il thread (se dorme), ma nessun messaggio è memorizzato sulla *signal mask*.  
La terminazione ha solo l'effetto di risvegliare il thread (se dormiente), ma non viene registrata alcuna consegna di messaggi nella maschera dei segnali. Tale azione viene realizzata manipolando un bit nel Thread Control Block, oppure lavorando sulla *signal mask*.

Possiamo creare un thread e collegarlo ad una RQ specifica, realizzando l'**affinità** su CPU:

```
struct task_struct *kthread_create_on_cpu(int (*function)(void *data),  
                                         void *data,  
                                         unsigned int cpu_id,  
                                         const char name[]);
```

Affinity settings for the new thread

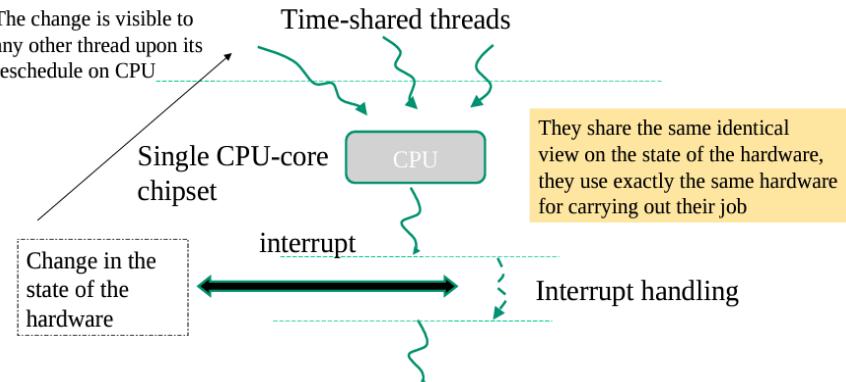
## Trap/interrupt architecture:

Vediamo trap e interrupt dal punto di vista architetturale, poi la loro relazione con il software.

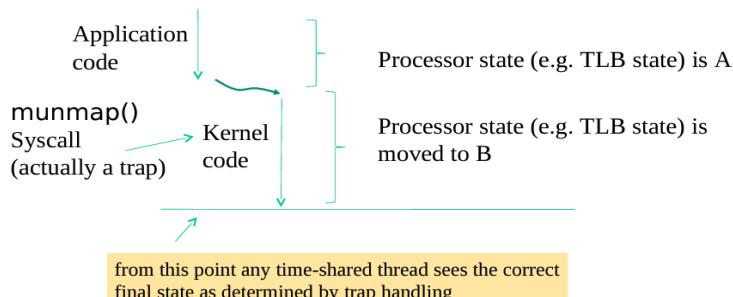
Storicamente su macchine single cpu single core l'interruzione era legata a due concetti inerenti al flusso di esecuzione:

- **Trap** causate dal flusso di esecuzione: oggetti sincroni.
- **Interrupt** causati dallo stato di dispositivi hardware esterni: oggetti asincroni.

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. L'handler va a lavorare sullo stato dell'hardware. Le modifiche che fa sono visibili da tutti i thread schedulati sulla CPU..



### Esempio con trap:



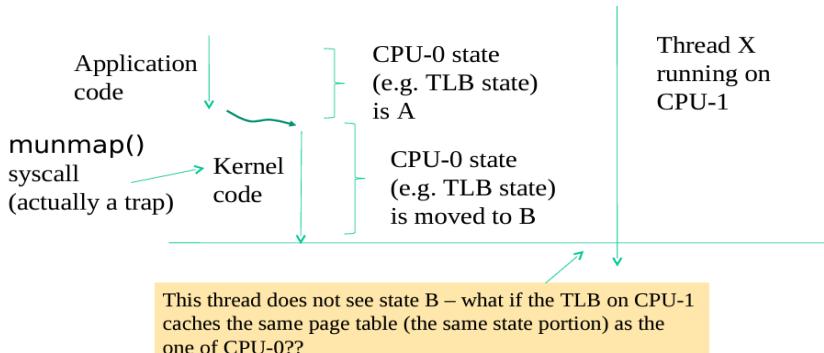
Consideriamo la trap data dalla system call **`munmap()`**, che unmappa determinate pagine di memoria. Il gestore di tale trap, oltre ad aggiornare la page table del processo chiamante, deve anche andare a invalidare il TLB presente nel processore. Fatto questo, poiché il processore è unico, qualunque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

### Passaggio al multi-cpu:

Passando ad un'architettura multi-cpu tutto ciò non è coerente.

La **`munmap()`** elimina parte dell'AS, dunque viene resettato qualcosa nella tabella delle pagine e si modifica il TLB. Il **TLB invalidato** è quello della **cpu su cui viene chiamata la `munmap`**.

Un thread su un'altra cpu deve vedere le stesse modifiche anche sul suo TLB.



Il sistema deve avere informazioni riguardo il SO *replicate su più cpu*, quando una trap o un interrupt modifica uno di questi oggetti, le modifiche vanno propagate agli altri.

Come si propaga una trap su un'altra cpu?

- Firmware: nell'hardware ho un sottosistema di controllo che quando accade qualcosa su un componente nella cpu0 lo propaga al corrispettivo componente sulla cpu1. Nel firmware non ci sono costrutti del tipo "if...else", esso si comporta in modo deterministico a seguito di un "command". Se i threads su cpu diverse operano su PT scorrelate il TLB non va invalidato su entrambe.
- Software: gli eventi vengono propagati e gestiti dal sistema operativo.

## IPI - Inter Processor Interrupt

E' un interrupt ricevibile da una CPU ed inoltrabile ad un'altra. IPI è un evento *sincrono* al mittente e *asincrono* al ricevente. 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 hanno due livelli di priorità diversi:

- *high*: il ricevente processa immediatamente
- *low*: porta all'accodamento degli IPI che poi verranno processati nel momento più opportuno (in stile top half / bottom half). Il TH è soggetto a schemi di priorità.

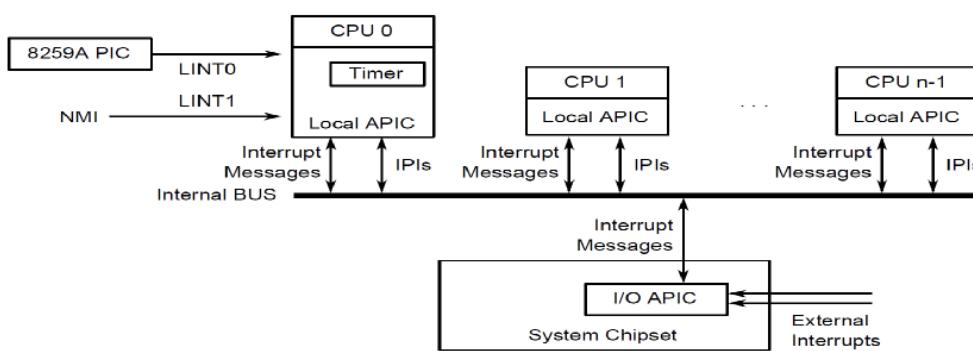
## Supporto x86 alle interrupt:

Nei sistemi x86, l'architettura *hardware* 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 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). Ulteriore componente è l'*I/O APIC*, che si occupa di accettare gli interrupt esterni (quelli da dispositivi esterni) e inoltrare i messaggi di interrupt ai CPU-core interessati (I/O APIC può parlare con tutti i LAPIC), sfruttando sempre l'APIC bus.

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

*Schema architettonico:*



## Nomenclatura:

**IRQ** è il codice effettivo associato ad una richiesta di interrupt.

**INT** è la linea di interruzione vista localmente dalla CPU ("quello che vede la CPU"); essenzialmente INT=F(IRQ). Su x86 INT=IRQ+32; queste targhe sono associate a qualcosa proveniente dall'esterno, i valori da 0 a 31 sono riservati alle trap.

## 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. In funzione della 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. Ad esempio, allo startup posso inoltrare gli interrupt solo ad alcune CPU.
- Logica: la richiesta di interrupt può coinvolgere diversi CPU-core, dove ne viene selezionato uno per volta secondo uno schema round-robin. Esistono attività "stabili", come il TLB shutdown (che interessa l'hardware) e le *function call*, per usare IPI senza cambiare stato dell'hardware. Vengono eseguite funzioni su richiesta di altre CPU. Esempio: scrivere una variabile per-cpu, come l'id.

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

Come si fa dal punto di vista software a gestire una variazione di flusso in base ad una targa? Questo meccanismo è legato al concetto di "*Trap interrupt table*"; è una tabella in memoria in cui la targa rappresenta l'offset nella tabella e nella entry troviamo ciò che serve ad identificare l'handler.

La IDT ha 256 entries (sia protected, sia long); è una struttura modulare, quindi sul processore abbiamo l'idtr, un registro packed che contiene l'indirizzo virtuale della IDT e la sua taglia.

L'idtr si manipola con:

- LIDT (load idt): se scrivo in idtr, cambio la IDT in uso sia per uno o più CPU-core.
- SIDT (set idt);

Correntemente Linux usa una sola IDT per tutti i processori, ciò non vieta di crearne un'altra.

Brevemente, alcune info sulle 256 entries:

- Le entries dalla 0 alla 19 sono per le *trap* e per i "non-maskable interrupts", ovvero la linea di interrupt relativi a ciò che avviene nella cpu.
- Le entries dalla 20 alla 31 sono Inter-reserved.
- L'entry 239 è per il LOCAL APIC timer interrupt.
- Le entries dalla 251 alla 255 sono per gli IPI; in particolare la 255 è *denominata entry spuria*: in una IDT tutte le entries devono essere valide, se una targa porta ad un'entry non valida il sottosistema di controllo non sa come raggiungere l'handler e ciò porta ad un errore nel flusso di esecuzione.

## Entries IDT in x86 protected mode:

Gli elementi della IDT sono struct di tipo "`desc_struct`", che prevede due `unsigned long` (32 bit).

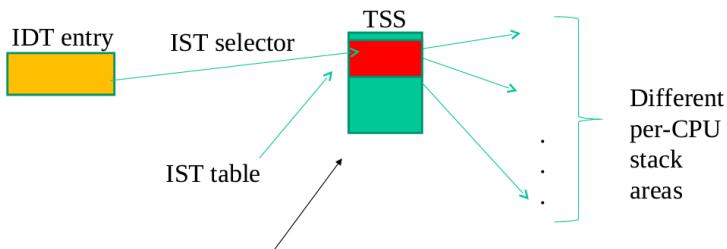
```
struct desc_struct { unsigned long a, b; }
```

Composta da:

- 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. Con l'offset sto descrivendo il GATE.

### Entries IDT in Long mode:

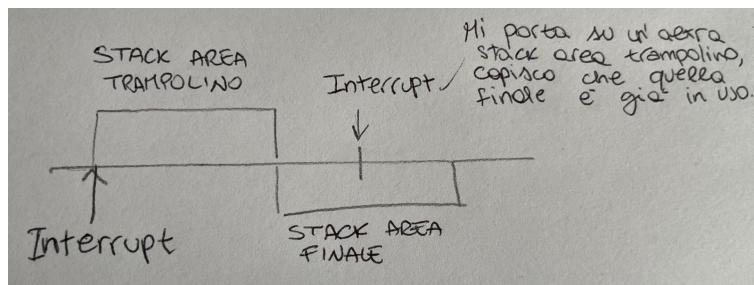
- L'offset non può più essere a 64 bit ma a 128. Si individua una parte high, medium (segment) e low.
- **IST(Interrupt Stack Table)** è una entry a 3 bit. Ha 8 possibili valori (in realtà il valore 0 indica che IST è disabilitato e usa TSS) per identificare 7 stack differenti. Ciò permette di usare stack aree diverse in funzione di tale gestore, senza usare la stessa che potrebbe essere "sporca". Il selettore IST ci porta nel TSS, che identifica stack area diverse se tale meccanismo è abilitato; altrimenti usa quella che viene detta "stack area trampolino".



These are typically the primary stacks (possibly of different size) for processing a given trap/interrupts  
Software will then switch to the classical kernel level stack of the running task if nothing prevents it (e.g. a double fault)

### Esempio IST:

Siano S e S' due stack areas diverse. Il cambio di stack area è fatto dal sottosistema di controllo. Dove si trova S? Da una cpu posso accedere a TSS, che riporta i metadati basici per il thread corrente. C'è anche lo stack pointer. Le stack areas si prendono mediante un allocatore. Quando si gira in user mode la stack area del kernel non è visibile. IST permette di prelevare dal TSS lo stack pointer che ci serve per eseguire l'handler; è una stack area trampolino.



### API x86 protected mode:

Ci sono tre api per settare le entries della IDT.

Specifichiamo il displacement dell'entry e l'indirizzo lineare dell'handler.

Queste 3 api internamente già sanno come settare i bit di controllo, per questo necessitano di soli due parametri.

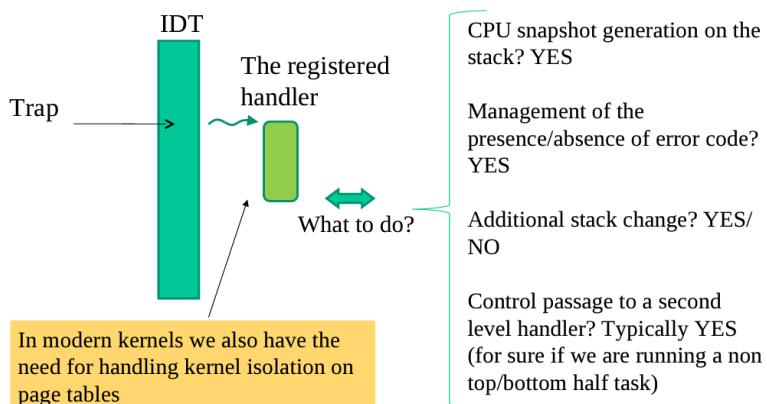
- La funzione `set_trap_gate(displacement,&symbol_name)` inizializza una voce nell'IDT in modo tale da definire il valore 0 come il livello di privilegio ammesso per l'accesso al GATE tramite software. Pertanto, non possiamo fare affidamento sull'istruzione in assembly INT a meno che non stiamo già eseguendo in modalità kernel.
- La funzione `set_intr_gate(displacement,&symbol_name)` sembra simile, tuttavia l'attivazione dell'handler si basa sull'interrupt masking.
- La funzione `set_system_gate(displacement,&symbol_name)` è simile a `set_trap_gate()`, tuttavia definisce il valore 3 come livello di privilegio ammesso per l'accesso al GATE.

**Recap sulle azioni degli handlers di trap/interrupt:**

Quando avviene una trap e si cambia flusso di esecuzione può esserci bisogno di un'informazione aggiuntiva; ad esempio se la trap è dovuta ad un'istruzione che vuole accedere in memoria, ma non può avvenire la risoluzione dell'indirizzo dalla PT. (Ciò è necessario in caso di trap, non in caso di interrupt). Il software deve essere quanto più semplice ed unico possibile, nella gestione delle trap la prima cosa che deve essere fatta è lo *snapshot di cpu*, oggetto unico che gestisce cose diverse, dunque il software deve saper gestire un caso generale; le informazioni aggiuntive se non presenti vengono inserite dummy.

Nella parte early (un preambolo) del gestore ci si pongono una serie di domande.

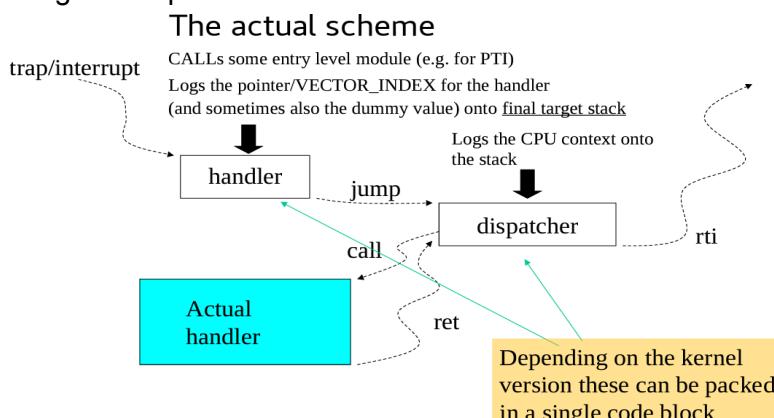
Dipende da cosa è previsto dalla entry della IDT (campo IST).

**Dispatcher:**

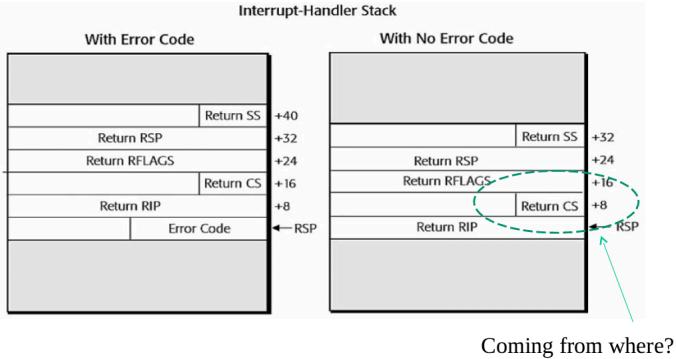
Gli handler di trap / interrupt vengono gestiti mediante un **dispatcher unico in tutto il sistema**.

- 1) 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, si memorizza un valore dummy (in modo tale che lo stack abbia sempre la stessa struttura, il che semplifica operazioni come la deallocazione dello stack stesso);
- 2) Si riporta sullo stack anche l'indirizzo della funzione che implementa l'handler di secondo livello.
- 3) 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 (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).
- 4) 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:



## Dettagli sulla conformazione dello stack usato dall'handler:



## Dettagli sullo snapshot di CPU:

### 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.

### 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)**

Un esempio di 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.

Una nuova versione è asmlinkage `void __kprobes do_page_fault(struct pt_regs *regs, unsigned long write, unsigned long address)` che prende in input anche il valore dell'indirizzo che causa il page fault (che è salvato in CR2).

## **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 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.

### **Gestione 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 t0 che gira sul CPU-core0 a un certo punto risveglia un thread t1 che è prioritario ma ha affinità solo su CPU-core1; allora t0 invia un IPI verso il CPU-core1 per far eseguire la funzione **schedule()** sul CPU-core1. Qui la entry della IDT che viene coinvolta è la **RESCHEDULE\_VECTOR** e può essere usata chiamando la funzione **reschedule\_interrupt()**.

Le API per gli IPI sono:

- **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

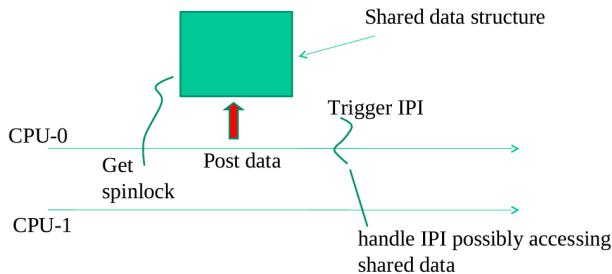
### **Sequenzializzazione della gestione IPI:**

Come è supportato IPI correntemente per l'esecuzione su cpu remote?

La chiamata a cpu remota coincide con l'esecuzione di un protocollo. Viene messa in atto una sorta di serializzazione. C'è necessità di passare parametri specifici, in particolare due informazioni: il *function pointer alla funzione da invocare* ed il *puntatore da passare alla funzione*.

Quest'area è unica, quindi quando si chiama la funzione su una cpu remota quest'area va bloccata mediante spinlock. Come si fa a sapere se l'altra cpu ha eseguito la funzione?

Vengono aggiornati i metadati del processore che ha chiamato l'IPI, sono strutture dati condivise.



Per chiamare una funzione su cpu remota si usa “`smp_call_function()`”.

Il terzo parametro dice se devo aspettare che gli altri abbiano processato la funzione; se non si deve attendere si rilascia il lock, altrimenti si sta in busy wait.

Se prima di entrare qui resetto il bit di interrompibilità ho un bug che porta a deadlock:

Supponiamo che CPU0 voglia prendere il lock. Questo però è mantenuto da CPU1 che sta aspettando che tutte le altre CPU (tra cui la 0 che è non-interrompibile) processino la funzione.

```
int smp_call_function(void (*_func)(void *_info), void *_info, int wait)
{
...
/* Can deadlock when called with interrupts disabled */
WARN_ON(irqs_disabled());
spin_lock_bh(&call_lock);
atomic_set(&scf_started, 0);
atomic_set(&scf_finished, 0);
func = _func;
info = _info;

for_each_online_cpu(i)
    os_write_file(cpu_data[i].ipi_pipe[1], "C", 1);

while (atomic_read(&scf_started) != cpus)
    barrier();

if (wait)
    while (atomic_read(&scf_finished) != cpus)
        barrier();

spin_unlock_bh(&call_lock);
return 0;
}
```

Beware this!!

La funzione da far eseguire ad un'altra cpu mi aspetto che sia rapida, chi riceve IPI esegue il top half per due motivi: *non sappiamo cosa stava facendo l'altro o io sono in busy wait*. Uso runqueue per cose più complesse.

### **Effetti addizionali API**

Gli IPI possono avere effetti sulla consistenza e sulle performance del sistema. 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. Di conseguenza, è necessario che il task non abbandoni mai la CPU nell'intervallo di tempo in cui utilizza una stessa variabile 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

### Preemption:

Per quanto riguarda `smp_call_function()` bisogna ricordare il problema dei deadlock: 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: 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 il parametro `wait` indica se il chiamante deve attendere che tutti i CPU-core reagiscano all'IPI o meno. 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. può essere decrementato tramite l'API `preempt_enable()`

```
preempt_enable() // decrement the preempt counter  
preempt_disable() // increment the preempt counter  
preempt_enable_no_resched() decrement, but do not  
immediately preempt  
preempt_check_resched() // if needed, reschedule  
preempt_count() return the preempt counter  
put_cpu() /get_cpu() //decrease/increase the counter  
(enable/disable preemption)
```

Variants of each other

A valle di queste considerazioni, la funzione `smp_call_function()` si è evoluta nel seguente modo:  
Io, da una CPU, chiedo ad altri delle cose, e io non voglio migrare su altre CPU. Non usiamo  
`smp_call_function()` in maniera massiva, perché alla fine ha il suo costo.

```
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

### **Conclusioni:**

Gli IPI sono una tecnologia potente, ma vanno considerati aspetti legati alla scalabilità.

Gli IPI che includono diversi core della cpu devono essere usati solo quando è obbligatorio usarli.

## Virtual file system:

E' il sottosistema che ci permette di eseguire le operazioni di I/O.

Vedremo anche dettagli su come si progetta un VFS e poi anche su qualcosa di più basso livello, come progettare device RAM.

### File system:

Gli oggetti di I/O che possiamo usare quando chiamiamo servizi di sistema come sono rappresentabili?

Abbiamo due rappresentazioni:

- In RAM: rappresentazione parziale o completa della struttura e del contenuto del filesystem.  
Rappresentazione completa ci fa capire che possono esistere FS le cui rappresentazioni esistono solo in memoria, non esistono su dispositivi di memoria di massa.
- On device: questa può essere non aggiornata.

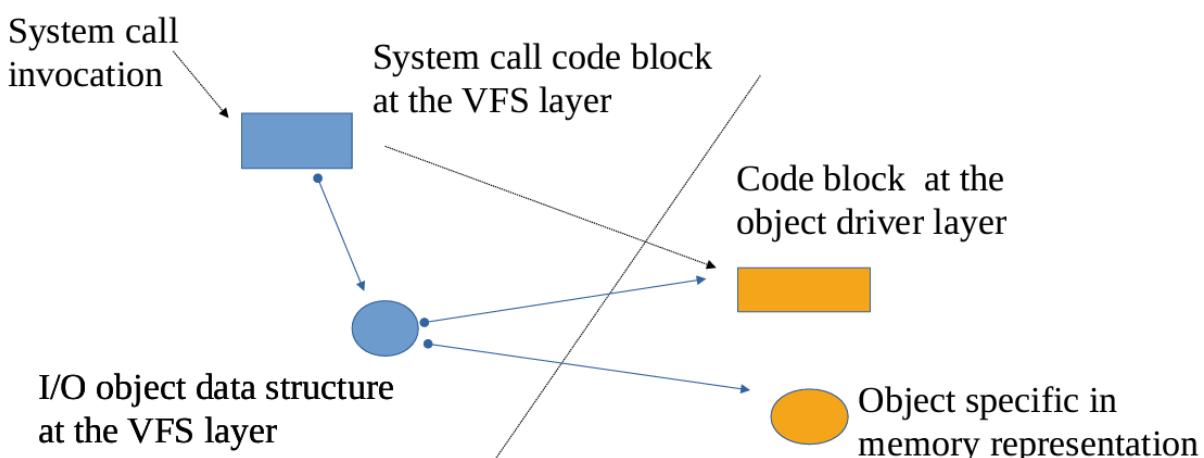
Se ragioniamo sul discorso RAM o device, abbiamo che ciò che sta sul device può essere non aggiornato rispetto a ciò che la cpu esegue in RAM su un oggetto.

Dobbiamo capire come questi due livelli si relazionano.

Per quanto riguarda l'accesso ai dati e la loro manipolazione si hanno due parti:

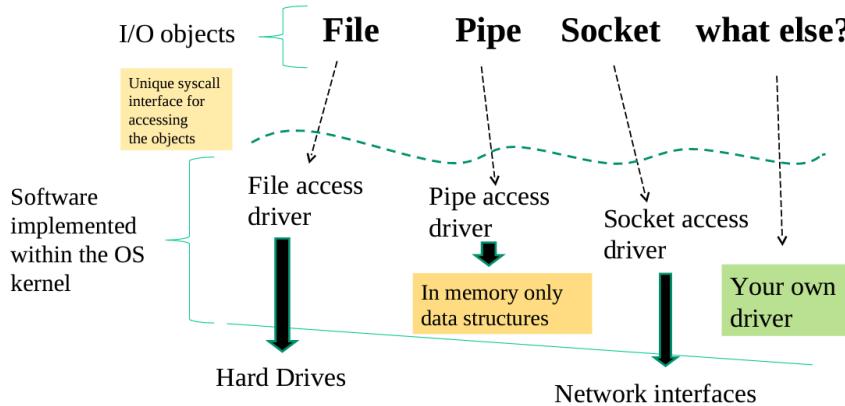
- Parte indipendente dal FS (FS-independent): è un layer del software che può essere invocato all'interno del kernel anche da parte degli *altri sottosistemi* e permette di lavorare sugli oggetti di I/O appartenenti al file system (e.g. i file o le directory) in modo indipendente dal file system stesso; infatti, tali oggetti sono rappresentati all'interno di strutture dati fatte apposta per il layer del software (come gli inode). 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 generiche. La parte indipendente usa un driver che non è altro che una tabella di function pointers che ci rimanda all'indirizzo di memoria specifico relativo alla parte dipendente.



Quando cominciamo a parlare di dispositivi, questi in un'architettura Linux come si possono gestire realmente dal punto di vista del software?

Nel VFS i dispositivi possono essere visti come **files**. Per gestirli viene utilizzato un **driver** (quindi abbiamo sia i driver che pilotano i dispositivi sia driver che lavorano a livello di 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

Quando vogliamo *mantenere qualcosa* su un filesystem effettivo, che ha una propria rappresentazione non solo in memory ma anche su un dispositivo, vanno fatte quattro azioni:

- 1) avere una funzione (servizio) che permette di leggere il super-blocco del dispositivo per scoprire dove sono posizionati i dati dei file.
- 2) quando lavoriamo su un file dobbiamo lavorare in RAM, eventualmente bisogna caricare altri blocchi in RAM.
- 3) dobbiamo leggere o scrivere in memoria e flushare queste cose scritte su blocchi di dispositivo.
- 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)

Come si mappa tutto ciò rispetto ai concetti che abbiamo espresso?

### **Block vs char device drivers**

Le prime tre funzioni precedenti sono collegate a **block-device drivers** che prevedono funzioni che leggono e scrivono i blocchi dei file (principalmente portano i blocchi sul device). Le ultime funzioni riguardano **char-device drivers** che solitamente vengono usati per lavorare in memory sui singoli caratteri (per prelevarli e consegnarli, usati per le tastiere) ma possono anche essere usati per pilotare dispositivi. Il funzionamento di una chiamata a system call su un dispositivo è il seguente:

- La system call viene intercettata dal FS indipendente che va ad invocare un char device driver.
- Siccome il file su cui si vuole eseguire la system call potrebbe non essere in memoria bisogna appoggiarsi al block device driver per lavorare sul dispositivo dove si trova il file. Se il file è in memoria non serve usare il block device driver.

### Esempio:

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.

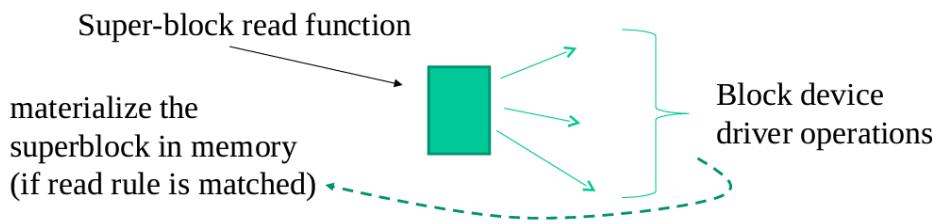
I driver (tra cui il block-device driver e il char-device driver) sono essenzialmente delle tabelle di *function pointer* che puntano all'implementazione effettiva delle operazioni che possono essere eseguite sull'oggetto target. Il fatto che driver differenti implementino in maniera diversa una stessa operazione, implica che una stessa entry di tabelle di function pointer diverse punti 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 systems Linux

### Superblock read function

Come abbiamo detto ci sono vari tipi di file systems. Linux mantiene una lista dei FS che gestisce. Ciascun elemento di tale lista ha un puntatore ad una *funzione di lettura di un superblocco*. Questa funzione fa affidamento sul *block-device driver* per istanziare il superblocco del dispositivo in memoria.

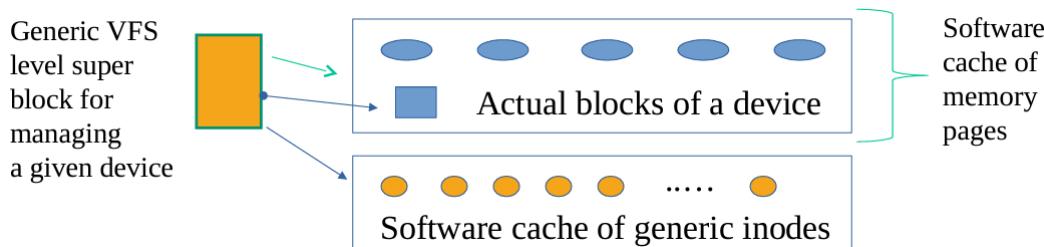
Se il filesystem è in RAM (volatile) non serve avere questa funzione.



Quando monto un FS c'è bisogno di un *mount point*, ovvero un punto di un FS già montato, quando mandiamo in startup Linux viene montato subito il root file system.

### Intermediario tra parte in memory e parte on device: buffer/page cache

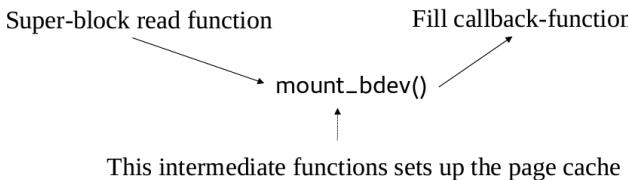
Il buffer/page cache è l'intermediario tra la parte in memory e quella on device. Non è che una cache in memoria RAM delle info del dispositivo. Permette alla superblock read function di leggere i blocchi del dispositivo passando attraverso una struttura dati che rappresenta un superblock generico. Questo non è altro che una struttura di accesso ai blocchi in cache del dispositivo. I blocchi cache hanno un proprio indice per poter essere selezionati nelle operazioni. Questo intermediario mantiene strutture dati separate per ciascun dispositivo. La head di tali strutture è appunto il superblock generico che mantiene un riferimento ad una cache dei blocchi attuali del filesystem (tra cui anche il superblocco effettivo) e una cache di inodes (che possono essere correlati ai blocchi)



La funzione di lettura del superblocco può sfruttare le API del kernel per fare il setup del superblocco a livello VFS, in particolare:

- `mount_bdev()`: fa setup della cache e delle altre strutture dati del VFS.
- `mount_single()`
- `mount_nodev()`: monta un file system che non è su un dispositivo fisico

Tutte queste funzioni prendono come parametro una funzione di callback che si occupa di finalizzare il superblocco (cioè lo riempie). Ovviamente tale funzione dipende da che tipo è il blocco che stiamo chiamando. Lo schema è il seguente:



### ***Magic number:***

Abbiamo parlato di una lista di FS gestibili da Linux con la loro funzione di lettura del superblocco.

Quando portiamo il blocco *in cache* verifichiamo se il super blocco mantiene una targa, il magic number, che identifica il tipo di file system e quindi la struttura dei relativi oggetti I/O e come accedervi.

Su Linux possiamo leggere i magic numbers di tutti i dispositivi con il comando:

`file [-s] /dev/{device-name}`

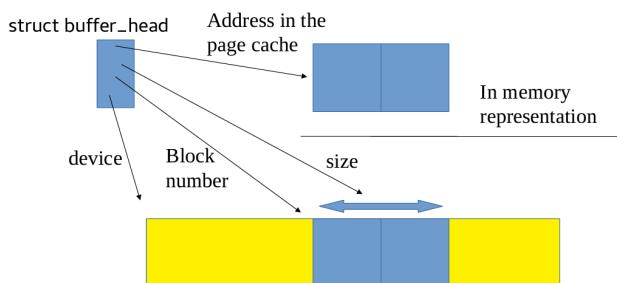
I nomi dei device si trovano in uno pseudo file.

Se un file system ha un magic number per cui non abbiamo la funzione di lettura del super blocco non sappiamo gestirlo.

### ***Dettagli buffer/page cache:***

L'intermediario buffer/page cache è rappresentato dalla struttura dati “`buffer_head`” con tali campi:

- `*b_data`: puntatore all'area di memoria dove si trovano i dati.
- `b_size`: taglia del buffer (posso usare taglie diverse).
- `*b_bdev`: block device da cui stiamo leggendo, lo identifica.
- `b_blocknr`: indice del blocco che ho caricato o che devo flushare.



### **Operazioni:**

Operazioni che mette a disposizione il page/cache buffer di Linux per leggere o scrivere su un block device:

- `__bread()`: legge un blocco che ha il numero e la taglia indicati in una struttura `buffer_head` e restituisce il puntatore a quest'ultima (NULL se errore).
- `sb_bread()`: rispetto a sopra, la taglia del blocco da leggere è presa dal superblocco.
- `mark_buffer_dirty()`: marca il buffer come dirty. Il buffer verrà riportato su disco in seguito da un demone.
- `brelse()`: avverte la page cache che il blocco prima richiesto (ad es. con una `__bread`) non serve più. Se nessuno lo sta usando il blocco può essere liberato.
- `map_bh()`: indica l'associazione tra il blocco puntato e il settore del dispositivo in cui esso si trova.

## Architettura:

Quando chiamiamo una system call, questa si relazione con l'interfaccia del VFS, che si relazione con l'intermediario (buffer/page cache) per interagire con il dispositivo.

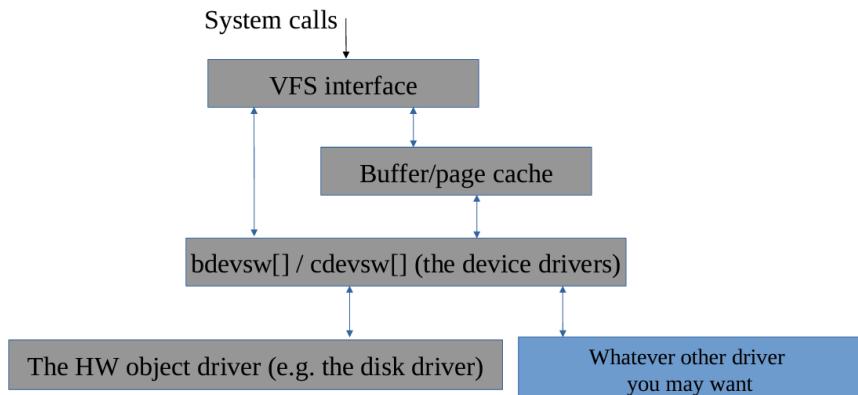
Ci si può anche ridirezionare direttamente sul device driver.

I FS usano oggetti già esistenti nel sistema, driver di basso livello.

Noi possiamo anche scrivere il driver per il nostro dispositivo di I/O.

I file system veri usano block device drivers, ma ad esempio read a standard input usa char device driver.

Questa differenza la fa capire al VFS il canale su cui lavoriamo, un canale permette di arrivare su una struttura di sessione.



## **File regolari vs file system:**

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”.

## **RAM file system:**

Per i file system totalmente in RAM, non c’è necessità di una rappresentazione on device, per cui la funzione di lettura del superblocco non ha bisogno di leggere blocchi da un dispositivo. Quello che si fa è semplicemente istanziare in memoria un nuovo superblocco che rappresenta la nuova incarnazione del filesystem (quello che succede allo startup Linux). Una tra le funzioni usate è `ramfs_get_inode()` che si occupa di inizializzare il superblocco e allocare inodes e dentry.

## **Inode e dentry**

Sono le strutture dati generiche usate dal VFS:

- Inode = struttura dati del VFS che rappresenta genericamente 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.

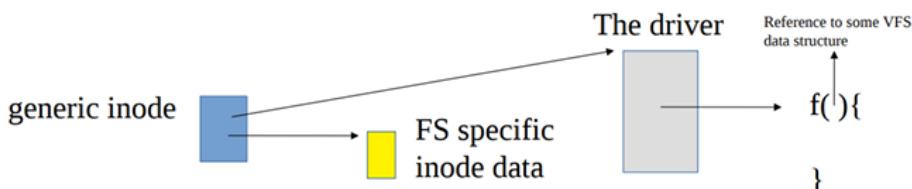
Due API per creare queste strutture dati sono riportate qui di seguito:

- `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). T

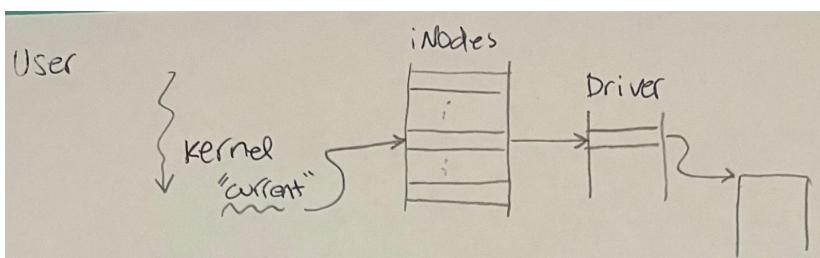
Tipicamente l’inode generico mantiene dei campi generici usati dal VFS e dei campi che servono a referenziare dati specifici del file system in cui ci troviamo.



Sappiamo che un driver è una tabella di puntatori. Quando un'operazione di tale driver è invocata, si passa tra i parametri l'indirizzo della relativa struttura dati. A partire da questo indirizzo il driver può accedere ai dati specifici del FS. Inoltre, una struttura dati nel VFS mantiene un riferimento al driver effettivo per le sue operazioni.



Quando un thread deve utilizzare un file, deve entrare nel livello kernel del sistema operativo tramite una system call. Se la system call riguarda un'operazione di input/output (I/O), essa riceve un codice che identifica il canale di I/O da utilizzare. Il kernel, partendo da “*current*” (processo in esecuzione) utilizza questo codice per trovare il puntatore all'inode corrispondente. All'interno dell'inode, c'è un puntatore al driver specifico del filesystem che gestisce quel file. Il kernel può chiamare la funzione specifica implementata nel software del driver per eseguire l'operazione richiesta, come leggere o scrivere il file.



Per esempio, quando un thread vuole aprire un file e chiama la system call `open`, il controllo passa al kernel. Il kernel utilizza il file descriptor per trovare il puntatore all'inode del file nel VFS. L'inode contiene un puntatore al driver del filesystem specifico (ad esempio, ext4, NTFS, ecc.). Il kernel quindi chiama la funzione `open` implementata dal driver di quel filesystem specifico.

### Lez.33 (13/12/23)

#### VFS startup in Linux:

Per inizializzare il VFS si invocano le seguenti funzioni:

- `vfs_cache_init()`: invocata dal thread di inizializzazione del kernel. Inizializza tutta la zona di memoria che serve per gestire il VFS e chiama internamente la funzione `mnt_init()`.
- `mnt_init()`: invoca le due funzioni successive:
- `init_rootfs()`: istanzia il FS “ancora” degli altri FS.
- `init_mount_tree()`: crea l'albero di montaggio.

Tipicamente si supportano almeno due tipi di filesystem che sono *Rootfs* e *Ext*. In realtà il kernel può essere configurato per non supportare alcun FS (la lista delle funzioni di lettura è vuota). In questo caso tutte le funzionalità che servono a far funzionare il kernel sono implementate direttamente all'interno del codice del kernel stesso e vengono eseguite da demoni.

```

>vfs_caches_init()
  >mnt_init()
    ✓init_rootfs()
    ✓init_mount_tree()

```

This tells we are instantiating at least one FS type – the **Rootfs**

### **Strutture dati “file system types”:**

La descrizione di un FS è fatta dalla struct `file_system_type`. Oltre a mantenere le informazioni riguardo al FS include anche un puntatore alla funzione di lettura del superblocco (campo `read_super`), il quale deve essere già stato creato e passato come parametro. Il campo `owner` ci indica se la funzione di lettura si trova all'interno di un modulo (in questo caso occorre bloccarlo per evitare che venga smontato mentre le funzioni sono usate).

Nelle versioni kernel più recenti la struct appare così:

```

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

```

In questo caso la funzione `mount` non ha un puntatore al superblocco ma controlla la lista delle funzioni per vedere se il FS indicato è gestibile. Se non è indicato alcun FS va a tentativi funzione su funzione.

```

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!!

```

### **Rootfs and basic fs-type API:**

L'allocazione delle strutture che tengono traccia di Rootfs viene fatta a compile time. Nel booting tale struct `file_system_type` mantiene le informazioni di Rootfs. Questo FS si trova solo in RAM e dunque viene inizializzato ad ogni boot del kernel. E' il punto di partenza per gli altri FS.

Per linkare una struct `file_system_type` di un FS alla lista si può utilizzare l'API  
`int register_filesystem(struct file_system_type *)`.

Nel caso della struct di Rootfs il linkage viene fatto dalla funzione `init_rootfs()`:

```

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

```

```

static struct file_system_type rootfs_fs_type = {
    .name        = "rootfs",
    .mount       = rootfs_mount,
    .kill_sb     = kill_litter_super,
};

int __init init_rootfs(void)
{
    int err = register_filesystem(&rootfs_fs_type);

    if (err)
        return err;

    if (IS_ENABLED(CONFIG_TMPFS) && !saved_root_name[0] &&
        (!root_fs_names || strstr(root_fs_names, "tmpfs")))
    {
        err = shmem_init();
        is_tmpfs = true;
    } else {
        err = init_ramfs_fs();
    }

    if (err)
        unregister_filesystem(&rootfs_fs_type);

    return err;
}

```

A few modifications in the structure  
of `init_rootfs()` are in kernel 5

Per vedere i FS correntemente gestiti dal kernel si può consultare il file `/proc/filesystems`. Accanto ai nomi del FS può essere presente la keyword `nodev` (indica che il Fs è in memory).

### **Strutture dati VFS:**

Ci sono quattro entries per gestire un file system:

- `struct vfsmount`: mantiene info come chi è il parent FS;
- `struct super_block`: mantiene metadati del FS.
- `struct inode`: contiene metadati degli oggetti I/O (files);
- `struct dentry`: mantiene il nome del FS e la gerarchia del FS.

Le prime due hanno informazioni dell'intero file system (una per ogni FS) mentre le ultime due hanno informazioni per ogni file e directory esistenti del FS (una per ogni file).

### **vfsmount**

E' la tabella per relazionare due file systems; possiamo stabilire se un file system in un VFS è montato su un altro e risalire alla tabella dell'altro; punta alla tabella del superblocco.

Questa struttura è stata resa più semplice; è stato introdotto un concetto interessante dal punto di vista della sicurezza. Si è visto che molti attacchi anche a livello kernel Linux sono basati sul ottenere in memoria queste informazioni, da cui reperire gli inodes. Avendo un inode posso chiamare funzioni.

Per evitare questo problema si utilizza `_randomize_layout`, quando una tabella è compilata con randomized layout non sappiamo in che posizione e a che distanza sono posizionati questi oggetti in memoria. Si usa a partire la kernel 4.8.

Per default, se abbiamo una struttura dati che contiene solo *function pointers* (come i driver), essa viene randomizzata anche se non è marcata con `_randomize_layout`.

La randomizzazione è applicata a compile time, quindi quando viene compilata l'immagine del kernel, quando si monta un modulo, che può dover usare quelle tabelle, devo conoscere come è stato randomizzato il kernel; la regola di randomizzazione si basa su seeds pseudo randomici; questi seeds si usano in "lead\_modules".

Torvalds in un articolo dice che questo serve solo in una release di Linux limitata, in cui non si prevedono updates con moduli.

### **Struttura dati superblocco:**

- `s_op` contiene le super operations che devono essere fatte per gestire il superblocco (non devono essere per forza presenti).
- `s_fs_info` è il puntatore al superblocco effettivo del FS.
- `s_user_ns` che indica il namespace del modulo.

```

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;

```

### Struttura dati dentry:

Registra il nome della directory ed è estremamente utile per navigare tra le directories.

Associa il nome della directory o file al relativo inode.

Anche in questo caso vi è un campo (`d_fsdata`) che referenzia un'area di memoria dove sono salvate le informazioni effettive del FS.

```

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_fsdata; /* fs-specific data */
    ...
    struct list_head d_child; /* child of parent list */
    struct list_head d_subdirs; /* our children */
    ...
} __randomize_layout;

```

### Struttura dati inode:

Un Node identifica due driver:

- `i_op` che è il driver dove sono memorizzate le funzioni per l'inode (quindi per lavorare sui metadati del file)
- `i_sb` che è il driver dove sono memorizzate le operazioni sul file (quindi per lavorare sui dati del file).

Inoltre, `i_pointer` è un puntatore ad un'area di memoria che mantiene le informazioni sul FS

```

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;

```

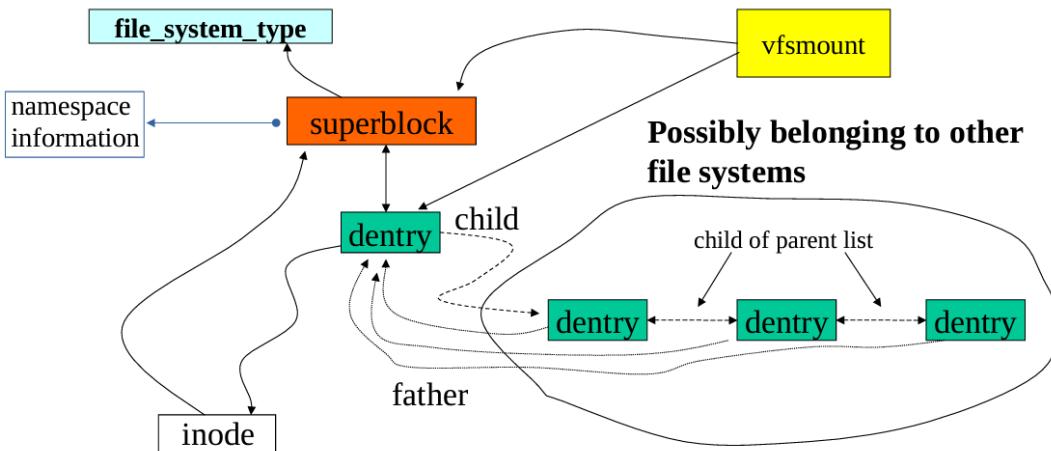
### Schema del virtual file system:

Ogni inode identifica il superblocco di appartenenza, quindi a quale FS appartiene l'inode, ogni dentry identifica l'inode.

“file\_system\_type” è collegato al superblocco, il superblocco permette di accedere anche a ulteriori informazioni sui namespaces, ci dicono a quale namespace afferisce un certo montaggio.

Di default, il montaggio è visibile su tutti i namespaces, se si vuole che la visibilità sia ridotta solo ad alcuni file systems si devono usare delle systemcall specifiche.

Le dentry figlie possono essere su un altro filesystem rispetto a quella genitore. (quindi essere montate su una directory di un altro FS).



### **Inizializzazione di istanza rootfs:**

I task eseguiti sono all'interno di “init\_mount\_tree()”.

Bisogna:

1. allocare le quattro strutture dati per Rootfs (**vsmount** per sapere il parent FS, **superblock** per i metadati FS, **inode** per i metadati degli oggetti I/O, **dentry** per il nome dell'oggetto I/O nel suo FS).
2. collegare le strutture dati.
3. associare il nome "/" alla root del filesystem
4. collegare il TCB dell'idle process a Rootfs (il TCB deve poter raggiungere il FS)

L'ultimo punto ci fa capire che su Linux abbiamo il supporto sul VFS per operazioni assolutamente differenziate.

Le prime tre funzioni sono svolte da `do_kern_mount()` o `vfs_kern_mount()` che si occupano di invocare la funzione di lettura del superblocco di Rootfs. Il linkage con idle process viene fatto da `set_fs_pwd()` e `set_fs_root()`.

Il montaggio del file system avviene nello stesso namespace del thread ma è visibile anche agli altri namespaces.

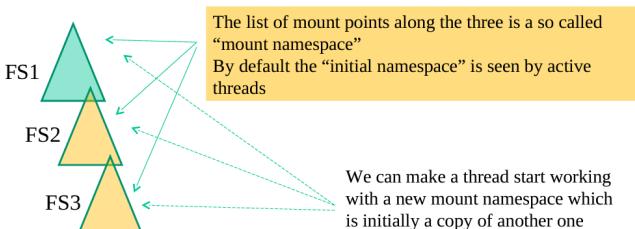
```
static void __init init_mount_tree(void){
    struct vfsmount *mnt;
    struct namespace *namespace;
    struct task_struct *p;

    mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
    if (IS_ERR(mnt))
        panic("Can't create rootfs");
    .....
    set_fs_pwd(current->fs, namespace->root,
               namespace->root->mnt_root);
    set_fs_root(current->fs, namespace->root,
               namespace->root->mnt_root);
}
```

.... very minor changes of this  
function are in kernel 4.xx/5.xx

## Montaggio di un filesystem e namespaces:

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.



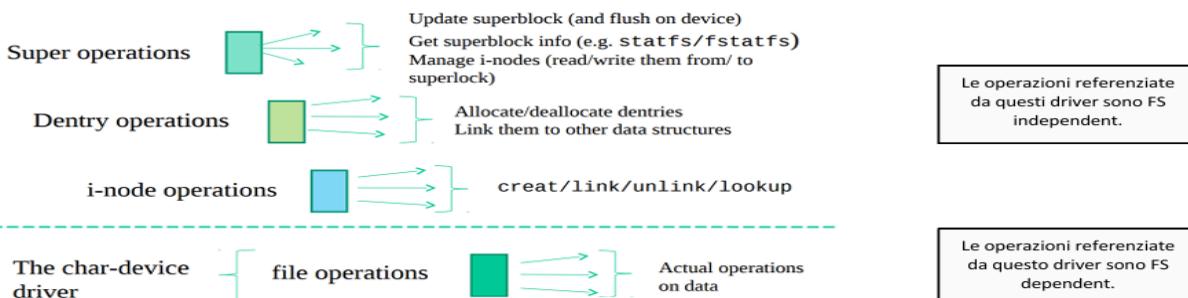
Moving to another mount namespace makes `mount/unmount` operations only acting on the current namespace (except if the mount operation is tagged with SHARED)

Nella figura qui sopra, 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 namespace. 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 lavoranti 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 apparirà a un nuovo namespace, per cui ciò che succederà col nuovo thread non impatterà il namespace corrente.

## Lez.34 (14/12/23)

### Visione del file system:

Come possiamo vedere ciascuna struttura dati ha associato un driver con le relative operazioni. Addirittura possiamo individuare un driver per le operazioni su file e uno per le operazioni su directory. Nel caso degli inodes l'operazione `lookup` è molto importante (chiede al vfs di associare un nome all'inode, potrebbe richiedere anche l'uso dell'inode parent per risolvere dipendenze a catena).



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.

### **TCB vs 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, infatti è condivisibile tra più threads.
- `altroot` indica altro root nella gerarchia, normalmente per fare operazioni particolari.

Nella versione più recente, 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`.

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

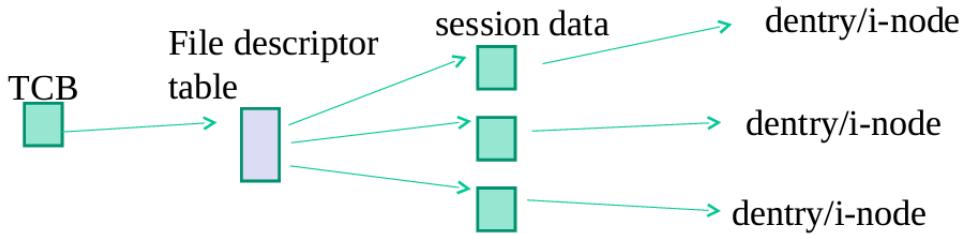
Dal kernel 4.8 si aggiunge `randomize_layout`.

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

### **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. È un array di puntatori a `struct_file`. 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 il codice canale IO, e poi si chiama il driver, puntato da i-node. 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.



TCB mantiene un campo `struct files_struct *files` che punta alla File Descriptor Table. Vediamo come è fatta `struct files_struct *files`:

```

struct files_struct {
    atomic_t count;
    rwlock_t file_lock; /* Protects all the below
    Nests           inside tsk->alloc_lock */                                members.

    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];
};

  
```

Tra i campi abbiamo `next_fd` che indica il primo elemento libero nell'array, `open_fs` che indica le entries libere dell'array, `fd_array` che è l'array di puntatori. I dati della sessione sono rappresentati dalla seguente struct

#### 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) perché 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;
};
  
```

Nelle versioni successive, si è evoluta, introducendo `__randomize_layout` e alcuni campi sono stati inclusi in altre strutture.

```

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

### ***Linux VFS api layering:***

In VFS possiamo operare con api assolutamente differenziate, alcune sono chiamate di systemcall, altre vengono eseguite da demoni senza che vengano chiamate esplicitamente system call.

Abbiamo syscall VFS layer che permette di fare il setup della sessione e il retrieve del channel id.

Poi abbiamo path based VFS layer, che permette di agire sul file system basandosi su un path, passato come parametro. Anche le systemcall usano il path, quindi anche il syscall layer lo sfruttano.

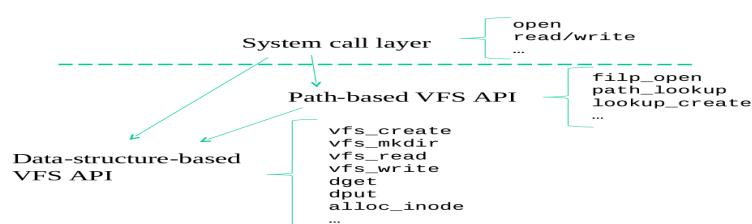
Data structure based VFS layer: è un layer per lavorare su strutture dati passando direttamente il puntatore alla struttura dati, ad esempio la struct file.

`filp_open`: permette di aprire una sessione a livello del file system. Ovviamente la sessione deve arrivare sul target, un oggetto con un certo nome. La sessione va agganciata ai file descriptors, se guardiamo l'implementazione effettiva della open sono poche righe in cui viene chiamata la `filp_open`, che va a chiamare tutto ciò che serve per arrivare al target che ci serve.

Con la read e write non usiamo path, ma vengono passati direttamente codici numerici.

`vfs_read` e `vfs_write` è il supporto reale a operazioni di read/write a livello di system call

Altre due operazioni interessanti sono la `dget` e la `dput`; la prima prende una entry, la seconda la rilascia; si utilizza un contatore atomico.

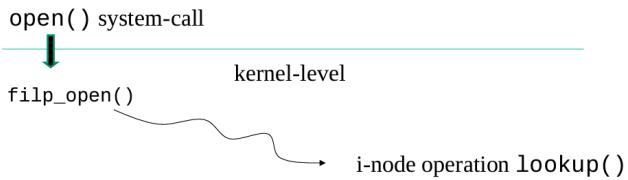


### **Path based API:**

`filp_open` tipicamente prende un puntatore ad una stringa che rappresenta un percorso nome, l'oggetto finale (target) di questo percorso nome è da aprire; su un percorso nome con più oggetti, per arrivare su uno dobbiamo necessariamente avere in memoria per il VFS una dentry per ciascun oggetto.

Se abbiamo la dentry abbiamo anche l'inode; bisogna essere in grado di fare il lookup per un certo inode. Tutto ciò viene usato a cascata.

## Esempio:



In the end we pass through dentry/i-node/char-dev/superblock drivers

## Data structure based:

`vfs_mkdir` per creare una directory passandogli una serie di informazioni, tra cui il parent.

Una `vfs_read` e `vfs_write` prendono il puntatore alla sessione e da quel puntatore fanno il retrieve del puntatore alla file operation read o alla file operation write da usare.

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
```

Creates an i-node and associates it with `dentry`. The parameter `dir` is used to point to a parent i-node from which basic information for the setup of the child is retrieved. `mode` specifies the access rights for the created object

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode)
```

Creates an i-node linked to the structure pointed by `dentry`, which is child of the i-node pointed by `dir`. The parameter `mode` corresponds to the value of the permission mask passed in input to the open system call. Returns 0 in case of success (it relies on the i-node-operation `create`)

```
static __inline__ struct dentry * dget(struct dentry *dentry)
```

Acquires a dentry (by incrementing the reference counter)

```
void dput(struct dentry *dentry)
```

Releases a dentry (this module relies on the dentry operation `d_delete`)

```
ssize_t vfs_read(struct file *file, char __user *buf,
```

`size_t count, loff_t *pos)`

```
ssize_t vfs_write(struct file *file, char __user *buf,
```

`size_t count, loff_t *pos)`

```

graph TD
    A[vfs_read()] --> B[file operation read()]
    A[vfs_write()] --> C[file operation write()]
    style B fill:#ffffcc
    style C fill:#ffffcc
  
```

In the end we traverse dentry/i-node structures to retrieve the file operations table associated with that dentry

## Lez.35 (18/12/23)

### **Relazione tra oggetti di I/O e drivers:**

Gli oggetti I/O sono rappresentati nei file system come `inode`.

Ogni inode può essere uno stream device o un block device.

Dal momento che i nodi non sono oggetti uniformi, ciascun nodo ha il proprio driver associato (nel caso di file regolari il driver è quello che eredita dal parent). In particolare, dopo aver creato il nodo in memoria (in un FS) e aver specificato di che tipologia è occorre linkarlo al corretto device driver. Tutti i device driver sono salvati in una tabella (in particolare ho una tabella per i block device driver e una per i stream device driver). Per identificare il giusto driver all'interno della tabella, ogni nodo ha il proprio **major number**. I nodi hanno anche un **minor number**: due nodi possono avere lo stesso major number (quindi lo stesso driver) ma diversi minor number (nel codice del driver posso fare operazioni diverse o usare strutture dati diverse a seconda del minor number, favorendo anche la concorrenza). Abbiamo detto che i nodi nel VFS vengono rappresentati tramite la struct `inode`. In particolare il campo `umode_t i_mode` tiene traccia del tipo di nodo. Solitamente i dispositivi che non sono file regolari vengono memorizzati nella directory `/dev`. Il campo `kdev_t i_rdev` mantiene le informazioni sui minor e major numbers. Questi sono salvati come una

maschera di bit per macchine x86, in particolare il major è il byte meno significativo mentre il minor è il secondo byte meno significativo. La macro MKDEV(*ma*,*mi*) è usata per creare la giusta maschera di bit. Usando le macro MAJOR e MINOR si recuperano tali numeri.

Quando lavoriamo su file system regolare tutto ciò non c'è, non ci interessa il major number, quando apriamo un oggetto di questo tipo vengono allocate le strutture per gestirlo e vengono portate a puntare ai drivers del file system, non c'è bisogno di rivolgersi a software esterno.

Il minor number permette di fare operazioni su più devices avendo isolamento delle operazioni.

Tutto ciò per quanto riguarda i block devices è una tabella di file\_operations.

### *mknod()*

Per creare un nodo generico si utilizza la funzione

`int mknod(const char *pathname, mode_t mode, dev_t dev)`. Il parametro mode: specifica i permessi che possono essere usati e il tipo di nodo. Per indicare il tipo di nodo si possono usare le macro S\_IFREG (file generico), S\_IFCHR (char device), S\_IFBLK (block device), S\_IFIFO. Se si usano la seconda e terza macro il parametro "dev" deve specificare il minor e major numbers (per le altre macro il parametro non viene usato).

Le funzioni del driver prendono tra i parametri di input la struct file. In questo modo accedendo alla sessione, alla dentry e all'inode possiamo recuperare il minor.

### **Char devices table:**

Gli elementi della char device table sono rappresentati tramite la seguente struct:

```
struct device_struct {
    const char * name;           ← Device name
    struct file_operations * fops; ← Device operations
};

static struct device_struct chrdevs[MAX_CHRDEV];
```

Per registrare un driver si usa la funzione:

`int register_chrdev(unsigned int major, const char * name, struct file_operations *fops).`

Se come major passiamo 0, il major viene scelto dal kernel e ritornato. Per rilasciare la entry al displacement major si usa la funzione `int unregister_chrdev(unsigned int major, const char * name)`, si fa un check sia sul major sia sul nome. MAX\_CHRDEV è il numero massimo di driver.

### *Evoluzione nel kernel 3:*

Dal kernel 3 in poi si usa una tabella hash (in questo modo si possono aggiungere anche più driver di quelli indicati come numero massimo grazie alle liste di trabocco)

```
#define CHRDEV_MAJOR_HASH_SIZE 255
static struct char_device_struct {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor; ← Minor number ranges
    int minorct;           ← already indicated and
    char name[64];          flushed to the cdev table
    struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Pointer to file-operations is here

major e baseminor creano un range di minor accettabili. Questi range evitano che il driver debba fare dei check. Infatti è il software del file system che guarda questi campi quando collega un inode al driver.

Le operazioni di aggiunta e rimozione del nodo vengono rimappate nel seguente modo:

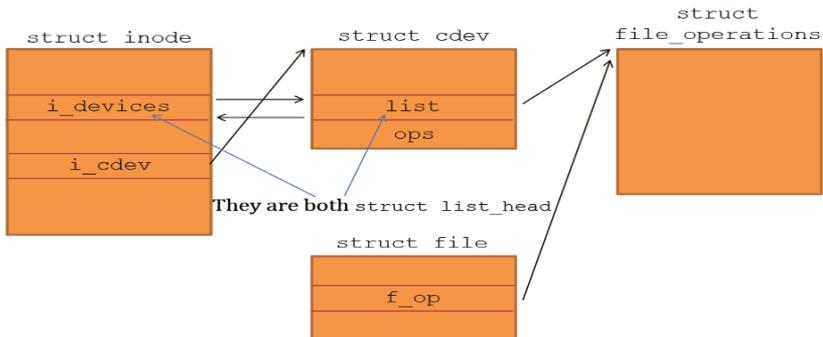
```
int register_chrdev(unsigned int major, const char
 *name, struct file_operations *fops)
```

New features → ~~int \_\_register\_chrdev(unsigned int major, unsigned
int baseminor, unsigned int count, const char
\*name, const struct file\_operations \*fops)~~

```
int unregister_chrdev(unsigned int major, const char *name)
```

void \_\_unregister\_chrdev(unsigned int major, unsigned
int baseminor, unsigned int count, const char \*name)

Lo schema per passare da inode alle file operation è il seguente:



## ***Finalizzazione del boot Linux:***

La funzione `rest_init()` è l'ultima invocata quando si chiama `"start_kernel()"`.

La prima cosa che fa è lanciare la creazione un kernel thread, che gira la funzione del kernel init, la quale finalizza il boot di Linux: lancia il file system root e lancia i programmi basici.

```
static void rest_init(void)
{
    kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
    unlock_kernel();
    current->need_resched = 1;
    cpu_idle();
}
```

## Dal kernel 3 in poj:

L'unica cosa che cambia è che è leggermente più complessa, perché siamo in grado di gestire la memoria NUMA, ad esempio “ `numa default policy`” fa sì che l'allocazione avvenga sul nodo NUMA locale.

```
static __attribute__((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     numa_default_policy();
408 }
```

Switch off round-robin to first-touch

### La funzione init():

init() si occupa di montare il *root filesystem* e attivare *INIT process*.

Chiama "do\_basic\_setup()", che registra i drivers, li prende montando i moduli presi dal RAM disk; questa zona di codice lavora sul RAM disk. Da qui in poi andiamo a lavorare sul filesystem di root.

Viene chiamata "prepare\_namespace()" la quale crea le directories /dev e /root sul filesystem rand.

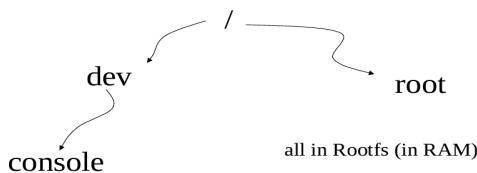
Sono separate perché su una metto i nodi e sull'altra monto il filesystem. Poi monto il filesystem.

```
static int init(void * unused){
    struct files_struct *files;
    lock_kernel();
    do_basic_setup(); ←
    prepare_namespace();                                registering drivers

    .....
    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.");
}
```

### La funzione mount\_root()

Struttura prima di chiamare mount\_root():



Viene invocata "mount\_root()", crea /dev/root, è un dispositivo (quindi un nodo), che verrà associato ad un certo major number; questo dispositivo è quello associato alla partizione dove è presente il filesystem di root che dobbiamo montare.

In create\_dev() abbiamo mknod per creare un nodo con un certo nome e specifichiamo dei parametri; poi si chiama la mount (mount\_block\_root); montiamo la sorgente /dev/root per popolare tutto in memoria; è interessante notare che viene detta solo la sorgente, la destinazione di default è quella cartella.

Non si specifica di che tipo è il dispositivo, esso è funzione dei parametri che abbiamo passato al kernel sulla riga di comando. Composta da queste due funzioni:

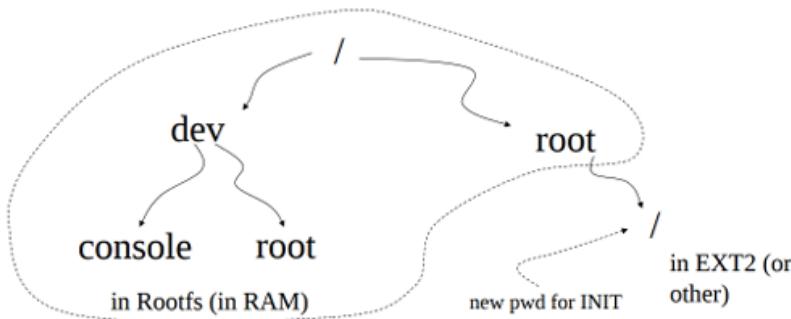
```
static void __init mount_root(void) {
    .....
    create_dev("/dev/root", ROOT_DEV,
              root_device_name);
    .....
    mount_block_root("/dev/root", root_mountflags);
}

static int __init create_dev(char *name, kdev_t dev,
                           char *devfs_name) {
    void *handle;
    char path[64];
    int n;

    sys_unlink(name);
    if (!do_devfs)
        return sys_mknod(name, S_IFBLK|0600,
                         kdev_t_to_nr(dev));
    .....
}
```

### La funzione init mount block root:

Tale funzione scorre gli array dei FS gestibili e cerca di montarli su /root. Se non trovo il driver per quel filesystem vado in panic(). Alla fine dell'esecuzione di mount root lo stato è il seguente



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.

Proprio come anticipato, il filesystem in RAM serve come punto di montaggio per il filesystem effettivo.

### Systemcall mount:

Abbiamo sorgente, target, tipo di filesystem, flag per il montaggio, puntatore ad una stringa con i parametri di interesse del driver separati da virgole.

```
int mount(const char *source, const char *target, const char *filesystemtype,
          unsigned long mountflags, const void *data);

MS_NOEXEC  Do not allow programs to be executed from this file system.
MS_NOSUID  Do not honour set-UID and set-GID bits when executing programs from this
file system.
MS_RDONLY  Mount file system read-only.
MS_REMOUNT  Remount an existing mount. This allows you to change the mountflags
and data of an existing mount without having to unmount and remount the file system.
source and target should be the same value specified in the initial mount() call;
filesystem type is ignored.
MS_SYNCHRONOUS Make writes on this file system synchronous (as though the O_SYNC
flag to open(2) was specified for all file opens to this file system).
```

Lo schema di montaggio è il seguente:

- 1) Il device che deve essere montato viene aperto e si carica il superblocco.
- 2) La funzione di lettura del superblocco è identificata per mezzo del file system type del device da montare.
- 3) La funzione di lettura del superblocco verifica che il superblocco è conforme con quello del FS di appartenenza.
- 4) Le 4 strutture dati vengono allocate. Il "mount point" è la directory indicata come punto di montaggio.

### Systemcall open:

- 1) Si ottiene il file descriptor (current->files->fd).
- 2) Si ottiene la dentry tramite `filp_open()` che interamente chiama la `open()` delle file operations (in tale open è possibile mettere l'acquisizione ad un lock, in modo da utilizzare il modulo per una singola istanza del device).

### Systemcall close:

- 1) Rilasciare la dentry tramite il fd e `filp_close()` che internamente chiama la `close()` delle file operations.
- 2) Rilasciare il file descriptor.

### Systemcall read/write:

- 1) Ottenerne la reference alla dentry tramite il fd (`fd->sessione->dentry`).
- 2) Ottenerne la reference alle file operations (driver).
- 3) Chiamare la relativa funzione nelle `file_operations`.

## Lez.36 (20/12/23)

Una delle cose che abbiamo visto è che un file system non ci permette di gestire solo operazioni di I/O come operazioni su stream di bytes, in particolare su Linux ci sono dati due filesystem che permettono di scambiare informazioni con il kernel: *proc* e *sys*.

### **Filesystem proc:**

Il primo a nascere tra i due. Ha un'architettura tradizionale; "sys" più moderna.

"*proc*" ci dà informazioni su programmi attivi, contenuti di memoria, settaggio del kernel,...

Nella directory in cui "*proc*" viene montato possiamo vedere directories e file di "*proc*".

Per ogni processo creato viene creata una nuova directory con "*proc*", targata con il pid del processo.

Partendo da un *thread control block* possiamo risalire a strutture collegate nel kernel che danno informazioni per il processo.

- **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

La funzione "*proc\_root\_init*" istanzia questo filesystem in memoria. La funzione di lettura del superblocco di questo oggetto non lo istanzia in memoria, ma lo aggancia a qualcosa di già esistente.

Ciò che viene creato è non solo la root, ma anche le sottodirectories.

Il montaggio avviene dopo il montaggio del filesystem applicativo, in cascata.

### Struttura e dati proc\_dir\_entry:

La rappresentazione di un file in memoria prevede di usare anche questa tabella, quindi la *proc\_dir\_entry* deve essere collegata in memoria all'inode ed alla dentry.

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;  
};
```

### *Release più recenti:*

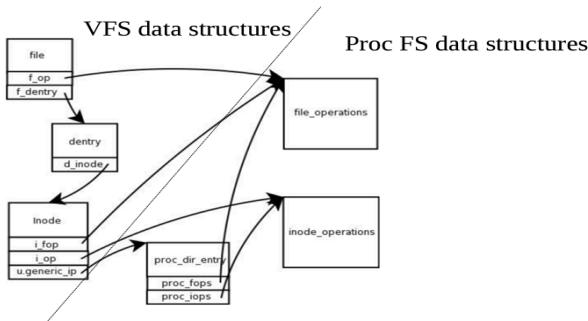
Nelle versioni più recenti non vi è più la tabella *file\_operation* ma solo quella *proc\_ops* (ha delle operazioni in meno). Vi è anche un puntatore diretto alla *write* (prioritaria). Se la tabella *proc\_ops* è NULL, per le operazioni si utilizzano le operazioni di default (e.s. apertura, lettura, ecc...).

```

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

```

### Layout delle strutture dati e montaggio di proc:



Ci sono pointers addizionali per relazionare aree di memoria ad un inode, possono essere informazioni addizionali per l'inode, ma possono essere anche dati. Su *proc* c'è un'API per creare questo oggetto e specificare cosa deve andare lì dentro.

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.

"*proc*" non viene montato subito, viene montato dalla init, che guarda in /etc/fstab.

In releases più recenti la init non guarda in /etc/fstab, ma usa informazioni precodificate al suo interno a seconda delle attività svolte.

### API:

```

struct proc_dir_entry *proc_mkdir(const char *name,
                                  struct proc_dir_entry *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)

```

Moved to struct proc\_ops

*proc\_mkdir* permette di creare una directory su *proc*, passiamo il nome ed il parent.

*proc\_create* serve per creare un file, bisogna indicare il driver per gestirlo, quindi si passa il puntatore alle file operations, su release moderne il puntatore alle proc operations.

```

static inline struct proc_dir_entry
*proc_create_data(const char *name, umode_t mode,
                  struct proc_dir_entry *parent,
                  const struct proc_ops *proc_fops,
                  void* data)

```



Get directly to some data  
via this pointer

Per sfruttare campi addizionali, anziché la `proc_create` si può usare la `proc_create_data`, grazie a cui possiamo collegare il driver ad aree di memoria con informazioni addizionali.  
L'ultimo puntatore è quello all'area di memoria addizionale; questa informazione viene riportata nell'inode.

La `read` e la `write` nelle proc operations hanno stessa segnatura delle file operations:

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *)
ssize_t (*write) (struct file *, const char *, size_t,
loff_t *);
```

La procedura di scrittura non prevede l'uso di un offset, non ci interessa perché con una scrittura diretta stiamo solo prendendo una certa quantità di dati per modificare qualche struttura nel kernel.

### **File system sys:**

È un file system disponibile a partire dal kernel 2.6 che è simile a proc ma si basa su strutture dati differenti. In particolare le directories sono kernel object e i files regolari sono *object attributes*. I kernel object sono strutture dati che giocano un ruolo molto ampio all'interno del kernel Linux. In particolare permettono di identificare un'entità a sé stante, un gruppo di oggetti, una tipologia di oggetti, gerarchie e associano oggetti della stessa tipologia. Ciascun ko ha delle facility interne per potersi relazionare con altri ko. I gruppi possono mantenere dei metadati che sono applicabili a tutti i ko che ne fanno parte

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.

### Struttura di un kernel object:

È un'entità che permette di:

- Identificare oggetti individuali o 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

`entry` viene usato per relazionarlo ad altri ko, `kset` identifica il gruppo di appartenenza, `ktype` punta ad una struttura che esplicita la tipologia del ko.

In particolare, la struct `kobj_type` descrive l'aspetto operativo del kernel object ed è fatta così:

```
struct kobj_type{
    void (*release)(struct kobject*);
    struct sysfs_ops *sysfs_ops;
    struct attribute **defaultAttrs;
}
```

We can have  
multiple  
attributes

Called when reference counting reaches the  
value zero

Actual operations to be executed on the object

In kobj\_type possiamo mantenere un puntatore a delle sysfs\_ops, quindi possiamo puntare ad un driver, non necessariamente è lo stesso per tutti gli oggetti dello stesso gruppo.

Con default\_attrs puntiamo ad un array che permette di registrare tutti gli attributi di questo oggetto.

Questi attributi possono essere usati per rappresentare un file nella directory, le informazioni che caratterizzano il file le troviamo nel kobject; in questo modo possiamo semplificare strutturalmente ciò che abbiamo quando vengono svolte operazioni.

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.

Sono solo due operazioni, una per mostrare il contenuto, l'altra per aggiornare il contenuto.

Le operazioni tipiche fatte su sys sono queste due, poi per fare operazioni specifiche si useranno quelle descritte nell'inode.

Tra i parametri passiamo il kernel object, il puntatore alla struct attribute, perché la show si chiama su un file specifico, quindi si passa il puntatore all'elemento che caratterizza l'esistenza dello specifico file nella struttura; poi si passa il buffer.

Il massimo numero di dati che possono essere consegnati di default è una pagina, non è un parametro di questa funzione.

Sulla store invece la size viene indicata.

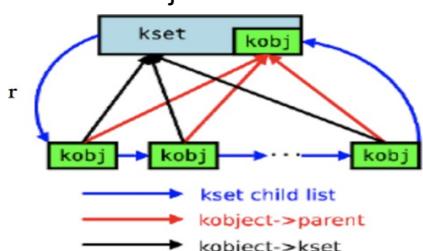
Il sys filesystem è fatto da oggetti piccoli, servono per chiedere o fornire informazioni al kernel.

### Cosa possiamo fare coi kernel object? kset

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:



Per quanto riguarda il file system linkage abbiamo che ogni oggetto può o meno essere associato a sys, un kobject è associato a sys se è associato ad un set.

#### 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)`: decrements the reference counter of a determined kset.
- `void kobject_set_name (struct kobject *kobj, char *name)`: assegna un nome a un determinato kernel object.

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. Per creare directories in /sys si usano le API (prendono in input ko prepopolati):

```
int sysfs_create_dir(struct kobject * k);
void sysfs_remove_dir(struct kobject * k);
int sysfs_rename_dir(struct kobject *, const char *new_name);
Main fields: parent - name
```

Creare una directory in sys significa agganciare un kobject ad una directory.  
Si può rinominare una directory.

#### API sui file:

Le operazioni su file sono gestite da questo secondo gruppo di API.

```
int sysfs_create_file(struct kobject *, const struct attribute *);
void sysfs_remove_file(struct kobject *, const struct attribute *);
int sysfs_update_file(struct kobject *, const struct attribute *);

struct attribute {
    char *name;
    struct module *owner;
    mode_t mode;
};
```

Minimal  
modifications along  
kernel releases

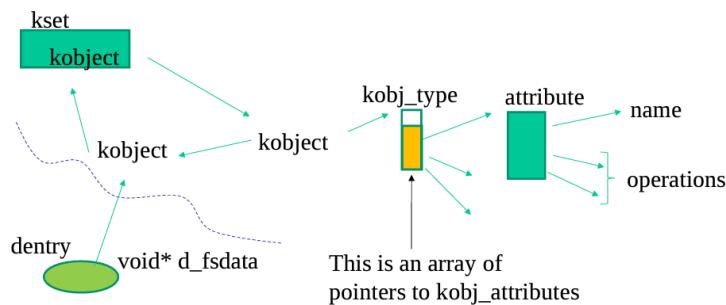
Il primo parametro deve indicare la directory cui appartiene il file.

La tabella di attributi deve dire come si chiama un file, quali sono i permessi di accesso ed eventualmente in che modulo sta il driver di gestione (tramite il campo owner).

Una delle cose che non abbiamo è specificare per un certo file un suo driver; è stata fatta un'espansione per associare a questi attributi un driver per gestire il contenuto dell'oggetto.  
Questi driver sono stati inseriti in una tabella che contiene gli attributi.

```
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 finale /sys:



Ogni directory ha il suo tipo, che identifica l'array dei puntatori agli attributi, al cui interno troviamo le operazioni, quindi i driver specifici.

Ciò che qui non è presente è il discorso relativo a major e minor number, che permettono di relazionare un elemento nel file system con un driver nel kernel.

Si può chiamare un'API per usare un driver associato ad un certo minor number: `device_create`; a cui va passata una classe, che è creata con l'API `class_create`.

```
static struct class* class_create(struct module* owner, char*
    class_name)

static struct class* device_create(static struct class* the_class, ...
    kdev_t i_rdev, ... char* name)
```

La classe identifica una classe di oggetti.

A livello di sys vengono create e allocate strutture che permettono di arrivare fino alla tabella dei driver associati ai major number di Linux.

## Software security aspects:

Alcuni concetti chiave:

- *ROP: return oriented programming*; è il nuovo standard di attacco. Ha bisogno a sua volta di innalzamento dei livelli di protezione: RM, reference monitor. Un SO orientato alla sicurezza deve essere basato sui reference monitor. In RM il root non è amministratore.

Baseline della IT security:

1. Il sistema va usato solo da utenti “legittimi”; se così non fosse il sistema sarebbe inutile
2. L'accesso è garantito sulla base di un'autorizzazione fornita da un amministratore. Come si fornisce questa autorizzazione? Abbiamo bisogno di due amministratori, un amministratore del sistema ed un amministratore della logica con cui il sistema deve operare.
3. 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).

### **DOS:**

Il DoS si basa su flooding di connessioni TCP, pacchetti (e.g. UDP) e richieste (protocolli applicativi). Posso avere un trade off rispetto alle richieste che si devono accettare; questo prescinde dalla costruzione del sistema, bisogna targare le richieste rispetto al fatto che siano o meno “buone”. Il metodo che vogliamo usare per fare questo va implementato in modo scalabile (multi-core, NUMA, algoritmi non bloccanti).

### **Legittimità:**

Cosa possiamo fare di non legittimo? Accedere a risorse importanti: dati e porzioni di codice, non risorse hardware.

Leggere dati che non dovremmo leggere ed eseguire codice, quindi eseguire qualcosa che non dovrebbe essere fatto in un sistema. L'utilizzo illegittimo di codice ha in realtà sempre a che vedere con utilizzo illegittimo di dati.

Come si fa ad avere un utilizzo illegittimo dei dati che non dipende da un utilizzo illegittimo del codice? Prendiamo ad esempio i side effect di meltdown, non si usa codice in maniera illegittima, ma si sfrutta ciò che il software fa in maniera speculativa.

### **Approcci di sicurezza:**

1. Crittografia
2. Autorizzazione/abilitazione
3. Security-enhance OS.

Tutto questo si usa in maniera combinata.

### Accessi illegittimi ai dati:

- side channel
- branch miss-prediction (varianti Spectre)
- speculazione sul path di esecuzione di trap (Meltdown)
- speculazione sul TAG-to-value (LT1 Terminal)
- hacking delle strutture kernel (syscalls, VFS)

### Contromisure:

- Gli accessi illegittimi possono essere in lettura o in scrittura.  
Per quanto riguarda la lettura abbiamo la **randomizzazione**, a compile time o a runtime.
- **Signature inspection:** quando carichiamo porzioni di codice possiamo vedere signatures che non ci piacciono, ad esempio quelle che cambiano il modo di esercizio del processore, l'abbiamo usata per caricare la systemcall table.
- **Cifratura dei dati**, per blocchi di dispositivo, pagine, locazioni di memoria e dati generici nel file system. Questa viene fatta tramite l'API `char *crypt(const char *key, const char *settings)` in cui passo password e metodo di cifratura. Il metodo di cifratura ed il **salt** utilizzato sono salvati in file specifici. Nel file `/etc/passwd` sono salvate le password cifrate ed è accessibile da tutti gli utenti. `/etc/shadow` è confidenziale per l'utente root e contiene dati critici per l'autenticazione come le passwords.

### Accesso illegittimo al codice:

- Mis-speculation su branch o traps.
- Hacking del kernel.
- Hacking del modo operazionale dell'hardware.

### Contromisure:

Le stesse di prima, ma anche qualcosa in più:

- Correzioni esplicite dei valori che, ad esempio, servono per spiazzarci all'interno di strutture dati (vedi la sanitizzazione)
- impedire di montare moduli kernel (che possono sovvertire le misure messe in atto sopra).  
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. 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**. E' quindi possibile deviare un thread dal suo control flow graph per fargli inserire del codice inserito dall'esterno. Uno dei modi per farlo è tramite **buffer overflow**.

## Lez.37 (21/12/23)

### **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).

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. Stiamo deviando dal suo control flow graph, e ciò è permesso ad un thread se siamo nel momento in cui esegue approcci legati al buffer overflow.

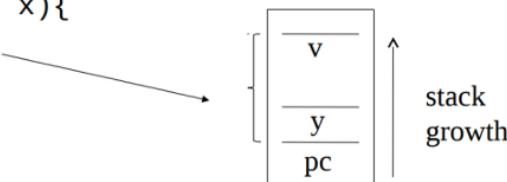
Consideriamo ad esempio lo stack frame associato a una certa funzione `f`. In fondo a tale stack frame si trova l'indirizzo di ritorno di `f`. Secondo un flusso di esecuzione legittimo, `f` dovrebbe poter manipolare sullo stack solo informazioni come i parametri e le variabili locali. Un attaccante può 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 salti verso una qualsiasi zona dell'address space.

Tipicamente il **software debole** prevede l'allocazione di un buffer sullo stack *senza controllo esplicito dei confini*. Ne sono esempi `scanf()` e `gets()`, le quali non fanno check sulla taglia della stringa passata. A questo punto un bug nell'uso dei puntatori può far sì che si vada a sovrascrivere il Program Counter, tornando il controllo ad una porzione arbitraria del codice originale o a codice iniettato. Questa tecnica è anche detta **stack exploit**.

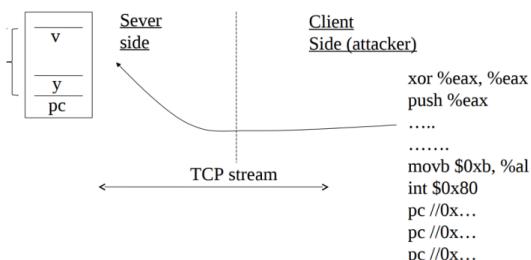
#### Esempio stack exploit:

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.

```
void do_work(int x){
    char v[SIZE];
    int y;
    ....
}
```



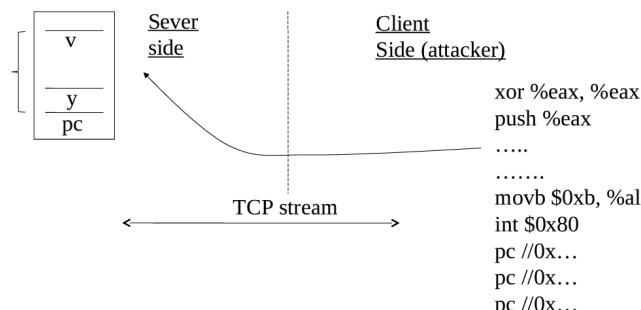
Se viene iniettato del codice target (quindi contenente delle istruzioni macchina) allora si parla di stack exploit with payload. In stack exploit con payload (realizzabile anche in remoto), l'indirizzo di ritorno che viene inserito nel program counter è relativo a una zona di codice iniettata. La funzione debole (e.g. `scanf()`) può sovrascrivere il PC. Oggi si usa `scanf()`, `scanf secure`; ma anche qui se specifico una size corretta ma il pointer all'area di memoria non è corretto posso andare in buffer overflow.



L'unico problema per l'attaccante è che la posizione dello stack è randomizzata, per cui è molto difficile indovinare l'indirizzo esatto del codice malevolo da inserire all'interno di PC.

La soluzione che si adotta consiste nell'aggiungere tante istruzioni 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).

Questo è un problema inherente ai bug nel software, se ci sono bug non siamo sicuri anche se usiamo funzioni sicure.

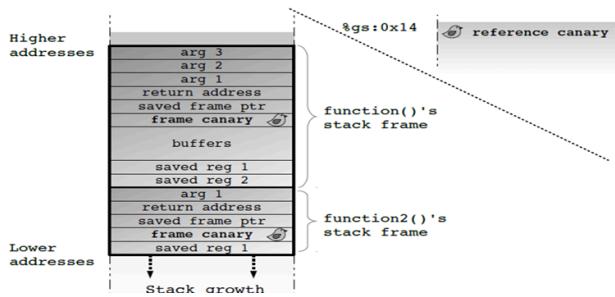


### Canary tag:

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.

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. Infatti è lo stesso valore per tutti i threads o processi figli di un tale processo, e possiamo sfruttare questo fatto per fare un brute force attack, almeno uno dei threads riuscirà a trovare il canary tag corretto.

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**.



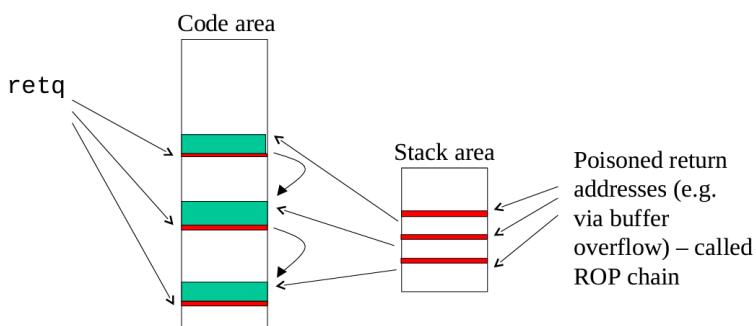
### Flag XD - Porzioni dell'AS eseguibili e non eseguibili:

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, perché marco le pagine come non eseguibili.** Per abilitare il fetch delle istruzioni si utilizza l'opzione `gcc -z exec_stack`.

Il codice è ancora vulnerabile se alloca aree di memoria sia scrivibili che eseguibili (come JVM). Tuttavia non siamo ancora sicuri. Infatti, per attivare un nuovo programma, basta eseguire poche istruzioni che potrebbero già essere presenti nell'eseguibile senza doverle eseguire. Questo ci porta alla tecnica **ROP**.

### Return Object Programming - ROP:

E' una tecnica di exploit che prevede di usare *più punti di ritorno sullo stack "avvelenati"*. Ciascun indirizzo di ritorno identifica un **gadget**, ossia un insieme di istruzioni già presenti all'interno del codice dell'applicazione o delle librerie usate, che termina con un'istruzione di return e compiono azioni "utili" come spostamento dei dati. *Sfruttando ad esempio buffer overflow, si forza una catena ROP*, dove ciascuna *return* del gadget andrà ad invocare il gadget successivo. Siccome ogni gadget consiste in poche istruzioni è possibile creare un flusso qualsiasi componendo le varie istruzioni.



Ovviamente è previsto che il return address venga "avvelenato" con le tecniche viste prima. Per trovare i gadget si può usare brute force. Dal momento che x86(-64) non impone l'allineamento di istruzioni in memoria, una singola zona di codice può essere usata in modi differenti da ROP.  
Ciò vuol dire che, iniziando a leggere da un byte piuttosto che un altro, posso avere interpretazioni totalmente differenti del codice.

### **Protezioni:**

#### Return Address Protection - RAP:

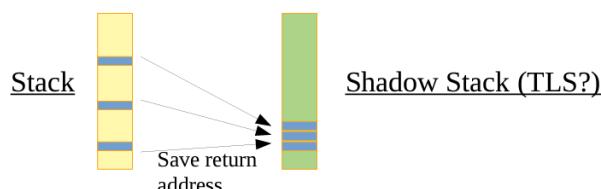
Ogni funzione viene compilata con un preambolo e una coda.

- Nel preambolo l'indirizzo di ritorno viene memorizzato in modo cifrato all'interno del registro rbx (al contrario dello stack i registri non possono essere sovrascritti), 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.

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 cifratura. Inoltre, si tratta di un meccanismo che ha un costo non minimale, soprattutto per le funzioni più brevi.

#### 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 (che può stare ovunque e potenzialmente essere sovrascrivibile). 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.

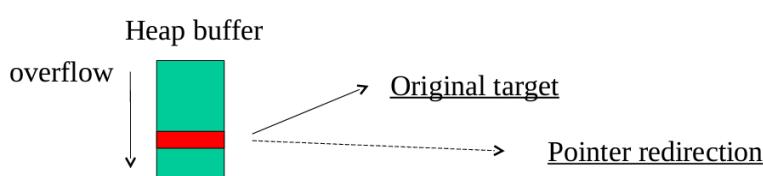


#### call/return branch predictor:

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 pezzo di codice esistente o anche del codice iniettato. Si può puntare a codice che va a fare l'exploit di dati sullo stack.



## **Contromisure per l'heap overflow:**

### 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. Questo check viene fatto dal thread stesso prima di fare la call. Tuttavia questa misura potrebbe non essere efficace nel caso di applicazioni multi-thread: infatti un thread potrebbe modificare il function pointer dopo il check ma prima della call di un altro thread.

### Sanitizzazione della memoria:

Quando usiamo `sanitize` in gcc, il software viene instrumentato per tracciare gli accessi in memoria, se si scrive fuori dalla zona che deve essere utilizzabile si può scoprire che sta accadendo qualcosa di non coerente. Ciò viene realizzato dal shadow 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. Tutto ciò costa moltissimo, si usa per fare analisi delle applicazioni.

### **Reale danno di un 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.

E' l'unico modo che abbiamo in un SO tradizionale per far sì che quando si esegue un'attività per un certo utente si possa poi cambiare livello di privilegio, anche temporaneamente.

## **User ID in Linux:**

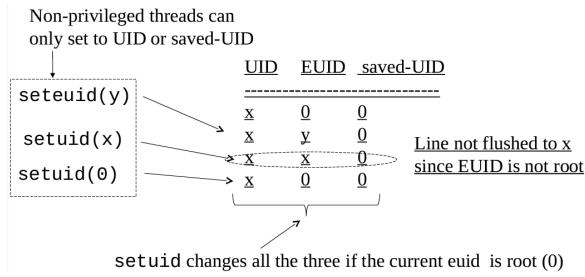
All'interno del kernel, lo username degli utenti è soltanto un placeholder. L'informazione che identifica veramente qual è lo user per conto del quale un programma sta girando è lo UID (user ID); analogamente, abbiamo il GID (group ID) per identificare i gruppi di utenti. Ciascun thread, in qualunque 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).

Tali targhe vengono ovviamente usate dal kernel per discriminare le capability di un thread (ad esempio se può eseguire una determinata istruzione).

Tra le API abbiamo `setuid()`; `seteuid()`; `getuid()` e `geteuid()`.

Se `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 read 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. Ecco un esempio:



### sudo e su:

sudo e su sono comandi setuid-root e, quando vengono utilizzati, richiedono all'utente di inserire una password.

- 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.

### **Linux capabilities:**

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 domini di protezione, grazie ai quali è possibile stabilire quali permessi assegnare ai vari thread.

Qui sotto riporto una rappresentazione di ciò che avviene:

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

Esiste una maschera di 32 o 64 bit che viene usata per determinare se un thread ha una qualche capability.

Ci sono varie bit-mask:

- 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: limite per i set permitted/inherit
- Ambient capability: indicano cosa possiamo fare coi programmi non SUID

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 è difficile gestirle all'unisono.

Tutto ciò ci fa capire che stiamo in un contesto di "domini dinamici", ovvero la protezione cambia nel tempo.

### **File capabilities:**

Dato che i file sono programmi anch'essi hanno delle capabilities associate.

```
#include <sys/capability.h>
cap_t cap_get_file(const char *path_p);
int cap_set_file(const char *path_p, cap_t cap_p);
cap_t cap_get_fd(int fd);
int cap_set_fd(int fd, cap_t caps);

Usable if the file system is mounted without the NOSUID option
Also, the user needs to have the CAP_SETFCAP capability
available to set capabilities for files
```

## Sistemi operativi Sicuri

Ci poniamo il seguente problema: come facciamo a evitare che si abbia un uso non legittimo da parte di un thread che esegue come root? 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. Così un attaccante ha minori possibilità di far danno, ad esempio, in termini di accesso o manipolazione dei dati. Un esempio di sistema operativo sicuro nel mondo Linux è SELinux (Secure Linux). Un sistema operativo sicuro fa affidamento ai *domini di protezione*.

### 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 entrambi), allora non può essere in alcun modo acceduta da quell'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. Di conseguenza, istanze differenti dello stesso programma possono essere associate a protection domain diversi. Quindi la riduzione di privilegio di alcune istanze non compromette le altre istanze.

### Security policy

I protection domain da soli non sono sufficienti per rendere sicuro un sistema operativo (infatti è possibile acquisire dinamicamente capabilities e non solo perderle). 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.
- 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). In questo caso non è possibile andare in escalation e acquisire privilegi.

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 il root user 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 sincronizza ACL con il sistema esterno può essere eliminato? Solo montando un modulo, ma per farlo serve thread rootId, ma se tale thread non può fare questa operazione (poiché 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.



La riconfigurazione runtime dei domini non si fa entrando nel sistema, ma viene fatta sul sistema esterno ed i due sistemi, quello che realmente lavora e quello con le ACL riconfigurate si parlano tra di loro.

### **Reference monitor:**

Servono a rinforzare i domini di protezione. Operano a kernel level nel contesto di un sistema operativo sicuro ma possono essere implementati anche in altri layer come i databases. Tipicamente questi moduli supervisionano l'esecuzione di una system call individuale. In particolare, quando viene invocata una system call, questa passa per il reference monitor, il quale consulta l'ACD (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. Notiamo che, a differenza dei domini di protezione, l'ACD non può essere modificato dai thread ma solitamente viene configurato dall'amministratore di sistema.

La configurazione non è cambiabile, quindi non posso toccare le policy di sicurezza.

