Compito del 17/06/2019

Domanda 1: L'algoritmo di Robert Tomasulo per la realizzazione di una pipeline Out Of Order (OOO) si prefigge di risolvere 3 conflitti possibili in un'architettura di tale tipo: date due istruzioni A e B, dove A $\rightarrow$ B nel program order, possono esserci:

- RAW: B legge un dato prima che A abbia scritto lo stesso dato;
- WAW: B scrive un dato prima che A abbia scritto lo stesso dato;
- WAR: B scrive un dato prima che A abbia letto lo stesso dato;

Per poter risolvere i conflitti RAW, è necessario tenere conto di quanto viene prodotto il dato da leggere e quindi occorre mettere in un buffer l'operazione di lettura finché tale dato non è stato prodotto.
Per risolvere i conflitti WAW e WAR, il supporto hardware necessario sono i renamed registers: tale soluzione architetturale prevede che quando si effettua un'operazione di scrittura su un registro, in realtà si vada ad operare su un multi-registro, quindi composto da diversi slot a cui sono associati dei TAG.
Quindi, un'operazione di scrittura non andrà mai a sovrascrivere dei valori precedenti poiché agisce suo uno degli slot, ed una volta che le istruzioni andranno ni commit, verrà ripristinato l'ordine dei valori scritti sul multi-registro.
Si potrà sempre operare sul registro renamed oppure su quello attuale, dove il renamed avrà:

- una storia passata;
- l'ultimo TAG standing che viene esposto per le operazioni di lettura
- una storia futura;

ogni scrittura genera un nuovo TAG standing, che verranno poi riconciliati nel tempo a seguito del retirement delle operazioni.
Sono così risolti i conflitti WAW, ma anche quelli WAR per cui in aggiunta vengono introdotte le reservation stations: tali "stazioni" sono dei buffer posti in corrispondenza dei componenti che possono svolgere una operazione richiesta (ad esempio le ALU) al cui interno viene salvato l'OP code dell'operazione da eseguire.
Inoltre, saranno indicate anche la coppia Q_j, Q_k di reservation stations che produrranno l'output necessario all'operazione o altrimenti i valori V_j, V_k da leggere dai renamed registers corretti; d'altronde, i registri saranno marcati con la reservation station che produrrà il risultato necessario, se esiste.
Tale componente permette dunque di salvare in buffer le operazioni, inoltre l'architettura hardware è composta anche da:

- Common Data Bus: un bus comune dove poter far fluire i dati necessari come input agli altri componenti
- ReOrder Buffer: componente che mantiene i metadati delle istruzioni non ancora committate. Tale componente può essere organizzato in entry oppure referenziare l'alias di registro necessario. Raccoglie inoltre i valori prodotti dalle nuove istruzioni e li mantiene uncommitted, può essere usato se serve dalle istruzioni che devono leggere tali valori uncommitted.

Domanda 2: Gli algoritmi di scheduling nel kernel hanno subito molte evoluzioni nel corso delle diverse release del kernel stesso: la prima versione era il "perfect load sharing", per poi passare al "load balancing" fino al "completely fair".
La prima differenza fra i 3 sta nel costo algoritmico, che è passato da O(n) per il load sharing ad O(1) per il load balancing per poi essere O(log(N)) nel caso peggiore per il completely fair.
Nel 1° dei 3 algoritmi l'assegnazione della CPU veniva effettuata in base ad una formula, che considerava la priorità statica associata ad ogni TCB dei thread, inoltre si consideravano anche dei fattori aggiuntivi. Il valore finale che determinava quale thread dovesse prendere la CPU a seguito di un context switch era dato dalla

funzione "goodness", che restituiva il peso di tale thread, nel calcolo di tale "goodness" si poteva aggiungere un +1 se il thread che si stava correntemente valutando come candidato condivideva l'address space con l'ultimo thread che aveva preso la CPU ed un +15 (su macchine SMP) se era proprio l'ultimo thread ad avere preso la CPU.
In ogni caso, se i ticks residui del thread erano pari a 0, il valore restituito dalla goodness era per default 0.
Nel caso del load balancing invece non si applica più una formula, ma si dividono i thread attivi, ovvero con numero di ticks residui > 0, in 140 code corrispondenti a 140 diversi livelli di priorità dove da 0 a 99 ci sono i livelli real-time, mentre da 100 a 139 i livello "other".
Vengono poi mantenuti due gruppi, uno con le code contenenti solo i thread che hanno ancora dei ticks da poter spendere in CPU ed una con i thread ne hanno 0 per l'epoca corrente, evitando così di analizzare anche i thread che non possono più prendere la CPU come avveniva nel 1° algoritmo.
All'interno della singola coda, per avere comunque discriminazione fra le priorità dei diversi thread, si utilizza un array di pesi, composto da 40 elementi.
Inoltre, grazie alla presenza dei due gruppi di code, è possibile riassegnare subito i ticks per l'epoca successiva ad un thread che ha finito quelli per l'epoca corrente, in quanto starà in una coda che non verrà correntemente analizzata.
L'algoritmo ha un costo O(1) in quanto viene preso il primo TCB trovato nella coda non vuota a priorità più alta, come indicato da un'apposita bitmap.

Nel terzo algoritmo, invece, per le priorità "other" i thread sono mantenuti in un red-black tree, ordinato in base alla virtual CPU (vCPU): tale vCPU lega la priorità statica del thread all'utilizzo di CPU effettivo, così da garantire fairness nello scheduling dei thread other, in quanto viene sempre scelto quello con il valore minore di vCPU.
Questo risolve un problema del 2° scheduler, dove invece veniva associata ai thread la priorità dinamica: se un thread si "comportava bene", la sua priorità dinamica aumentava e veniva portato in classi superiori, ma così facendo si rischiava di schedulare un thread di questo tipo prima di uno più prioritario, in quanto avente priorità dinamica più alta nonostante quella statica fosse più bassa.
Per i thread real time invece, nel 3° algoritmo, viene mantenuta la stessa soluzione adottata dal 2° algoritmo.
Ulteriore differenza fra i 3 scheduler è che nel primo non era possibile eseguire l'algoritmo in parallelo sui diversi CPU core: guardando alla stessa runqueue, era necessario prendere un lock per evitare che due CPU schedulassero lo stesso thread.
Con gli algoritmi 2 e 3 le runqueue sono per CPU, avendo inoltre la garanzia che la probabilità di collisione sulla stessa coda sia molto bassa.

Domanda 3: La prima tipologia di attacco buffer overflow è il così detto "stack exploit": in tale attacco, supponiamo di considerare un'applicazione in cui si utilizza un buffer di una certa taglia finita per ricevere dati da un utente, tramite funzioni come `scanf` o `gets`. Tale utilizzo, nel momento in cui l'utente passa dei dati la cui lunghezza eccede la taglia del buffer, può portare a sovrascrivere la memoria, portando il PC a fare fetch di istruzioni tali per cui il thread non è più compliant con il suo control flow graph.
Tali istruzioni vengono passate dall'attaccante come payload e possono portare il programma vittima ad eseguire qualunque comando.
Le contro-muisure possibili sono diverse:

- Le classiche contro-misure di sicurezza nel software:
  - randomizzazione dell'address space: vengono randomizzate le posizioni in memoria di strutture dati e istruzioni.
    Questo renderebbe complesso lo stack exploit, in quanto occorre saltare alla prima istruzione del payload la cui posizione può essere randomizzata, ma è possibile inserire delle NOP come preambolo, per far si che si ricada su una di esse e si porti il processore poi ad eseguire il resto del payload;

  - signature ispection: si verifica, ad esempio all'atto del montaggio di un modulo del kernel, se vi sono istruzioni che possono essere usate in maniera malevola, così che si proceda allo smontaggio del modulo;
  - cifratura, ad esempio dell'address space, in modo che le pagine una volta accedute vengano decifrate per accedere al contenuto reale.

- in x86 long mode, è possibile specificare un bit di protezione, XD, per far si che le pagine della page table del processo non possano contenere delle istruzioni eseguibili dal processore

- i compilatori, a meno che non venga disabilitato, inseriscono per default nello stack dei "tag canarino" e nel momento in cui tale tag viene sovrascritto, l'applicazione termina.

Il secondo tipo di attacco è basato sulla Return Oriented Programming (ROP): in questo caso, lo scopo dell'attaccante è portare l'esecuzione in una particolare zona dell'address space in cui sono contentute delle istruzioni dette "gadget".
Tali gadget sono vari e sparsi nell'address space ma componendoli in modo che ogni gadget ritorni il controllo a quello successivo, costruendo così la ROP-chain, è possibile far si che venga ricostruito il vettore di attacco.
La contro-misura possibile prevede di usare delle shadow stack area, all'interno delle quali si salva l'indirizzo di ritorno della funzione, così che se nello stack reale questo valore viene modificato il check con la shadow stack non è valido.
Un'altra possibile contro-misura prevede di verficare eventuali miss-prediction nell'unità di predizione dei branch, che è però un controllo costoso da implementare.
Il terzo ed ultimo tipo di attacco è l'heap overflow, dove si va a modificare un function pointer per far si che esso punti verso una funzione voluta dall'attaccante.
Anche qui, la soluzione consiste nel fare memory sanitize, inserendo dei valori immutabili nella memoria. Tutte queste contro-misure non possono comunque risolvere la problematica legata ai bug sui puntatori, che possono portare ad esegire in zone arbitrarie dell'address space, rendendo impossibile l'effettivo controllo.

Compito del 16/09/2019

Domanda 1: La problematica della coerenza della cache fa riferimento a sistemi multi-core, in cui ciascuno dei core ha una cache di livello L1 privata. Per tale motivo, vengono violati i seguenti punti relativi alla coerenza dei valori:

1. se una CPU legge un certo dato X ed è l'ultima che ha effettuato una scrittura su X, allora il dato letto è l'ultimo scritto dalla CPU se nessun altro l'ha aggiornato

2. un'operazione di lettura da una CPU che segue la scrittura di un'altra CPU restituisce il valore aggiornato se la distanza temporale fra le due è adeguata

3. l'ordine con cui vengono viste le operazioni deve essere lo stesso per tutti i CPU core.

Adottando i classici approcci della gestione delle cache, non si riescono a garantire tutti e 3 i punti: se si suppone di avere 2 CPU, CPU_0 e CPU_1, che portano in cache uno stesso dato X:

- se la cache è write through, la CPU_0 aggiorna X, ponendo il valore ad 1 e scrive il valore in memoria. Quando la CPU leggerà X lo farà dalla sua cache privata, quindi il valore letto non è aggiornato

- se la cache è write back, la CPU_0 può scrivere il valore 1 su X, in seguito la CPU_1 può scrivere 2 su X. Se la CPU_1 riscrive il valore in memoria prima della CPU_0 è stato invertito l'ordine delle operazioni

Il problema è quindi sempre relativo al fatto che le CPU possono avere dei valori stale in cache, per risolvere tali problemi vengono usati dei protocolli per la gestione delle operazioni in cache: tali protocolli saranno caratterizzati dagli stati in cui è ammissibile che un blocco di cache sia, l'insieme di eventi ammissibili e di transizioni di stato possibili.
Si possono distinguere due famiglie di protocolli:

- invalidate protcols, dove chi scrive un dato rende invalide tutte le altre copie nell'architettura distribuita
- update protcols, dove ogni volta che viene effettuata una modifica, questa è propagata verso tutte le copie affinché siano sempre aggiornate.

Negli invalidate protocols è possibile distinguere protocolli che operano secondo un'architettura di snooping cache, come avviene ad esempio in processori x86: tale organizzazione prevede che vi sia un bus comune dove far transitare le richieste, su cui vi è la serializzazione, tale bus verrà gestito dal controller di ciascuna cache quando è necessario effettuare un'operazione che richiede un cambiamento di stato.
In questo protocollo, si parla per un'intera linea di cache, vi sono diverse varianti:

- MSI: tale protocollo prevede che gli stati ammissibili per la linea siano 3:
  - Modified: il dato è stato scritto
  - Invalid: la copia del dato non è valida
  - Shared: è stata ricevuta una copia da un reader o writer
- MESI: viene aggiunto in più lo stato Exclusive. Se ad esempio un program flow utilizza frequentemente un dato in cache, può richiedere mediante una apposita Request For Ownership sul bus (RFO) di prendere la linea di cache ad uso esclusivo, così da poter transitare nello stato Modified in caso di scrittura senza dover generare richieste sul bus.
- MOESI: si aggiunge lo stato Owner, tale per cui chi ha la linea di cache in tale stato non deve passare nuovamente per lo stato di Exclusive per effettuare una modifica, evitando così ulteriore spreco di banda. Inoltre, se si richiede una copia del dato all'Owner, esso può renderla ma potrà anche modificare il valore senza generare richieste sul bus, rendendo così la copia data non aggiornata.

Tale approccio di snooping cache è applicabile ad architetture piccole ma è poco scalabile, per cui si può considerare di adottare protocolli directory based: qui, l'interazione avviene solo con una directory centralizzata, che mantiene i metadati per i dati in cache.
Se ad esempio una CPU P ha un cache miss, passa per la directory per verificare a chi dover chiedere la copia del dato.
In questo modo, è possibile mantenere nel componente centralizzato chi ha una copia dirty del dato, così da sapere anche chi deve invalidarla nel caso fosse necessario.

Domanda 2: L'architettura di SoftIRQ è un'architettura software usata in Linux per la gestione degli interrupt secondo il paradigma del Top Bottom Half Programming (TBH).
Qui, nel momento in cui arriva un interrupt, il top half prevede che venga risvegliato un apposito demone di sistema, il SoftIRQ deamon, e viene flaggata ad 1 una la entry della SoftIRQ table corrispondente all'interrupt ricevuto.
Nel momento in cui il demone verrà schedulato in CPU, andrà a guardare nella tabella per verificare quale entry è stata portata ad 1 e quindi quale bottom half eseguire, dove il bottom half può essere stato precedentemente registrato nella tabella.
Questa architettura risolve diversi problemi associati alle task queue che si utilizzavano in versioni precedenti di Linux, in particolare:

- permette la gestione parallela degli interrupt, in quanto il SoftIRQ deamon risvegliato è preso da un pool di N thread, tanti quanti i core della macchina

- la gestione del bottom half è delegata ad un demone apposito e non più ad un thread generico come poteva avvenire con le task queue, dove era possibile rischiare di rallentare thread ad esempio real time in quanto essi dovevano svolgere tutto il lavoro indicato nella coda nel caso in cui venisse chiamata la API corrispondente.

- è inoltre possibile creare affinità fra il SoftIRQ deamon e la CPU facendo si che determinati task vengano eseguiti da determinati demoni, in base alle configurazioni di apposite bitmask

L'architettura non prevede di per se la possibilità di accodare del lavoro per il bottom half, cosa che viene resa possibile con le tasklet: tali strutture specificano delle code, sono presenti in due entry della SoftIRQ table e prevedono una coda ad alta priorità ed una a priorità normale. Quindi, quando viene flaggata la entry corrispondente ad una di esse, il SoftIRQ deamon scorre la coda ed esegue i staci registrati, la serie di lavori può essere inserita mediante apposite API.
Le teste delle due liste di tasklet sono definite come variabili per-CPU, così che ogni thread possa operare su di esse senza andare in conflitto con gli altri.
Le API permettono di creare nuove tasklet, anche in stato disabled: queste potranno successivamente, mediante ulteriori API, essere attivate per poter eseguire il lavoro specificato all'interno.
È importante quindi verificare eventuali dipendenze con delle tasklet in stato disabled, per evitare problemi dovuti ad esempio allo smontaggio di moduli del kernel prima che i task vengano eseguiti, inoltre importante che il bottom half specificato nelle tasklet non sia bloccante, o si renderebbe intrinsecamente bloccante anche il SoftIRQ deamon che le processa.

Domanda 3: Per la costruzione dei Sistemi Operativi orientati alla sicurezza, è importante considerare la definizione di due aspetti: i domini di protezione e le policy di sicurezza.
Un dominio di protezione è dato dalla coppia <risorsa, modalità di accesso> ed è specificabile per differenti risorse del sistema, come utenti o program.
Si potrà quindi avere un program di una certa utenza a cui è associato un dominio D, mentre ad un altro program della stessa utenza è possibile associare un dominio D' e tali domini possono cambiare nel tempo.
Questo concetto in Linux ha avuto una sola implementazione con le capabilities: queste permettono di avere dei thread i cui privilegi sono "nel mezzo" fra quelli di un thread di root e quelli di un utente non privilegiato. Infatti, ad un thread di root è tipicamente permesso tutto all'interno di un sistema, mentre per un thread utente vi sono i privilegi minimi, necessari affinché tale utente possa svolgere lo stretto necessario all'interno del sistema.
Con le capabilities, un thread bypassa determinati controlli di sicurezza del kernel, in Linux vengono definite:

- Permitted capabilities: quelle permesse al thread

- Effective capabilities: quelle che è il thread possiede correntemente

- Inherited capabilities: quelle che i child processes ereditano a seguito di una `exec`

Vi sono poi ulteriormente:

- Bounding capabilities: pongono dei limiti alle Permitted ed alle Inherited

- Ambient capabilities: specificano cosa è permesso ad un thread non-SUID

Le security policies definiscono invece a chi è permesso di poter manipolare i domini di protezione, vengono distinte:

- politiche discrezionarie: gli utenti del sistema, come root, possono modificare i domini di protezione

- politiche mandatorie: la definizione dei domini di protezione è a carico di un utente esterno, che non può loggarsi nel sistema

L'implementazione di una politica mandatoria si può ottenere con un oggetto come il reference monitor: tale modulo software viene caricato nel sistema ed interagisce con un database di Access Control, dove vi è la definizione dei domini di protezione per tutte le risorse del sistema.
Ad esempio, quando viene invocata una syscall, il reference monitor, sulla base di quanto contenuto nell'AC DB decide se permettere la chiamata oppure negarla e questo si può applicare anche su un thread con privilegi di root. Tale monitor, una volta installato nel sistema ed una volta definito il DB dei permessi, non è più modificabile nemmeno dall'esterno.

Compito del 24/01/2020

Domanda 1: Su processori x86, l'architettura hardware di trap ed interrupt prevede la possibilità di specificare 255 linee di interrupt, usate per differenti scopi. Vi è infatti l'architettura APIC, componente locale a ciascuno dei core del sistema, che permette l'invio dei messaggi di interrupt così come anche degli InterProcessor Interrupt (IPI) su un bus condiviso in broadcast.
Gli IPI sono una tipologia di interrupt sincroni per la CPU che invia ed asincroni per quella che li riceve e possono essere utilizzati per comunicare di effettuare diverse operazioni. Ad esempio, in Linux, sono usati per:

- comunicare ad una CPU di dover flushare le entry del suo TLB
- comunicare ad una CPU di eseguire una funzione, le cui informazioni vengono scritte in un'area di memoria condivisa
- se una CPU effettua lo spostamento di un TCB sulla runqueue di un'altra, può richiedere a quest'ultima mediante un IPI di schedulare il prima possibile tale thread in CPU

Gli IPI possono essere distinti in:

- fixe/physical: tale IPI è inviato verso una specifica CPU, definendo quindi una sorta di affinità fra l'interrupt e la CPU che lo processa
- logical/low priority: l'interrupt viene inviato ad un gruppo di CPU, magari usando un algoritmo round-robin

È possibile inoltre distinguere due livelli di priorità degli interrupt:

- alto, che permette che vi sia un solo interrupt alla volta standing in processamento
- basso, dove è possibile avere più interrupt standing, il cui processamento verrà serializzato

Per ogni interrupt è possibile specificare un codice, IRQ_x, a cui in realtà viene applicata in x86 una apposita funzione e quindi il codice risultante sarà dato da $F(IRQ_x)$, questo poiché le prime 32 linee sono associate alle trap supportate dall'architettura, quindi in questo modo se il pool di tali trap aumenta si potrà facilmente modificare solo la funzione F.
Uno dei componenti usati in Linux dell'architettura APIC è il timer LAPIC-T, un timer che è possibile programmare per l'invio di interrupt periodici o one-shot. Linux lo utilizza per inviare interrupt periodici, ogni 1ms in macchine single core e 4ms in macchine multicore, dove l'effetto è quello di decrementare i ticks di CPU residui per il thread correntemente in CPU.
All'interno di Linux, le informazioni riguardo APIC sono mantenute in appositi pseudofile del VSF, nella directory /proc, fra cui ad esempio vi è un file che mostra le informazioni per tutti gli interrupt, fra cui anche quante volte sono state invocate le funzioni per la gestione di tali interrupt.
Vi è fra queste la definizione per una entry associata ad un interrupt spurio: infatti, quando si controlla la linea di interrupt, questa può avere due valori (alto/basso) associabili al fatto che l'interrupt sia presente o no.
Siccome il passaggio da un valore all'altro non avviene direttamente ma può essere transitorio, si utilizza una

threshold per determinare se l'interrupt è presente o meno, per cui la entry è associata al caso in cui non si riesca a determinare con esattezza l'outcome corretto.

Domanda 2: vedi compito del 17/06/2019

Domanda 3: Un cache side channel è un effetto collaterale che permette, in base all'accesso in cache, di inferire informazioni riguardo il valore di un certo dato di interesse, basandosi sul tempo impiegato per accedere a tale dato.
Infatti, se tale dato è presente nell'architettura di caching, allora il tempo di accesso necessario per leggere il valore sarà molto inferiore al caso in cui tale dato va preso dalla memoria centrale.
Per poter effettuare tale misura, è necessario avere un timer ad alta precisione, che ad esempio in x86 è rdtsc: con l'istruzione macchina corrispondente è possibile ottenere il tempo trascorso nella granularità dei cicli di clock passati fino a quel momento, restituito come un valore a 64 bit, di cui i 32 più significativi vengono messi nel registro ecx ed i 32 meno significativi nel registro eax.
L'utilizzo di tale istruzione in user mode senza privilegi è permesso in base al fatto che venga o meno settato un bit di uno dei registri di controllo CR* del processore, ma questo è spesso vero in quanto tale timer viene utilizzato da molti software usati per fare profiling delle applicazioni.
Negli attacchi di spectre e meltdown, tale side channel viene utilizzato per inferire il valore di un byte del kernel. In entrambi i casi (meltdown e spectre v1), l'attaccante ha un array user mode composto da 256 entry, quindi il numero possibile di valori di un byte, ciascuna entry è pari a 4096 byte e quindi alla dimensione di una pagina. Tale byte kernel viene quindi usato per accedere allo 0-esimo byte di una pagina per andare ad indicizzare l'array e tale pagina verrà portata in cache. Andando successivamente a fare il timing sulla cache, si potrà determinare, accedendo alle pagine dell'array, quale di esse è presente in cache e quindi poter scoprire il valore del byte kernel.
È anche necessario, per l'attacco, aver ripulito la cache prima di effettuare la prima operazione di lettura dell'array, cosa possibile mediante l'istruzione di x86 CLFLUSH, che prende come parametro un pointer ad un byte e va a flushare la linea di cache contenente tale byte.

Compito del 28-02-2020

Domanda 1: L'Out-Of-Order pipeline è stata introdotta come uno dei metodi per poter risolvere le criticità della pipeline, relative al fatto che una istruzione che deve produrre dei dati necessari ad altre istruzioni può impiegare diversi cicli di clock per produrre tali dati, rendendo così necessario lo stallo delle istruzioni dipendenti da essa.
Con l'OOO è possibile far si che le altre istruzioni superino quella bloccata in pipeline, arrivando eventualmente anche al commit point, ma senza esporre i loro risultati a livello dell'ISA.
Nel momento in cui verranno risolte le criticità che avevano comportato il blocco, verrà ripristinato l'ordine di programma e quindi committate le istruzioni e rese effettive le modifiche a livello ISA.
In questo modo, si riesce ad ottenere un throughput tale per cui ad ogni ciclo di clock entra una istruzione in pipeline e ne esce una.

Per il resto della risposta, vedere compito del 17/06/2019.

Domanda 2: Per la gestione della memoria in Linux, su architetture x86, è necessario distinguere due casi: nel primo caso, si hanno i servizi disponibili durante la fase di boot del kernel, quando non è ancora a steady state.
Qui, il codice del kernel comincia già ad esprimere gli indirizzi virtuali, quindi è già presente la prima struttura fondamentale che è la kernel page table. Tale page table verrà espansa, alla fine del boot, per poter arrivare ad indicizzare tutta la memoria disponibile sulla macchina.
Inoltre, è la page table che viene utilizzata come base per la realizzazione delle tabelle di tutti gli altri processi del sistema.

Durante la fase di boot, è possibile allocare e deallocare memoria, mediante l'utilizzo dei servizi offerti dalla bootmem: questa mantiene una bitmap delle zone di memoria libere/occupate, per poter permettere l'allocazione di memoria qualora necessaria.
Vi è anche una versione NUMA-aware, che è la memblock, fondamentalmente simile alla bootmem.
Una volta raggiunto lo steady state, vi è la coremap come struttura di base: questa tiene traccia delle aree di memoria complessivamente libere o occupate, sopra di essa vengono costruite le free list che offrono una API per poter richiedere l'allocazione e deallocazione di memoria.
Tali liste sono specifiche per CPU, quindi mantenute nella per-CPU memory, e mantengono informazioni sulla zona di memoria ad esse associata.
Infatti, un allocatore di questo tipo è un così detto buddy allocator, che permette di allocare e deallocare memoria di diverso ordine:

- una pagina
- una coppia di pagine
- una doppia coppia di pagine
- etc...

quindi ogni free list manterrà in un array gli ordini di memoria che è possibile allocare per una determinata zona. Se ad esempio non è possibile allocare memoria di ordine 2, si possono accorpare due pagine di ordine 1, viceversa è possibile spezzare una coppia di pagine in due singole pagine ed allocarne una di esse.
L'interazione con un buddy allocator pervede l'utilizzo di spinlock e non è quindi adeguata per l'allocazione e deallocazione di strutture dati molto usate dal kernel, che avviene frequentemente. Per questo motivo, è possibile usare altri servizi che prevedono che venga pre-riservata della memoria da poter allocare, andando quando necessario ad interagire con il buddy allocator sottostante.
Tali servizi in Linux sono implementati con le quicklist, strutture dati che non bloccano e non falliscono, e tali per cui sono definite anche esse come variabili per-CPU.
Vi è poi la possibilità di allocare memoria con una granularità diversa da quella della singola pagina, mediante i così detti allocatori SLAB: tali servizi allocano memoria da un buddy allocator ed utilizzano le pagine restituite sia per allocare memoria a grana più fine che per contenere i loro stessi metadati.
In Linux, abbiamo i seguenti servizi:

- kmalloc / kfree: permettono di allocare e deallocare memoria fino ad un massimo di 128KB. L'aspetto interessante è che la memoria allocata cade sempre all'inizio di una linea di cache
- vmalloc / vfree: permettono di allocare e deallocare memoria virtuale per il kernel fino al massimo di 128 MB.

Tutti i servizi elencati allocano memoria virtuale che è anche allineata in memoria fisica, mediante la regola del direct mapping. L'unico servizio ad allocare memoria contigua solo virtualmente e non fisicamente è la vmalloc, che ha quindi anche importanti impatti sulla page table e sul TLB.