Capitolo 1

COS'È UN KERNEL

Il kernel rappresenta il nucleo di un sistema operativo e racchiude tutte le funzioni principali del sistema stesso come gestione della memoria, gestione delle risorse, lo scheduling e il file system. Le applicazioni in esecuzione nel sistema possono richiedere particolari servizi al kernel tramite chiamate di sistema (system call) senza accedere direttamente alle risorse fisiche. L'accesso diretto all'hardware può risultare anche molto complesso pertanto il kernel implementa una o più astrazioni dell'hardware, il cosiddetto Hardware Abstraction Layer. Queste astrazioni servono a nascondere la complessità e a fornire un'interfaccia pulita ed omogenea dell'hardware sottostante.
I kernel si possono classificare in quattro categorie:

- **Kernel monolitici** un unico aggregato di procedure di gestione mutuamente coordinate e astrazioni hardware.

- **Micro kernel** semplici astrazioni dell'hardware gestite e coordinate da un kernel minimale, basate su un paradigma client/server, e primitive di message passing.

- **Kernel ibridi** simili a micro kernel con la sola differenza di eseguire alcune componenti del sistema in kernel space per questione di efficienza.

- **Exo-kernel** non forniscono alcuna astrazione dell'hardware sottostante ma soltanto una collezione di librerie per mettere in contatto applicazioni con le risorse fisiche.

1.1 Il kernel Linux

Nell'aprile del 1991 Linus Torvalds, uno studente finlandese di informatica, comincia a sviluppare un semplice sistema operativo chiamato Linux. L'architettura del kernel sviluppato da Torvalds è di tipo monolitico a discapito di una struttura più moderna e flessibile come quella basata su micro kernel.

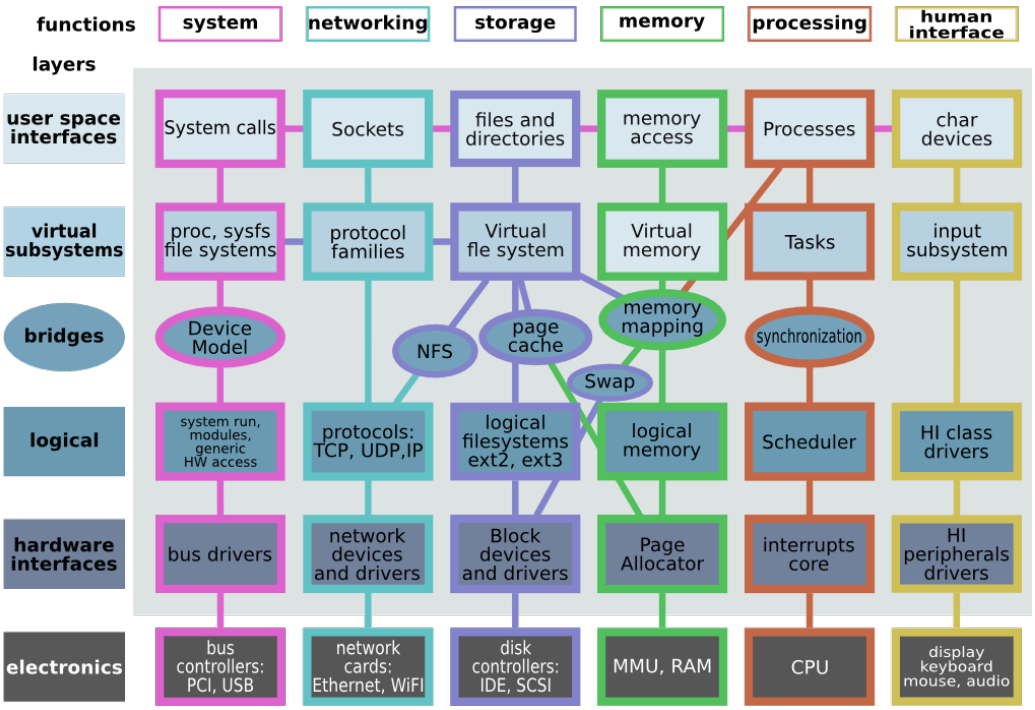

Figura 1.1: L'architettura del kernel Linux.

Sebbene oggi il kernel possa essere compilato in modo da ottenere un'immagine binaria ridotta al minimo, e i driver possono essere caricati da moduli esterni, l'architettura originaria è chiaramente visibile: tutti i driver, infatti,

devono avere una parte eseguita in kernel mode, anche quelli per cui ciò non sarebbe affatto necessario (come ad esempio i driver dei file system).
Attualmente il kernel Linux è distribuito con licenza GNU General Public License ed è in continua evoluzione grazie ad una vastissima comunità di sviluppatori da ogni parte del mondo che contribuisce attivamente al suo sviluppo.
Il kernel Linux trova larghissima diffusione ed utilizzo: infatti grazie alla sua flessibilità viene utilizzato dai personal computer ai grandi centri di calcolo, dai più moderni sistemi embedded agli smartphone.
Android, il sistema mobile più diffuso al mondo, si basa su una versione lightweight del kernel Linux.

1.2 Linux kernel v4.0

Il 12 Aprile 2015 è stato rilasciata la versione 4.0 del kernel Linux. Il cambio di versione da 3.x a 4.0 non è dovuto a nessun particolare miglioramento del kernel: la nuova versione stabile sarebbe dovuta essere la 3.20 ma su proposta di Linus Torvalds si è deciso di incrementare la numerazione alla versione 4.0 per non creare confusione con numeri molto grandi.
La nuova versione del kernel, quindi, non introduce particolari nuove feature. Una delle novità degna di nota è pero il *live patching*: ovvero la possibilità di installare pacchetti e estensioni al kernel senza dover riavviare la macchina. Questo può essere cruciale, ad esempio, per aspetti legati alla sicurezza.

1.3 Linux networking

I moduli del networking (e relativi device driver) occupano buona parte del codice del kernel Linux.
Vi sono due strutture particolarmente importanti utilizzante largamente in tutto lo stack di rete del kernel Linux.

Socket buffer La struttura sk_buff mantiene le informazioni relative a ciascun pacchetto inviato o ricevuto lungo lo stack di rete di Linux.
La struttura socket buffer contiene diversi campi che memorizzano le informazioni del pacchetto come il puntatore alla queue a cui è accodato o il socket a cui è associato.
Essendo lo stack di rete implementato su più livelli, dove ciascun layer aggiunge (o rimuove) delle proprie informazioni di intestazione ad un messaggio, mantenere per ogni livello di rete una struttura diversa per identificare un certo pacchetto appartenente a quel dato layer potrebbe risultare parecchio oneroso in termini di memoria in quanto si finirebbe per copiare grandi quantità di dati da un buffer ad un altro. Anche questo motivo, nel kernel Linux, si è deciso di mantenere un'unica struttura accessibile da qualsiasi layer dello stack di rete.
La struttura sk_buff mantiene un union field per ciascun livello di rete dello stack TCP/IP (trasporto, network, data-link) dove ciascun campo (h, nh, mac) rappresenta un header di un protocollo di comunicazione adotabbile per quel livello. Ciascun campo di queste union punteranno poi effettivamente a un header per quel livello.
Il campo *data* punta all'inizio di tutti i dati relativi al pacchetto (header compresi) variando a seconda del layer in cui il socket buffer è utilizzato: in particolare quando una funzione ad un certo livello dello stack di rete tratta una struttura sk_buff il campo data punterà all'header del messaggio per quel livello. Ad esempio in fase di ricezione di un pacchetto a livello network il campo nh di sk_buff sarà inizializzato all'header puntato dal campo data. Se si vorrà accedere, per qualche motivo al payload del datagram IP si potrà farlo calcolando l'offset a partire dal puntatore data; inoltrando il pacchetto ad un layer superiore l'header relativo al livello di rete corrente potrà essere rimosso tramite la routine **skb_pull()**.
In fase di invio il procedimento è molto simile ma con la difficoltà di dover appendere eventuali header man mano che si percorre lo stack di rete verso il basso. A tale scopo vi è una funzione di manipolazione delle strutture

sk_buff chiamata **skb_reserve**, in genere invocata dopo un allocazione di un socket buffer, che consente di riservare dello spazio per gli header dei diversi protocolli.

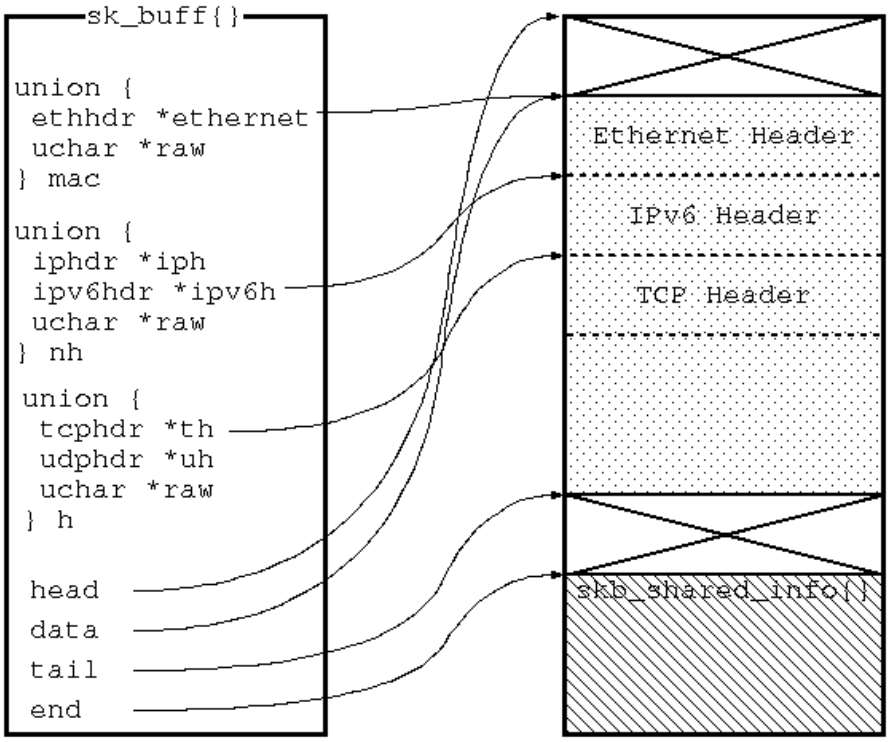

Figura 1.2: La struttura *sk_buff*.

struct net_device questa struttura rappresenta una scheda di rete, anche virtuale: può essere infatti che ci si riferisca ad una scheda di rete virtuale ottenuta dopo aver effettuato un *bonding*, ovvero è possibile assegnare un unico indirizzo IP a due o più NIC (vedendole così come un unico device) in modo da migliorare le performance.
La struttura net_device contiene campi che identificano l'interfaccia di rete come il valore massimo della sua MTU e l'indirizzo MAC.

Nello sviluppo del progetto di tesi è stato esteso il comportamento del workflow del kernel Linux per quanto riguarda la trasmissione di un datagram UDP. Per tanto, qui di seguito, descriveremo brevemente buona parte dei moduli che rendono possibile la trasmissione di un pacchetto UDP a partire da un applicazione fino al device driver.
Quando viene invocata una routine che interagisce con la rete tramite socket (come ad esempio send, sendto o sendmsg) il controllo viene passato al *socket interface layer*: se il socket è di tipo TCP il controllo verrà dato alla funzione tcp_sendmsg altrimenti, se di tipo UDP, alla primitiva udp_sendmsg.
Appena il flusso dell'esecuzione è passato alla routine udp_sendmsg vengono effettuati alcuni controlli sui parametri passati come ad esempio sulla lunghezza specificata per il payload del datagram UDP o sugli indirizzi mittente e destinatario e le porte. Dopodiché, a seconda se specificato il flag UDP_CORK o meno (che consente quando abilitato tramite la system call *setsockopt* di accumulare tutti i dati in output sul socket e inviarli in unico datagram quando disabilitato), il controllo dell'esecuzione passerà alla primitiva di livello rete ip_append_data() oppure a ip_skb_make() che in entrambi casi, comunque, si appoggeranno sulla funzione __ip_append_data() che tra le altre cose si occuperà di bufferizzare tutte le richieste di tramissione e di generare uno o più sk_buff rappresentanti ciascun pacchetto (o frammento) IP.
A questo punto il flusso esecutivo torna a livello trasporto dove a un certo punto verrà invocata la primitiva udp_push_pending_frames che genera l'header UDP e passa definitivamente il controllo a livello rete invocando la funzione ip_push_pending_frame().
In ip_push_pending_frame() verrà generato l'header IP per ogni socket buffer in coda sul buffer in uscita per quel socket e dopo diversi controlli se il datagram IP necessita di frammentazione la funzione ip_fragment si occuperà di frammentare il pacchetto IP in diverse strutture di tipo sk_buff.
Tra i compiti a carico del livello network dello stack di rete vi è quello del *routing*: in particolare a supporto del routing a livello rete vi è la funzione

ip_route_output_flow.
Se la fase di routing termina con successo verrà invocata la funzione di livello *data-link* dev_queue_xmit() che, se l'interfaccia richiesta per la trasmissione è attiva, accoderà il socket buffer nella coda in uscita del net_ device associato all' sk_buff in invio e invocherà la funzione dev_hard_start_xmit per cominciare a processare l'invio di un sk_buff.
Nel caso di un interfaccia di rete *softMAC*, ovvero che il MLME (MAC Sublayer Management Entity, ovvero dove viene implementata la logica del Medium Access Control) è implementato via software (vi sono dispositivi detti *fullMAC* che implementano il MLME direttamente in hardware) questa funzione si appoggerà sul sottosistema mac80211.

Sottosistema mac80211 Il sottosistema mac80211 è stato rilasciato nel 2005 e si occupa di implementare la logica MLME per device softMAC interpretando e generando frame IEEE 802.11. Prima della sua adozione all'interno del kernel Linux la gestione e l'implementazione di IEEE 802.11 era lasciata ai device driver.
Oggi la maggior parte dei device supportano questo sistema e in molti dei vecchi dispositivi i device driver sono stati riscritti tenendo conto di esso.
In fase di trasmissione di un messaggio il controllo dal livello data-link viene ceduto al sottosistema mac80211 invocando la sua entry-point ieee **ieee80211_xmit** che tra le altre cose si occupa dell'inizializzazione del frame 802.11 e del suo header. Per far ciò il sottosistema mac80211 si serve di una serie di handler appositamente settati, tra cui:

- **ieee80211_tx_h_dynamic_ps** per la gestione del power saving.
- **ieee80211_tx_h_select_key** si occupa di selezionare una chiave di cifratura.
- **ieee80211_tx_h_michael_mic_add** si occupa di aggiungere il Message Integrity Code al frame IEEE 802.11.

- **ieee80211_tx_h_rate_ctrl** che seleziona il bit rate con la quale il frame IEEE 802.11 deve essere trasmesso.

- **ieee80211_tx_h_sequence** calcola il *sequence number* del frame e lo assegna all'header 802.11.

- **ieee80211_tx_h_fragment** si occupa, eventualmente, di frammentare il frame IEEE 802.11 in base al valore della MTU della scheda di rete wireless e a quella dell'access point.

- **ieee80211_tx_h_encrypt** cifra il frame IEEE 802.11 con l'algoritmo designato.

- **ieee80211_tx_h_calculate_duration** si occupa di calcolare il campo *duration* del frame IEEE 802.11 che indica per quanto tempo il canale sarà impegnato dalla trasmissione del frame.

- **ieee80211_tx_h_stats** setta una serie di variabili utili al mantenimento di alcune statistiche di trasmissione.

Una volta che il frame IEEE 802.11 è stato trasmesso la struttura socket buffer associata non viene immediatamente deallocata: mac80211 provvederà a ritrasmettere il messaggio per un certo numero di volte, dipendente dalla NIC e dallo stato in cui si trova la rete alla quale è connessa, qualora non dovesse ricevere l'ACK relativo a quel frame.
Alla fine di questo processo viene invocata la routine ieee80211_tx_h_status che fornisce alcune informazioni riguardanti l'esito della trasmissione per un certo pacchetto come ad esempio quante volte il pacchetto è stato ritrasmesso e se è stato effettivamente consegnato o meno all'access point.
A questo punto sia che il frame IEEE 802.11 sia stato ricevuto dal first-hop o meno il socket buffer associato viene definitivamente deallocato.

1.4 Customizzazione Kernel

Può esserci l'esigenza di personalizzare il proprio kernel per adattarlo alle proprie esigenze. Alcuni vantaggi che si possono ottenere dalla customizzazione del kernel sono:

- Velocizzare l'avvio del sistema.
- Sfruttare un'ottimizzazione basata sul proprio processore per avere un sistema operativo un po' più reattivo.
- Ottenere una configurazione leggermente diversa e più adatta alla propria macchina.
- Cercare di migliorare le prestazioni del proprio dispositivo.
- Ridurre i consumi.
- Installare solo i moduli necessari.
- Rendere il sistema più leggero.

Per personalizzare il kernel, però, serve una buona conoscenza della propria macchina, ed in particolare del proprio hardware. Se non ci conosce il proprio hardware, infatti, si rischia di peggiorare le prestazione e si rischia di togliere dei moduli che possono influire sul funzionamento del sistema.

Non è necessario compilare un kernel nei casi in cui l'hardware non funzioni alla perfezione o le periferiche non vengano completamente riconosciute. A volte per far riconoscere correttamente al sistema una data periferica basta caricare i moduli necessari con le dovute opzioni. È utile ricompilare il kernel solo se tali moduli non sono presenti o se si è certi che i driver della periferica sono presenti solo in una versione diversa da quella attualmente in uso. Inoltre, l'aumento di prestazioni tende a essere irrilevante, soprattutto su computer già veloci. È anche bene tenere presente che compilare un nuovo kernel significa, nella sostanza, cambiare sistema operativo, in quanto esso ne costituisce il motore; inoltre è richiesta un buona conoscenza del proprio

hardware.
Dopo aver analizzato i vantaggi e gli svantaggi della personalizzazione del kernel, spieghiamo ora come si può fare la customizzazione. Per modificare il kernel bisogna determinare quali driver sono necessari, compilare i kernel ed i moduli ed infine installare l'immagine del kernel.
I passi da eseguire per portare a termine una customizzazione sono:

Scaricare una versione stabile del kernel ed estrarlo

```
# cd /directory/
# wget https://www.kernel.org/pub/linux/kernel/
v4.x/linux-4.0.1.tar.xz
# tar -xvJf linux-4.0.1.tar.xz
```

Il link indicato è quello relativo alla pagina ufficiale, dove si possono trovare le ultime versioni dei kernel. La versione scelta da noi è stata la 4.0.1 in quanto era l'ultima stabile.

Scegliere la configurazione

```
# cd linux-4.0.1
# sudo apt-get install libncurses5
# sudo apt-get install libncurses5-dev

# sudo make menuconfig
```

Per poter eseguire il comando di menuconfig è necessario installare prima i componenti *libncurses5*. Una volta eseguito il comando di configurazione, si aprirà una finestra come quella indicata in figura 1.3 Nel caso in cui si volessero aggiungere o rimuovere moduli, bisognerà andare a selezionarli all'interno di questo menu.

Compilare il Kernel

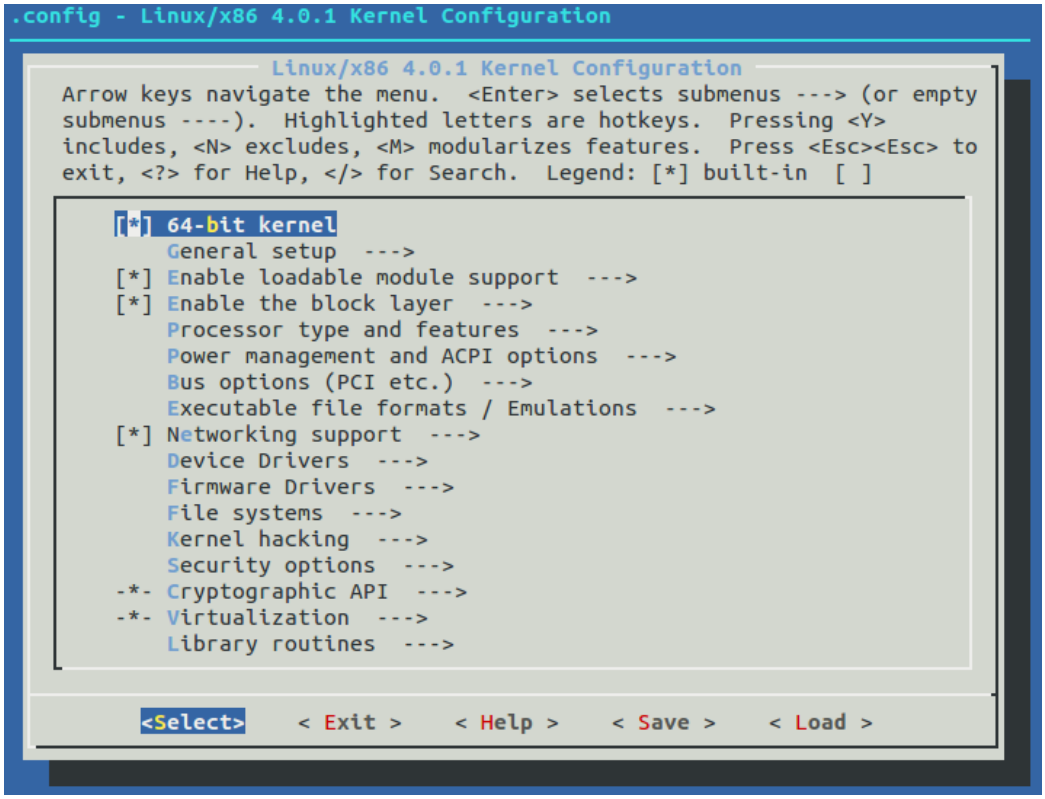

Figura 1.3: Esecuzione del comando menuconfig.

```
# sudo make
```

Con questo comando si va a compilare il kernel. Questa operazione può richiedere diverso tempo, in particolare in dispositivi con prestazione ridotte. È possibile anche specificare l'opzione -jN con N che indica il numero di core più (ad esempio *-j5* per un quad-core) per velocizzare l'operazione.

Compilare i moduli ed installarli

```
# sudo make modules
# sudo make modules_install
```

Installare il Kernel

```
# sudo make install
# sudo reboot
```

Come ultima operazione si va ad installare l'immagine del nuovo kernel. Una volta riavviato il sistema è possibile utilizzare il proprio kernel customizzato.