IL SISTEMA OPERATIVO LINUX

Nel capitolo online “Il Sistema Operativo FreeBSD” si è discusso il funzionamento interno del sistema operativo 4.3BSD. BSD è soltanto uno dei sistemi UNIX-like. Linux è un altro sistema UNIX-like che ha guadagnato sempre maggiore popolarità negli ultimi anni. In questo capitolo, si darà uno sguardo alla storia e alla progettazione di Linux, oltre che alle interfacce per l'utente e per il programmatore che il sistema operativo mette a disposizione. Si discuterà inoltre la struttura interna che realizza tali interfacce. Dato che Linux è stato progettato per essere compatibile con il più alto numero di applicazioni standard UNIX, esso ha molto in comune con le implementazioni esistenti di UNIX. Per questo motivo non verrà duplicata la descrizione di base di UNIX data nel capitolo “Il Sistema Operativo FreeBSD”.

Linux è un sistema operativo in rapida evoluzione. Questo capitolo descrive nello specifico il kernel 2.2 di Linux, rilasciato nel gennaio 1999, riportando alcune delle novità aggiunte dal kernel 2.6 pubblicato alla fine del 2003.

1 Storia

Linux ha tutte le caratteristiche di ogni altro sistema UNIX. In effetti, la compatibilità con UNIX è uno degli obiettivi principali del design di Linux. Comunque, Linux è molto più giovane della maggior parte dei sistemi UNIX. Lo sviluppo iniziò nel 1991, quando uno studente finlandese, Linus Torvalds, scrisse e battezzò **Linux**, un piccolo ma autonomo kernel per processori 80386, il primo processore a 32-bit nella produzione di CPU per PC compatibili di Intel.

Presto, durante lo sviluppo, il codice sorgente di Linux fu reso disponibile gratuitamente su Internet. Come risultato, la storia di Linux è stata segnata dalla collaborazione di molti utenti da ogni parte del mondo, in contatto quasi esclusivamente tramite Internet. Dal kernel iniziale, che realizzava solo parzialmente una piccola parte dei servizi UNIX, Linux è cresciuto fino ad aggiungere molte funzionalità di UNIX.

Nei suoi primi anni, lo sviluppo di Linux si rivolse in larga misura intorno al kernel centrale del sistema operativo - il core, un ideale direttore privilegiato che gestisce tutte le risorse del sistema e interagisce direttamente con l'hardware del computer. Ovviamente, c'è bisogno molto più del kernel per ottenere un sistema operativo completo. È utile distinguere tra il kernel e il sistema Linux. Il kernel è un software completamente originale, sviluppato da zero dalla comunità Linux. Questo sistema, come lo si conosce oggi, include una moltitudine di componenti, alcuni dei quali scritti autonomamente, altri presi in prestito da altri progetti e altri ancora creati in collaborazione con altri gruppi di sviluppo.

Il sistema Linux di base è un ambiente standard per le applicazione e la programmazione da parte dell'utente, ma non rispetta nessuno standard per la gestione generale delle funzionalità disponibili. Con la maturazione di Linux, è sorta la necessità di inserire un nuovo livello di funzionalità in cima al sistema. Una **distribuzione Linux** include i componenti standard del sistema Linux, oltre a una serie di tool di amministrazione aventi lo scopo di semplificare l'installazione iniziale, gli aggiornamenti successivi, e la gestione dell'installazione e della disinstallazione sul sistema di altri pacchetti. Una moderna distribuzione comprende solitamente dei tool per la gestione del file system, la creazione di account di utenti, l'amministrazione di rete, eccetera.

1.1 Il kernel di Linux

Il primo kernel di Linux, rilasciato al pubblico, è stato la versione 0.0.1, il 14 maggio 1991. Era privo di gestione della rete, era eseguibile solo su processori compatibili con Intel 80386 e l'hardware di un PC, ed aveva un supporto molto limitato di driver di periferiche. Il sottosistema di memoria virtuale era anch'è molto basilare e non includeva il supporto per i file mappati in memoria; comunque, anche questa prima incarnazione sopportava le pagine condivise con la copia in scrittura. I file system Minix era l'unico supportato: i primi kernel di Linux sono stati sviluppati su piattaforma Minix. In ogni caso, il kernel implementava propriamente i processi di UNIX con lo spazio di indirizzi protetti.

La successiva versione importante, Linux 1.0, è stata rilasciata il 14 marzo 1994, ed è stata il culmine di tre anni di rapido sviluppo del kernel di Linux. Forse la singola novità più grossa è stata il supporto di rete: 1.0 ha incluso il supporto per il protocollo di rete standard di UNIX TCP/IP, oltre a interfacce di socket compatibili con BSD per la programmazione di rete. È stato aggiunto supporto per i driver delle periferiche per eseguire IP su Ethernet o (usando i protocolli PPP o SLIP) su linee seriali o modem.

Il kernel 1.0 incluse un file system nuovo è molto migliorato senza le limitazioni del file system originario di Minix, e sopportò una gamma di controller SCSI per l'accesso ai dischi con alte prestazioni. Gli sviluppatori estesero il sottosistema di memoria virtuale per sopportare la paginazione per scambiare file e la mappatura in memoria di file arbitrari (ma nella versione 1.0 fu realizzata solo la mappatura in sola lettura).

In questa versione, fu incluso anche supporto per vari hardware addizionali, anche se ancora ristretto alla piattaforma PC Intel, thread, sino ad includere lettori di floppy disk e di CDROM, oltre a schede sonore, vari mouse, e tastiere internazionali. Fu fornita anche l'emulazione dei floating-point per gli utenti dell'80386 che non avevano il coprocessore matematico 80387, e fu realizzata la comunicazione fra processi (**interprocess communication, IPC**) nello stile di UNIX System V, inclusi la memoria condivisa, i semafori, e le code di messaggi. Fu aggiunto anche un semplice supporto per i moduli del kernel caricabili e scaricabili dinamicamente.

A questo punto iniziò lo sviluppo del filone 1.1, ma in seguito furono rilasciati numerosi bug fix per la 1.0. Questo comportamento fu adottato come la convenzione di numerazione standard dei kernel di Linux: i kernel con un numero di versione minore dispari, quali 1.1, 1.3 o 2.1 sono **kernel di sviluppo** (**development kernel**); quelli con numeri minori di versione pari sono i **kernel di produzione** (**production kernel**) stabili. Le modifiche sui kernel stabili sono intese solo come versioni di ripiego, laddove i kernel di sviluppo possono includere funzionalità nuove e ancora non testate.

Nel marzo 1995 fu rilasciato il kernel 1.2, che non offrì lo stesso incremento di funzionalità della versione 1.0, mai incluse il supporto per una quantità molto più ampia di hardware, inclusa la nuova architettura dei bus PCI. Gli sviluppatori aggiunsero un'altra caratteristica specifica dei PC - il supporto della CPU 80386 per la modalità virtuale 8086 - per permettere l'emulazione del sistema operativo DOS per PC. Fu modificato lo stack di rete per fornire supporto al protocollo IPX, e per rendere più completa l'implementazione di IP, includendo le funzionalità di accounting e di firewall.

Il kernel 1.2 fu anche l'ultimo kernel di Linux solo per PC. La distribuzione sorgente di Linux 1.2 incluse supporto parzialmente implementato per le CPU SPARC, Alpha e MIPS, ma la piena integrazione di queste altre architetture non cominciò se non dopo il rilascio del kernel 1.2 stabile.

Il rilascio di Linux 1.2 si concentrò su un supporto hardware più ampio e su un'implementazione più completa delle funzionalità esistenti: molte nove funzionalità erano in fase di sviluppo a quei tempi, ma l'integrazione del nuovo codice nel codice sorgente del kernel principale fu posticipato a

dopo il rilascio del kernel 1.2 stabile. Come risultato il flusso di sviluppo dell'1.3 vide a una grande quantità di nove funzionalità aggiunte al kernel.

Questo lavoro fu finalmente rilasciato come Linux 2.0 nel Giugno del 1996. A questo rilascio fu dato un incremento del numero maggiore di versione a seguito di due nuove principali possibilità: il supporto per più architetture, fra cui una porta ad una Alpha nativo completamente a 64 bit, e il supporto per le architetture multiprocessore. Le distribuzione di Linux basate sul 2.0 furono disponibili anche per i processori Motorola della serie 68000 e per i sistemi SPARC della Sun. Una versione derivata di Linux eseguibile su microkernel Mach funzionava anche sui sistemi PC e PowerMac.

I cambi nella 2.0 non si fermarono qui: il codice per la gestione della memoria fu migliorato sostanzialmente per fornire una cache unificata per i dati del file system, e indipendente dalla cache delle periferiche a blocchi. Come risultato di questo cambiamento, il kernel offrì prestazioni sul file system e sulla memoria virtuale ampiamente migliorate. Per la prima volta, la cache del file system fu estesa ai file system in rete, e furono supportate regioni mappate di memoria scrivibili.

Il kernel 2.0 incluse anche prestazioni ampiamente migliorate per TCP/IP, e fu aggiunta una quantità di nuovi protocolli di rete, fra cui AppleTalk, le reti AX.25 dei radioamatori ed il supporto

ISDN, oltre alla possibilità di montare volumi remoti Netware e SMB (Microsoft LanManager).

Altri miglioramenti principali nella 2.0 furono il supporto per i thread interni del kernel, per la gestione delle dipendenze fra moduli caricabili, e per il caricamento automatico dei moduli su richiesta. La configurazione dinamica del kernel durante l'esecuzione fu molto migliorata attraverso interfacce di configurazione nuove e standardizzate. Altre nuove caratteristiche includono le quote del file system e le classi compatibili POSIX per la schedulazione dei processi real time.

Nel gennaio 1999 fu rilasciato Linux 2.2 che proseguì i miglioramenti aggiunti da Linux 2.0. Fu aggiunta una versione per UltraSPARC. La gestione della rete fu migliorata attraverso un firewall più flessibile, un miglior routing e una migliore gestione del traffico, come il supporto per finestre TCP larghe e di ack selettivi. I dischi Acorn, Apple e NT potevano essere letti, fu migliorato NFS e fu aggiunto un demone NFS in modalità kernel. La gestione dei segnali, gli interrupt e parte dell'I/O furono definiti meglio e corretti rispetto a prima per migliorare le prestazioni di SMP.

I miglioramenti nei successivi kernel 2.4 e 2.6 includono un miglior supporto per i sistemi SMP (symmetric multiprocessor), file system con il supporto per il journaling e miglioramenti nel sistema di gestione della memoria. Nella versione 2.6, inoltre, è stata modificata la schedulazione dei processi per fornire un algoritmo di schedulazione efficiente con complessità computazionale costante. Inoltre il kernel 2.6 supporta la prelazione, permettendo l'interruzione di un processo mentre è in esecuzione in modalità kernel.

1.2 Il sistema Linux

In molti sensi, il kernel di Linux rappresenta il cuore del progetto, ma altri componenti concorrono a formare il sistema operativo Linux completo.

Considerato che il kernel è composto interamente di codice scritto da zero specificatamente per il progetto Linux, molto del software di supporto che compone il sistema non è esclusivo del sistema operativo, ma comune ad un certo numero di sistemi UNIX.

In particolare, Linux utilizza molti tool sviluppati come parte del sistema operativo BSD di Berkeley, dell'X Window System del MIT, e del progetto GNU della Free Software Foundation.

Questa condivisione di tool ha funzionato in entrambe le direzioni. Le librerie principali di sistema di Linux ebbero origine dal progetto GNU, ma la comunità Linux le migliorò enormemente

segnalando omissioni, inefficienze o bug. Altri componenti come il **compilatore GNU C** (**gcc**) erano di qualità già sufficientemente alta per essere usati direttamente in Linux. Il tool di amministrazione di rete sotto Linux deriva dal codice sviluppato per 4.3BSD, ma in cambio, più recenti derivati di BSD, come ad esempio FreeBSD, hanno preso in prestito porzioni di codice da Linux, come la libreria matematica di emulazione del floating point per Intel, o i driver per le periferiche hardware audio.

Il sistema Linux nella sua interezza è mantenuto da un gruppo autonomo di sviluppatori che collaborano tramite Internet, con piccoli gruppi di persone con la responsabilità di mantenere l'integrità di componenti specifici.

Un numero ridotto di siti ftp pubblici su Internet viene utilizzato come repository (deposito) standard per questi componenti. Il documento **File System Hierarchy Standard** (standard della gerarchia del file system) viene anch'esso mantenuto dalla comunità Linux, come mezzo per ottenere la compatibilità tra i vari componenti di sistema. Questo standard specifica la struttura (layout) globale di un file system Linux standard; esso determina in quali directory devono essere presenti i file di configurazione, le librerie, i file binari di sistema e i file di dati run time.

1.3 Le distribuzioni di Linux

In teoria, chiunque può installare un sistema Linux prelevando da un sito ftp le ultime versioni dei componenti di sistema necessari e compilandole. Nel primo periodo di Linux, questa operazione era proprio quella che spesso restava a carico dell'utente. Con la maturazione di Linux, comunque, varie persone e gruppi provarono a rendere questo processo meno oneroso, distribuendo un set di package standard precompilati, allo scopo di rendere l'installazione più semplice.

Questi insiemi di componenti, o distribuzioni, includevano molto più che la sola base del sistema Linux. Venivano tipicamente inclusi programmi extra di utilità per l'installazione o la gestione del sistema, così come package precompilati pronti da installare e contenenti molti dei tool comuni di UNIX, come, ad esempio, news server, navigatori web, elaboratori di testo, tool di editing e perfino giochi.

La prima distribuzione permetteva la gestione dei package semplicemente mettendo a disposizione un tool per decomprimere tutti i file nelle posizioni appropriate. Uno dei contributi importanti delle distribuzioni moderne, ad ogni modo, consiste nella gestione avanzata dei package. Le distribuzioni Linux del giorno d'oggi includono un database di tracciamento dei package, che ne permette la facile installazione, l'aggiornamento e la rimozione.

Nel lontano passato di Linux, la distribuzione SLS fu la prima raccolta di package cui ci si poteva riferire come ad una distribuzione completa. Nonostante la possibilità di installare package come singole entità, SLS non comprendeva tool di gestione dei package, che ci si aspetta invece di trovare nelle nuove distribuzioni. La distribuzione **Slackware** rappresentò un grande miglioramento nella qualità complessiva, nonostante la gestione dei package fosse ancora carente; questa distribuzione è ancora oggi una delle più largamente installate dalla comunità Linux.

A partire dal rilascio di Slackware, sono state rese disponibili un grande numero di distribuzioni commerciali e non. **Red Hat** e **Debian** sono distribuzioni particolarmente popolari, mantenute rispettivamente da una società commerciale di supporto Linux, e dalla comunità Linux per il software gratuito. Altre versioni commerciali di UNIX sono le distribuzioni di **Caldera, Craftworks** e **WorkGroup Solutions**. Un grande seguito di utenti in Germania ha fatto sì che fossero immesse sul mercato numerose distribuzioni dedicate in lingua tedesca, come è avvenuto, ad esempio, per le versioni **SuSE** e **Unifix**. Esistono troppe distribuzioni di Linux in circolazione

perché le si possa elencare tutte. La varietà delle distribuzioni non esclude a priori la compatibilità tra le varie versioni. La maggioranza delle distribuzioni utilizza o supporta il formato dei file di package RPM, e le applicazioni commerciali distribuite in questo formato possono essere installate ed eseguite su qualsiasi distribuzione di Linux che supporti questo tipo di file.

1.4 La licenza di Linux

Il kernel di Linux è distribuito sotto la licenza GNU General Public Licence (GPL), la cui sigla è stata coniata dalla Free Software Foundation. Linux non è un software di **pubblico dominio**, poiché con esso è previsto che l'autore rinunci al copyright sul software, mentre invece il copyright su Linux è ancora di proprietà dei vari creatori del codice. Linux è un software gratuito, ad ogni modo, nel senso che ognuno può copiarlo, modificarlo, utilizzarlo in qualunque modo si voglia, e ridistribuirlo senza alcuna restrizione.

La conseguenza principale dei termini di licenza di Linux è che chiunque voglia utilizzarlo o creare una propria versione derivata del sistema (una operazione legittima), non è autorizzato a rendere la propria versione proprietaria. Il software rilasciato con licenza GPL non può essere ridistribuito con solo i file binari. Se si rilascia un software che include componenti coperti dalla GPL, allora con questo tipo di licenza è necessario rendere disponibili nella distribuzione anche il codice sorgente, insieme ai binari. (Questa restrizione non proibisce di produrre o perfino vendere distribuzioni di soli file binari, ma chi riceve la distribuzione binaria deve anche avere la possibilità di ottenere i codici sorgente ad un prezzo ragionevole).

2 Principi di progettazione

Nel suo progetto generale, Linux assomiglia a tutte le altre implementazioni tradizionali di UNIX senza microkernel. È un sistema multiutente, multitasking con un set completo di tool compatibili con UNIX. Il file system di Linux segue da vicino la semantica tradizionale di UNIX, e il modello di networking standard di UNIX è stato implementato in modo completo. I dettagli interni della progettazione di Linux sono stati influenzati pesantemente dalla storia di questo sistema operativo.

Nonostante che Linux sia compatibile con un'ampia varietà di piattaforme, il sistema fu sviluppato esclusivamente sull'architettura PC. Una grossa fetta degli sviluppi, nel primo periodo, fu portata avanti da singole persone entusiaste, piuttosto che da ben sovvenzionati laboratori di ricerca e sviluppo; così, fin dall'inizio, Linux tentò di ottenere la maggior funzionalità possibile a partire da risorse limitate. Oggi, Linux può essere eseguito senza problemi su una macchina multiprocessore con migliaia di megabyte di memoria principale e molti gigabyte di spazio su disco, e allo stesso modo su una macchina con 4MB di RAM.

Con i PC che diventavano sempre più potenti, e con memorie e hard disk più economici, i kernel originali, minimalisti, crebbero per implementare più funzionalità di UNIX. La velocità e l'efficienza sono ancora obiettivi importanti nella progettazione, ma molto del lavoro speso in tempi recenti è concentrato su un terzo obiettivo principale: la standardizzazione.

Uno dei prezzi da pagare per la diversità delle implementazioni di UNIX attualmente disponibili è che il codice sorgente scritto per un dialetto potrebbe non compilare o non eseguirsi correttamente su un altro. Perfino quando sono presenti le stesse chiamate di sistema su due diversi sistemi UNIX, queste non si comportano necessariamente nello stesso identico modo. Lo standard POSIX consiste di un set di specifiche di diversi aspetti del comportamento di un sistema operativo. Esistono

documenti POSIX per le funzionalità comuni dei sistemi operativi e per le estensioni, come ad esempio i thread dei processi e le operazioni real-time. Linux è stato progettato per essere coerente con i documenti POSIX di rilievo; almeno due distribuzioni Linux hanno ottenuto una certificazione ufficiale in tal senso.

Per il fatto di presentare delle interfacce standard sia per il programmatore che per l'utente, Linux presenta poche sorprese a chiunque sia familiare con UNIX. Non descriveremo nel dettaglio tali interfacce nel contesto di Linux. Le sezioni sull'interfaccia di programmazione (Paragrafo A.3) e sull'interfaccia utente (Paragrafo A.4) di 4.3BSD possono essere applicate allo stesso modo a Linux. Per default, comunque, l'interfaccia di programmazione di Linux aderisce alla semantica SVR4 UNIX, piuttosto che al comportamento di BSD. È disponibile un set separato di librerie che le implementano, in situazioni in cui i due comportamenti sono profondamente diversi.

Esistono molti altri standard nel mondo UNIX, ma le certificazioni che attesta che Linux li implementa vengono a volte rallentate, poiché esse sono costose, e la spesa per certificare la bontà del sistema operativo nel seguire la maggior parte degli standard è sostanziale. In ogni caso, il fatto di supportare una grossa base di applicazioni è importante per qualsiasi sistema operativo, per questo motivo l'implementazione di questi standard è uno degli obiettivi principali per lo sviluppo di Linux, seppure non certificato ufficialmente. In aggiunta allo standard POSIX, Linux attualmente supporta l'estensione per i thread e un sottoinsieme delle estensioni per il controllo dei processi real-time.

2.1 Componenti di un sistema Linux

Il sistema Linux è composto di tre parti principali di codice, come accade per la maggior parte delle implementazioni tradizionali di UNIX.

1. **Kernel.** Il kernel è responsabile della gestione di tutte le principali astrazioni del sistema operativo, tra le quali la memoria virtuale e i processi.
2. **Librerie di sistema.** Le librerie di sistema definiscono un set di funzioni standard tramite le quali le applicazioni possono interagire con il kernel; esse implementano molte delle funzionalità del sistema operativo che non necessitano dei privilegi in possesso del codice del kernel
3. **Utilità di sistema.** Le utilità di sistema sono programmi che eseguono funzioni di gestione singole e specializzate. Alcune di queste utility potrebbero essere richiamate solo una volta, al fine di inizializzare o configurare alcuni aspetti del sistema; altre- dette, nella terminologia UNIX, *demoni*, possono essere in esecuzione in modo permanente, al fine di gestire delle funzioni come la risposta alle connessioni di rete in ingresso, accettare richieste di logon dai terminali, o aggiornare i file di log.

La figura 1 mostra le varie componenti che formano un sistema Linux completo. La distinzione più importante, in questo momento, è quella tra il kernel e tutto il resto. Tutto il codice del kernel viene eseguito in modalità privilegiata del processore, con accesso totale a tutte le risorse fisiche del computer. In Linux ci si riferisce a questo tipo di accesso come **kernel mode**, equivalente al monitor mode, descritto nel paragrafo 2.5.1. Sotto Linux, non è presente nessuna porzione di codice user mode all'interno del kernel. Qualsiasi porzione di codice di supporto al sistema operativo che non abbia necessità di essere eseguita in modalità kernel viene posta nelle librerie di sistema.

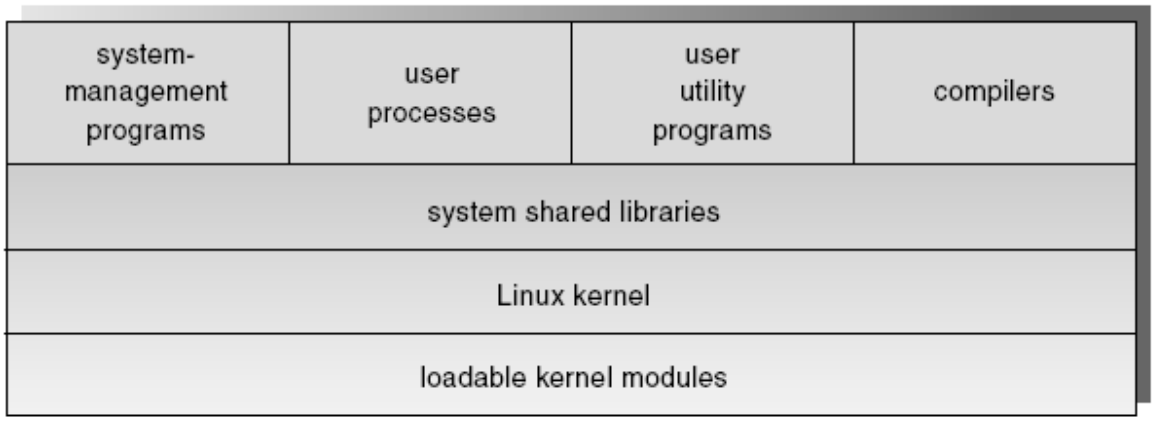

Figura 1. Componenti del sistema Linux.

system management programs = programmi di gestione del sistema
user process = processo utente
user utility programs = programmi di utilità per utente
compilers = compilatori
system shared libraries = librerie condivise di sistema
Linux kernel =kernel di Linux
loadable kernel modules = moduli caricabili del kernel

Nonostante il fatto che molti sistemi operativi moderni abbiano adottato una architettura basata sullo scambio di messaggi, per le operazioni interne del kernel, Linux ha mantenuto il modello storico di UNIX: il kernel è composto da una libreria singola e monolitica. Il motivo principale è quello di migliorare le prestazioni: dato che il codice del kernel e le strutture dati sono tenute in un unico spazio di indirizzi, non si rendono necessari cambi di contesto, quando un processo richiama una funzione del sistema operativo o quando viene segnalato un interrupt hardware. Non solo il codice per la schedulazione del core e per la memoria virtuale occupa questo spazio di indirizzi; *tutto* il codice del kernel, inclusi i driver delle periferiche, il codice per il file system e per il networking, è presente nello stesso unico spazio di indirizzi.

Il fatto che tutto il kernel condivida questo stesso miscuglio di componenti non significa che non ci sia attenzione per la modularità. Nello stesso modo in cui un'applicazione dell'utente può caricare librerie condivise a run time per utilizzare un pezzo di codice che le è necessario, il kernel di Linux è in grado di caricare (e scaricare) dinamicamente i moduli a run time. Il kernel non ha necessità di conoscere in anticipo quali moduli devono essere caricati: tali operazioni vengono eseguite in modo del tutto indipendente.

Il kernel di Linux rappresenta il cuore del sistema operativo. Esso fornisce le funzionalità principali per eseguire processi, e rende possibili i servizi di sistema, al fine di consentire un accesso

protetto e supervisionato alle risorse hardware. Il kernel implementa tutte le funzionalità necessarie perché si possa parlare di un sistema operativo. Preso a parte, comunque, il sistema operativo fornito dal kernel di Linux non somiglia per niente a un sistema UNIX. Sono assenti numerose funzioni extra di UNIX, e le caratteristiche implementate non sono necessariamente nel formato che un'applicazione UNIX si aspetterebbe. L'interfaccia del sistema operativo, resa visibile alle applicazioni in esecuzione, non viene mantenuta direttamente dal kernel. Piuttosto, le applicazioni eseguono delle chiamate a librerie di sistema, che a turno richiamano i servizi del sistema operativo a seconda delle necessità.

Le librerie di sistema rendono possibili vari tipi di funzionalità: al livello più semplice fanno in modo che le applicazioni siano in grado di eseguire richieste di servizi di sistema del kernel. Per eseguire una chiamata di sistema è necessario il trasferimento del controllo dalla modalità utente non privilegiata a quella kernel privilegiata; i dettagli di questo trasferimento cambiano da architettura ad architettura. Le librerie hanno il compito di ricevere gli argomenti delle chiamate di sistema e, se necessario, elaborarli nella forma specifica necessaria all'esecuzione della chiamata.

Le librerie potrebbero anche fornire versioni più complesse delle chiamate di sistema di base. Ad esempio, le funzioni per la gestione dei file con l'uso di buffer del linguaggio C sono tutte implementate nelle librerie di sistema, rendendo possibile un controllo più avanzato dell'I/O di un file, rispetto alle chiamate di sistema di base del kernel. Le librerie forniscono anche procedure che non corrispondono affatto a chiamate di sistema, come, ad esempio, algoritmi di ordinamento, funzioni matematiche, procedure di manipolazione di stringhe: tutte le funzioni necessarie al supporto dell'esecuzione di applicazioni UNIX o POSIX sono implementate nelle librerie di sistema.

Il sistema Linux include un'ampia varietà di programmi in modalità utente - utilità sia di sistema che per l'utente. Le utilità di sistema comprendono tutti i programmi necessari all'inizializzazione del sistema, come, ad esempio, i programmi per configurare le periferiche di rete, o per caricare moduli del kernel. I programmi server che sono continuamente in esecuzione fanno parte delle utilità di sistema; tali programmi gestiscono le richieste di login da parte dell'utente, le connessioni di rete in ingresso e le code di stampa.

Non tutte le utility standard forniscono funzioni chiave per l'amministrazione del sistema. L'ambiente dell'utente di UNIX contiene un gran numero di utility standard che permettono di effettuare le operazioni semplici di tutti i giorni, come la lista delle directory, lo spostamento e la cancellazione dei file o la visualizzazione del contenuto. Utility più complesse possono eseguire funzioni di elaborazione di testi, come, ad esempio, l'ordinamento di dati testuali, o l'esecuzione di ricerche di pattern all'interno di un testo in input. Insieme, queste utility formano un insieme standard che l'utente si aspetta di trovare su ogni sistema UNIX; anche se esse non eseguono nessuna funzione del sistema operativo, sono una parte importante del sistema Linux di base.

3 Moduli del kernel

Il kernel di Linux ha la capacità di caricare e scaricare sezioni arbitrarie di codice del kernel su richiesta. Questi moduli del kernel vengono eseguiti in modalità kernel mode privilegiata, e di conseguenza hanno l'accesso totale a tutte le caratteristiche hadware della macchina su cui essi vengono eseguiti. In teoria, non esistono restrizioni su quello che un modulo del kernel è abilitato a fare; solitamente, un modulo implementa un driver di periferica, un file system o un protocollo di rete.

I moduli del kernel sono vantaggiosi per molte ragioni. Il codice sorgente di Linux è gratuito, così chiunque voglia scrivere il proprio codice del kernel è in grado di compilarne uno modificato e

di riavviare il sistema per caricare la nuova funzionalità In ogni caso, la compilazione, il link e il caricamento dell'intero kernel sono un processo ingombrante da mettere in atto, se si sta sviluppando un nuovo driver. Se si utilizzano moduli del kernel, non si ha bisogno di avere un nuovo kernel per testare il nuovo driver - quest'ultimo può venire compilato autonomamente e caricato nel kernel in corso di esecuzione. Ovviamente, una volta che un nuovo driver è stato scritto, esso può essere distribuito come un modulo, così che altri utenti ne possano beneficiare senza che si renda necessario ricompilare il kernel.

Quest'ultimo punto ha un'ulteriore implicazione. Essendo coperto dalla licenza GPL, il kernel di Linux non può essere rilasciato con l'aggiunta di componenti proprietari, a meno che anche i componenti stessi siano rilasciati con questo tipo di licenza e i codici sorgente siano resi disponibili su richiesta. L'interfaccia dei moduli del kernel permette a terze parti di scrivere e di distribuire, secondo i propri termini di contratto, driver di periferica o file system che non potrebbero essere distribuiti con la licenza GPL.

I moduli kernel permettono a Linux di essere configurato con un kernel standard ridotto, senza nessuna periferica aggiuntiva collegata. Ogni driver di periferica di cui l'utente ha bisogno può essere caricata esplicitamente dal sistema al momento dell'avvio, caricata automaticamente dal sistema in caso di richiesta, e scaricata quando non più utilizzata. Ad esempio, i driver per il CD-ROM dovrebbero essere caricati quando viene inserito un CD, e scaricati quando esso viene estratto e quindi eliminato dal file system.

Il supporto ai moduli sotto Linux consta di tre componenti:

1. La **gestione dei moduli** consente ai moduli di essere caricati in memoria e di parlare con il resto del kernel.
2. La **registrazione dei driver** permette ai moduli di comunicare al resto del kernel che si è reso disponibile un nuovo driver.
3. Un **meccanismo di risoluzione dei conflitti** permette a diversi driver di periferica di riservare delle risorse hardware e di proteggere tali risorse dall'uso accidentale da parte di un altro driver.

3.1 Gestione dei moduli

Il caricamento dei moduli richiede molto più che il semplice trasferimento del contenuto binario nella memoria del kernel. Il sistema deve anche assicurarsi che ogni riferimento fatto dal modulo a simboli interni al kernel o a particolari entry point siano aggiornati in modo da far riferimento alla corretta posizione nello spazio di indirizzi del kernel. Linux affronta questa problematica di aggiornamento dei riferimenti dividendo il job di caricamento dei moduli in due sezioni separate: la gestione delle sezioni di codice del modulo all'interno della memoria del kernel, e la gestione dei simboli a cui i moduli sono autorizzati a far riferimento.

Linux mantiene una tabella di simboli interna nel kernel. Questa tabella di simboli non contiene l'intero set di simboli definiti nel kernel durante l'ultima compilazione: al contrario, un simbolo deve essere esportato esplicitamente dal kernel. Il set di simboli esportato costituisce un'interfaccia ben definita, tramite la quale un modulo è in grado di interagire con il kernel.

Nonostante il fatto che l'operazione di esportazione dei simboli venga eseguita a fronte di una richiesta esplicita del programmatore, non sono necessari particolari sforzi per importare questi simboli in un modulo. Chi scrive il modulo usa semplicemente il link esterno standard del linguaggio C: ogni simbolo esterno cui il modulo si riferisce, ma che non è dichiarato da esso, viene

segnalato come non risolto nel contenuto binario del modulo, al termine della compilazione. Quando un modulo deve venir caricato nel kernel, una utility di sistema, per prima cosa, cerca questi riferimenti non risolti nel modulo. Tutti i simboli che ancora devono essere risolti vengono ricercati nella tabella relativa del kernel, e infine l'indirizzo corretto del simbolo trovato nel kernel corrente viene sostituito nel codice del modulo.

Solo a questo punto il modulo viene passato al kernel per essere caricato. Se l'utility di sistema non riesce a risolvere un qualsiasi riferimento nel modulo, dopo aver cercato nella tabella di simboli del kernel, allora il modulo in questione viene rifiutato.

Il caricamento del modulo viene eseguito in due fasi. Per prima cosa, l'utility module loader richiede al kernel di riservare un'area di memoria virtuale contigua per il modulo. Il kernel restituisce l'indirizzo della memoria allocata, così che l'utility di caricamento possa usare l'indirizzo per posizionare il codice macchina del modulo nel punto corretto. Una seconda chiamata di sistema, in seguito, passa il modulo e l'eventuale tabella di simboli, che il modulo ha necessità di esportare al kernel. Il modulo stesso viene poi copiato nello spazio precedentemente allocato, e la tabella dei simboli del kernel è aggiornata con l'aggiunta di quelli esportati dal modulo, in modo che altri moduli ancora non caricati ne possano usufruire.

L'ultima componente nella gestione dei moduli è costituita dal richiedente del modulo. Il kernel definisce un'interfaccia di comunicazione alla quale un programma di gestione dei moduli può connettersi. Quando viene stabilita questa connessione, il kernel informa il processo di gestione del fatto che un processo ha richiesto l'accesso a un driver di periferica, file system o servizio di rete che non sono stati ancora caricati; in questo modo il gestore è ora in grado di poterlo caricare, e la richiesta del servizio ha termine a caricamento ultimato. Il processo di gestione interroga ad intervalli regolari il kernel, per avere visione di quali moduli caricati dinamicamente siano ancora in uso e per scaricare i moduli che non sono più necessari.

3.2 Registrazione dei driver

Una volta che un modulo viene caricato, non è altro che una regione di memoria isolata, se non comunica al resto del kernel quali sono le nuove funzionalità messe a disposizione. Il kernel mantiene delle tabelle dinamiche riguardo tutti i driver conosciuti, e mette a disposizione un insieme di rpocedure per permettere ai driver stessi di essere aggiunti o rimossi da queste tabelle in ogni momento. Il kernel si assicura che venga chiamata una procedura di avvio del modulo, quando questo viene caricato, e richiama la procedura di cleanup (pulitura) appena prima che il modulo venga scaricato: queste procedure sono responsabili della registrazione delle funzionalità dei moduli.

Un modulo può registrare vari tipi di driver. Un singolo modulo può registrare driver di tutti i tipi elencati sotto, e all'occorrenza può registrarne anche più di uno. Ad esempio, un driver di periferica potrebbe voler registrare due meccanismi separati per accesso ad una periferica. Le tabelle di registrazione comprendono i seguenti tipi:

- **Driver di periferica:** Questi driver includono dispositivi a carattere (come, ad esempio, tastiere, terminali e mouse) e periferiche di interfaccia di rete.
- **File system:** Il file system può essere qualunque cosa che implementi le procedure del file system virtuale di Linux. Si potrebbe implementare un formato per il salvataggio dei file su disco, o allo stesso modo gestire un file system di rete, come il NFS, o un file system virtuale il cui contenuto viene generato su richiesta, come accade nel *proc* di Linux.

- **Protocolli di rete:** Un modulo può implementare un intero protocollo di rete, come, ad esempio, l'IPX, oppure semplicemente un nuovo set di regole di filtro dei pacchetti per un firewall di rete.
- **Formato binario:** Questo formato specifica un mezzo per riconoscere e caricare un nuovo tipo di file eseguibile.

In aggiunta, un modulo può registrare un nuovo set di voci nelle tabelle *sysctl* e */proc*, al fine di consentire al modulo di essere configurato dinamicamente (Paragrafo 7.4).

3.3 Risoluzione dei conflitti

Solitamente, le implementazioni commerciali di Linux vengono vendute perché funzionino sull'hardware scelto dal venditore. Un vantaggio di questa soluzione che coinvolge un solo fornitore è che quest'ultimo ha un'idea chiara di quali configurazioni hardware siano possibili. L'hardware dei PC IBM, d'altra parte, viene venduto in una vasta gamma di configurazioni, con un gran numero di possibili driver per le periferiche come le schede di rete, i controller SCSI, e adattatori video. Il problema della gestione delle configurazioni hardware diventa più gravoso nel momento in cui sono supportati i driver di periferica modulari, poiché il set corrente di periferiche attive può variare dinamicamente.

Linux fornisce un meccanismo di risoluzione dei conflitti centrale, per organizzare arbitrariamente gli accessi alle risorse hardware. Gli obiettivi del meccanismo sono i seguenti:

- Evitare che i moduli vadano in crash durante l'accesso alle risorse hardware.
- Evitare che i driver con **autorilevazione** della configurazione della periferica interferiscano con i driver di periferica esistenti.
- Risolvere i conflitti tra i driver, nel caso in cui più di uno di essi cerchi di accedere allo stesso hardware; ad esempio, il driver della stampante parallela e il driver di rete parallel- line IP (PLIP) potrebbero tentare entrambi di comunicare con la porta della stampante parallela.

A questo scopo, il kernel mantiene delle liste delle risorse hardware allocate. Il PC ha un numero limitato di porte di I/O (gli indirizzi nello spazio di indirizzi dell'hardware di I/O), canali di interrupt e canali DMA; nel momento in cui un driver di periferica voglia accedere a una risorsa di questo tipo, quest'ultima viene riservata al driver tramite il database del kernel. Questo permette all'amministratore del sistema di determinare quale risorsa è stata allocata da un dato driver in un particolare momento.

Ci si aspetta che un modulo utilizzi questo meccanismo per riservarsi in anticipo l'accesso a una risorsa hardware che si presume abbia necessità di usare. Se la richiesta di accesso riservato viene rifiutata, perché la risorsa non è presente o è attualmente in uso, è a discrezione del modulo decidere come comportarsi. Si potrebbe pensare di far fallire l'inizializzazione e richiedere lo scaricamento del modulo, in mancanza delle condizioni per continuare, oppure proseguire ugualmente, utilizzando risorse hardware alternative.

4 Gestione dei processi

Un processo è il contesto di base all'interno del quale ogni attività richiesta da un utente viene elaborata, all'interno del sistema operativo. Per essere compatibile con altri sistemi UNIX, Linux deve utilizzare un modello di processo simile a questi ultimi. Linux si comporta in modo diverso da UNIX in pochi punti chiave, comunque. In questo paragrafo, verrà preso in esame il modello di processo dei sistemi UNIX tradizionali (Paragrafo 3.2 del capitolo online "Il Sistema Operativo FreeBSD") e si introdurrà il modello di threading di Linux.

4.1 Il modello dei processi basati su fork() e exec()

Il principio di base della gestione dei processi UNIX è quello di distinguere due operazioni: la creazione dei processi e l'esecuzione di un nuovo programma. Un nuovo processo viene creato dalla chiamata di sistema `fork`, mentre un nuovo programma viene eseguito dopo una chiamata a `execve`. Quelle appena descritte sono due funzioni separate. Un nuovo processo può essere creato tramite `fork` senza che un nuovo programma venga eseguito – il nuovo sottoprocesso continua semplicemente l'esecuzione dello stesso identico programma che il padre, che lo ha creato, stava eseguendo. Allo stesso modo, eseguire un nuovo programma non richiede che venga prima creato un processo: qualunque processo può richiamare execve in ogni momento. Il programma attualmente in esecuzione viene terminato immediatamente, e il nuovo programma comincia l'esecuzione nel contesto del processo esistente.

Questo modello ha il vantaggio di essere molto semplice. Piuttosto che dover specificare ogni dettaglio sull' l'ambiente relativo a un nuovo programma nella chiamata di sistema che lo manda in esecuzione, i nuovi programmi si eseguono nel proprio ambiente preesistente. Se un processo padre volesse modificare l'ambiente nel quale deve venire eseguito un nuovo programma, esso può richiamare `fork` e, in seguito, restando in esecuzione lo stesso programma in un processo figlio, eseguire ogni chiamata di sistema necessaria per modificare quel processo figlio prima di lanciare effettivamente il nuovo programma.

Sotto UNIX, quindi, un processo comprende tutte le informazione che il sistema operativo deve conservare per tracciare il contesto della singola esecuzione di un singolo programma. Sotto Linux, è possibile spezzare questo contesto in più sezioni specifiche. Nel generico, le proprietà dei processi appartengono a tre gruppi: l'identità, l'ambiente e il contesto.

4.1.1 Identità dei processi

L'identità di un processo consiste principalmente nei seguenti elementi:

- **Process ID (PID):** Ad ogni processo viene assegnato un identificatore univoco. I PID vengono utilizzati per specificare un processo al sistema operativo quando un applicazione esegue una chiamata di sistema per segnalare, modificare o restare in attesa di un altro processo. Ulteriori identificatori vengono utilizzati per associare un processo a un gruppo (tipicamente, un albero di processi su cui è stato eseguito `fork` da un singolo comando di un utente) e ad una sessione di login.

- **Credenziali:** Ogni processo deve avere uno user ID associato e uno o più group ID (i gruppi di utenti sono discussi nel Paragrafo 11.6.2; i gruppi di processi non sono descritti in questa sede) che determina i diritti che un processo ha di accedere alle risorse del sistema e ai file.
- **Personalità:** La personalità dei processi non viene tradizionalmente implementata nei sistemi UNIX, ma sotto Linux ad ogni processo viene assegnato un identificatore di personalità, che può modificare leggermente la semantica di determinate chiamate di sistema. Le personalità sono principalmente usate dalle librerie di emulazione, al fine di richiedere che le chiamate di sistema siano compatibili con determinate caratteristiche di UNIX.

La maggior parte di questi identificativi sono sotto il controllo limitato del processo stesso. Il gruppo di processi e l'identificativo di sessione possono essere modificati se il processo è sul punto di eseguire un nuovo gruppo o sessione. Le proprie credenziali possono venir modificate, a patto che si superino determinati test di sicurezza. Comunque, il PID primario di un processo non è modificabile e identifica il processo stesso fino al suo termine.

4.1.2 Ambiente dei processi

L'ambiente di un processo viene ereditato dal proprio processo padre, ed è composto da due vettori terminati da null: il vettore degli argomenti e quello dell'ambiente. Il **vettore degli argomenti** contiene semplicemente una lista degli argomenti usati nella linea di comando per eseguire il programma corrente, e solitamente comincia con il nome del programma stesso. Il **vettore di ambiente** è una lista di coppie nome-valore che associa nomi di variabili di ambiente a valori testuali arbitrari. L'ambiente non viene mantenuto nella memoria del kernel, ma immagazzinato nello spazio di indirizzi in modalità user-mode del processo stesso come prima informazione, sulla base dello stack del processo.

I vettori degli argomenti e dei processi non vengono alterati quando un processo viene creato: il nuovo figlio erediterà l'ambiente posseduto dal processo padre. Ad ogni modo, viene configurato un nuovo ambiente al lancio dell'esecuzione di un nuovo programma. Al momento della chiamata ad execve, un processo deve fornire le informazioni dell'ambiente per il nuovo programma. Il kernel passa queste variabili di ambiente al prossimo programma, sostituendo l'ambiente corrente del processo. In caso contrario, il kernel ignora i vettori di ambiente e della linea di comando: la loro implementazione è lasciata interamente alle librerie user-mode e alle applicazioni.

Il passaggio delle variabili di ambiente da un processo a quello successivo, e l'ereditarietà di tali variabili da parte del figlio di un processo, garantiscono un metodo flessibile per passare informazioni a componenti del software di sistema user-mode. Numerose variabili di ambiente importanti hanno significati convenzionali, collegati ad altre parti del software di sistema. Ad esempio, la variabile `TERM` è utilizzata per dare un nome del tipo di terminale connesso alla sessione di login di un utente. Molti programmi utilizzano questa variabile per determinare in che modo eseguire determinate operazioni sul display dell'utente, come il movimento del cursore o lo scorrimento di un'area di testo. I programmi con supporto multilingua utilizzano la variabile `LANG` per determinare in che lingua visualizzare i messaggi di sistema per i programmi che prevedono il supporto multilingua.

Il meccanismo delle variabili di ambiente altera il sistema operativo a livello di singolo processo, piuttosto che per l'intero sistema. Gli utenti possono scegliere il proprio linguaggio o selezionare i propri editor indipendentemente gli uni dagli altri.

4.1.3 Contesto dei processi

L'identità del processo e le proprietà di ambiente sono solitamente configurate quando il processo viene creato, e non vengono più modificate finché il processo non termina. Un processo può scegliere di modificare alcuni aspetti della propria identità, se è necessario fare ciò, oppure può modificare il proprio ambiente. Il contesto del processo, d'altra parte, rappresenta lo stato del programma in esecuzione in qualsiasi momento; dunque esso cambia continuamente.

- **Contesto di schedulazione.** La parte più importante del contesto del processo è il contesto di schedulazione: le informazione di cui lo schedulatore ha bisogno per sospendere e fare riprendere un processo. Queste informazioni comprendono delle copie di tutti i registri del processo. I registri floating-point sono immagazzinati separatamente e sono utilizzati solo quando necessario, così che i processi che non utilizzano il floating point non debbano aver l'onere di salvare quel determinato stato. Il contesto di schedulazione include anche informazioni riguardo la priorità di schedulazione e su qualsiasi segnale eccezionale che aspetta di essere consegnato al processo. Una parte chiave del contesto di schedulazione è lo stack del kernel del processo: un'area separata di memoria del kernel riservata all'esecuzione esclusiva di codice kernel-mode. Sia le chiamate di sistema che gli interrupt che si verificano mentre il processo è in esecuzione faranno uso di questo stack.
- **Accounting.** Il kernel mantiene delle informazioni riguardo le risorse attualmente in uso da ciascun processo, e il totale delle risorse utilizzate dal processo nella sua storia, fino al momento corrente.
- **Tabella dei file.** La tabella dei file è un vettore di puntatori a strutture di file nel kernel. Quando vengono eseguite chiamate di sistema per l'I/O su file, i processi creano dei riferimenti ai file grazie agli indici presenti in questa tabella.
- **Contesto del file-system.** Mentre la tabella dei file rappresenta una lista dei file attualmente aperti, il contesto del file system entra in gioco per richiedere l'apertura di nuovi file. La root corrente e le directory di default da usare per le ricerche di file sono immagazzinate qui.
- **Tabella di gestione dei segnali.** I sistemi UNIX permettono la trasmissione di segnali asincroni verso i processi, in risposta a vari eventi esterni. La tabella di gestione dei segnali definisce la procedura da richiamare, nello spazio di indirizzi del processo, all'arrivo di un segnale specifico in ingresso.
- **Contesto della memoria virtuale.** Il contesto della memoria virtuale descrive il contenuto completo di uno spazio di indirizzi privato di un processo; questo argomento verrà discusso nel Paragrafo 6.

4.2 Processi e thread

La maggior parte dei sistemi operativi moderni prevede il supporto di processi e thread. Anche se le differenze esatte tra i due dipende spesso dall'implementazione, le distingueremo nel seguente modo: i *processi* rappresentano l'esecuzione di programmi singoli, mentre i *thread* rappresentano

esecuzioni separate e concorrenti all'interno di un singolo processo che esegue un singolo programma.

Ciascun processo ha a disposizione il proprio spazio di indirizzi, anche se più processi possono utilizzare la memoria condivisa per rendere comune una parte (ma non la totalità) della propria memoria virtuale. In contrasto, due thread all'interno dello stesso processo condivideranno *lo stesso* spazio di indirizzi, e non uno *simile*. Qualsiasi modifica alla struttura della memoria virtuale eseguita da un thread sarà visibile immediatamente agli altri thread del processo, poiché essi sono in esecuzione all'interno dello stesso spazio di indirizzi.

I thread possono essere implementati in diversi modi. L'implementazione può essere definita nel kernel del sistema operativo, come un oggetto di proprietà di un processo, o essere un'entità completamente indipendente; Può anche non essere affatto definita nel kernel: i thread possono essere implementati all'interno del codice di applicazioni o librerie, con l'aiuto dei timer messi a disposizione dal kernel.

Il kernel di Linux gestisce con facilità la differenza tra processi e thread, utilizzando la stessa rappresentazione interna per entrambi, ed infatti utilizza generalmente il termine *task* per definire entrambi, dal momento che non fa una reale distinzione fra i due. Un thread è solamente un nuovo processo che condivide lo stesso spazio di indirizzi del processo padre. La distinzione tra le due entità viene fatta solamente quando è creato un nuovo thread, attraverso la chiamata di sistema clone. Mentre `fork` crea un nuovo processo che presiede un proprio contesto, completamente nuovo, `clone` crea un nuovo processo che ha una propria identità, ma è abilitato a condividere le strutture dati del proprio padre.

La distinzione può essere fatta poiché Linux non mantiene l'intero contesto del processo all'interno della struttura dati principale del processo stesso; piuttosto, tale contesto viene mantenuto all'interno di sottocontesti indipendenti. Il contesto di file-system, le tabelle di descrizione dei file e di gestione dei segnali, e il contesto della memoria virtuale sono mantenuti in strutture dati separate. La struttura dati del processo contiene semplicemente puntatori a queste strutture separate, così che un numero qualsiasi di processi possa condividere uno stesso sottocontesto puntando a quello appropriato.

La chiamata di sistema `clone` accetta un parametro che specifica quali sottocontesti copiare e quali condividere, al momento della creazione di un nuovo processo. A quest'ultimo viene sempre assegnata una nuova identità e un nuovo contesto di scheduling; a seconda dei parametri passati, comunque, il processo può creare nuove strutture dati di sottocontesto, e inizializzarle in modo che siano una copia di quelle del padre, oppure configurare il nuovo processo perché vangano utilizzate le stesse strutture dati del sottocontesto utilizzate dal padre. La chiamata di sistema `fork` non è altro che un caso particolare di `clone` in cui vengono copiati tutti i sottocontesti, e nessuno di essi viene condiviso. L'utilizzo di `clone` rende possibile un controllo preciso di tutto ciò che due thread hanno in condivisione. I flag che compongono il parametro passato alla chiamata `clone` sono:

- `CLONE_FS` : per la condivisione di informazioni riguardanti il file system.
- `CLONE_VM` : per la condivisione dello stesso spazio di memoria.
- `CLONE_SIGHAND` : per la condivisione dei gestori dei segnali.
- `CLONE_FILE` : per la condivisione dell'insieme dei file aperti.

I processi creati con la chiamata `clone` ricevono sempre una nuova identità ed un nuovo contesto di schedulazione; secondo gli argomenti passati, comunque, possono creare nuove strutture dati dei sottocontesti, inizializzate per essere copie di quelle del padre, o impostare il nuovo processo affinché utilizzi le stesse strutture dati del sottocontesto, usate dal padre.

I gruppi di lavoro POSIX hanno definito un'interfaccia di programmazione, specificata nello standard POSIX.1c, per rendere possibile l'esecuzione di thread multipli da parte delle applicazioni.

Le librerie di sistema di Linux supportano due meccanismi separati che implementano questo standard in modi diversi. Un'applicazione può scegliere se utilizzare il package di threading basato sulla modalità utente o sulla modalità kernel. La libreria di thread in modalità utente evita al kernel i lavori supplementari di schedulazione e delle chiamate di sistema relative ai thread, ma è limitata dal fatto che tutti i thread vengono eseguiti in un singolo processo. La libreria di thread supportata dal kernel utilizza la chiamata di sistema `clone` per implementare la stessa interfaccia di programmazione, ma, dato che vengono creati numerosi contesti di schedazione, ha il vantaggio di rendere possibile, da parte di un'applicazione, l'esecuzione di thread su più processori allo stesso tempo, su sistemi multiprocessore. In questo modo è anche possibile l'esecuzione simultanea di chiamate di sistema del kernel da parte di più thread.

5 Schedulazione

La schedulazione è l'attività di allocare il tempo della CPU a compiti diversi all'interno di un sistema operativo. Normalmente pensiamo alla schedulazione come all'esecuzione e all'interruzione dei processi, ma c'è un altro aspetto importante della schedulazione in Linux: l'esecuzione delle varie attività del kernel, che comprendono sia quelle richieste dai processi in esecuzione, sia quelle eseguite internamente per conto dei driver delle periferiche.

5.1 La sincronizzazione del kernel

Il modo in cui il kernel schedula le proprie operazioni è fondamentalmente diverso da quello col quale processa la schedulazione. Una richiesta di esecuzione in modalità kernel può avvenire in due modi distinti. Un programma in esecuzione può richiedere un servizio del sistema operativo sia esplicitamente, tramite una chiamata di sistema, che implicitamente – ad esempio, quando si verifica un page fault. In alternativa, un driver di periferica può scatenare un interrupt hardware che provoca l'esecuzione da parte della CPU dell'handler definito dal kernel per quel particolare interrupt.

Il problema che si pone al kernel è dato dal fatto che tutti questi task potrebbero cercare di accedere alle stesse strutture dati interne. Se un task del kernel è nel mezzo di un accesso ad una struttura dati, e in quel momento viene eseguita una procedura di servizio di un interrupt, allora quella procedura non può accedere o modificare gli stessi dati senza rischiare di danneggiarli. Questo fatto si collega all'idea di sezioni critiche: le porzioni di codice che accedono a dati condivisi e che non devono essere autorizzate all'esecuzione concorrente.Come risultato, la sincronizzazione del kernel implica molto più che la sola elaborazione della schedulazione. È necessario un framework (struttura) che consenta l'esecuzione di una sezione critica del kernel, senza violare l'integrità dei dati condivisi.

Analizziamo ora l'approccio a questo problema, come è realizzato nel kernel 2.2. La prima parte della soluzione al problema, adottata da Linux, risiede nel rendere il normale codice del kernel non interrompibile (non-preemptable). Solitamente, quando il kernel riceve un interrupt dal timer, viene richiamato lo schedulatore di processi, così da poter potenzialmente sospendere il processo attualmente in esecuzione e riprendere l'esecuzione di un altro – attuando il sistema di condivisione del tempo, naturale di qualsiasi sistema UNIX. Comunque, quando l'interrupt del timer viene ricevuto nel momento in cui un processo sta eseguendo una procedura di servizio di sistema, la rischedulazione non viene messe in atto immediatamente. Piuttosto, viene settato il flag `need_resched` del kernel, per segnalare la necessità di eseguire lo schedulatore dopo che la

chiamata di sistema è stata completata, quando il controllo è in procinto di tornare alla modalità utente.

Una volta che un pezzo di codice kernel inizia l'esecuzione, è garantito che esso sarà l'unico di questo tipo, finché non si verificherà una delle seguenti situazioni:

- un interrupt,
- un page fault,
- una chiamata del codice kernel alla funzione di schedulazione stessa.

Gli interrupt rappresentano un problema solamente nel momento in cui contengono sezioni critiche. Gli interrupt del timer non causano mai direttamente la rischedulazione di un processo. Richiedono solamente che tale schedulazione venga eseguita successivamente, così che un qualsiasi interrupt non possa influire sull'ordine di esecuzione del codice del kernel non interrompibile. Una volta che il servizio di interrupt termina, l'esecuzione ritorna semplicemente allo stesso codice del kernel in esecuzione nel momento in cui l'interrupt era stato ricevuto.

I page fault sono problemi potenziali; se una procedura del kernel tenta di leggere o scrivere nella memoria utente, può verificarsi un page fault che richieda che l'I/O su disco venga completato; il processo in esecuzione sarà sospeso fino a che questo l'operazione in corso non viene portata a termine. Allo stesso modo, se, mentre si è in modalità kernel, una procedura di servizio di una chiamata di sistema invoca lo schedulatore (sia esplicitamente, richiamando direttamente il codice dello schedulatore, sia implicitamente, richiamando una funzione che attenda che l'I/O venga completato), allora il processo verrà sospeso e si verificherà una rischedulazione. Quando il processo diventa nuovamente eseguibile, esso continuerà l'esecuzione in modalità kernel, continuando dall'istruzione successiva a quella che ha causato la chiamata allo schedulatore.

Si può assumere, quindi, che il codice kernel non sarà mai interrotto da un altro processo, e che non è necessaria un'attenzione particolare per proteggere le sezioni critiche. La sola richiesta che viene fatta è che le sezioni critiche non contengano riferimenti alla memoria dell'utente o che siano in attesa del completamento di I/O.

La seconda tecnica di protezione utilizzata da Linux si applica alle sezioni critiche che si verificano all'interno delle procedure di servizio degli interrupt. Il mezzo principale, mediante il quale viene gestita questa situazione, consiste nell'hardware di controllo degli interrupt del processore. Disabilitando gli interrupt durante una sezione critica, il kernel garantisce il proseguimento delle operazioni senza il rischio di accessi concorrenti alle strutture dati condivise.

Esiste uno svantaggio nel disabilitare gli interrupt. Sulla maggior parte delle architetture hardware, le istruzioni di attivazione e disattivazione degli interrupt sono dispendiose. Inoltre, fin quando gli interrupt rimangono disattivati, qualunque tipo di I/O è disabilitato, e ogni periferica in attesa di un servizio rimarrà in questo stato fino a che gli interrupt non saranno riabilitati; in questo modo le prestazioni si degradano. Il kernel di Linux utilizza un'architettura di sincronizzazione che consente l'esecuzione di lunghe sezioni critiche, per tutta la loro durata, e senza che gli interrupt siano disabilitati. Questa caratteristica è utile in special modo nel codice di rete: un interrupt in un driver di una periferica di rete può segnalare l'arrivo di un intero pacchetto di rete, e la qual cosa può portare a una grossa mole di codice da eseguire per disassemblare, instradare e inoltrare il pacchetto all'interno della procedure di servizio dell'interrupt.

Linux implementa questa architettura tramite la separazione delle procedure di servizio degli interrupt in due sezioni: la metà superiore (top half) e la metà inferiore (bottom half). La **metà superiore è** una procedura di servizio di interrupt normale, e viene eseguita con gli interrupt ricorsivi disabilitati; gli interrupt a più alto livello possono comunque interrompere la procedura, ma quelli di uguale o minore priorità sono disabilitati. La **metà inferiore** di una procedura di servizio viene eseguita, con tutti gli interrupt abilitati, tramite uno schedulatore "in miniatura", che assicura

che le metà inferiori non si interrompano tra loro. Lo schedulatore della metà inferiore viene chiamato automaticamente all'uscita di una procedura di servizio di interrupt.

Questa separazione significa che una qualsiasi elaborazione complessa che deve essere compiuta in risposta a un interrupt può essere completata dal kernel senza che quest'ultimo si preoccupi del fatto che qualcuno possa interromperlo. Se si verifica un altro interrupt mentre una metà inferiore è in esecuzione, quello stesso interrupt può richiedere che venga eseguita la stessa metà inferiore; la richiesta sarà esaudita quando l'istanza attualmente in esecuzione sarà terminata. Ogni esecuzione di una metà inferiore può essere interrotta da una metà superiore, ma mai da un'altra metà inferiore.

L'architettura "metà inferiore / metà superiore" viene completata da un meccanismo che consente la disabilitazione di determinate metà inferiori durante l'esecuzione di normale codice del kernel in foreground. Il kernel, in questo modo, può gestire le sezioni critiche con facilità: gli handler degli interrupt possono codificare le proprie come metà inferiori; quando il kernel, in foreground, entra in una sezione critica, esso disabilita le metà inferiori rilevanti per evitare che altre sezioni critiche possano interromperlo. Alla fine dell'esecuzione di questa sezione, il kernel può riabilitare le metà inferiori ed eseguire i task che sono stati accodati, durante la criticità, dalle procedure di servizio di interrupt delle metà superiori.

La figura 2 riassume i vari livelli di protezione degli interrupt all'interno del kernel. Ogni livellopuò essere interrotto da codice eseguito a un più alto livello, ma mai da codice eseguito a un livellouguale o minore; con l'eccezione del codice in modalità utente, i processi dell'utente possono esseresempre interrotti a favore di un altro processo, nel momento in cui si verifica un interrupt di schedulazione del time-sharing.

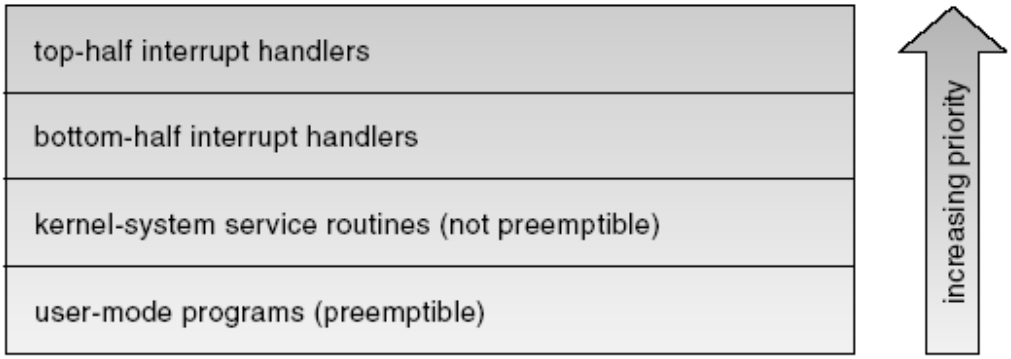

Figura 2. Livelli di protezione degli interrupt.

top–half interrupt handlers = handler dell'interrupt della metà superiore
bottom–half interrupt handlers = handler dell'interrupt della metà inferiore
kernel – system service routines (not preemptible) = routine di servizio del kernel (senza sospensione)
user–mode programs (preemptible) = programmi in modalità utente (sospendibili)
increasing priority = priorità in aumento

Con la versione 2.6, il kernel di Linux è diventato completamente preemptive (interrompibile), quindi un task può essere interrotto mentre è in esecuzione in modalità kernel.

Il kernel di Linux fornisce spinlock e semafori (insieme alla versione lettore-scrittore di questi due meccanismi) per i lock del kernel. Sulle macchine SMP, il meccanismo fondamentale di lock è lo spinlock; il kernel è progettato in modo che gli spinlock siano tenuti solo per brevi periodi. Sulle macchine a singolo processore, gli spinlock sono un meccanismo inappropriato e sono sostituiti dall'abilitazione o disabilitazione della interrompibilità del kernel, ossia: su macchine a processore singolo un task disabilita l'interrompibilità del kernel, invece di usare uno spinlock, e la riabilita quando dovrebbe rilasciare lo spinlock. Questo comportamento è riassunto così:

processore singolo	multi processore
disabilitazione dell'interrompibilità del kernel	acquisizione di uno spinlock
abilitazione dell'interrompibilità del kernel	rilascio dello spinlock

Linux utilizza un approccio interessante all'abilitazione e disabilitazione dell'interrompibilità del kernel, fornendo due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`, per la disabilitazione e l'abilitazione. Inoltre il kernel non è interrompibile se un task del kernel possiede un lock. Per rafforzare questa regola, ogni task del sistema ha una struttura `thread-info` che include il campo `preempt_count`, che è un contatore del numero di lock posseduti dal task, e viene incrementato all'acquisizione di un lock, e decrementato al suo rilascio. Se il campo `preempt_count` del task correntemente in esecuzione è maggiore di zero, interrompere il kernel è una operazione non sicura, poiché il task in esecuzione possiede un lock. Se il contatore è invece a zero, il kernel può essere interrotto in sicurezza, assumendo che non ci siano chiamate in esecuzione di `preempt_disabile()`.

Gli spinlock, come l'abilitazione e la disabilitazione dell'interrompibilità del kernel, sono utilizzati nel kernel solo quando i lock sono tenuti per un breve periodo, altrimenti vengono usati i semafori.

La seconda tecnica di protezione usata da Linux, si applica alle sezioni critiche delle procedure di gestione degli interrupt. Lo strumento principale è l'hardware del processore di controllo degli interrupt. Disabilitando gli interrupt (o usando gli spinlock) durante una sezione critica, il kernel garantisce di poter procedere senza il rischio di accesso concorrente alle strutture dati condivise.

5.2 Schedulazione dei processi

Nel kernel 2.2, una volta che il kernel raggiunge un punto di rischedulazione – se si è verificato un interrupt o un processo del kernel in esecuzione è bloccato in attesa di un segnale di wakeup – esso deve decidere quale processo eseguire successivamente. Linux dispone di due algoritmi separati per la schedulazione dei processi. Il primo è un algoritmo di time-sharing equo per la schedulazione preemptive tra più processi; l'altro è progettato per i task in real time, nei quali le priorità assolute sono più importati dell'equità.

Una parte dell'identità di ciascun processo è costituita da una classe di schedulazione che definisce quali di questi algoritmi devono essere applicati al processo stesso. La classe di schedulazione usata da Linux è definita nell'estensione per l'elaborazione real time dello standard POSIX (POSIX.4, oggi conosciuto come POSIX.1b).

Per i processi con time sharing, Linux utilizza un algoritmo basato su priorità e crediti. Ogni processo possiede un certo numero di crediti di schedulazione; quando è necessario decidere quale

task deve essere eseguito, viene scelto il processo con il maggior credito. Ogni volta che si verifica un interrupt timer, il processo attualmente eseguito perde un credito; quando il numero di crediti arriva a zero, il processo stesso viene sospeso e ne viene scelto un altro.

Se non esistono processi eseguibili che possiedono crediti, allora Linux esegue un'operazione di riaccredito, aggiungendone ad ogni processo nel sistema, e non solo a quelli eseguibili, secondo la seguente regola:

$$\text{crediti} = \frac{\text{crediti}}{2} + \text{priorità}$$

Questo algoritmo tende a elaborare insieme due fattori: la storia del processo e la priorità. Metà dei crediti che appartengono al processo, a partire dall'ultimo riaccredito, saranno mantenuti dopo che l'algoritmo è stato applicato, conservando così uno storico del comportamento recente del processo. I processi continuamente in esecuzione tendono ad esaurire rapidamente i propri crediti, ma allo stesso modo, quelli che restano sospesi per molto tempo riescono ad accumulare crediti in molteplici operazioni di riaccredito, trovandosi, di conseguenza, con un numero di crediti più elevato al termine dell'operazione. Questo sistema di crediti assegna automaticamente un'alta priorità a processi interattivi o limitati dall'I/O, per i quali è importante che ci siano tempi di risposta rapidi.

L'utilizzo ai fini del calcolo dei nuovi crediti consente alla priorità del processo di essere regolata con precisione. I job di tipo batch eseguiti in background avranno una bassa priorità; essi riceveranno automaticamente meno crediti, rispetto ai job interattivi degli utenti, e quindi sarà ad essi riservata una percentuale più bassa di CPU time, nei confronti dei processi a priorità più elevata. Linux utilizza questo sistema per implementare il meccanismo standard di priorità dei processi di UNIX , chiamato *nice*.

Nella versione 2.5 del kernel di Linux l'algoritmo di schedulazione utilizzato per le procedure e i task a condivisione di tempo è stato fortemente migliorato e viene eseguito con complessità computazionale costante, indipendentemente dal numero di task nel sistema. Il nuovo schedulatore fornisce anche un supporto migliorato per SMP, fra cui l'affinità per un processore e il bilanciamento del carico, oltre al mantenimento dell'equità ed al supporto per i task interattivi.

Lo schedulatore di Linux è un algoritmo preemptive basato sulla priorità con due gruppi di priorità separati: il gruppo **real time** da 0 a 99 e un valore **nice** (gentile) che va da 100 a 140. Questi due gruppi si uniscono in uno schema globale di priorità dove i valori più bassi indicano priorità più alte. A differenza degli schedulatore di molti altri sistemi, quello di Linux assegna ai task ad alta priorità quantum di tempi più lunghi e viceversa. A causa della natura unica dello schedulatore, questo è appropriato per Linux, come vedremo presto. La relazione tra le priorità e le porzione di tempo è mostrata in figura 3.

Un task eseguibile è considerato eleggibile per l'esecuzione sulla CPU fintanto che ha tempo rimanente nella sua porzione. Quando un task ha terminato la sua porzione viene considerato **expired** (spirato) e non è eleggibile per l'esecuzione fintanto che tutti gli altri task non hanno esaurito il loro quantum di tempo. Il kernel mantiene una lista di tutti i task eseguibili in una coda di esecuzione (**runqueue**). Per il supporto all'SMP, ciascun processore mantiene la sua coda di esecuzione e si schedula indipendentemente. Ogni coda contiene array di due priorità: attivi e spirati. L'array degli attivi contiene tutti task che hanno tempo rimanente nella loro porzione, mentre quello degli spirati contiene tutti i task spirati. Ciascuno di questi array contiene una lista di task indicizzati secondo la priorità (figura 4) e lo schedulatore sceglie il task con la priorità maggiore dall'array degli attivi per eseguirlo sulla CPU. Sulle macchine multiprocessore questo significa che ogni processore schedula il task con la priorità più alta dalla sua coda di esecuzione. Quando tutti i task hanno esaurito la loro porzione di tempo (cioè l'array degli attivi è vuoto), i due array vengono scambiati. Ai task viene assegnata una priorità dinamica basata sul valore di *nice* più o meno un valore sino a 5 basato

sull'interattività del task. L'interattività di un task è determinata da quanto a lungo ha dormito in attesa di I/O: i task più interattivi tipicamente hanno durate di sleep più lunghe e quindi è più facile che abbiano una correzione vicina a –5, poiché lo schedulatore favorisce i task interattivi. Al contrario quelli con tempi più brevi sono spesso legati alla CPU e pertanto la loro priorità verrà abbassata. La rivalutazione della priorità dinamica di un task avviene all'esaurirsi del quantum di tempo, quando deve essere mosso nell'array degli spirati.

priorità numerica	priorità relativa		quantum di tempo
0	più alto	task real time	200 ms
•			
•			
•			
99			
100		altri task	
•			
•			
•			
140	più basso		10 ms

Figura 3. Relazione fra priorità e lunghezza della porzione di tempo.

array degli attivi		**array degli spirati**	
priorità	lista dei task	priorità	lista dei task
[0]	○—○	[0]	○—○—○
[1]	○—○—○	[1]	○
•	•	•	•
•	•	•	•
•	•	•	•
[140]	○	[140]	○—○

Figura 4. Lista dei task indicizzati secondo la priorità.

Sia nel kernel 2.2 che nel kernel 2.6, la schedulazione in real time di Linux è ancora più semplice. Linux implementa le due classi di scheduling in real time richieste dallo standard POSIX.1b: il first-come, first-served (FCFS) e il round-robin (Paragrafi 6.3.1 e 6.3.4, rispettivamente). In entrambi i casi, ciascun processo ha una priorità associata, in aggiunta alla classe di scheduling. Nello scheduling in time-sharing, comunque, i processi con diversa priorità possono competere tra loro, entro un certo limite; nello scheduling in real time, invece, lo scheduler esegue sempre per primo il processo con la priorità più alta. Se esistono processi con la stessa priorità associata, allora viene eseguito il processo in attesa da più tempo. L'unica differenza tra lo scheduling FCFS e il round-robin

è che i processi gestiti con FCFS continuano l'esecuzione fino a che essi terminano o si bloccano, mentre un processo gestito da round-robin verrà bloccato dopo un certo periodo, e sarà spostato in coda nella lista di scheduling, così che i processi gestiti con round-robin attueranno automaticamente tra loro la strategia di time sharing.

Lo scheduling in real time di Linux è di tipo leggero. Lo scheduler offre precise garanzie riguardo le priorità relative dei processi real time, ma il kernel, invece, non offre alcuna certezza riguardo la velocità con cui un processo verrà schedulato, dal momento in cui esso è nelle condizioni di essere eseguito.

È necessario ricordare che il codice del kernel di Linux, nelle versioni precedenti al 2.6, non può mai essere bloccato dal codice in modalità utente. Se si verifica un interrupt che "risveglia" un processo real time mentre il kernel sta eseguendo una chiamata di sistema per conto di un altro processo, il processo real time dovrà semplicemente aspettare che la chiamata di sistema in corso completi l'esecuzione o si blocchi.

5.3 Multiprocessing simmetrico

Il kernel 2.0 di Linux fu il primo a supportare in maniera stabile l'hardware per il **multiprocessing simmetrico (SMP)**. Singoli processi o thread possono essere eseguiti in parallelo su processori separati. Comunque, per rispondere al requisito di una sincronizzazione non preemptive del kernel, l'implementazione del SMP in questa versione impone la restrizione che un solo processore alla volta possa eseguire codice in modalità kernel. L' SMP utilizza un singolo spinlock del kernel (lo spinlock è un tipo di lock in cui un thread richiede ciclicamente l'accesso a una risorsa) per mettere in pratica questo vincolo. Questo spinlock non crea problemi nel caso di task di elaborazione, mentre, nel caso in cui sia necessaria un'abbondante attività da parte del kernel, si possono creare dei colli di bottiglia rilevanti.

Il kernel 2.2 di Linux rende l'implementazione del SMP più scalabile, tramite la divisione dello spinlock singolo del kernel in molteplici lock, ognuno dei quali protegge dal rientro (l'esecuzione contemporanea dello stesso codice da più entità) solo una piccola parte delle strutture dati del kernel. Attualmente, la gestione dei segnali, gli interrupt ed alcune procedure di I/O utilizzano lock multipli per permettere a più processori di eseguire simultaneamente codice in modalità kernel.

Il kernel 2.6 fornisce ulteriori miglioramenti all'SMP, fra cui l'affinità per i processori ed il bilanciamento del carico.

6 Gestione della memoria

La gestione della memoria in Linux ha due componenti: la prima riguarda l'allocazione e la deallocazione della memoria fisica: pagine, gruppi di pagine, e piccoli blocchi di memoria; la seconda si occupa della memoria virtuale, che è la memoria mappata nello spazio di indirizzamento dei processi in esecuzione.

Descriviamo queste due componenti, e poi esaminiamo il meccanismo tramite il quale i componenti caricabili di un nuovo programma sono portati nella memoria virtuale di un processo come risultato di una chiamata di sistema `exec`.

6.1 Gestione della memoria fisica

A seguito di caratteristiche dell'hardware, Linux separa la memoria fisica in tre **zone** differenti che identificano regioni di memoria differenti: `ZONE_DMA`, `ZONE_NORMAL`, e `ZONE_HIGHMEM`.

Queste zone sono specifiche dell'architettura. Per esempio, nell'architetture dell'80x86, certe periferiche ISA (Industry Standard Architecture) possono accedere solo ai sedici MB di memoria fisica utilizzando il DMA. In questi sistemi, i primi sedici MB di memoria fisica contengono lo `ZONE_DMA`. `ZONE_NORMAL` identifica la memoria fisica mappata nello spazio degli indirizzi della CPU ed è utilizzata per molte richieste di memoria delle procedure. Per le architetture che non limitano l'accesso del DMA, `ZONE_DMA` non esiste e viene utilizzato `ZONE_NORMAL`. Infine `ZONE_HIGHMEM` (sta per "memoria alta") si riferisce alla memoria fisica che non è mappata nello spazio di indirizzi del kernel. Per esempio nell'architettura Intel a 32 bit (dove 2^{32} genera uno spazio di indirizzi di 4 GB), il kernel è mappato nei primi 896 MB dello spazio degli indirizzi, mentre ci si riferisce alla memoria rimanente come **memoria alta** ed è allocata da `ZONE_HIGHMEM`. La relazione tra le zone e la memoria fisica nell'architettura 80x86 è mostrata nella figura 5. Il kernel mantiene una lista di pagine libere per ogni zona; quando arriva una richiesta di memoria fisica, il kernel la soddisfa utilizzando la zona appropriata.

zona	memoria fisica
`ZONE_DMA`	< 16 MB
`ZONE_NORMAL`	16..896 MB
`ZONE_HIGHMEM`	> 896 MB

Figura 5. Reazione fra le zone e gli indirizzi fisici nell'Intel 80x86.

Il gestore principale della memoria fisica nel kernel di Linux è il **page allocator**. Tale componente è responsabile dell'allocazione e della cancellazione di tutte le pagine fisiche, ed è in grado di allocare su richiesta dei range di pagine fisicamente contigue. L'allocatore usa un **algoritmo buddy-heap** per tener traccia delle pagine fisiche disponibili. Un algoritmo di questo tipo accoppia unità adiacenti di memoria disponibile (da qui il nome, buddy [compagno] heap [accumulare]). Ogni regione di memoria allocabile ha un partner adiacente (o buddy), e nel momento in cui due regioni partner vengono liberate, esse vengono combinate in una unica e più grande. Tale regione risultante avrà anch'essa un partner, col quale si potrà combinare per formare un'area di memoria ancora più grande. Alternativamente, se una modesta richiesta di memoria non può venir soddisfatta dall'allocazione di una piccola regione di memoria esistente, allora un'area di memoria più grande verrà suddivisa in due regioni partner, al fine di soddisfare la richiesta.

Per la registrazione di zone di memoria libera di qualsiasi dimensione consentita, vengono utilizzate diverse liste collegate; sotto Linux, la dimensione minima allocabile con questo meccanismo è pari alla dimensione di una pagina singola. La figura 6 mostra un esempio di allocazione col meccanismo buddy-heap: sta per essere allocata una regione di 4 KB, ma la più piccola regione disponibile è di 16 KB. La regione, quindi, viene spezzata ricorsivamente, finché non viene recuperato un segmento della dimensione desiderata.

In definitiva, tutte le operazioni di allocazione di memoria nel kernel di Linux avvengono sia in maniera statica, tramite i driver che riservano un'area di memoria contigua durante il boot del sistema, che dinamicamente, tramite il page allocator. Comunque, le funzioni del kernel non hanno

necessità di utilizzare l'allocatore di base al fine di riservare la memoria. Diversi sottosistemi di gestione della memoria utilizzano il page allocator sottostante per gestire i propri pool di memoria. I più importanti sono i sistemi di memoria virtuale, descritti nel paragrafo 6.2; l'allocatore a lunghezza variabile `kmalloc`; i due sistemi di data cache del kernel: il cache di buffer e di pagina.

Molti componenti del sistema operativo Linux hanno necessità di allocare intere pagine all'occorrenza, ma spesso vengono richiesti blocchi di memoria più piccoli. Il kernel mette a disposizione un allocatore addizionale per richieste di dimensioni arbitrarie, in cui la dimensione stessa della richiesta non è conosciuta in anticipo, e potrebbe essere di pochi bytes piuttosto che di un'intera pagina. Questo servizio, **`kmalloc`**, analogo alla funzione `malloc` del linguaggio C, alloca intere pagine, su richiesta, ma successivamente le divide in pezzi più piccoli. Il kernel mantiene un insieme di liste di pagine attualmente utilizzate dal servizio `kmalloc`, dove tutte le pagine di una data lista sono state divise in pezzi di una determinata dimensione. L'allocazione della memoria implica il processo di elaborazione della lista appropriata, e successivamente, la scelta di recuperare la prima area libera disponibile nella lista, o di allocare una nuova pagina e di dividerla.

Le regioni di memoria richieste dal sistema **`kmalloc`** vengono allocate permanentemente finché non vengono liberate esplicitamente. Il sistema **`kmalloc`** non è in grado di rilocare o richiedere queste regioni in caso di mancanza di memoria.

Un'altra strategia adottata da Linux per allocare memoria del kernel è nota come allocazione di slab (blocchi). Uno slab è utilizzato per allocare la memoria per le strutture dati del kernel ed è costituito da una o più pagine fisicamente contigue. Una **cache** consiste di uno o più slab ed esiste una singola cache per ciascuna struttura dati unica del kernel: per esempio una cache per la struttura dati che rappresenta i descrittori dei processi, una cache per gli oggetti file, una per i semafori, e così via. Ciascuna cache è popolata da **oggetti** che sono istanze della struttura dati del kernel che la cache rappresenta. Per esempio, la cache che rappresenta i semafori contiene istanze di oggetti semaforo, la cache che rappresenta i descrittori dei processi contiene istanze degli oggetti descrittori dei processi eccetera. La relazione fra slab, cache e oggetti è mostrata nella figura 7, che mostra due oggetti del kernel di 3 KB e tre oggetti da 7 KB. Questi oggetti sono contenuti nelle rispettive cache per oggetti da 3 e 7 KB.

L'algoritmo di allocazione degli slab utilizza le cache per immagazzinare gli oggetti del kernel. Quando una cache viene creata, le vengono assegnati un numero di oggetti inizialmente marcati come **liberi**. Il numero di oggetti nella cache dipende dalla dimensione degli slab associati: per esempio, uno slab da 12 KB (composto da tre pagine contigue da 4 KB) può contenere sei oggetti da 2 KB. Inizialmente tutti gli oggetti nella cache sono marcati come liberi. Quando è necessario un nuovo oggetto per una struttura dati del kernel l'allocatore può soddisfare la richiesta assegnando un qualunque oggetto libero dalla cache. L'oggetto assegnato dalla cache viene marcato come **in uso**.

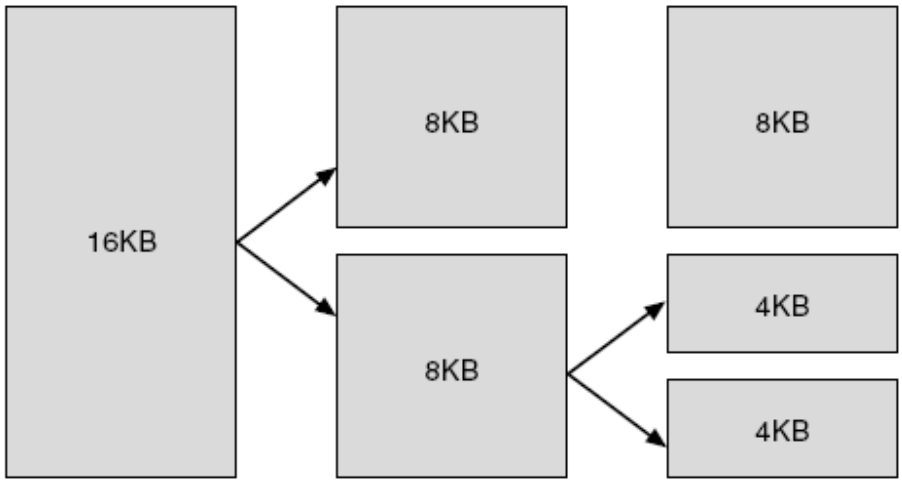

Figura 6. La divisione della memoria in un algoritmo buddy heap.

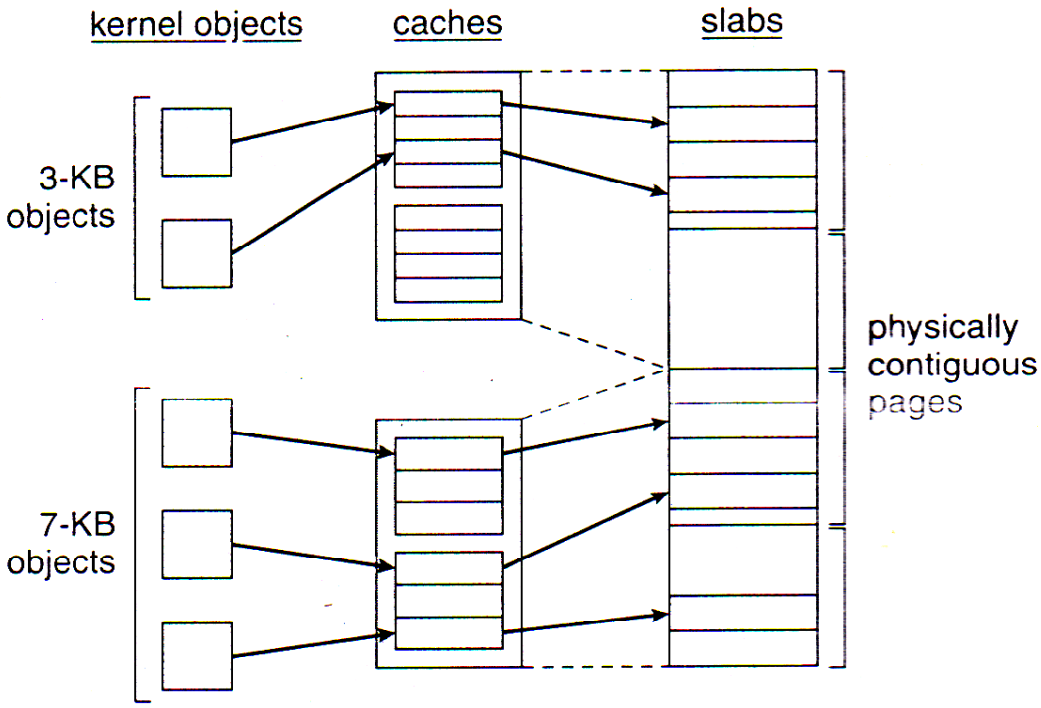

Figura 7. Allocatore di slab in Linux.

kernel objects = oggetti del kernel
caches = cache; slabs = slab
3–KB objects = oggetti da 3 KB
7–KB objects = oggetti da 7 KB
physically contiguous pages = pagine fisicamente contigue

Consideriamo uno scenario nel quale il kernel richieda memoria dall'allocatore degli slab per un oggetto che rappresenta un descrittore di un processo. Nel sistema Linux, un descrittore di un processo è di tipo `struct task_struct`, che richiede circa 1,7 KB di memoria. Quando il kernel di Linux crea un nuovo task, richiede alla cache la memoria necessaria per un `struct task_struct`. La cache soddisferà la richiesta utilizzando un oggetto `struct task_struct` già allocato nello slab e marcato come libero.

In Linux uno slab può essere in uno di questi tre possibili stati:

1. **pieno**: tutti gli oggetti nello slab sono marcati come in uso.
2. **vuoto**: tutti gli oggetti nello slab sono marcati come liberi.
3. **parziale**: lo slab contiene oggetti sia in uso che liberi.

L'allocatore degli slab tenta prima di soddisfare le richieste con un oggetto libero in uno slab parziale e se non ne esistono assegna un oggetto libero da uno slab vuoto. Se non ci sono slab vuoti ne viene allocato uno da una pagina fisica contigua e viene assegnato alla cache; la memoria per l'oggetto viene allocata da questo slab.

Gli altri due sottosistemi principali che svolgono la propria gestione delle pagine fisiche sono strettamente collegati. Si tratta della cache di pagina e del sistema di memoria virtuale. La cache di pagina è il sistema principale di caching utilizzato dal kernel per la gestione di periferiche block-oriented, e per i file mappati in memoria; questo meccanismo è quello principale per l'esecuzione dell'I/O su questo tipo di periferiche. Sia i file system nativi basati sui dischi di Linux che il file system di rete NFS utilizzano la cache di pagina. Quest'ultima gestisce una cache in cui vengono salvate intere pagine di contenuti di file, e non è limitata alle periferiche a blocchi; essa consente anche di gestire la cache per i dati sulla rete. Il sistema di memoria virtuale gestisce il contenuto dello spazio di indirizzi virtuale di ciascun processo.

Questi due sistemi interagiscono strettamente tra loro. La lettura di una pagina di dati nella cache di pagina richiede la mappatura delle pagine nella cache di pagina utilizzando il sistema di memoria virtuale. Nelle seguenti sezioni analizziamo in maggior dettaglio il sistema di memoria virtuale.

6.2 Memoria virtuale

La memoria virtuale di un sistema Linux è responsabile di mantenere lo spazio di indirizzi visibile a ciascun processo, creando pagine di memoria virtuale a richiesta, e gestendo il caricamento di queste pagine dal disco o il loro swap su disco, quando necessario. In Linux il gestore della memoria virtuale mantiene due viste separate dello spazio di indirizzi di un processo: come insieme di regioni separate, e come insieme di pagine.

La prima vista di uno spazio di indirizzi è quella logica, che descrive le istruzioni che il sistema di memoria virtuale ha ricevuto riguardo alla struttura dello spazio degli indirizzi. In questa vista, lo spazio degli indirizzi consiste di un insieme di regioni non sovrapposte che rappresentano un sottoinsieme continuo e allineato rispetto alle pagine (page alligned) dello spazio di indirizzi. Ciascuna regione è descritta internamente da una singola struttura **vm_area_struct** che definisce le proprietà della regione, inclusi i permessi di lettura, scrittura ed esecuzione del processo in quella regione e le informazioni su qualunque file associato con una regione. Le regioni per ciascuno spazio di indirizzi sono inserite in un albero binario bilanciato per permettere una ricerca veloce della regione corrispondente a ciascun indirizzo virtuale.

Il kernel mantiene anche che una seconda vista fisica di ciascuno spazio di indirizzi. Questa vista è salvata nelle tabelle hardware delle pagine per quel processo. Le voci della tabella delle pagine determinano l'esatta locazione corrente per ciascuna pagina della memoria virtuale, sia che essa sia

su disco sia che sia nella memoria fisica. La vista fisica è gestita da un insieme di procedure invocata dai gestori degli interrupt software del kernel ogni volta che un processo prova ad accedere ad una pagina che non è correntemente presente nelle tabelle delle pagine. Ogni **vm_area_struct** contiene, nella descrizione dello spazio degli indirizzi, un campo che punta alla tabella delle funzioni che implementano le funzioni chiave di gestione delle pagine per ogni regione della memoria virtuale. Tutte le richieste di lettura o di scrittura, per una pagina non disponibile, vengono eventualmente inoltrate al gestore appropriato nella tabella delle funzioni del **vm_area_struct**, cosicché le procedure centrali di gestione della memoria non debbano conoscere i dettagli della gestione di ciascuno possibile tipo di regione di memoria.

6.2.1 Regioni della memoria virtuale

Linux gestisce molti tipi di regioni di memoria virtuale. La prima proprietà che caratterizza un tipo di memoria virtuale è il backing store della regione, che descrive da dove vengono le pagine di una regione. Molte regioni di memoria si appoggiano su un file o sul nulla. Una regione poggiata sul nulla costituisce il tipo più semplice di memoria virtuale e rappresenta la **demand-zero memory** (memoria con nessuna richiesta): quando un processo prova a leggere una pagina in una regione di questo tipo, semplicemente ottiene una pagina di memoria riempita con degli zeri.

Una regione poggiata su un file agisce come uno spiraglio su una sezione di quel file: ogni volta che il processo prova ad accedere ad una pagina all'interno di quella regione, la tabella delle pagine viene riempita con l'indirizzo di una pagina all'interno della cache delle pagine del kernel corrispondente all'appropriato spiazzamento nel file. La stessa pagina di memoria fisica viene utilizzata sia dalla cache della pagina, sia dalle tabelle delle pagine del processo, dunque ogni modifica fatta al file dal file system è immediatamente visibile ad ogni processo che ha mappato quel file nel suo spazio degli indirizzi. Un numero qualunque di processi può mappare la stessa regione dello stesso file, e tutti utilizzeranno la stessa pagina di memoria fisica per questo scopo.

Una regione di memoria virtuale è anche definita dalle sue reazioni alle scrittura: la mappatura di una regione nello spazio di indirizzi di un processo può essere sia *privata* che *condivisa*. Se un processo scrive in una regione mappata privatamente, allora il paginature si accorge che è necessaria una copy-on-write per mantenere i cambiamenti locali al processo. D'altra parte, le scritture in una regione condivisa danno luogo alla modifica dell'oggetto mappato nello regione, cosicché le modifiche saranno visibili immediatamente a ogni processo che mappa quell'oggetto.

6.2.2 Periodo di vita di uno spazio di indirizzamento virtuale

Il kernel creerà un nuovo spazio di indirizzamento virtuale in esattamente due situazioni: quando un processo esegue un nuovo programma con la chiamata di sistema `exec`, e alla creazione di un nuovo processo mediante la chiamata di sistema `fork`. Il primo caso è semplice: quando un nuovo programma viene eseguito, al processo viene dato un nuovo spazio di indirizzamento virtuale completamente vuoto. Dipende dalle procedure di caricamento del programma riempire lo spazio degli indirizzi con le regioni della memoria virtuale.

Nel secondo caso, la creazione di un nuovo processo tramite la `fork` richiede la creazione di una copia completa dello spazio di indirizzamento virtuale del processo già esistente. Il kernel copia i descrittori **vm_area_struct** del processo padre, e quindi crea un nuovo insieme di tabelle di pagine per il figlio. Le tabelle delle pagine del padre sono copiate direttamente in quelle del figlio

incrementando il contatore dei puntatori a ciascuna pagina; pertanto, dopo la `fork`, il padre e il figlio condividono le stesse pagine fisiche di memoria del loro spazio degli indirizzi.

Un caso speciale capita quando le operazioni di copia raggiungono una regione di memoria virtuale mappata in modo privato. Ogni pagina all'interno di una tale regione, nella quale il processo padre ha scritto, è privata, e le successive modifiche a queste pagine da parte del padre o del figlio non devono modificare la pagina nello spazio degli indirizzi dell'altro processo. Quando le voci della tabella delle pagine per queste regioni vengono copiate, vengono specificate come a sola lettura e marcate per la copy-on-write. Finché nessuno dei processi modifica queste pagine, entrambi condividono la stessa pagina di memoria fisica; d'altra parte, se entrambi i processi provano a modificare una pagina in copy-on-write, viene controllato il contatore dei puntatori alla pagina. Se la pagina è ancora condivisa, il processo copia i contenuti della pagina in una nuova pagina di memoria fisica e utilizza la propria copia. Questo meccanismo assicura che le pagine di dati private siano condivise fra i processi ogni volta che sia possibile, e che vengano effettuate copie solo quando è assolutamente necessario.

6.2.3 Swap e paginazione

Un compito importante per un sistema di memoria virtuale è la riallocazione delle pagine di memoria dalla memoria fisica al disco quando quella memoria diventa necessaria. I primi sistemi UNIX effettuavano questa riallocazione effettuando in un solo colpo lo swap dei contenuti dell'intero processo, ma le versioni moderne si basano maggiormente sulla paginazione - il movimento di pagine singole di memoria virtuale fra la memoria fisica e il disco. Linux non implementa lo swap dell'intero processo, ma la paginazione.

Il sistema di paginazione può essere diviso in due sezioni: prima, **l'algoritmo di policy** decide quali pagine devono essere scritte sul disco e quando scriverle; secondo, **il meccanismo di paginazione** esegue il trasferimento e pagina nuovamente i dati nella memoria fisica quando sono nuovamente necessari.

La **pageout policy** (**politica di depaginazione**) di Linux non utilizza una versione modificata dell'algoritmo dell'orologio standard (o second chance) descritta nel paragrafo 10.4.5.2. In Linux, viene utilizzato un orologio a passaggi multipli, e ogni pagina ha un'età che viene modificata a ogni passaggio dell'orologio. L'età è più precisamente una misura della gioventù di una pagina, o di quanta attività la pagina ha visto recentemente. Le pagine a cui si accede frequentemente otterranno un valore di età maggiore, mentre quelle a cui si accede poco frequentemente crolleranno circa a zero ad ogni passaggio. Questa valutazione dell'età permette al paginatore di selezionare le pagine da impaginare basandosi su una politica least-frequently-used (LFU).

Il meccanismo di paginazione supporta la paginazione sia per le periferiche e partizioni dedicate allo swap, sia per i normali file, anche se lo swap di un file è significativamente più lento a causa dell'overhead extra causato dal file system. I blocchi vengono allocati dalle periferiche di swap in accordo con le mappe di vita dei blocchi usati, che vengono continuamente mantenuti nella memoria fisica. L'allocatore utilizza un algoritmo next-fit per provare a scrivere le pagine in sezioni continue di blocchi del disco per migliorare le prestazioni. L'allocatore registra il fatto che una pagina è stata scritta sul disco utilizzando una caratteristica delle tabelle delle pagine dei moderni processori: viene impostato il bit di pagina non presente nella voce della tabella delle pagine, permettendo che il resto della voce della tabella delle pagine sia riempita con un indice che identifica dove è stata scritta la pagina.

6.2.4 Memoria virtuale del kernel

Linux riserva per il proprio uso interno una regione costante e dipendente dall'architettura, di spazio di indirizzi virtuali per ogni processo. Le voci nella tabella delle pagine che mappano queste pagine del kernel sono marcate come protette, cosicché le pagine non siano visibili o modificabili quando il processore è in esecuzione in modo utente. Questa area di memoria virtuale del kernel contiene due regioni. La prima sezione ha un'area statica che contiene i riferimenti alla tabella delle pagine ad ogni pagina di memoria fisica disponibile nel sistema, cosicché venga effettuata, quando viene eseguito codice del kernel, una semplice traslazione dagli indirizzi di fisici a quelli virtuali. Il nucleo del kernel, più tutte le pagine allocate dal allocatore del normali pagine, risiede in questa regione.

Il resto della sezione dello spazio di indirizzi riservata dal kernel, non è riservata per nessuno scopo specifico. Le voci della tabella delle pagine in questo spazio di indirizzi può essere modificato dal kernel per puntare a ogni altra area di memoria, secondo le necessità. Il kernel fornisce un paio di utilità che permettono ai processi di utilizzare questa memoria virtuale. La funzione `vmalloc` alloca un numero arbitrario di pagine di memoria fisica, e le mappa in una sola regione della memoria virtuale del kernel, permettendo l'allocazione di larghi blocchi di memoria contigua anche se non c'è sufficiente spazio fisico adiacente per soddisfare la richiesta. La funzione `vremap` mappa una sequenza di indirizzi virtuali per puntare a un'area di memoria usata dal driver di una periferica per l'I/O memory-mapped.

6.3 Caricamento ed esecuzione dei programmi utente

L'esecuzione del kernel di Linux dei programmi utente è innescata da una invocazione alla chiamata di sistema `exec`. Questa chiamata ordina al kernel di eseguire un nuovo programma all'interno del processo corrente, sovrascrivendo completamente il contesto di esecuzione corrente con il contesto iniziale del nuovo programma. Il primo compito di questo servizio di sistema è di verificare che il processo chiamante abbia il permesso per l'esecuzione del file. Una volta verificato, il kernel invoca una procedura di caricamento per iniziare l'esecuzione del programma. Il loader non carica necessariamente i contenuti del file del programma nella memoria fisica, ma perlomeno allestisce la mappatura del programma nella memoria virtuale.

Non ci sono singole procedure in Linux per il caricamento di un nuovo programma. Invece, Linux mantiene una tabella di possibili funzioni di caricamento, e dà a ciascuna di queste funzioni l'opportunità di provare a caricare il file assegnato quando viene eseguita una chiamata di sistema `exec`. La ragione iniziale per questa tabella di caricatori era che, fra il rilascio dei kernel 1.0 e 1.2, il formato standard dei file binari di Linux è stato modificato. I kernel di Linux più vecchi capivano il formato a.out per i file binari: un formato relativamente semplice, comune sui vecchi sistemi UNIX. I sistemi Linux più recenti usano il più moderno formato **ELF**, adesso sopportato dalla maggior parte delle implementazioni correnti di UNIX. ELF ha una quantità di vantaggi rispetto ad a.out, fra cui la flessibilità e l'estensibilità: nuove sezioni possono essere aggiunte a un binario ELF (per esempio, per aggiungere informazioni di debug aggiuntive), senza che le procedure di caricamento diventino confuse. Consentendo la registrazione di più procedure di caricamento, Linux può facilmente supportare i formati binari ELF ed a.out con una singola esecuzione di sistema.

Nei paragrafi 6.3.1 e 6.3.2, ci concentriamo esclusivamente sul caricamento l'esecuzione dei binari in formato ELF. La procedura per il caricamento dei binari a.out è più semplice, ma è un'operazione simile.

6.3.1 Mappatura dei programmi in memoria

In Linux, il caricatore di file binari non carica un file nella memoria fisica, piuttosto, le pagine del file binario vengono mappate in regioni della memoria virtuale. Solo quando il programma prova ad accedere ad una pagina si otterrà un page fault nel caricamento della pagina nella memoria fisica.

La costituzione della mappatura iniziale della memoria è responsabilità del caricatore di binari del kernel. Un file binario in formato ELF consiste di una header seguito da varie sezioni allineate per pagina. Il caricatore dei file ELF funziona leggendo l'header e mappando le sezioni del file in regioni separate della memoria virtuale.

La figura 8 mostra la struttura tipica delle regioni di memoria organizzata dal loader ELF. In una regione riservata a un estremo dello spazio degli indirizzi si trova il kernel, nella sua regione privilegiata di memoria virtuale inaccessibile ai normali programmi in modalità utente. La parte restante della memoria virtuale è disponibile per le applicazioni, che possono utilizzare le funzioni di mappatura della memoria del kernel per creare regioni che mappino una porzione di un file o che siano disponibili per i dati dell'applicazione.

Il compito del loader consiste nell'allestire la mappatura iniziale della memoria per consentire l'esecuzione del programma da avviare. La regione che necessita di essere inizializzata include lo stack e le regioni di testo e di dati del programma.

Lo stack viene creato in cima alla memoria virtuale in modalità utente, e cresce in giù in direzione degli indirizzi con numero più basso, includendo copie degli argomenti e delle variabili dell'ambiente passati al programma nella chiamata di sistema `exec`. Le altre regioni vengono create vicino al fondo della memoria virtuale. Le sezioni del file binario che contengono il testo del programma o i dati di sola lettura sono mappate in memoria come una regione protetta dalla scrittura. Dopo di essi sono mappati i dati scrivibili inizializzati, e quindi ogni dato non inizializzato viene mappato come una regione privata zero-demand.

Subito dopo questa regione di lunghezza fissa si trova una regione di dimensione variabile, che può venire espansa dai programmi secondo necessità, per memorizzare i dati allocati durante l'esecuzione. Ogni processo ha un puntatore, *brk*, che punta all'area corrente di questa regione di dati, e i processi possono estendere o contrarre la loro regione *brk* con una sola chiamata di sistema.

Dopo queste mappature, il loader inizializza il registro program-counter del processo col punto di inizio registrato nell'header ELF, e il processo può essere schedulato.

6.3.2 Link statico e dinamico

Una volta che il programma è stato caricato ed ha cominciato l'esecuzione, tutti i contenuti necessari del file binario sono stati caricati nello spazio di indirizzamento virtuale del processo. Comunque, molti programmi necessitano anche di eseguire delle funzioni dalle librerie di sistema, e anche queste funzioni di libreria devono essere caricate. Ma nel caso più semplice, quando un programma costruisce un'applicazione, le funzioni di libreria necessarie vengono inglobate direttamente nel file binario eseguibile del programma. Un programma di questo tipo è collegato staticamente alle sue librerie, e può iniziare l'esecuzione non appena viene caricato.

Lo svantaggio principale del link statico è che ogni programma generato deve contenere esattamente le copie delle stesse funzioni comuni di libreria del sistema. È molto più efficiente, sia in termini di memoria fisica, sia di utilizzo di spazio su disco, caricare le librerie in memoria solo una volta. Il link dinamico permette che ciò avvenga.

Linux implementa il link dinamico nello spazio utente attraverso una speciale libreria del linker. Ogni programma linkato dinamicamente, contiene una piccola funzione collegata staticamente che viene chiamata all'avvio del programma e che mappa la link library in memoria e esegue il codice che la funzione contiene. La link library legge la lista delle librerie dinamiche necessarie al programma e i nomi delle variabili delle funzioni necessarie in queste librerie, leggendo le informazioni contenute in alcune sezioni del binario ELF, e mappa le librerie in mezzo alla memoria virtuale, risolvendo i riferimenti ai simboli contenuti in queste librerie. Non importa esattamente dove siano mappate in memoria queste librerie condivise: sono compilate in **codice indipendente dalla posizione** (**PIC**, **position-independent code**), che può essere eseguito in ogni indirizzo di memoria.

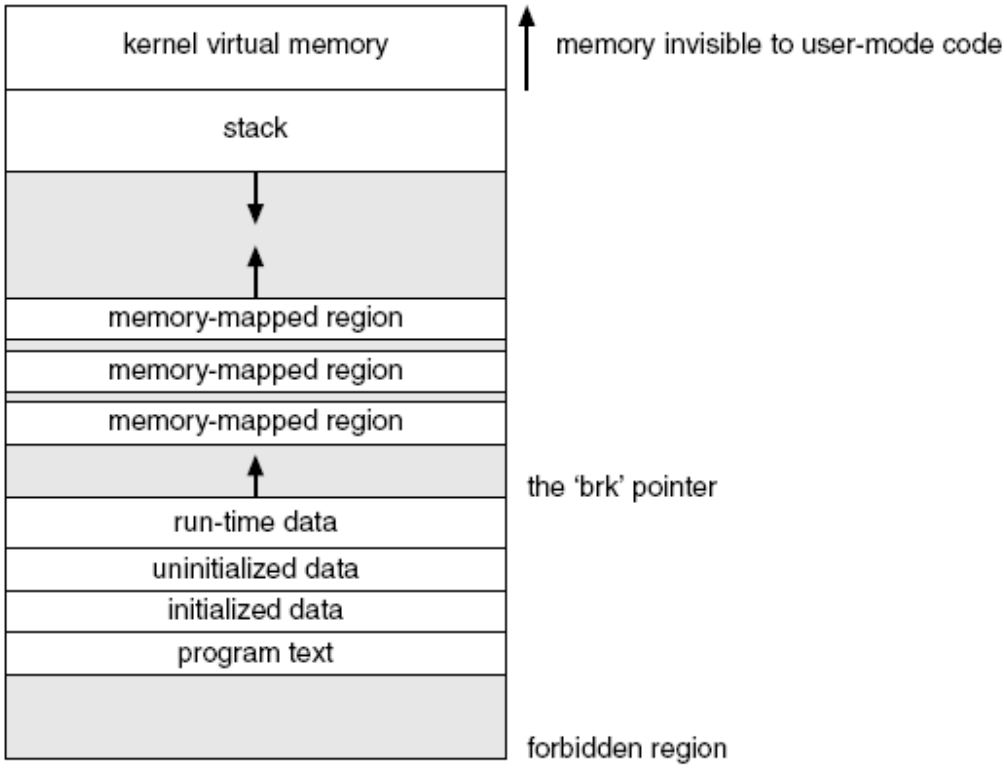

Figura 8. Struttura della memoria per i file ELF.

kernel virtual memory = memoria virtuale del kernel
stack
memory invisisble to user–mode code = memoria invisibile al codice in modalità utente
memory–mapped region = regione a memoria mappata
the 'brk' pointer= il puntatore 'brk'
run–time data = dati run–time
uninizialized data = dati non inizializzati
inizialized data = dati inizializzati
program text = testo del programma
forbidden region = regione vietata

7 File system

Linux mantiene il modello del file system standard di UNIX, in cui, un file non è necessariamente un oggetto memorizzato su disco o preso attraverso la rete da un file server remoto. Al contrario, i file di UNIX possono essere qualunque cosa in grado di gestire l'ingresso o l'uscita di un flusso di dati. I driver delle periferiche possono sembrare dei file, e i canali di comunicazione fra i processi o le connessioni di rete possono apparire all'utente come dei file.

Il kernel di Linux gestisce tutti i tipi di file nascondendo i dettagli di implementazione di ogni singolo tipo di file dietro a uno strato di software: il virtual file system (VFS: file system virtuale).

7.1 Il file system virtuale

Il VFS di Linux è progettato secondo principi object oriented. Esso è composto da due componenti: un set di definizioni che specificano quali caratteristiche un oggetto file sia autorizzato ad avere, e uno strato di software che ha come obiettivo quello di manipolare questi oggetti. I tre tipi di oggetto principali definiti dal VFS sono l'**oggetto inode** e le strutture **oggetto file**, che rappresentano file individuali, e **oggetto file system** (o **oggetto superblocco**), che rappresenta l'intero file system. Nel kernel 2.6 troviamo anche il **dentry object**, che rappresenta una singola voce di una directory.

Per ognuno di questi tre tipi di oggetto, il VFS definisce insiemi di operazioni che devono essere implementate dalla struttura specifica. Ogni oggetto di uno di questi tipi contiene un puntatore a una tabella di funzioni. Questa tabella contiene una lista di indirizzi delle funzioni reali che implementano quelle operazioni per quel particolare oggetto. Così, lo strato di software del VFS può eseguire un'operazione su uno di questi oggetti richiamando una funzione appropriata dalla tabella di funzioni specifica, senza il bisogno di conoscere in anticipo il tipo di oggetto con cui precisamente si ha a che fare. Il VFS non conosce, e non se ne preoccupa, se un inode rappresenta un file di rete, o sul disco, o una socket di rete, o una directory. La funzione appropriata per l'operazione di lettura di quel file sarà sempre nella stessa posizione della propria tabella di funzioni, e lo strato software del VFS chiamerà quella funzione senza preoccuparsi di come i dati verranno realmente letti.

Gli oggetti inode e file rappresentano i meccanismi utilizzati per accedere ai file. Un oggetto inode rappresenta il file nel suo complesso, e l'oggetto file è un punto di accesso ai dati all'interno del file. Un processo non può accedere al contenuto di un inode senza prima aver ottenuto un oggetto file che punta a quel determinato inode. L'oggetto file tiene traccia della posizione del file in cui il processo sta leggendo o scrivendo attualmente, per tener traccia dell'I/O sequenziale su file. Tale oggetto file memorizza inoltre l'eventuale richiesta di accesso di scrittura da parte del processo, al momento dell'apertura del file; infine, monitora l'attività del processo, nel caso fosse necessario eseguire operazioni di read-ahead adattivo, caricando dati in memoria in anticipo rispetto al processo che li richiede, al fine di migliorare le prestazioni.

Gli oggetti file appartengono tipicamente a un singolo processo, ma gli oggetti inode non seguono questa prassi. Anche quando un file non è più utilizzato da nessun processo, l'inode corrispondente potrebbe ancora essere presente nella cache del VFS, allo scopo di migliorare le prestazioni, nel caso in cui il file venisse utilizzato nuovamente a breve. Tutti i dati di file nella cache sono collegati ad una lista contenuta nell'oggetto inode relativo al file. L'inode mantiene anche le informazioni standard riguardo ogni file, come, ad esempio, il proprietario, la dimensione, e la data dell'ultima modifica.

I file di directory vengono gestiti in modo leggermente differente rispetto agli altri file. L'interfaccia di programmazione di UNIX definisce un certo numero di operazioni effettuabili sulle

directory, come, ad esempio, la creazione, la cancellazione o la rinominazione un file in una directory. Diversamente dalle operazione di lettura e scrittura, in cui il file dove venire necessariamente aperto, le chiamate di sistema per queste operazioni non richiedono che l'utente apra il file in questione. Il VFS, quindi, definisce queste operazioni sulle directory nell'oggetto inode, piuttosto che nell'oggetto file.

L'oggetto file system rappresenta un insieme di file connessi tra loro a formare una gerarchia di directory ***selfcontained***. Il kernel del sistema operativo mantiene un oggetto file system singolo per ogni periferica registrata come file system e per ogni file system di rete connesso al momento. La responsabilità maggiore dell'oggetto file system è quella di garantire l'accesso agli inode. Il VFS identifica ogni inode con una coppia univoca (file system – numero inode), e trova l'inode corrispondente a un particolare numero richiedendo all'oggetto file system di recuperare l'inode con quel dato numero.

Infine un oggetto dentry rappresenta una voce in una directory che può includere il nome il nome di una directory nel percorso di un file (come `/usr`) o il file stesso (come `stdio.h`). per esempio il file `/usr/include/stdio.h` contiene le directory (1) `/`, (2) `usr`, (3) `include` e (4) `stdio.h`. Ciascuno di questi valori è rappresentato da un oggetto dentry separato.

Come esempio di come sono usati gli oggetti dentry, si consideri la situazione in cui un processo desidera aprire il file con percorso `/usr/include/stdio.h` utilizzando un editor. Poiché Linux tratta i nomi di directory come dei file, la traduzione di questo percorso richiede per prima cosa di ottenere l'inode della root /. Il sistema operativo deve quindi leggere questo file per ottenere l'inode del file `include` e deve proseguire in questo processo finché non ottiene l'inode del file `stdio.h`. Poiché la traduzione di un percorso può essere una attività che richiede tempo, Linux mantiene una cache di oggetti dentry, che viene consultata durante la traduzione di un percorso. Ottenere un inode dalla cache dei dentry è considerevolmente più veloce che leggere i file su disco.

7.2 Journaling

Ci sono molti file system disponibili per Linux. Una caratteristica popolare fra i file system è il journaling, mediante il quale si scrivono progressivamente in un giornale (journal) le modifiche effettuate sul file system. L'insieme di operazioni che esegue un compito specifico è una **transazione**. Una volta che una transazione è stata scritta nel journal, è considerata committata, e la chiamata di sistema che modifica il file system (per esempio write()) può tornare al processo utente, pemettendogli di proseguire con l'esecuzione, mentre le voci del journal relative alla transazione vengono eseguite nel file system reale. Mentre vengono eseguite le operazioni, viene aggiornato un puntatore per dire quali operazioni siano complete e quali ancora da eseguire, e quando un'intera transazione viene completata, viene rimossa dal journal. Il journal, che è implementato come un buffer circolare, può risiedere in una sezione separata del file system, o addirittura in uno spindle separato del disco. È più efficiente, ma più complesso averlo sotto testine di lettura e di scrittura separate, diminuendo così la contesa delle testine ed il tempo di ricerca.

Se il sistema subisce un crash, ci saranno zero o più transazioni nel journal, che non sono state completate sul file system, anche se è stata eseguita la commit dal sistema operativo, e che devono essere completate. Le transazioni possono essere eseguite dal puntatore sino alla fine del lavoro, e la struttura del file system rimane consistente. L'unico problema accade quando una transazione è stata abortita, ossia non è stata fatta la commit prima del crash del sistema: ogni operazione svolta da quella transazione deve essere disfatta, preservando la consistenza del file system. Questi recuperi

sono tutto ciò che va fatto dopo un crash del sistema, eliminando i problemi di controllo della consistenza.

I file system con journaling sono anche tipicamente più veloci di quelli senza, visto che le modifiche sono molto più veloci quando sono applicate al giornale in memoria piuttosto che direttamente alle strutture dati su disco. la ragione di questo miglioramento risiede nel vantaggio dell'esecuzione di I/O sequenziale invece che casuale. Le costose scritture sincrone sul file system sono tramutate in meno costose scritture sequenziali sincrone nel journal del sistema, e poi rieseguite in modo asincrono attraverso scritture casuali nelle strutture appropriate. Il risultato complessivo è un incremento significativo delle prestazioni delle operazioni su metadati del file system, come la creazione e la cancellazione dei file.

Il file system ext2fs non fornisce journaling, ma esso è presente in un altro file system comunemente disponibile nei sistemi Linux : ext3, che è basato su ext2fs.

7.3 Il file system ext2fs di Linux

Il file system su disco standard utilizzato da Linux si chiama *ext2fs*, per ragioni storiche. Linux è stato originariamente programmato con un file system compatibile con Minix, per facilitare lo scambio di dati con il sistema di sviluppo Minix, il cui file system era però pesantemente limitato dalla restrizione di 14 caratteri nel nome dei file e di 64 MB di dimensione massima del file system. Il file system di Minix è stato rimpiazzato da un nuovo file system, battezzato **extended file system (extfs)**. Una successiva riprogettazione di questo file system, volta a migliorare le prestazioni e la scalabilità e ad aggiungere alcune caratteristiche mancanti, ha portato al **second extended file system (ext2fs)**.

Il file system ext2fs ha molto in comune con il fast file system di BSD (ffs) (paragrafo 7.7 del capitolo online "Il Sistema Operativo FreeBSD"). Utilizza un meccanismo simile per localizzare i blocchi di dati appartenenti a uno specifico file, per immagazzinare i puntatori ai blocchi di dati nei blocchi indiretti attraverso il file system sino a tre livelli di indirezione. Come in ffs, le directory sono salvate su disco come normali file, benché i loro contenuti siano interpretati differentemente. Ciascun blocco in una directory, consiste in una lista di voci, dove ciascuna voce contiene la propria lunghezza, il nome di un file, e il numero dell'inode a cui si riferisce.

La differenza principale tra ext2fs e ffs risiede nella politica di allocazione del disco. In ffs, il disco viene allocato per il file in blocchi di 8 KB, a loro volta suddivisi in frammenti di 1 KB per immagazzinare i piccoli file o blocchi riempiti parzialmente alla fine di un file. Al contrario, ext2fs non usa per niente i frammenti, ma esegue tutte le sue locazioni in unità più piccole; la dimensione di default dei blocchi è di 1 KB, benché siano supportati anche blocchi di 2 o 4 KB.

Per mantenere alte le prestazioni, il sistema operativo deve provare ad eseguire l'I/O in parti ampie ogni qualvolta, se possibile, raggruppando le richieste di I/O fisicamente adiacenti. Il raggruppamento riduce l'overhead per la richiesta in cui incorrono i driver delle periferiche, i dischi, e l'hardware dei controller dei dischi. Una dimensione di richiesta di I/O di 1KB è troppo piccola per mantenere delle buone prestazioni, pertanto ext2fs utilizza delle politiche di allocazione progettate per mettere blocchi di un file logicamente adiacenti in blocchi di disco fisicamente adiacenti, così da poter inoltrare una richiesta di I/O per vari blocchi del disco come un'operazione singola.

La politica di allocazione di ext2fs avviene in due parti. Come in ffs, un file system ext2fs è partizionato in più **gruppi di blocchi**. ffs utilizza il concetto simile di **gruppi di cilindri**, dove ciascun gruppo corrisponde a un singolo cilindro di un disco fisico. Comunque, l'attuale tecnologia

dei dischi raggruppa i settori sul disco a differenti densità, e pertanto con differenti dimensioni dei cilindri a seconda di quanto distante è la testina del disco dal suo centro. Pertanto, i gruppi di cilindri di dimensione fissa non corrispondono necessariamente alla geometria del disco.

Quando alloca un file, ext2fs deve prima scegliere un gruppo di blocchi per quel file. Per i blocchi di dati, prova a scegliere lo stesso gruppo di blocchi nel quale è stato allocato l'inode del file. Per l'allocazione degli inode, seleziona lo stesso gruppo di blocchi della directory di appartenenza, per i file che non sono directory. Le directory non sono tenute insieme, ma piuttosto sparpagliate attraverso i gruppi di blocchi disponibili. Queste politiche sono progettate per mantenere all'interno dello stesso gruppo di blocchi le informazioni correlate, ma anche per distribuire il carico del disco fra i diversi gruppi di blocchi, al fine di ridurre la frammentazione di ogni area del disco.

All'interno di un gruppo di blocchi, ext2fs cerca di mantenere le allocazioni fisicamente contigue, se possibile, riuscendo la frammentazione, se ci riesce. ext2fs mantiene una mappa di bit di tutti i blocchi liberi in un gruppo di blocchi; quando alloca i primi blocchi per un file comincia cercando un blocco libero dall'inizio del gruppo di blocchi; quando estende un file continua la ricerca dal blocco allocato più recentemente per quel file. La ricerca viene effettuata in due passi: prima ricerca un intero byte libero nella mappa di bit e se fallisce nel trovarne uno, cerca un bit libero; la ricerca di byte liberi mira ad allocare lo spazio del disco in gruppi di almeno otto blocchi laddove sia possibile.

Una volta identificato un blocco libero, la ricerca viene estesa all'indietro sino a raggiungere un blocco allocato. Quando un byte libero viene trovato nella mappa di bit, questa estensione all'indietro impedisce a ext2fs di lasciare un buco fra gli blocco più recentemente allocato nel precedente byte diverso da zero e il byte a zero trovato. Una volta che nella ricerca di bit o di byte è stato trovato il prossimo blocco da allocare, ext2fs estende l'allocazione in avanti per un massimo di otto blocchi e **prealloca** questi blocchi extra per il file, aiutando a ridurre la frammentazione durante l'alternarsi delle scritture a file differenti, e riducendo inoltre il costo per la CPU dell'allocazione del disco, allocando simultaneamente più blocchi. I blocchi preallocati vengono riportati nella mappa di bit e dello spazio libero quando viene chiuso il file.

La figura 9 illustra le politiche di allocazione. Ciascuna riga rappresenta una sequenza di bit impostati e non impostati in una mappa di bit di allocazione, indicando i blocchi utilizzati o liberi su disco. Nel primo caso, se possiamo trovare dei blocchi liberi sufficientemente vicini all'inizio della ricerca, li allochiamo indipendentemente da quanto possano essere frammentati. La frammentazione è parzialmente compensata dal fatto che i blocchi siano vicini fra di loro e possano probabilmente essere tutti letti senza incorrere in ricerche sul disco; allocarli tutti allo stesso file alla lunga risulta meglio che allocare blocchi isolati a file differenti, una volta che siano diventate scarse le aree libere ampie sul disco. Nel secondo caso, non abbiamo trovato immediatamente un blocco libero vicino, quindi cerchiamo avanti per un intero byte libero nella mappa di bit. Se allocassimo quel byte completamente, finiremmo per creare un'area frammentata di spazio libero prima di esso, pertanto prima di allocarlo, torniamo indietro fino a incontrare l'allocazione che lo precede, e quindi allochiamo in avanti per soddisfare l'allocazione di default di otto blocchi.

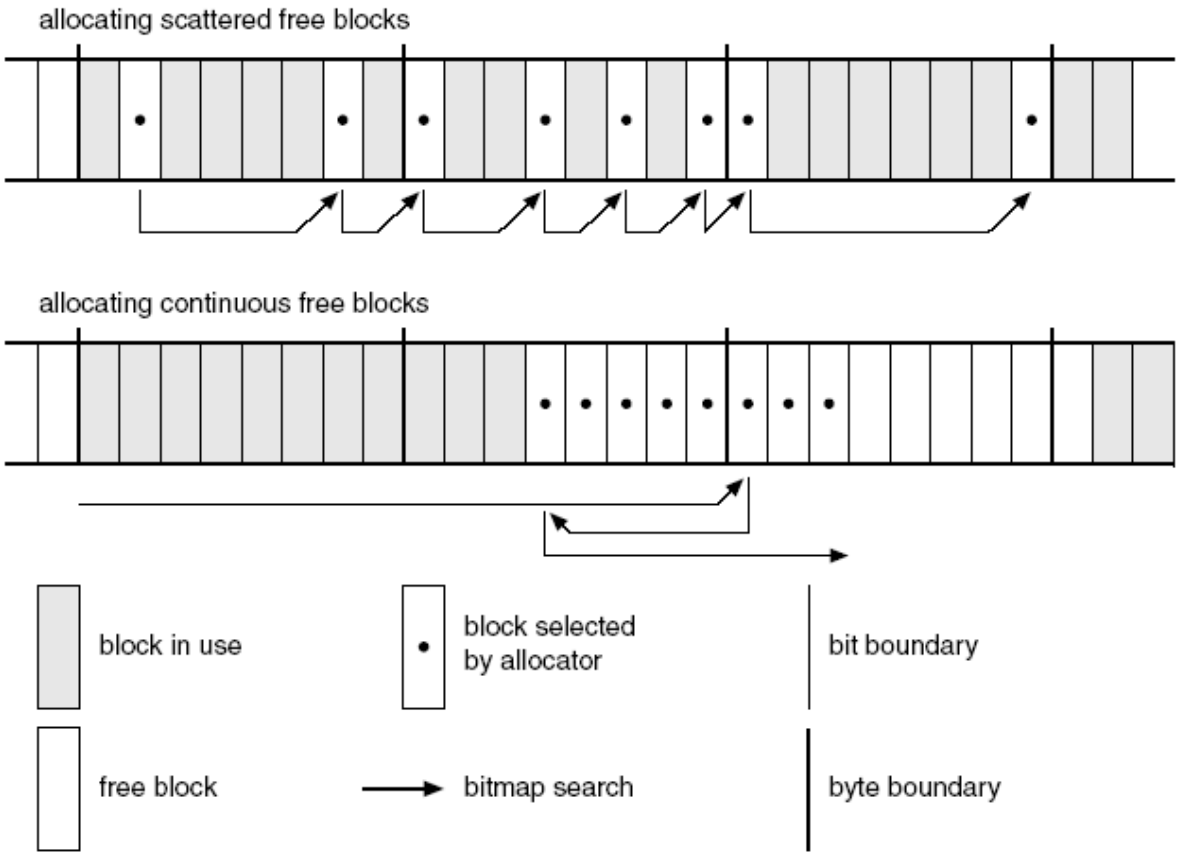

Figura 9. Politiche di allocazione del blocco ext2fs.

allocating scattered free blocks = blocchi liberi allocati sparpagliati
allocating continuous free blocks = blocchi liberi allocati contigui
block in use = blocco in uso
block selected by allocator =blocco selezionato dall'allocatore
bit boundary = bordo dei bit
free blocks = blocchi liberi
bitmap search = ricerca della mappa di bit
byte boundary = bordo dei byte

7.4 Il file system proc di Linux

Il VFS di Linux e sufficientemente flessibile da rendere possibile l'implementazione di un file system che non memorizza affatto dati persistenti, ma fornisce semplicemente un'interfaccia ad alcune altre funzionalità. Il **process file system** di Linux, conosciuto come file system *proc*, è un esempio di file system i cui contenuti non sono immagazzinati da nessuna parte ma vengono calcolati quando è necessario in accordo alle richieste di I/O degli utenti.

Il file system *proc* non è unico di Linux; SVR4 UNIX ha introdotto il file system *proc* come un'interfaccia efficiente a supporto del processo di debug del kernel: ciascuna sotto directory del file system non corrisponde ad una directory su un qualunque disco, ma ad un processo attivo nel sistema. Un'esplorazione della file system rivela una directory per ogni processo, in cui il nome della directory è la rappresentazione decimale ASCII dell'identificatore univoco del processo (process unique identifier, PID).

Linux implementa il file system *proc*, ma lo estende fortemente aggiungendo un numero di directory extra e file di testo sotto la directory radice del file system, corrispondenti a varie statistiche del kernel e ai relativi driver caricati. Il file system *proc* fornisce ai programmi un modo per accedere a queste informazioni come file di testo, per la cui elaborazione sono forniti strumenti potenti dall'ambiente standard di UNIX. Per esempio, in passato, il comando tradizionale di UNIX *ps* per elencare gli stati di tutti i processi in esecuzione, è stato implementato come un processo privilegiato che leggeva lo stato dei processi direttamente dalla memoria virtuale del kernel. In Linux, questo comando è implementato come un programma completamente non privilegiato che semplicemente interpreta e formatta le informazioni da *proc*.

Il file system *proc* deve implementare due cose: una struttura di directory ed i file contenuti dentro di essa. Posto che un file system UNIX sia definito come un insieme di inode di file e di directory identificati dai loro numeri di inode, il file system *proc* deve definire un numero di inode unico e persistente per ciascuna directory e per i file associati. Una volta che esiste questa mappatura, può utilizzare questo numero di inode per identificare proprio quale operazione è richiesta quando un utente prova a leggere da un particolare file o esegue una ricerca in una particolare directory. Quando i dati vengono letti da uno di questi file, il file system *proc* raccoglie le informazioni appropriate, le formatta in una forma testuale, e le mette nel buffer di lettura del processo richiedente.

La mappatura tra i numeri di inode e i tipi di informazione, divide i numeri di inode in due campi. In Linux, un PID è lungo 16 bit, mentre il numero di inode è di 32 bit, quindi i primi 16 bit del numero di inode sono interpretati come un PID, mentre i bit rimanenti definiscono che tipo di informazione viene richiesta riguardo al processo.

Un PID di zero non è valido, pertanto un campo di PID a zero nel numero di inode è interpretato col significato che quel inode contiene informazioni globali invece che specifiche di un processo. Esistono diversi file globali all'interno di *proc* per riportare informazioni come la versione del kernel, la memoria libera, le statistiche sulle prestazioni, ed i driver in esecuzione in quel momento.

Non tutti i numeri di inode in questo insieme sono riservati: il kernel può allocare dinamicamente nuove mappature di inode in *proc*, mantenendo una mappa di bit di numeri di inode allocati, e anche una struttura dati ad albero di voci globali del file system *proc* registrate: ciascuna voce contiene il numero dell'inode del file, il nome del file, i permessi di accesso, e le funzioni speciali utilizzate per generare i contenuti del file. I driver possono inserire o togliere voci in questo albero in ogni momento, e una sezione speciale dell'albero, posizionata sotto la directory */proc/sys*, è riservata per le variabili del kernel. Un insieme di handler comuni è utilizzata per gestire il file

all'interno di questo albero, e permettono la lettura e la scrittura di queste variabili, cosicché un amministratore di sistema può coordinare il valore dei parametri del kernel semplicemente scrivendo in ASCII i nuovi valori desiderati nel file appropriato.

Per permettere un accesso efficiente a queste variabili dall'interno delle applicazioni, il sottoalbero */proc/sys* è reso disponibile mediante una speciale chiamata di sistema, `sysctl`, che legge e scrive le stesse variabili in binario, piuttosto che in formato testo, senza l'overhead del file system. `sysctl` non è una funzionalità extra, semplicemente legge l'albero delle voci dinamiche di *proc* per decidere a quale variabile l'applicazione si stia riferendo.

8 Input e Output

All'utente, il sistema di I/O di Linux sembra molto simile a quello di un qualunque UNIX, ossia, nei limiti del possibile, tutti i driver di periferica sembrano dei normali file. Un utente può aprire un canale di accesso ad una periferica nello stesso modo in cui può aprire un qualunque altro file: le periferiche possono apparire come oggetti all'interno del file system. L'amministratore del sistema può creare file speciali all'interno del file system che contengono riferimenti ad uno specifico driver di periferica, e un utente che apre un simile file sarà in grado di leggere o scrivere dalla periferica a cui si riferisce. Utilizzando il normale sistema di protezione dei file, che determina chi può accedere a quale file, l'amministratore può impostare i diritti di accesso ad ogni periferica.

Linux divide tutte le periferiche in tre classi: le periferiche a blocchi (block device), le periferiche a caratteri (character device), e le periferiche di rete (network device). La figura 10 illustra la struttura di insieme del sistema di driver delle periferiche. Le **periferiche a blocchi** includono tutte le periferiche che permettono un accesso casuale a blocchi di dati di dimensione fissa e completamente indipendenti, ed includono gli hard disk, i floppy disk e i CD-ROM. Le periferiche a blocchi sono tipicamente utilizzate per immagazzinare sistemi di dati, ma è permesso anche l'accesso diretto alle periferica a blocchi in modo che i programmi possano creare e riparare il file system contenuto nella periferica. Le applicazioni possono anche accedere a queste periferiche a blocchi direttamente, se lo desiderano; per esempio, un'applicazione di base di dati, può scegliere di eseguire l'immagazzinamento dei dati nel disco con un proprio sistema ben calibrato, piuttosto che utilizzare un file system general purpose.

Le **periferiche a caratteri** includono la maggior parte delle altre periferiche, con la principale eccezione delle periferiche di rete. Queste periferiche non necessitano di supportare tutte le funzionalità dei normali file; per esempio, l'altoparlante permette che gli vengano inviati dei dati, ma non ne supporta la lettura. In modo analogo, la ricerca di una certa posizione nel file potrebbe essere supportata da una periferica per nastri magnetici, ma non avrebbe senso il posizionamento di una periferica come il mouse.

Le **periferiche di rete** sono gestite in modo differente dalle periferiche a blocchi o a caratteri: gli utenti non possono trasferire direttamente dati alle periferiche di rete, ma devono invece comunicare indirettamente aprendo una connessione al sottosistema di rete del kernel. Discuteremo l'interfaccia alle periferiche di rete separatamente nel paragrafo 10.

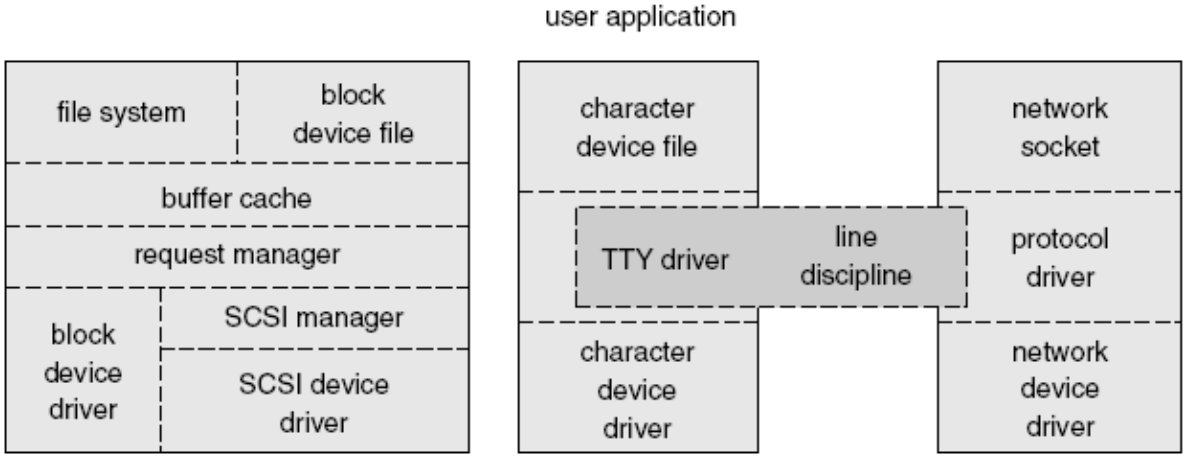

Figura 10. Struttura del blocco del device driver.

user application = applicazione utente
file system
block device file = file del dispositivo dei blocchi
buffer cache = cache del buffer
request manager = gesture della richiesta
block device driver = driver del dispositivo del blocco
SCSI manager = gestore SCSI
SCSI device driver = driver del dispositivo SCSI
character device file = file del dispositivo a caratteri
TTY driver = driver TTY
Line discipline = struttura a righe
character device driver = driver del dispositivo a caratteri
network socket = socket di rete
protocol driver = driver del protocollo
network device driver = driver del dispositivo di rete

8.1 Block device

Le periferiche a blocchi forniscono la principale interfaccia a tutti i dischi del sistema. Le prestazioni sono particolarmente importanti per i dischi, ed il sistema delle periferiche a blocchi deve fornire le funzionalità che garantiscano che l'accesso ai dischi sia il più veloce possibile. Questa funzionalità è ottenuta mediante l'uso di due componenti di sistema: la **block buffer cache** (**cache dei buffer dei blocchi**) ed il **request manager** (**manager delle richieste**).

8.1.1 La Block Buffer Cache

La **block buffer cache** di Linux ha due funzioni principali: agisce come un pool di buffer per l'I/O attivo, e come una cache per l'I/O che è già stato completato. La cache di buffer consiste di 2 parti. La prima comprende i buffer stessi, set di pagine di dimensioni scelte dinamicamente, allocate direttamente dal memory pool principale del kernel. Ciascuna pagina viene suddivisa in un certo numero di buffer di uguale lunghezza. La seconda parte consiste in un set di descrittori del buffer, le *intestazioni del buffer*, una per ogni buffer della cache.

Le *intestazioni del buffer* contengono tutte le informazioni riguardanti i buffer che il kernel mantiene. L'informazione principale è l'identità del buffer. Ogni buffer è identificato da una terna di dati: il block device al quale il buffer appartiene, l'offset dei dati all'interno di quella particolare periferica e la dimensione del buffer. Tali buffer sono anche contenuti in numerose liste, a seconda che siano vuoti, "sporchi", o bloccati, ed oltre a questi viene mantenuta una lista di buffer liberi. I buffer vengono considerati liberi e inseriti nella lista dal file system (ad esempio, quando un file viene cancellato), o da una funzione di refill freelist, che viene chiamata quando il kernel necessita di ulteriori buffer. Il kernel riempie la lista dei buffer liberi espandendo il pool di buffer o riciclando quelli esistenti, a seconda che la quantità di memoria disponibile sia o meno sufficiente. Infine, ogni buffer, non presente nella lista di quelli liberi, viene indicizzato tramite una funzione di hash con la propria periferica e numero del blocco, e viene collegato a una lista di lookup corrispondente.

Il sistema di gestione del buffer del kernel si occupa automaticamente della scrittura di buffer "sporchi" sul disco. Due demoni in background servono da assistenti in queste operazioni: uno di questi si attiva ad intervalli regolari e richiede che tutti i dati "sporcati" da un tempo maggiore di uno prestabilito vengano scritti su disco. L'altro demone è un thread del kernel che viene attivato quando la funzione di refill freelist si accorge che una percentuale troppo alta della cache di buffer è "sporca".

8.1.2 Il Request Manager

Il request manager è lo strato di software che gestisce la lettura e la scrittura di contenuti dei buffer da e verso i driver del block device. Il sistema di richieste si appoggia a una funzione, `ll_rw_block`, che esegue letture e scritture del block device a basso livello. Questa funzione richiede come argomenti in input una lista di descrittori di buffer e un flag di lettura/scrittura, e imposta il progresso delle operazioni di I/O per tutti quei buffer, senza attendere che l'I/O venga completato.

Le richieste di I/O rilevanti vengono memorizzate in strutture di `richiesta`. Una `richiesta` viene considerata rilevante quando riguarda la lettura o scrittura di molteplici settori contigui su un singolo block device. Dato che più di un buffer può dover essere chiamato in causa nel

trasferimento, la `richiesta` contiene un puntatore al primo di una lista di *intestazioni di buffer* collegate, che verranno utilizzate per il trasferimento.

Nel kernel 2.2, viene mantenuta una lista di richieste separata per ogni driver di un singolo block device, e tali richieste sono schedulate tramite un algoritmo del tipo ad ascensore unidirezionale (C-SCAN), che sfrutta l'ordine in cui le richieste sono state inserite e rimosse dalle liste per periferica. Le liste di richieste vengono mantenute ordinate per il numero di settore di partenza. Quando una richiesta è pronta per essere processata da un driver del block device, essa non viene rimossa dalla lista; tale operazione viene eseguita solo quando l'I/O è completato; a questo punto, il driver continua con la richiesta successiva, anche se una nuova richiesta viene inserita nella lista prima di quella attiva.

Quando vengono inoltrate nuove richieste di I/O, il request manager prova a fondere le richieste nella lista divisa per periferiche. Passare una richiesta singola di grandi dimensioni, piuttosto che tante di piccole dimensioni è spesso più efficiente in termini di sfruttamento della periferica sottostante. Tutte le *intestazioni dei buffer,* in tutte le richieste aggregate di questo tipo, vengono bloccate quando la richiesta iniziale di I/O viene effettuata. In seguito, quando la richiesta viene processata dal driver di una periferica, le *intestazioni di buffer* singole, che costituiscono la richiesta aggregata, vengono sbloccate una alla volta; un processo in attesa di un solo buffer non deve quindi attendere che vengano sbloccati tutti gli altri buffer della richiesta. Questo fattore è particolarmente importante perché assicura che il read-ahead (la lettura anticipata dei dati contigui a quelli effettivamente richiesti) venga eseguito in modo efficiente.

Nel kernel 2.6, la schedulazione delle operazioni di I/O è in qualche modo cambiata. Il problema fondamentale con l'algoritmo dell'ascensore è che operazioni di I/O concentrate in una regione specifica del disco possono dare luogo a starvation delle richieste localizzate in altre regioni del disco. Il **deadline I/O scheduler** (lo schedulatore di I/O con scadenze) usato nel paragrafo 2.6 funziona in modo simile all'algoritmo dello schedulatore, ma associa una deadline (scadenza) ad ogni richiesta, affrontando così il problema della starvation. Di default la deadline per le richieste di lettura è di 0,5 secondi, mentre per quelle di scrittura è di 5 secondi. Lo schedulatore con deadline mantiene una **coda ordinata** per numero di settore di operazioni di I/O pendenti, ma mantiene anche due altre code: una **coda di lettura** per le operazioni di lettura, ed una **coda di scrittura** per le operazioni di scrittura, ordinate secondo la deadline. Ogni richiesta di I/O viene inserita sia nella coda ordinata, che in una delle code, di lettura o di scrittura, come appropriato. Tipicamente le operazioni di I/O avvengono dalla coda ordinata, ma se viene oltrepassata una deadline per una richiesta nella coda di lettura o di scrittura, allora le operazioni di I/O vengono schedulate dalla coda che contiene la deadline superata. Questa politica garantisce che nessuna operazione di I/O aspetti per più tempo di quello che fa scadere la deadline.

Sia nel kernel 2.6, che già nel kernel 2.2, un'ulteriore caratteristica del request manager riguarda il fatto che possono essere eseguite le richieste di I/O senza passare dalla cache del buffer. La funzione di basso livello per l'I/O di pagina, `brw_page`, crea un set di *intestazioni di buffer* temporanee per etichettare il contenuto di una pagina di memoria, allo scopo di sottomettere le richieste di I/O al request manager. Comunque, queste *intestazioni di buffer* temporanee non sono collegate alla cache del buffer, e una volta che il buffer della pagina ha completato l'I/O, l'intera pagina stessa viene sbloccata e i *buffer_heads* scartati.

Questo meccanismo di aggiramento della cache viene utilizzato dal kernel quando il chiamante sta utilizzando un sistema di caching dei dati indipendente, o quando si conosce a priori che non è più possibile usare la cache per i dati che si prendono in considerazione. La cache di pagina riempie le proprie pagine in questo modo, al fine di evitare di utilizzare la cache anche quando non è necessario, per i dati presenti sia nella cache di pagina e in quella di buffer. Il sistema di memoria virtuale aggira anch'esso la cache, quando si trova a eseguire operazioni di I/O verso l'unità di swap.

8.2 Character device

Un driver di una periferica a caratteri può essere quasi un qualunque driver di periferica che non fornisce l'accesso casuale a un blocco di dati. Tutti i driver di periferiche a caratteri, registrati presso il kernel di Linux, devono anche registrare un insieme di funzioni che implementano le operazioni di I/O sui file che il driver può gestire. Il che non esegue quasi nessuna operazione di preprocessamento delle richieste di lettura o scrittura verso una periferica a caratteri, ma passa semplicemente la richiesta alle periferiche in questione, e lascia che questa si occupi della sua gestione.

L'eccezione principale a questa regola è il sottoinsieme speciale driver di periferiche a blocchi che implementano i terminali. Il kernel mantiene un'interfaccia standard per questi driver per mezzo di un insieme di strutture *tty_struct*, ciascuna delle quali fornisce buffer e controllo del flusso sullo stream di dati dal terminale, e li fornisce di una struttura a righe (line discipline).

Una **line discipline** è un interpretazione delle informazioni provenienti da un terminale. La più famosa line discipline è quella di `tty`, che attacca il flusso di dati del terminale ai flussi dello standard input e standard output dei processi di un utente, permettendo loro di comunicare direttamente con il terminale dell'utente. Questo lavoro è complicato dal fatto che più di un processo di questo tipo può essere in esecuzione simultaneamente, e che la `tty` line discipline è responsabile di attaccare e staccare l'input e l'output del terminale dai vari processi ad essa connessi, quando vengono sospesi o risvegliati dal utente.

Sono implementate anche altre line discipline che non hanno niente a che fare con l'I/O verso un processo dell'utente. I protocolli di rete PPP e SLIP sono modi di codificare una connessione di rete su un terminale come una linea seriale. Questi protocolli sono implementati su Linux come driver che ad un estremo appaiono al sistema terminale come delle line discipline, e che all'altro estremo appaiono al sistema di rete come driver di periferiche di rete. Dopo che è stata abilitata una di queste line discipline su un terminale, ogni dato che vi appare sarà indirizzato direttamente al driver di periferiche di rete appropriato.

9 Comunicazione fra processi

UNIX fornisce un ricco ambiente per permettere ai processi di comunicare fra di loro. La comunicazione può essere una questione per consentire agli altri processi di conoscere che è capitato un qualche evento, o può comportare il trasferimento di dati da un processo all'altro.

9.1 Sincronizzazione e segnali

Il meccanismo standard di UNIX per informare un processo dell'avvenimento di un evento è il **segnale** (**signal**). I segnali possono essere inviati da un qualunque processo a qualunque altro processo, con delle restrizioni per i segnali inviati a processi di proprietà di un altro utente. Comunque, è disponibile un numero limitato di segnali ed essi non possono portare informazioni: per un processo è disponibile solo il fatto che è arrivato un segnale. I segnali non devono necessariamente essere generati da un altro processo: anche il kernel ne genera internamente; per

esempio, può inviare un segnale a un processo server quando arrivano dei dati su un canale di rete, a un processo padre quando un figlio termina, o quando scade un timer.

Internamente, il kernel di Linux non utilizza i segnali per comunicare con processi in esecuzione in modo kernel: se un tale processo attende che accada un evento, normalmente non utilizzerà i segnali per riceverne la notifica; piuttosto, la comunicazione riguardo gli eventi asincroni in entrata all'interno del kernel è effettuata tramite l'uso delle strutture degli stati di schedulazione e della `wait_queue`. Questi meccanismi permettono a processi in modalità kernel, di informarsi a vicenda riguardo agli eventi rilevanti, e permettono anche che gli eventi siano generati dai driver delle periferiche o dalla sistema di rete. Ogni qualvolta un processo vuole attendere il completamento di un qualche evento, si mette nella `wait_queue` associata con quell'evento e dice allo schedulatore che non è più eleggibile per l'esecuzione. Una volta che l'evento è completato, sveglierà ogni processo nella `wait_queue`. Questa procedura permette a più processi di aspettare un singolo evento. Per esempio, se più processi stanno provando a leggere un file dal disco, saranno tutti i risvegliati quando i dati sono stati letti in memoria con successo.

Benché i segnali siano sempre stati il meccanismo principale per comunicare in modo asincrono gli eventi fra i processi, Linux implementa anche il meccanismo dei semafori di UNIX System V. Un processo può attendere ad un semaforo in modo altrettanto semplice di come può attendere un segnale, ma i semafori hanno due vantaggi: può essere condiviso tra più processi indipendenti un grande numero di semafori, e le operazioni su più semafori possono essere eseguite in modo atomico. Internamente, il meccanismo standard di Linux della `wait_queue` sintonizza i processi che comunicano con i semafori.

9.2 Passaggio di dati fra processi

Linux offre molti meccanismi per il passaggio di dati fra processi. Il meccanismo standard delle **pipe** di UNIX permette a un processo figlio di ereditare un canale di comunicazione dal suo genitore; i dati scritti da un lato della pipe possono essere letti dall'altro. In Linux, le pipe appaiono come un qualunque altro tipo di inode per il software del file system virtuale, e ogni pipe ha un paio di wait_queue per sincronizzare il lettore e lo scrittore. UNIX definisce anche un insieme di funzionalità di rete che possono inviare flussi di dati a processi sia locali che remoti. La rete è trattata nel paragrafo 10.

Sono disponibili altri due metodi di condivisione dei dati fra i processi. Il primo è la memoria condivisa, che offre un meccanismo estremamente veloce per comunicare grandi o piccole quantità di dati; ogni dato scritto da un processo, in una regione di memoria condivisa, può essere letto immediatamente da ogni altro processo che ha mappato quella ragione nel proprio spazio di indirizzi. Lo svantaggio principale della memoria condivisa è che, da parte sua,, non offre nessuna sincronizzazione: un processo non può né chiedere al sistema operativo se un pezzo di memoria condivisa è stato scritto, né sospendere l'esecuzione sino a che non avviene una scrittura. La memoria condivisa diviene particolarmente potente quando è utilizzata congiuntamente con un altro meccanismo di comunicazione fra processi che fornisca la sincronizzazione mancante.

Una regione di memoria condivisa in Linux è un oggetto persistente che può essere creato o cancellato da un processo, ed è trattato come se fosse un piccolo spazio di indirizzi indipendente: gli algoritmi di paginazione di Linux possono scegliere di spostare su disco una pagina di memoria condivisa, come possono farlo per una pagina di dati del processo. L'oggetto di memoria condivisa funziona come un deposito secondario per le regioni di memoria condivisa, nello stesso modo in cui un file può funzionare da deposito secondario per le regione di memoria memory-mapped. Quando

un file è mappato in una regione dello spazio di indirizzi virtuale, ogni fault di pagina che capita, causa la mappatura nella memoria virtuale della pagina appropriata del file. In modo simile, le mappature della memoria condivisa fanno sì che gli errori di pagina mappino nelle pagine da un oggetto di memoria condivisa persistente. Inoltre, proprio come per i file, gli oggetti di memoria condivisa ricordano i loro contenuti anche se nessun processo li stia mappando in quel momento nella memoria virtuale.

10 Struttura di rete

La rete è l'area chiave delle funzionalità di Linux, che non solo supporta i protocolli Internet standard utilizzati per la maggior parte delle comunicazioni da UNIX a UNIX, ma implementa anche un numero di protocolli nativi di altri sistemi operativi non UNIX. In particolare, dal momento che Linux è stato originariamente implementato principalmente sui PC, invece che su grandi workstation o su sistemi di classe server, supporta molti dei protocolli utilizzati tipicamente sulle reti di PC, come AppleTalk e IPX.

Internamente, le funzionalità di rete nel kernel di Linux sono implementate da tre strati di software:

1. l'interfaccia socket,
2. i driver dei protocolli,
3. i driver delle periferiche di rete.

Le applicazioni utente eseguono tutte le richieste di rete attraverso l'interfaccia dei socket, progettata per sembrare simile allo stato dei socket di BSD4.3, così che ogni programma progettato per fare uso dei socket di Berkley funzioni sotto Linux senza nessun cambiamento del codice sorgente. Questa interfaccia è descritta nel paragrafo 9.1 del capitolo online “Il Sistema Operativo FreeBSD”. L'interfaccia dei socket di BSD è abbastanza generale per rappresentare gli indirizzi di rete per un'ampia gamma di protocolli di rete. Questa interfaccia unica non è utilizzata da Linux per accedere solo a quei protocolli implementati dai sistemi BSD standard, ma a i tutti protocolli supportati dal sistema.

Lo strato successivo del software è lo stack di protocollo, che è simile nell'organizzazione al framework di BSD. Ogni volta che arrivano dei dati a questo strato, sia da un socket dell'applicazione sia dal driver delle periferiche di rete, ci si aspetta che siano marcati con un identificatore che specifichi quale protocollo di rete contengono. I protocolli possono comunicare tra loro, se lo desiderano; per esempio, all'interno dei protocolli di Internet, protocolli diversi gestiscono il routing, le informazioni sugli errori, e la ritrasmissione affidabile dei dati persi.

Lo strato di protocollo può riscrivere i pacchetti, creare nuovi pacchetti, dividere o riassemblare pacchetti in frammenti, o semplicemente scartare dati in ingresso. In definitiva, una volta che ha terminato di processare un insieme di pacchetti, li passa, in su all'interfaccia dei socket se i dati sono destinati a una connessione locale, o in giù ai driver delle periferiche se i pacchetti devono essere trasmessi remotamente. Lo strato di protocollo decide a quale socket o periferica inviare il pacchetto.

Tutte le comunicazioni fra gli strati dello stack di rete, sono eseguite passando singole strutture **skbuff**, che contengono un insieme di puntatori in una singola area di memoria continua, che rappresenta un buffer dentro al quale i pacchetti di rete possono essere costruiti. I dati validi in un `skbuff` non necessitano di cominciare all'inizio del buffer dello `skbuff` e non devono necessariamente andare fino alla fine. Il codice di rete può aggiungere o togliere dati da entrambi gli

estremi del pacchetto, sin tanto che il risultato può ancora essere contenuto in uno `skbuff`. Questa possibilità è particolarmente importante sui microprocessori moderni dove gli aumenti di velocità della CPU hanno ampiamente superato le prestazioni della memoria centrale: l'architettura dello `skbuff` permette flessibilità nella manipolazione degli header e dei checksum dei pacchetti, evitando ogni copia dei dati non necessaria.

Un insieme più importante di protocolli nei sistema di rete di Linux, è la Internet protocol (IP) suite, che comprende una quantità di protocolli diversi. Il protocollo IP implementa il routing fra macchine differenti in qualunque parte della rete. In cima a protocollo di routing sono costruiti i protocolli UDP, TCP e ICMP. Il protocollo UDP porta singoli datagrammi arbitrari fra gli host. Il protocollo TCP implementa connessioni affidabili fra gli host con la consegna dei pacchetti garantita in ordine, e ritrasmissione automatica dei dati persi. Il protocollo ICMP è utilizzato da portare vari errori e messaggi di stato fra gli host.

Ci si aspetta che i pacchetti (`skbuff`) in arrivo allo strato di rete del protocollo, siano già marcati con un identificatore interno che indica a quale protocollo appartengano. Driver di periferiche di rete differenti codificano il tipo di protocollo in modi differenti sui loro mezzi di comunicazione; pertanto, l'identificazione del protocollo per i dati in ingresso deve essere effettuata all'interno dei driver delle periferiche che utilizzano una tabella di hash degli identificatori dei protocolli di rete per cercare il protocollo appropriato, e passare il pacchetto a quel protocollo. Possono essere aggiunti nuovi protocolli alla tabella di hash come moduli caricabili del kernel.

I pacchetti IP in arrivo vengono consegnati al driver IP. Il lavoro di questo strato è di eseguire l'instradamento: stabilisce dove un pacchetto è destinato e lo inoltra al driver del protocollo interno appropriato per essere consegnato localmente, o non lo inserisce nella coda di un driver di una periferica di rete affinché sia inoltrato verso un altro host. Il driver effettua le decisioni di instradamento usando due tabelle: la base persistente di informazioni di inoltro (forwarding information base, FIB), e una cache di decisioni recenti di instradamento. La FIB mantiene le informazioni di configurazione di instradamento e può specificare i percorsi basandosi sia su uno specifico indirizzo del destinatario sia su wildcard che rappresentano destinazioni multiple. La FIB è organizzata con insieme di tabelle di hash indicizzate dall'indirizzo di destinazione; le tabelle che rappresentano i percorsi più specifici sono sempre controllate per prime. Le ricerche con successo in questa tabella sono aggiunte alla tabella di cache di instradamento, in modo che le ricerche possano essere effettuate velocemente. Una voce nella cache di routing viene rimossa dopo un periodo prefissato in cui non viene usata.

In vari passaggi, il software IP passa pacchetti alla sezione separata di codice per la **gestione del firewall**: filtro selettivo di pacchetti in accordo a criteri arbitrari, generalmente a scopo di sicurezza. Il gestore del firewall mantiene una quantità separata di **firewall chain** (**catene di firewall**), e consente che uno `skbuff` sia controllato in ogni chain. Le catene sono separate per diversi scopi: una è utilizzata per i pacchetti inoltrati, una per i pacchetti inseriti verso l'host, e una per i dati generati dall'host. Ogni catena contiene una lista ordinata di regole, dove una regola specifica una delle possibili funzioni per la decisione del firewall, più alcuni dati arbitrari con cui eseguire il confronto.

Altre due funzioni eseguite dal driver IP sono smembramento e riassemblaggio di pacchetti voluminosi: se il pacchetto in uscita è troppo grande per essere accodato ad una periferica, viene semplicemente diviso in **frammenti** (**fragment**) più piccoli, che vengono tutti accodati al driver. Nella macchina ricevente, questi frammenti devono essere riassemblati. Il driver IP mantiene un oggetto `ipfrag` per ciascun frammento che attende di essere riassemblato, e un `ipq` per ciascun datagramma riassemblato. I frammenti in arrivo vengono confrontati con ciascun `ipq` noto, e se viene trovata la corrispondenza, il frammento viene aggiunto adesso; altrimenti ne viene creato uno

nuovo. Una volta che è arrivato il frammento finale di un `ipq`, viene costruito un `skbuff` completamente nuovo per contenere il nuovo pacchetto che viene passato al driver IP.

I pacchetti che IP riconosce come destinati a questo host vengono passati oltre ad un altro driver di protocollo. I protocolli UDP e TCP condividono l'associazione dei pacchetti con i socket sorgente e destinazione: ogni coppia di socket connessi è unicamente identificata dai suoi indirizzi sorgente e destinazione, e dai numeri di porta sorgente e destinazione. Le liste di socket sono connesse in tabelle di hash, le cui chiavi sono composte a questi quattro valori dell'indirizzo-porta, per la ricerca del socket all'arrivo dei pacchetti. Il protocollo TCP deve occuparsi anche di connessioni non affidabili, pertanto mantiene liste ordinate dei pacchetti in uscita che non hanno ricevuto una conferma di ricezione (acknowledge), per ritrasmetterli dopo un timeout, e dei pacchetti in arrivo non ordinati da presentare al socket quando saranno arrivati i dati mancanti.

11 Sicurezza

Il modello di sicurezza di Linux è fortemente correlato ai meccanismi tipici di sicurezza di UNIX. Le questioni di sicurezza possono essere classificate in due gruppi:

1. **Autenticazione:** assicurarsi che nessuno possa accedere al sistema senza prima aver dimostrato di avere i diritti di farlo.
2. **Controllo degli accessi:** fornitura di un meccanismo per controllare se un utente ha i diritti di accesso ad un determinato oggetto, e prevenirne l'accesso se necessario.

11.1 Autenticazione

In UNIX, l'autenticazione è stata tipicamente effettuata attraverso l'utilizzo di un file di password leggibili: la password di un utente è combinata con un valore casuale (detto "salt"), codificata con una funzione di trasformazione non invertibile e immagazzinata in un file di password. L'uso della funzione non invertibile significa che la password originale non può essere dedotta dal file di password se non per tentativi o con errori. Quando un utente presenta una password al sistema, questa viene di combinata con il salt e passata attraverso la stessa trasformazione non invertibile; se il risultato combacia con il contenuto del file, la password è accettata.

Storicamente, le implementazioni di UNIX di questo meccanismo hanno avuto molti problemi: le password erano spesso limitate a otto caratteri, e il numero dei possibili valori del salt era troppo basso, permettendo così che all'attaccante potesse facilmente combinare un dizionario di password comunemente usate, con ogni possibile valore del salt, e avere una buona possibilità di trovare uno o più password nel file, ottenendo come risultato l'accesso non autorizzato ad ogni account che veniva così compromesso. Sono state introdotte delle estensioni al meccanismo di password per mantenere il file di password criptato in un file che non era pubblicamente leggibile, che permetteva password più lunghe, o che utilizzava meccanismi più sicuri per codificare le password. Sono stati introdotti altri meccanismi di autenticazione che limitano le volte in cui un utente può connettersi a un sistema o che distribuiscono informazioni di autenticazione a tutti i sistemi correlati in una rete.

Un nuovo meccanismo di sicurezza è stato sviluppato dai venditori di UNIX per affrontare questi problemi. Il sistema **pluggable authentication modules** (**PAM, moduli inseribili di autenticazione**) è basato su una libreria condivisa che può essere utilizzata da ogni componente del sistema che necessiti di autenticare gli utenti. Un'implementazione di questo sistema è disponibile

per Linux. PAM permette che i moduli di autenticazione siano caricati su richiesta, come specificato in un file di configurazione di sistema. Se un nuovo meccanismo di autenticazione viene aggiunto successivamente, può essere aggiunto al file di configurazione e tutti i componenti del sistema saranno immediatamente in grado di avvantaggiarsi di esso. I moduli PAM possono specificare i metodi di autenticazione, le restrizioni degli account, le funzioni di configurazione della sessione, o le funzioni di cambio password (così che, quando gli utenti cambiano la loro password, tutti i meccanismi di autenticazione necessari possono essere aggiornati contemporaneamente).

11.2 Controllo degli accessi

Il controllo degli accessi nei sistemi UNIX, incluso Linux, viene effettuato attraverso l'utilizzo di identificatori numerici univoci. Un identificatore di utente (user identifier, uid) identifica un singolo utente o un singolo insieme di diritti di accesso. Un identificatore di gruppo (group identifier, gid) è un identificatore aggiuntivo che può essere utilizzato per identificare i diritti che appartengono a più di un utente.

Il controllo degli accessi è applicato a vari oggetti nel sistema: ogni file disponibile nel sistema è protetto dal meccanismo standard di controllo degli accessi; in aggiunta, altri oggetti condivisi, come le sezioni di memoria condivisa e i semafori, utilizzano lo stesso sistema di accesso.

Ogni oggetto in un sistema UNIX sotto il controllo dell'accesso dell'utente del gruppo, ha un solo uid ed un solo gid associati. Anche i processi utente hanno un singolo uid, ma possono avere più di un gid. Se lo uid di un processo corrisponde allo uid di un oggetto, allora il processo ha i **diritti utente** (**user rights**) o i **diritti del proprietario** (**owner rights**) per quell'oggetto, altrimenti se uno dei gid del processo corrisponde a quello dell'oggetto, gli vengono conferiti i **diritti di gruppo** (**group rights**); altrimenti il processo ha i **diritti del mondo** (**world rights**) su quell'oggetto.

Linux effettua il controllo degli accessi assegnando agli oggetti una **maschera di protezione** (**protection mask**) che specifica quali modi di accesso: lettura, scrittura o esecuzione debbano essere assegnati ai processi con i diritti di proprietario, gruppo o mondo. Pertanto, il proprietario di un oggetto può avere diritti di piena lettura, scrittura e di esecuzione di un file; agli altri utenti in un certo gruppo potrebbero essere stati dati i diritti di lettura ma non a quelli di scrittura; e chiunque altro potrebbe non avere nessun diritto in assoluto.

L'unica eccezione è lo uid privilegiato di **root**. Un processo con lo uid speciale possiede automaticamente l'accesso a ogni oggetto nel sistema, ignorando il normale controllo degli accessi. Questi processi hanno anche i permessi di eseguire operazioni privilegiate quali la lettura di una qualunque memoria fisica o l'apertura di socket di rete riservati. Questo meccanismo permette al kernel di impedire che gli utenti normali accedano a queste risorse: molte delle risorse chiave interne del kernel sono implicitamente possedute dallo uid di root.

Linux implementa il meccanismo standard `setuid` di UNIX, descritto nel paragrafo 3.2 del capitolo online “Il Sistema Operativo FreeBSD”, che permette a un programma di essere eseguito con privilegi differenti da quelli dell'utente che lo esegue: per esempio il programma lpr (chi invia un job sulla coda di una stampante) ha accesso alle code di stampa del sistema anche se l'utente che lo invoca non lo ha. L'implementazione di UNIX di `setuid` distingue fra uid *reale* ed *effettivo* del processo: quello reale è quello dell'utente che esegue il programma, quello effettivo è quello del proprietario del file.

In Linux questo meccanismo è potenziato in due modi. Per prima cosa, Linux implementa le specifiche POSIX del meccanismo `saved user-id`, che permette a un processo di abbandonare e

riacquisire ripetutamente il suo uid effettivo. Per ragioni di sicurezza, un programma può voler eseguire molte delle proprie operazioni in modalità sicura, rinunciando ai privilegi offerti dal suo stato `setuid`, ma può desiderare di eseguire alcune operazioni con tutti i suoi privilegi. Le implementazioni standard di UNIX eorniscono questa capacità solo scambiando lo uid reale con quello effettivo; lo uid effettivo procedente viene ricordato ma lo uid reale del programma non corrisponde sempre a quello dell'utente che lo esegue. Gli uid salvati permettono a un processo di impostare il proprio uid da effettivo a reale e poi di tornare indietro ai valori precedenti dello uid effettivo, senza dover modificare in nessun momento quello reale.

La seconda miglioria fornita da Linux è l'aggiunta di una caratteristica del processo che garantisce un sottoinsieme di diritti dello uid effettivo. Le proprietà **fsuid** e **fsgid** di un processo sono utilizzate quando vengono garantiti i diritti di accesso ad un file, e vengono impostati ogni volta che viene impostato lo uid o il gid effettivo. Comunque, fsuid e fsgid possono essere impostati indipendentemente dagli id effettivi, permettendo a un processo di accedere ai file per conto di un altro utente senza prendere l'identità dell'altro utente in nessun altro modo. In particolare, i processi server possono utilizzare questo meccanismo per fornire file a un certo utente senza diventare vulnerabili all'uccisione o alla sospensione da parte di quell'utente.

Linux fornisce un altro meccanismo che è diventato comune nelle versioni moderne di UNIX per il passaggio flessibile dei diritti da un programma all'altro: quando un socket della rete locale è stato impostato fra due processi nel sistema, entrambi i processi possono mandare l'uno all'altro un file descriptor per ognuno dei loro file aperti; l'altro processo riceve un duplicato del file descriptor per lo stesso file. Questo meccanismo permettere a un client di passare l'accesso a un singolo file selettivamente a un qualche processo server, senza garantire a quel processo nessun altro privilegio. Per esempio, non è più necessario per una server di stampa essere in grado di leggere tutti i file di un utente che invia una nuova attività di stampa: il client può semplicemente passare al server il file descriptor per ogni file che deve essere stampato, negando al server l'accesso a ogni altro file dell'utente.

12 Sommario

Linux è un moderno sistema operativo free basato sugli standard di UNIX. È stato progettato per girare in modo efficiente e affidabile sull'hardware dei comuni PC, ma può girare anche su una varietà di altre piattaforme. Fornisce un'interfaccia di programmazione e un'interfaccia utente compatibili con il sistema UNIX standard e può eseguire una vasta gamma di applicazioni UNIX, fra cui un numero crescente di applicazioni supportate commercialmente.

Linux non si è evoluto nel vuoto: un sistema Linux al completo include molti componenti che sono stati sviluppati in modo indipendente. Il nucleo del kernel del sistema operativo Linux è interamente originale, ma permette l'esecuzione di molto software free esistente per UNIX, dando origine ad un sistema operativo completamente compatibile con UNIX e libero da codice proprietario.

Il kernel di Linux è implementato come un kernel tradizionale monolitico per ragioni di prestazioni, ma è sufficientemente modulare nella sua progettazione da permettere a molti driver di essere caricati dinamicamente e scaricati durante l'esecuzione.

Linux è un sistema multiutente, che fornisce protezione tra processi e l'esecuzione di più processi secondo una schedulazione a condivisione di tempo. I processi appena creati possono condividere alcune parti del loro ambiente di esecuzione con i loro processi genitori, permettendo la programmazione multithread. La comunicazione fra processi è supportata sia dai meccanismi di

System V: code di messaggi, semafori, e memoria condivisa, sia dall'interfaccia socket di BSD. Attraverso l'interfaccia socket si può accedere a più protocolli di rete simultaneamente.

All'utente il file system appare come un albero di cartelle organizzate gerarchicamente che obbediscono alla semantica di UNIX. Internamente Linux utilizza uno strato di astrazione per gestire più tipi differenti di file system. Sono supportati il file system virtuale, di rete, e orientato alle periferiche. I file syistem orientati alle periferiche accedono all'immagazzinamento su disco attraverso due cache: i dati sono inseriti nella cache delle pagine che è unificata con un sistema di memoria virtuale, mentre i metadati sono inseriti in una cache di buffer: una cache separata indicizzata da blocchi fisici del disco.

Il sistema di gestione della memoria utilizza la condivisione delle pagine e la copia durante la scrittura per minimizzare la duplicazione dei dati condivisi da processi differenti. Le pagine vengono caricate su richiesta quando avviene il primo riferimento ad esse, e vengono scaricate in accordo a un algoritmo LFU, se c'è bisogno di memoria fisica.

Esercizi

1 Linux funziona su una varietà di piattaforme hardware. Quali passi devono compiere gli sviluppatori di Linux per assicurare che il sistema sia portabile su processori e architetture di gestione della memoria differenti, e per minimizzare la quantità di codice del kernel specifico per l'architettura?

2 I moduli del kernel caricati dinamicamente danno flessibilità quando vengono aggiunti dei driver al sistema. Hanno anche degli svantaggi? Sotto quali circostanze un kernel dovrebbe essere compilato come un singolo file binario? Quando sarebbe meglio mantenerlo spezzato in moduli? Giustificare le risposte.

3 Il multithreading è una tecnica di programmazione comunemente usata. descrivere tre modi in cui possono essere implementati i thread. spiegare come questi tre modi si comparano con il meccanismo `clone` di Linux. Quando ciascuno di questi meccanismi può essere meglio o peggio dell'uso dei cloni?

4 In quali costi aggiuntivi si incorre creando e schedulando un processo, rispetto al costo dei thread clonati?

5 Lo schedulatore di Linux implementa la schedulazione *soft* real time. Quali caratteristiche mancano, necessarie per certi task di programmazione real time? Come potrebbero venire aggiunte al kernel?

6 Il kernel di Linux non permette di depaginare la memoria del kernel. Quali effetti ha questa restrizione sulla progettazione del kernel? Quali sono due vantaggi e due svantaggi di questa decisione progettuale?

7 In Linux, le librerie condivise svolgono molte operazioni fondamentali per il sistema operativo. Quale è il vantaggio di tenere queste funzionalità fuori dal kernel? Ci sono delle controindicazioni? Spiegare la risposta.

8 Quali sono i tre vantaggi del link dinamico (o condiviso) delle librerie rispetto a quello statico? Quali sono i due casi in cui quello statico è preferibile?

9 Si confronti l'uso dei socket di rete con l'uso della memoria condivisa come meccanismo per comunicare i dati fra i processi su un singolo computer. Si citino due vantaggi di ciascun metodo. Quando ciascuno dei metodi può essere preferibile?

10 I sistemi UNIX erano soliti usare l'ottimizzazione della struttura del disco basati sulla posizione di rotazione dei dati sul disco, ma le implementazioni moderne, incluso Linux, ottimizzano semplicemente per l'accesso sequenziale ai dati. Perché si comportano così? Di quali caratteristiche hardware si avvantaggia l'accesso sequenziale? Perché l'ottimizzazione rotazionale non è più così utile?

11 Il codice sorgente di Linux è liberamente e ampiamente disponibile su Internet o da venditori di CD-ROM. Quali sono tre implicazioni di questa disponibilità riguardo la sicurezza di Linux?

Note bibliografiche

Il sistema Linux è un prodotto di Internet; come risultato, molta documentazione disponibile su Linux è disponibile in qualche forma su Internet. I seguenti siti fondamentali riportano molte delle informazioni utili disponibili:

- Le Linux Cross-Reference Pages presso http://lxr.Linux.no/ mantengono liste aggiornate del kernel di Linux, visibili via web e con riferimenti incrociati completi.
- Linux-HQ presso http://www.linuxhq.com/ ospita una grande quantità di informazioni riguardo al kernel di Linux 2.x. Questo sito include anche collegamenti alle home page della maggior parte delle distribuzioni di Linux, come pure archivi delle principali mailing list.
- Il Linux Documentation Project presso http://sunsite.unc.edu/linux/ elenca molti libri su Linux che sono disponibili in formato sorgente come parte del progetto di documentazione di Linux. Il progetto ospita anche le guide *how-to* (come fare) di Linux, che contengono una serie di suggerimenti e trucchi riguardanti aspetti di Linux.
- *Kernel Hackers' Guide* è una guida su Internet alle parti interne del kernel in generale. Questo sito in costante espansione è situato presso http://www.redhat.com:8080/HyperNews/get/khg.html.
- Il sito Kernel Newbies (http://www-kernelnewbies.org/) fornisce una risorsa per introdurre il kernel di Linux ai novizi.

Sono disponibili anche molte mailing list riguardo a Linux. Le più importanti sono mantenute da un gestore che può essere raggiunto tramite l'indirizzo e-mail majordomo@vger.rutgers.edu. Inviate una e-mail a questo indirizzo con la sola linea "help" nel corpo della e-mail per informazioni su come accedere al server delle liste e per iscriversi a ciascuna lista.

Infine, il sistema Linux stesso può essere ottenuto tramite Internet. Distribuzioni complete di Linux possono essere ottenute dai siti delle relative compagnie; la comunità di Linux mantiene in parecchi posti su Internet anche degli archivi dei componenti correnti del sistema. I più importanti sono:

- ftp://tsx-11.mit.edu/pub/linux/
- ftp://sunsite.unc.edu/pub/linux/
- ftp://Linux.kernel.org/pub/linux/

Oltre a investigare le risorse su Internet, sono disponibili, riguardo alle parti interne del kernel di Linux, Bovet e Cesati [2002] e Love [2004].