IL SISTEMA OPERATIVO FREEBSD

Questo capitolo presenta un esame approfondito del sistema operativo FreeBSD, una versione di UNIX. Analizzando un sistema completo e reale, potremo vedere come i concetti che abbiamo discusso finora siano relazionati sia fra loro che con le problematiche pratiche. Vedremo dapprima una breve storia di UNIX e presenteremo le interfacce utente e di programmazione del sistema operativo. Poi, discuteremo le strutture dati interne e gli algoritmi usati dal kernel di UNIX per supportare l'interfaccia utente e di programmazione.

1 Storia

La prima versione di UNIX è stata sviluppata nel 1969 da Ken Thompson del Gruppo di Ricerca dei Laboratori Bell per utilizzare un PDP-7, altrimenti inutilizzato. A Thompson presto si è unito Dennis Ritchie e, con altri membri del gruppo di ricerca, hanno prodotto le versioni iniziali di UNIX. Ritchie precedentemente aveva lavorato al progetto MULTICS, che ha avuto una forte influenza sul nascente sistema operativo. Anche il nome *UNIX* è un gioco di parole riferito a *MULTICS*. L'organizzazione di base del file system, l'idea dell'interprete dei comandi (o shell) come processo utente, l'uso di un processo separato per ogni comando, i caratteri originali di editing di linea (# per cancellare l'ultimo carattere e @ per cancellare l'intera linea) e numerose altre caratteristiche sono derivate direttamente da MULTICS. Sono state inoltre sfruttate idee provenienti da altri sistemi operativi, come CTSS dell'MIT ed il sistema XDS-940.

Ritchie e Thompson hanno lavorato tranquillamente su UNIX per molti anni. Con una seconda versione fu portato su un PDP-11/20 e per una terza versione, hanno riscritto la maggior parte del sistema operativo con il linguaggio di programmazione C, sostituendo il linguaggio Assembler, usato precedentemente. Il linguaggio C è stato sviluppato presso i laboratori della Bell per supportare UNIX. UNIX inoltre è stato spostato sui modelli più grandi di PDP- 11, come l'11/45 e l'11/70. La multiprogrammazione ed altri miglioramenti furono aggiunti quando fu riscritto in C e portato su sistemi (come l'11/45) che avevano un supporto hardware per la multiprogrammazione.

Durante lo sviluppo, UNIX veniva ampiamente usato all'interno dei laboratori della Bell e gradualmente si è diffuso in alcune università. La prima versione disponibile, ampiamente utilizzata al di fuori dei laboratori della Bell, fu la versione 6, rilasciata nel 1976. (Il numero di versione per i primi sistemi UNIX corrisponde al numero dell'edizione del *Manuale di programmazione di UNIX* corrente quando veniva fatta la distribuzione; il codice ed il manuale venivano modificati indipendentemente.)
Nel 1978 venne distribuita la versione 7; questa versione del sistema UNIX, he girava su PDP-11/70 e su Interdata 8/32, è il predecessore della maggior parte dei moderni sistemi UNIX. In particolare, presto fu portato su altri modelli PDP-11 e sulla linea di calcolatori VAX. La versione disponibile su VAX era conosciuta come 32V. Da allora la ricerca è continuata.

1.1 Il gruppo di supporto di UNIX

Dopo la distribuzione della versione 7 del 1978, il gruppo di supporto di UNIX (USG) ha assunto il controllo amministrativo e la responsabilità dal Gruppo di Ricerca per le distribuzioni di UNIX all'interno dell'AT&T, l'organizzazione a cui appartengoni i Bell Laboratories. UNIX stava

trasformandosi in un prodotto, invece di un semplice strumento di ricerca. Comunque il Gruppo di Ricerca ha continuato a sviluppare le versioni di UNIX per supportare le necessità di elaborazione interne. La versione 8 ha incluso il **sistema di I/O a flusso** (*stream I/O system*), che permette una configurazione flessibile dei moduli IPC del kernel. Inoltre ha incluso RFS, un file system remoto simile a quello NFS di Sun. La versione corrente è la 10, rilasciata nel 1989 e disponibile solo all'interno dei Laboratori Bell.

USG ha fornito principalmente supporto per UNIX in seno all'AT&T. La prima distribuzione esterna fornita da USG (nel 1982) fu il System III, che comprende caratteristiche tratte dalle versioni 7 e 32V, come pure caratteristiche di parecchi sistemi UNIX sviluppati da gruppi diversi da quello di Ricerca. Per esempio, sono state incluse nel System III le caratteristiche di UNIX/RT, un sistema UNIX in real-time e parecchie porzioni del pacchetto di strumenti software del Programmer's Work Bench (PWB).

USG ha rilasciato il System V nel 1983; in gran parte derivato dal System III. Lo scorporamento di varie aziende operative della Bell da parte di AT&T ha lasciato AT&T in una posizione tale da favorire l'introduzione aggressiva sul mercato del System V. USG è stato ristrutturato come Laboratorio di Sviluppo del System V di UNIX (USDL), che ha rilasciato la Release 2 (V.2) del System V di UNIX nel 1984. Il rilascio 2, versione 4 (V.2.4) del System V ha aggiunto una nuova realizzazione della memoria virtuale con copia in scrittura su paginazione e la memoria condivisa. USDL è stato, a sua volta, sostituito dai Sistemi Informativi di AT&T (ATTIS), che hanno distribuito System V Release 3 (V.3) nel 1987. V.3 adatta la realizzazione del sistema di I/O a flusso (stream) della V.8 e lo rende disponibile come *STREAM*. Inoltre include RFS, un file system remoto simile a NFS.

1.2 Berkeley inizia lo sviluppo

La piccola dimensione, la modularità e il disegno pulito dei primi sistemi UNIX portarono a lavori basati su UNIX da parte di numerose altre organizzazioni informatiche, quali Rand, BBN, le Università dell'Illinois, di Harvard, di Purdue e DEC. Il gruppo di sviluppo più influente, ad esclusione dei Laboratori Bell e dei gruppi di sviluppo UNIX di AT&T, è stata l'Università di California a Berkeley.

Bill Joy e Ozalp Babaoglu hanno fatto il primo lavoro sul Berkeley VAX UNIX nel 1978; hanno aggiunto la memoria virtuale, la paginazione su richiesta e la sostituzione di pagina alla versione 32V per produrre UNIX 3BSD. Questa versione era la prima a realizzare una di queste funzionalità su un qualsiasi sistema UNIX. Il grande spazio di memoria virtuale di 3BSD ha permesso lo sviluppo di programmi molto grandi, quale il LISP di Berkeley. Il lavoro sulla gestione della memoria ha convinto il Defense Advanced Research Projects Agency (DARPA) a finanziare Berkeley per lo sviluppo di un sistema standard UNIX per uso governativo; ne risultò 4BSD UNIX.

Il prodotto 4BSD per DARPA è stato guidato da un comitato di coordinamento che ha incluso molta gente notevole dalle comunità della rete e di UNIX. Uno degli obiettivi di questo progetto era di fornire il supporto per i protocolli di rete Internet di DARPA (TCP/IP). Questo supporto è stato fornito in un modo generale: è possibile in 4.2BSD comunicare uniformemente fra vari servizi di rete, comprese le reti locali (quali Ethernet e token ring) e reti più estese (quali NSFNET). Questa realizzazione era la motivazione più importante per la popolarità di questi protocolli. È stata usata come la base per la messa a punto di molti fornitori di sistemi di elaborazione UNIX e perfino con altri sistemi operativi. Ha consentito ad Internet di svilupparsi: da 60 reti collegate nel 1984, a più di 8.000 reti e 10 milioni di utenti stimati nel 1993.

In più, Berkeley ha adattato molte caratteristiche dei sistemi operativi contemporanei per migliorare l'architettura e la realizzazione di UNIX. Molte funzioni di editing di linea del terminale del sistema

operativo TENEX (TOPS-20) sono state fornite da un nuovo driver di terminale. A Berkeley sono stati scritti: una nuova interfaccia utente (la C Shell), un nuovo editor di testi (ex/vi), i compilatori per Pascal ed il LISP e molti nuovi programmi di sistema. Per 4.2BSD, certi miglioramenti di efficienza sono stati ispirati dal sistema operativo VMS.

Il software di UNIX proveniente da Berkeley è rilasciato nelle **Berkeley Software Distribution (BSD)**. È conveniente riferirsi ai sistemi del Berkeley VAX UNIX dopo 3BSD, come 4BSD, anche se ci sono stati realmente parecchi rilasci specifici, in particolare 4.1BSD e 4.2BSD. Le sigle generiche BSD e 4BSD sono usate per le distribuzioni su VAX e su PDP-11 di UNIX Berkeley. 4.2BSD, distribuito nel 1983, era il culmine del progetto originale UNIX Berkeley DARPA. 2.9BSD è la versione equivalente per i sistemi PDP-11.

Nel 1986 è stato rilasciato 4.3BSD. Era così simile a 4.2BSD che i suoi manuali descrivevano 4.2BSD più esaurientemente dei manuali originali per 4.2BSD. Includeva numerosi cambiamenti interni, tuttavia comprendeva soluzioni di anomalie e miglioramenti nelle prestazioni. Inoltre sono state aggiunte alcune nuove funzionalità, compreso il supporto ai protocolli Xerox Network System. 4.3BSD Tahoe è stata la versione seguente, rilasciata nel 1988. Introduceva un controllo migliorato di congestione della rete e delle prestazioni di TCP/IP. Le configurazioni del disco sono state separate dai driver di periferica e venivano ora lette al di fuori dei dischi stessi. Fu incluso un supporto ampliato per i fusi orari. 4.3BSD Tahoe fu veramente sviluppato sopra e per il sistema CCI Tahoe (Computer Console, Inc., Power 6 computer), piuttosto che per la solita base VAX. La versione corrispondente per PDP-11 è la 2.10.1BSD, distribuita dall'associazione USENIX che inoltre pubblica i manuali di 4.3BSD. Il rilascio di 4.32BSD Reno ha visto l'introduzione di una realizzazione del modello di rete ISO/OSI.

L'ultimo rilascio della Berkeley, il 4.4BSD, è terminato nel giugno 1993. Include il nuovo supporto della rete X.25 e la conformità agli standard POSIX. Inoltre ha un'organizzazione radicalmente nuova del file system, con una nuova interfaccia virtuale al file system e il supporto per pile di file system (*file system stack*), permettendo di sovrapporre i file system a strati uno sull'altro per facilitare l'introduzione di nuove caratteristiche. Una realizzazione di NFS è inclusa nella versione (Capitolo 16), come un nuovo file system basato su log (si consulti il Capitolo 14). Il sistema di memoria virtuale 4.4BSD è derivato da Mach (descritto nel paragrafo 9 del capitolo online “Sistemi operativi storici”). Vengono introdotti parecchi altri cambiamenti, quali l’aumentata sicurezza e una struttura migliorata del kernel. Con il rilascio della versione 4.4, Berkeley ha fermato i suoi sforzi di ricerca.

1.3 La diffusione di UNIX

4BSD era il sistema operativo scelto per i VAX fin dal suo rilascio iniziale (nel 1979) fino al rilascio di Ultrix, una realizzazione BSD della DEC. 4BSD è ancora la scelta migliore per molte installazioni di rete e di ricerca. Molte organizzazioni comprerebbero una licenza 32V e ordinerebbero 4BSD da Berkeley senza nemmeno preoccuparsi di ottenere un nastro 32V.

Comunque l'attuale insieme dei sistemi operativi UNIX non è limitato a quelli provenienti dalla Bell (che attualmente è di proprietà di Lucent Technology) e Berkeley. Sun Microsystems ha contribuito a diffondere UNIX BSD installandolo sulle proprie workstation. Siccome UNIX è cresciuto in popolarità, esso è stato portato su molti calcolatori e sistemi di elaborazione. E’stata creata un'ampia varietà di sistemi operativi UNIX e simil UNIX. DEC supporta, sulle proprie workstation, la propria versione di UNIX (chiamata Ultrix) e sta sostituendo Ultrix con un altro sistema operativo derivato da UNIX: OSF/1; Microsoft ha riscritto UNIX per la famiglia Intel 8088, chiamandolo XENIX ed il suo nuovo sistema operativo Windows NT è pesantemente influenzato da UNIX; IBM ha UNIX (AIX) sui

propri pc, workstation e mainframe. UNIX, infatti, è disponibile su quasi tutti i calcolatori di uso generale: funziona sui personal computer, sulle workstation, sui minicomputer, sui mainframe e sui supercomputer, da Apple Macintosh II a Cray II. In virtù della sua ampia disponibilità, è usato in ambienti che vanno da quello accademico a quello militare fino al controllo di processi di fabbricazione. La maggior parte di questi sistemi è basata sulla versione 7, sul System III, su 4.2BSD, o su System V.

La grande popolarità di UNIX fra i fornitori di computer, lo ha reso il più portabile dei sistemi operativi e gli utenti si possono aspettare un ambiente UNIX indipendente dallo specifico produttore di computer. Ma il gran numero di realizzazioni del sistema ha condotto a notevoli variazioni nelle interfacce di programmazione e utente distribuite dai fornitori. Ma per una vera indipendenza dal fornitore, gli sviluppatori di applicazioni hanno bisogno di interfacce di programmazione uniformi. Tali interfacce permetterebbero a tutte le applicazioni "UNIX" di funzionare su tutti i sistemi UNIX, cosa che non rispecchia certamente la situazione attuale. Questa questione è diventata importante allorché UNIX è diventato la piattaforma di sviluppo più importante e preferita per applicazioni che variano dalle basi di dati alla grafica e alle connessioni di rete: ciò ha determinato una forte domanda di mercato di uno standard per UNIX.

Sono in corso parecchi progetti di standardizzazione, a partire dallo */usr/group 1984 Standard* sostenuto dal gruppo di utenti industriali UniForum. Da allora, molti organismi ufficiali dello standard hanno proseguito lo sforzo, compresi IEEE e ISO (lo standard POSIX). Il consorzio internazionale X/Open Group ha completato XPG3, un ambiente di applicazioni comuni, che include lo standard di interfaccia IEEE. Purtroppo, XPG3 è basato su una bozza dello standard ANSI C, invece che sulla specifica finale e quindi ha bisogno di essere rifatto. XPG4 è stato rilasciato nel 1993. Nel 1989, l'organismo di normalizzazione ANSI ha standardizzato il linguaggio di programmazione C, producendo una specifica dell'ANSI C che i fornitori adottarono rapidamente.

Man mano che questi progetti proseguiranno, le varietà di UNIX convergeranno e condurranno ad un'unica interfaccia di programmazione di UNIX, permettendo a UNIX di diventare ancora più popolare. Infatti, due gruppi separati di potenti fornitori di UNIX stanno lavorando a questo problema: il gruppo guidato da AT&T, UNIX International (UI) e Open Software Foundation (OSF) si sono entrambi accordati per seguire lo standard POSIX. Recentemente, molti dei fornitori coinvolti in questi due gruppi si sono accordati per un'ulteriore standardizzazione (l'accordo COSE) nell'ambiente grafico a finestre Motif, e ONC+ (che comprende RPC di Sun e NFS) e i servizi di rete DCE (che include AFS e un pacchetto RPC).

AT&T ha sostituito il proprio gruppo ATTIS nel 1989 con UNIX Software Organization (USO), che ha distribuito il primo UNIX unificato: System V Release 4. Questo sistema unisce le caratteristiche di System V, 4.3BSD e SunOS della Sun, compresi i nomi di file lunghi, il file system Berkeley, la gestione della memoria virtuale, i collegamenti simbolici, gruppi multipli di accesso, il controllo dei job e segnali affidabili; è inoltre conforme allo standard pubblicato da POSIX: POSIX.1. Dopo che USO ha prodotto SVR4, è divenuta una filiale indipendente di AT&T, chiamata Unix System Laboratories (USL); nel 1993, è stato comprata da Novell, Inc.
La Figura 1 ricapitola le relazioni fra le varie versioni di UNIX.

Il sistema UNIX si è sviluppato su un progetto personale di due impiegati dei laboratori Bell fino a diventare un sistema operativo che è definito dagli Istituti di normalizzazione multinazionali. Per di più UNIX è un veicolo eccellente per lo studio accademico e crediamo che rimarrà una parte importante della teoria e della pratica dei sistemi operativi. Per esempio, Tunis, il sistema operativo Xinu e Minix è basato su concetti presi da UNIX, ma sviluppati esplicitamente per la didattica. C'è una pletora di sistemi collegati a UNIX per scopi di ricerca, inclusi Mach, Chorus, Comandos e Roisin. Gli sviluppatori originali, Ritchie e Thompson, furono premiati nel 1983 dall'Association for Computing Machinery Turing per il loro lavoro su UNIX.

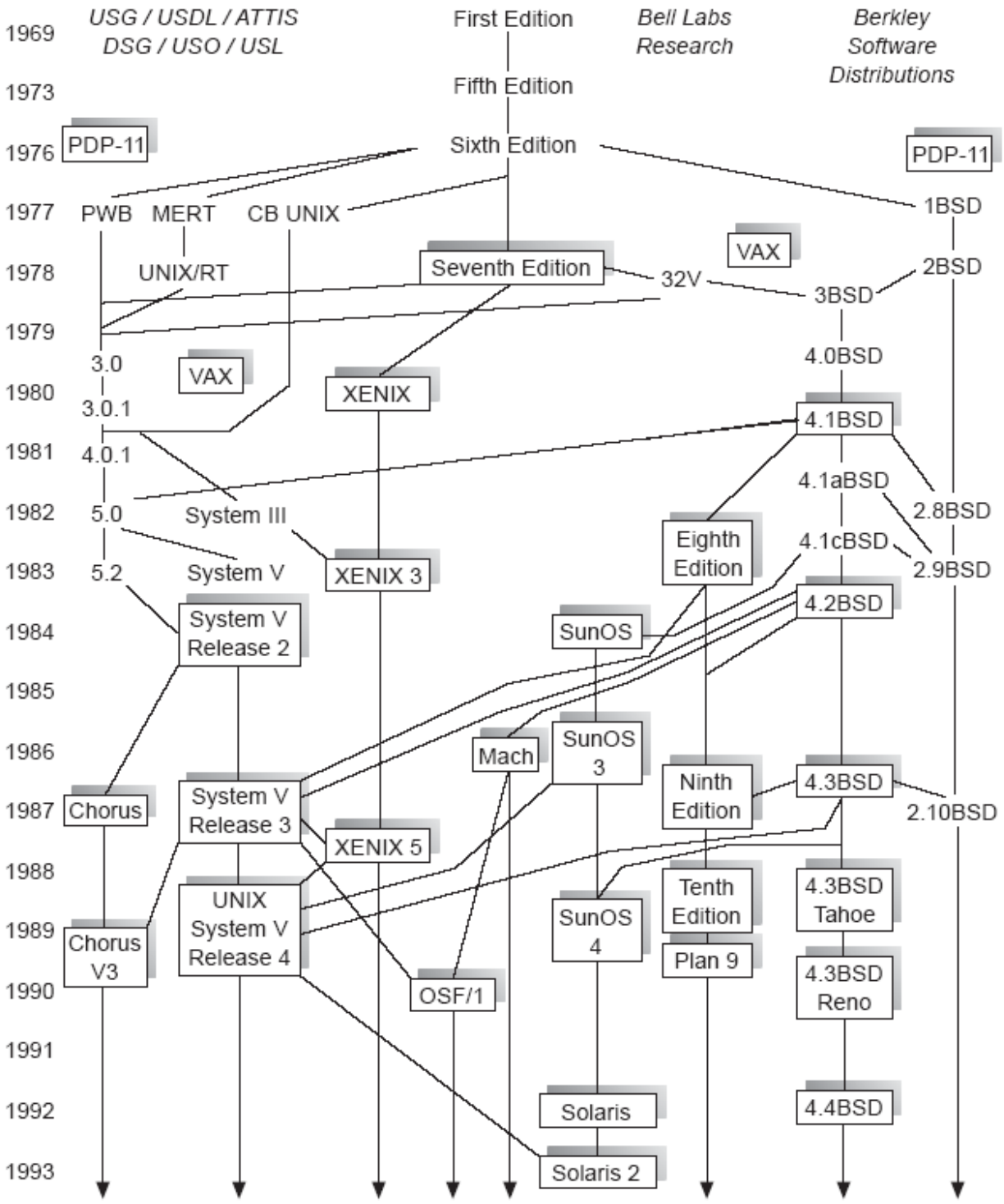

Figura 1. Storia delle versioni di UNIX**.**

First Edition = prima edizione; Fifht Edition = quarta edizione
Sixth Edition = sesta edizione; Seventh Edition = settima edizione
Eight Edition = ottava edizione; Ninht Edition = nona edizione
Tenth Edition = decima edizione; Release = Versione
Berkley software distribution = distribuzione software di Berkley

1.4 FreeBSD

La specifica versione di UNIX usata in questo capitolo è la versione Intel di FreeBSD. Questo sistema operativo realizza molti interessanti concetti riguardanti i sistemi operativi, quali la paginazione su richiesta con clustering (raggruppamento) e la gestione della rete. Il progetto FreeBSD è cominciato all'inizio del 1993 per produrre uni istantanea di 386BSD per risolvere problemi che non si potevano risolvere mediante patch (correzioni degli eseguibili). 386BSD è stato derivato da 4.3BSD-Lite (Net/2) ed è stato rilasciato nel mese di giugno del 1992 da William Jolitz. FreeBSD 1.0 (nome coniato da David Greenman) è stato rilasciato nel mese di dicembre del 1993. FreeBSD 1.1 è stato rilasciato nel mese di maggio 1994 ed entrambe le versioni erano basate sul codice 4.3BSD- Lite. Questioni legali fra UCB e Novell hanno provocato che il codice 4.3BSD-Lite non potesse più essere usato, di conseguenza il rilascio finale di 4.3BSD-Lite è avvenuto nel mese di luglio del 1994 (FreeBSD 1.1.5.1).

FreeBSD è stato reinventato basandosi sul codice di base del 4.4BSD-Lite, che era incompleto; fu così rilasciato FreeBSD 2.0 nel novembre del 1994. I rilasci successivi includono i rilasci di 2.0.5 nel giugno del 1995, 2.1.5 nell'agosto del 1996, 2.1.7.1 nel febbraio del 1997, 2.2.1 in aprile del 1997, 2.2.8 nel novembre del 1998, 3.0 nel mese di ottobre del 1998, 3.1 nel febbraio del 1999, 3.2 nel maggio 1999, 3.3 in settembre del 1999, 3.4 in dicembre del 1999, 3.5 nel giugno del 2000, 4.0 nel marzo del 2000, 4.1 nel luglio 2000 e 4.2 nel novembre del 2000.

L'obiettivo del progetto FreeBSD è di fornire un software che può essere utile per qualsiasi scopo senza vincoli. L'idea è che il codice otterrà il più ampio uso possibile e fornirà grandi benefici. Fondamentalmente, è la stessa idea descritta in McKusick et al. [1984] con l'aggiunta di una memoria virtuale incorporata, di una cache del buffer del file system, delle code del kernel e degli aggiornamenti leggeri del file system. Attualmente, funziona principalmente su piattaforme Intel, anche se sono supportate le piattaforme Alpha. È in corso un lavoro di trasferimento su altre piattaforme.

2 Principi progettuali

UNIX è stato progettato per essere un sistema a condivisione del tempo. L'interfaccia utente standard (shell) è semplice e, se lo si desidera, può essere sostituita da un'altra. Il file system è un albero multilivello, che permette agli utenti di creare i propri sottodirettori. Ogni file di dati dell'utente è semplicemente una sequenza di byte.

I file su disco e le periferiche di I/O sono trattati nel modo più simile possibile. Pertanto, le dipendenze dalla periferica e le particolarità sono mantenute il più possibile nel kernel; anche nel kernel, la maggior parte di esse sono confinate nei device-driver (driver del dispositivo).

UNIX supporta i processi multipli. Un processo può facilmente creare nuovi processi. La schedulazione della CPU è un semplice algoritmo basato sulle priorità. FreeBSD usa la paginazione su

richiesta come meccanismo per supportare la gestione della memoria e le decisioni di schedulazione della CPU. Lo swap viene usato nel caso un sistema sia sottoposto a eccessiva paginazione.

Poiché UNIX è stato progettato da Thompson e da Ritchie come un sistema operativo adatto alle loro esigenze, era abbastanza piccolo da poter essere capito. La maggior parte degli algoritmi sono stati scelti in base alla loro *semplicità*, non per la velocità o la sofisticazione. L'intenzione era di avere il kernel e le librerie che fornissero un piccolo insieme di funzionalità abbastanza potenti per permettere ad una persona di sviluppare, in caso di necessità, un sistema più complesso. Il progetto elegante di UNIX ha avuto numerose imitazioni ed elaborazioni.

Anche se i progettisti di UNIX avevano una significativa conoscenza di altri sistemi operativi, UNIX non ha avuto una progettazione elaborata presentata prima della sua implementazione. Questa flessibilità sembra essere uno dei fattori chiave nello sviluppo del sistema. Furono coinvolti alcuni principi progettuali, anche se non furono resi espliciti all'inizio.

Il sistema UNIX è stato progettato da programmatori per programmatori. Perciò è stato sempre interattivo e le funzionalità per lo sviluppo di applicazioni sono sempre state prioritarie. Tali funzionalità includono il programma *make* (che può essere usato per controllare quali file sorgente di un programma necessitano di venire compilati e poi di eseguire la compilazione) ed il Sistema di Controllo del Codice Sorgente (SCCS), che viene usato per mantenere disponibili versioni successive dei file senza dover salvare l'intero contenuto ad ogni passo. Il sistema principale di controllo della versione, usato da freebsd, è *Concurrent Versions System* (CVS) per controllare il grande numero di sviluppatori operanti su un codice comune.

Il sistema operativo è scritto principalmente in C, linguaggio che è stato sviluppato per supportare UNIX, poiché né Thompson né Ritchie amavano programmare in Assembler. La mancanza del linguaggio Assembler era pure necessaria a causa dell'incertezza sulle macchine in cui UNIX avrebbe funzionato. Ciò ha notevolmente facilitato i problemi di trasporto di UNIX da un sistema hardware ad un altro.

All'inizio, i sistemi di sviluppo di UNIX hanno avuto tutti i codici sorgente disponibili on-line, e gli sviluppatori hanno usato i sistemi in sviluppo come loro sistemi primari. Questo modello di sviluppo ha notevolmente facilitato la scoperta di anomalie e la loro soluzione, come pure nuove possibilità e rispettive realizzazioni. Ha pure incoraggiato un gran numero di varianti di UNIX attualmente esistenti, ma i benefici hanno superato gli svantaggi: se qualcosa non funziona correttamente, può essere corretto sul sito locale; senza la necessità di aspettare il successivo rilascio del sistema. Tali correzioni, come pure le nuove funzionalità, possono essere incorporate nelle distribuzioni successive.

I vincoli di dimensione del PDP-11 (e dei primi calcolatori utilizzati per UNIX) hanno forzato una certa eleganza. Mentre altri sistemi usano procedure elaborate per gestire condizioni patologiche, UNIX esegue solo un arresto controllato chiamato *panic*; invece di tentare di curare tali condizioni, UNIX cerca di prevenirle. Dove altri sistemi userebbero la bruta forza o la macroespansione, UNIX ha dovuto principalmente sviluppare soluzioni più ingegnose, o almeno più semplici.

Questi primi punti di forza di UNIX hanno contribuito parecchio alla sua popolarità, e a loro volta hanno prodotto nuove richieste che ne hanno messo alla prova la solidità. UNIX è stato usato per scopi quali la gestione di reti, la grafica e operazioni in real-time, che non sempre erano previste nel modello originario orientato verso l'elaborazione di testi. Pertanto, i cambiamenti sono stati fatti a determinate funzionalità interne e sono state aggiunte nuove interfacce di programmazione. Queste nuove funzionalità ed altre, in particolare interfacce a finestre, hanno richiesto grandi quantità di codice per supportarle, con un sostanziale aumento della dimensione del sistema. Per esempio, il supporto di rete e le finestre hanno raddoppiato la dimensione del sistema. Questo modello a sua volta ha evidenziato la forza di UNIX: ogni volta che c'è bisogno di un nuovo sviluppo per l'industria, UNIX può solitamente assorbirlo, ma rimane sempre UNIX.

3 L'interfaccia di programmazione

Come la maggior parte dei sistemi operativi, UNIX consiste di due parti separabili: il kernel ed i programmi di sistema. Possiamo considerare il sistema operativo UNIX come fatto a strati, come è mostrato in Figura 2. Tutto ciò che è sotto l'interfaccia di chiamata di sistema e sopra l'hardware fisico è il *kernel*. Il kernel fornisce il file system, la schedulazione della CPU, la gestione della memoria ed altre funzioni del sistema operativo attraverso chiamate di sistema. I programmi di sistema usano le chiamate di sistema supportate dal kernel per fornire utili funzionalità, quali la compilazione e la manipolazione di file

Le chiamate di sistema definiscono l'*interfaccia di programmazione* di UNIX; l'insieme dei programmi di sistema, comunemente disponibili, si definisce *interfaccia utente*. L'interfaccia utente e di programmazione definiscono il contesto che il kernel deve supportare.
La maggior parte dei programmi di sistema è scritta in C, e il *Manuale di Programmazione UNIX* descrive tutte le chiamate di sistema come funzioni C. Un programma di sistema scritto in C per FreeBSD sul Pentium può essere portato su un sistema FreeBSD su Alpha e semplicemente ricompilato, anche se i due sistemi sono proprio differenti. I dettagli delle chiamate di sistema sono conosciuti soltanto al compilatore; questa caratteristica è una fattore importante per la portabilità dei programmi UNIX.

Le chiamate di sistema di UNIX possono essere approssimativamente raggruppate in tre categorie: manipolazione dei file, controllo del processo e manipolazione delle informazioni. Nel Capitolo 3, abbiamo elencato una quarta categoria, gestione delle periferiche, ma poiché le periferiche in UNIX sono trattate come file (speciali), le stesse chiamate di sistema supportano sia i file che le periferiche (anche se c'è una chiamata di sistema supplementare per la configurazione dei parametri del dispositivo).

3.1 I file

Un file in UNIX è una sequenza di byte. Differenti programmi si aspettano vari tipi di struttura, ma il kernel non impone una struttura ai file. Per esempio, la convenzione per i file di testo è quella di avere linee di caratteri ASCII separati da un singolo carattere "a capo" (newline), che in ASCII è il carattere "linefeed", ma il kernel non conosce niente su questa convenzione.
I file sono organizzati in *direttori* strutturati ad albero. I direttori sono essi stessi file che contengono informazioni su come trovare altri file. Un percorso ad un file (*path name)* è una stringa di testo che identifica un file specificando il percorso del file attraverso la struttura dei direttori. Sintatticamente, consiste di elementi individuali del file-name separati dal carattere "*/*". Per esempio, in /usr/local/font, il primo "*/*" indica la radice dell'albero del direttorio, chiamato *root*. Il successivo elemento: *usr*, è un sottodirettorio della root, *local* è un sottodirettorio di *usr* e *font* è un file o un direttorio nel direttorio *local*. Se *font* è un normale file o direttorio, non può essere determinato a partire dalla sintassi del path-name.

Il file system di UNIX ha sia *path-name assoluti* che *path-name relativi*, quelli assoluti incominciano dalla root del file system e sono distinti da uno slash iniziale nel path-name: /usr/local/font è un path-name assoluto. I path-name relativi cominciano dal *direttorio corrente*, che è un attributo del processo che accede al path-name. Pertanto, *local/font* indica un file o un direttorio che si chiama *font* nel direttorio *local* del direttorio corrente, che potrebbe essere o meno /usr.

Un file può essere individuato da più di un nome in uno o più direttori. Tali nomi multipli sono conosciuti come *link* (collegamenti) e tutti i link sono trattati nello stesso modo dal sistema operativo.

FreeBSD supporta inoltre *link simbolici* (symbolic link), che sono file che contengono il path-name di un altro file. I due tipi di link sono anche noti come *hard link* e *soft link*. I soft link (simbolici), a differenza degli hard link, possono puntare a direttori e possono attraversare i confini del file system.

Il nome del file "." in un direttorio è un hard link al direttorio stesso. Il nome di file ".." è un hard link al direttorio padre. Quindi, se il direttorio corrente è */user/jlp/programs*, allora *../bin/wdf* si riferisce a */user/jlp/bin/wdf*.

<table>
<tr><td colspan="3">(gli utenti)</td></tr>
<tr><td colspan="3">shell e comandi
compilatori e interpreti
librerie di sistema</td></tr>
<tr><td colspan="3">chiamate di sistema - interfaccia al kernel</td></tr>
<tr><td>terminale
sistema I/O a caratteri
drivers di terminale</td><td>file system
blocchi di swapping del sistema di I/O
driver per disco e nastro</td><td>schedulazione della CPU
sostituzione di pagina
paginazione su richiesta
memoria virtuale</td></tr>
<tr><td colspan="3">interfaccia del kernel all'hardware</td></tr>
<tr><td>controller dei
terminali</td><td>controller delle periferiche
dischi e nastri</td><td>controller della memoria
memoria fisica</td></tr>
</table>

Figura 2. Struttura a strati del 4.4BSD.

Le periferiche hardware hanno i nomi nel file system. Questi *file speciali di periferica* o *file speciali* sono conosciuti dal kernel come interfacce di periferica, ma nonostante ciò l'utente accede ad essi con le stesse chiamate di sistema degli altri file.

La figura 3 mostra un tipico file system di UNIX. La root (/) contiene normalmente un piccolo numero di direttori come */kernel*, l'immagine binaria per l'avvio (*bootstrap*) del sistema operativo; */dev* contiene i file speciali delle periferiche, quali */dev/console*, */dev/lp0*, */dev/mt0* e così via; */bin* contiene i file binari dei programmi di sistema essenziali UNIX. Altri file binari possono trovarsi in */usr/bin* (programmi per applicazioni di sistema, quali i formattatori del testo), in */usr/compat* per programmi provenienti da altri sistemi operativi quali Linux o in */usr/local/bin* (per programmi di sistema scritti localmente). I file di libreria, come le sottoprocedure di libreria di C, Pascal e Fortran, sono tenuti in */lib* (o */usr/lib* o */usr/local/lib*).

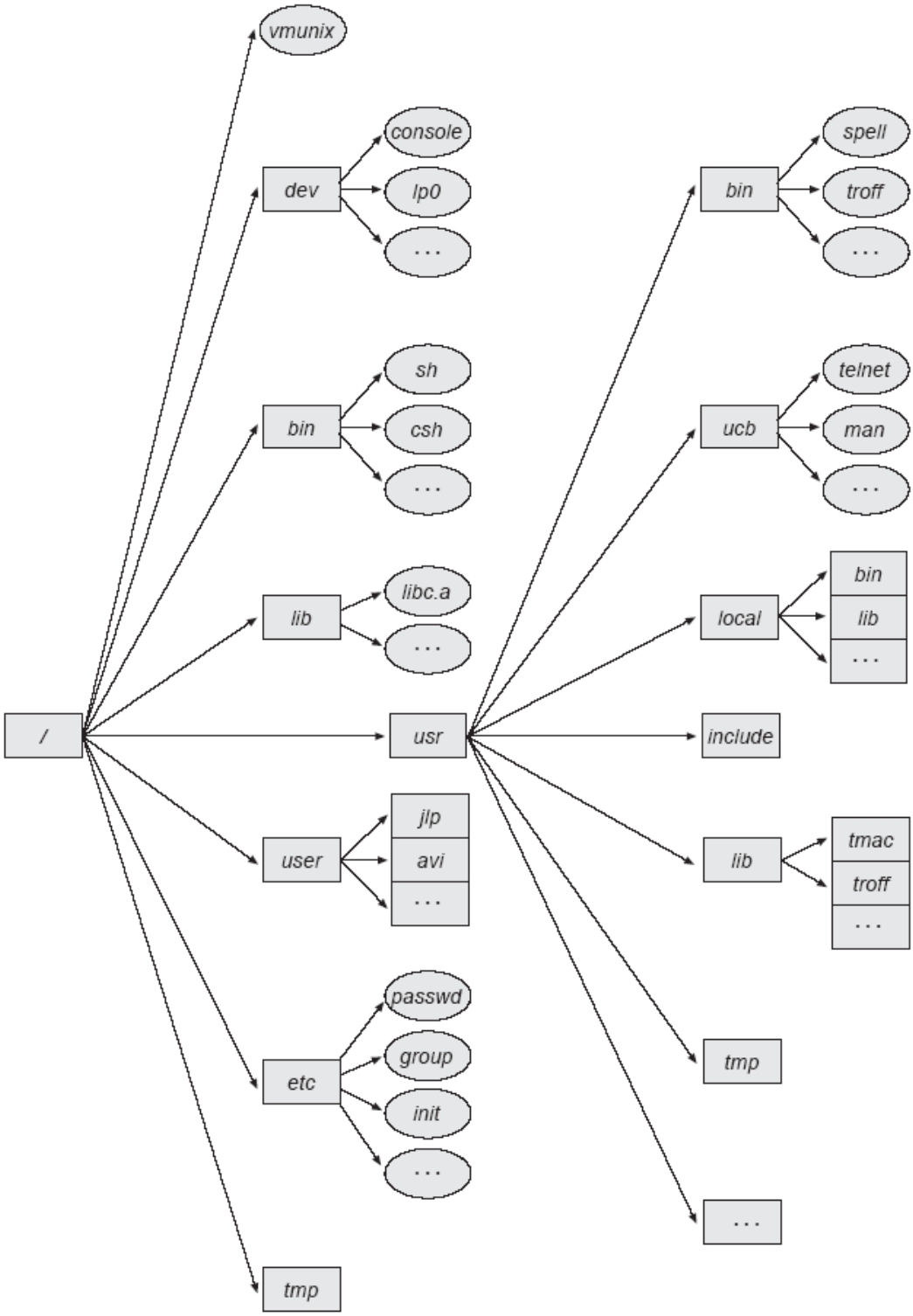

Figura 3. Una tipica struttura del direttorio UNIX.

I file degli utenti sono memorizzati in un direttorio separato per ogni utente, tipicamente in */usr*. Così, il direttorio utente di *Carol* si trova normalmente in */usr/carol*. In un grande sistema, questi direttori possono essere ulteriormente raggruppati per facilitarne la gestione, creando una struttura di file con */usr/prof/avi* e */usr/staff/carol*. I file di tipo amministrativo ed i programmi, quale il file delle *password*, sono tenuti in */etc*. I file temporanei possono essere messi in */tmp*, che viene normalmente cancellato durante la fase di avvio del sistema, o in */usr/tmp*.

Ciascuno di questi direttori può avere più strutture diverse. Per esempio, le tabelle descrizione dei font per il formattatore troff per l'impaginatore Merganthaler 202 sono tenute in */usr/lib/troff/dev202*. Tutte le convenzioni riguardanti la posizione di file specifici e di direttori sono state definite dai programmatori e dai loro programmi; il kernel del sistema operativo ha bisogno solo di */etc/init*, che viene usato, per essere operativo, per inizializzare i processi di terminale.

Le chiamate di sistema per la manipolazione di base dei file sono **creat**, **open**, **read**, **write**, **close**, **unlink** e **trunc**. La chiamata **creat**, una volta assegnato un path-name, genera un file vuoto o ne tronca uno esistente. Mediante la chiamata **open** viene aperto un file esistente che prende un path-name e una modalità (quale read, write, o read-write) e ritorna un piccolo numero intero, chiamato *descrittore del file*. Un descrittore del file può poi essere passato ad una chiamata di sistema **read** o **write** (insieme con l'indirizzo di un buffer e ad il numero di byte da trasferire) per effettuare trasferimenti di dati verso il file o dal file. Un file viene chiuso quando il proprio descrittore del file è passato alla chiamata di sistema **close**. La chiamata **trunc** riduce la lunghezza di un file a 0.

Un descrittore del file è un indice in una piccola tabella di file aperti per il processo. I descrittori cominciano da 0 e raramente, in programmi tipici, assumono valori superiori a 6 o a 7, a causa del numero massimo di file aperti simultaneamente.

Ogni chiamata **read** o **write** aggiorna lo spiazzamento (offset) corrente nel file, che è associato con l'ingresso corrispondente nella tabella dei file ed è usato per determinare la posizione nel file per la successiva **read** o **write**. La chiamata di sistema **lseek** permette di ripristinare esplicitamente la posizione; inoltre permette la creazione di file sparsi (cioè file contenenti "buchi").

Le chiamate di sistema **dup** e **dup2** possono essere usate per produrre un nuovo descrittore del file che è una copia di uno esistente. Anche la chiamata **fcntl** può fare questa duplicazione ed in più può esaminare o configurare i vari parametri di un file aperto. Ad esempio, può fare sì che ogni **write** successiva scriva in un file aperto aggiungendo (append) alla fine di quel file. Esiste una chiamata supplementare, **ioctl,** per manipolare i parametri di una periferica; per esempio, si può stabilire il baud rate di una porta seriale, o riavvolgere un nastro.

Le informazioni su un file (quali la dimensione, le protezioni di accesso, il proprietario ed e così via) possono essere ottenute dalla chiamata di sistema **stat**. Parecchie chiamate permettono di cambiare alcune di queste informazioni: **rename** (cambia il nome del file), **chmod** (cambia il modo di protezione) e **chown** (cambia il proprietario ed il gruppo). Molte di queste chiamate hanno varianti che si applicano ai descrittori del file anziché ai nomi del file. La chiamata di sistema **link** crea un hard link per un file esistente, creando un nuovo nome per un file esistente. Un link viene rimosso con la chiamata **unlink**; se è l'ultimo link, il file viene cancellato. La chiamata di sistema **symlink** crea un link simbolico.

I direttori sono creati mediante la chiamata di sistema **mkdir** e vengono cancellati con **rmdir**. Il direttorio corrente viene cambiato con **cd**. Diversamente dai file standard i direttori hanno una struttura interna che deve essere preservata. Invero, un altro insieme di chiamate viene fornito per aprire un direttorio, scorrere ogni file all'interno del direttorio, chiuderlo e per eseguire altre funzioni; queste sono **opendir, readdir, closedir** e altre.

3.2 I processi

Un *processo* è un programma in esecuzione. I processi sono identificati dal proprio *identificatore del processo*, che è un numero intero. Un nuovo processo viene creato dalla chiamata di sistema **fork**. Il nuovo processo consiste in una copia dello spazio di indirizzamento del processo originale (lo stesso programma e le stesse variabili con gli stessi valori). Entrambi i processi (padre e figlio) continuano l'esecuzione in corrispondenza dell'istruzione dopo **fork** con una differenza: il codice di ritorno di **fork** è zero per il nuovo processo (figlio), mentre un identificatore del processo figlio (diverso da zero) viene restituito al padre.

Tipicamente, la chiamata di sistema **execve** è usata dopo una fork di uno dei due processi per sostituire lo spazio di memoria virtuale dei processi con un nuovo programma. La chiamata **execve** carica un file binario in memoria (distruggendo l'immagine della memoria del programma che contiene la chiamata di sistema **execve**) e inizia la propria esecuzione.

Un processo può terminare usando la chiamata di sistema **exit** ed il processo padre può aspettare quell'evento usando la chiamata di sistema **wait**. Se il processo figlio si blocca, il sistema simula la chiamata **exit**. La chiamata di sistema **wait** fornisce l'identificatore di un processo figlio terminato, in modo che il padre possa dire quale dei tanti processi figlio è terminato. Una seconda chiamata di sistema **wait3**, è simile a **wait,** ma permette in più che il padre raccolga statistiche sulle prestazioni dei figli. Fra l'istante di tempo in cui il figlio è in esecuzione e quello in cui il padre completa una delle chiamate **wait**, il figlio è definito *defunto* (defunct). Un processo defunto non può fare niente, ma esiste solo in modo che il padre possa raccogliere le informazioni di stato. Se il processo padre di un processo defunto termina prima di quello figlio, il processo defunto viene ereditato dal processo *init* (che a sua volta attende che termini) e si trasforma in un processo *zombie*. Un uso tipico di queste funzionalità è indicato nella Figura 4.

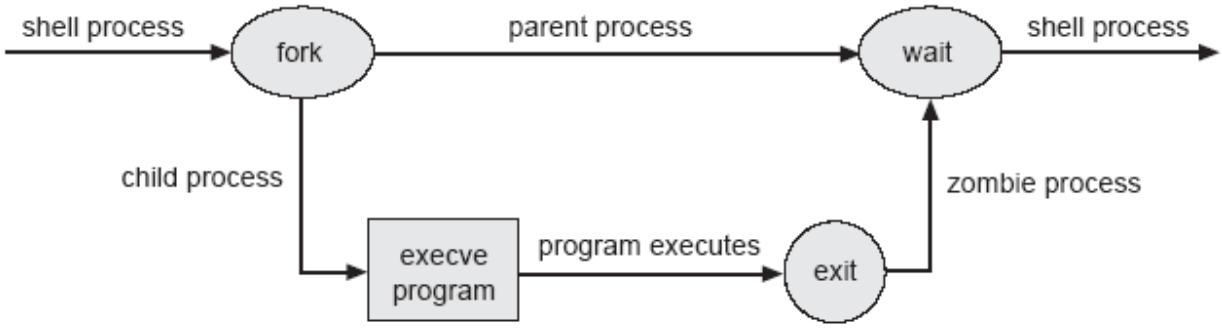

Figura 4. Una shell crea un sotto processo per eseguire un programma.

Shell process =processo shell; fork
parent process = processo padre; wait =attendi
shell process = processo shell; child process = processo figlio
zombie process = processo zombie; execve program = programma execve
program executes = esecuzione del programma
exit = uscita

La forma più semplice di comunicazione fra processi è realizzata tramite le *pipe*, che possono essere create prima della **fork,** i cui punti terminali sono installati fra **fork** e **execve**. Una pipe è essenzialmente una coda di byte fra due processi. Si accede ad una pipe tramite un descrittore del file, come se fosse un ordinario file. Un processo scrive nella pipe e l'altro legge dalla pipe. La dimensione della pipe originale viene fissata dal sistema. In FreeBSD le pipe sono realizzate sopra il sistema dei socket, con dei buffer di dimensioni variabili. La lettura da una pipe vuota o la scrittura in una pipe piena causa un blocco del processo finché non cambia lo stato della pipe. Operazioni speciali sono necessarie affinché una pipe sia messa fra un padre e un figlio (in modo che solo uno sia in lettura e l'altro in scrittura).
Tutti i processi utente sono discendenti di un processo originale, chiamato *init*. L'utente ha un processo *getty* generato da init. Il processo *getty* inizializza i parametri del terminale di linea ed attende che l'utente inserisca il suo nome di *login*, che viene passato attraverso una **execve** come argomento ad un processo di *login*. Il processo di *login* prende la password dell'utente, la cripta e confronta il risultato con una stringa criptata presa dal file */etc/passwd*. Se il confronto ha successo, all'utente viene permesso di entrare. Il processo di *login* esegue una *shell*, o interprete dei comandi, dopo aver configurato l'*identificativo utente* del processo relativamente all'utente che sta effettuando l'operazione di *login*. (La shell e l'identificativo utente vengono trovati in */etc/passwd* attraverso il nome di login dell'utente.) È con questa shell che l'utente normalmente comunica per il resto della sessione di login; la shell stessa crea sottoprocessi per i comandi che l'utente ordina di eseguire.

L'identificativo dell'utente è usato dal kernel per determinare i permessi dell'utente per certe chiamate di sistema, ed in particolare per quelle che riguardano gli accessi ai file. C'è anche un *identificativo del gruppo*, che è usato per fornire privilegi similari ad un gruppo di utenti. In FreeBSD un processo può appartenere simultaneamente a parecchi gruppi. Il processo di login mette la shell in tutti i gruppi consentiti all'utente per mezzo dei file */etc/passwd* e */etc/group.*

Il kernel utilizza due diversi identificativi utente: l'*identificativo utente effettivo* è usato per determinare i permessi di accesso ai file. Se un file di un programma caricato da una **execve** ha posto il bit **setuid** nel proprio inode, l'identificativo utente effettivo del processo è posto uguale all'identificativo utente proprietario del file, mentre l'*identificativo utente reale* è lasciato come era. Questo schema permette a certi processi di avere più dei privilegi ordinari anche se sono eseguiti da utenti ordinari. L'idea del **setuid** è stata brevettata da Dennis Ritchie (brevetto U.S. 4.135.240) ed è una delle caratteristiche peculiari di UNIX. Per i gruppi esiste un bit simile: **setgid**. Un processo può determinare il suo identificativo utente reale ed effettivo, rispettivamente con le chiamate **getuid** e **geteuid**. Le chiamate **getgid** e **getegid** determinano rispettivamente per un processo l'identificativo reale ed effettivo del gruppo. Il resto dei gruppi dei processi può essere trovato con la chiamata di sistema dei **getgroups**.

3.3 I segnali

I *signal* (segnali) sono una funzionalità per la gestione di circostanze eccezionali simili agli interrupt software. Ci sono 20 segnali differenti, ciascuno dei quali corrisponde ad uno stato differente. Un segnale può essere generato da un interrupt di tastiera, da un errore in un processo (quale un errato riferimento alla memoria), o da un certo numero di eventi asincroni (quali contatori (timer) o segnali di controllo di un job dalla shell). Quasi tutti i segnali possono inoltre essere generati dalla chiamata di sistema **kill**.

Il segnale di *interrupt*, SIGINT, è usato per arrestare un comando prima di completare l'esecuzione. È prodotto solitamente dal carattere ^C (ASCII 3). A partire da 4.2BSD, i caratteri importanti della tastiera sono definiti da una tabella per ogni terminale e possono essere ridefiniti facilmente. Il segnale *quit*, SIGQUIT, è prodotto solitamente dal carattere ^bs (ASCII 28), e arresta il programma attualmente in esecuzione e inoltre provoca il salvataggio (*dump)* della propria immagine corrente di memoria in un file chiamato *core* nel direttorio corrente. Il file *core* può essere usato dai programmi di debug. SIGILL è prodotto da un'istruzione illegale e SIGSEGV da un tentativo di indirizzare la memoria fuori dello spazio legale di memoria virtuale di un processo.

Si può fare in modo che la maggior parte dei segnali siano ignorati (non abbiano effetto), o che venga chiamata una procedura nel processo utente (un signal handler). Una funzione signal handler può fare, in modo sicuro, una di queste due cose prima di ritornare dall'aver preso in carico un segnale: eseguire la chiamata di sistema **exit** o modificare una variabile globale. Un segnale (il segnale *kill*, numero 9, SIGKILL) non può essere ignorato o gestito da un signal handler. SIGKILL è usato, per esempio, per uccidere un processo instabile che sta ignorando altri segnali quali SIGINT o SIGQUIT.

I segnali possono essere persi: se un altro segnale dello stesso tipo è trasmesso prima che uno precedente sia stato accettato dal processo verso cui è diretto, il primo segnale sarà sovrascritto e solo l'ultimo segnale sarà visto dal processo. In altre parole una chiamata al signal handler avvisa un processo che c'è almeno un segnale. Inoltre, non c'è priorità relativa fra i segnali di UNIX. Se due segnali differenti sono trasmessi allo stesso processo nel medesimo istante, non possiamo sapere quale dei due sarà ricevuto per primo dal processo.

I segnali sono stati originariamente concepiti per trattare eventi eccezionali. Come è vero per l'uso della maggior parte delle caratteristiche di UNIX, tuttavia, l'uso dei segnali è stato ampliato costantemente. BSD4.1 ha introdotto il controllo dei job, che usa i segnali per far partire ed arrestare i sottoprocessi a richiesta. Questa funzione permette ad una shell di controllare molti processi: far partire, arrestare e mettere in background un processo, come l'utente desidera. Il segnale SIGWINCH, inventato da Sun Microsystems, è usato per informare un processo che la finestra, in cui è visualizzato l'output, ha cambiato dimensione. I segnali sono pure usati per trasmettere dati urgenti alle connessioni di rete.

Gli utenti inoltre volevano segnali più affidabili e una correzione di un baco (bug) in una corsa critica congenita nella vecchia realizzazione dei segnali. Così, 4.2BSD ha portato una nuova messa a punto della gestione dei segnali, priva di corse critiche, affidabile e realizzata separatamente; permette che segnali individuali siano bloccati durante le sezioni critiche, e possiede una nuova chiamata di sistema per permettere ad un processo di dormire (sleep) fino a che non venga interrotto. È simile alla funzionalità di interrupt hardware; tale funzionalità fa ora parte dello standard POSIX.

3.4 Gruppi di processi

Capita frequentemente che gruppi di processi collegati cooperino per eseguire un task comune. Per esempio, i processi possono generare e comunicare su pipe. Un tale insieme di processi è chiamato *gruppo di processi*, o *job*. I segnali possono essere inviati a tutti i processi in un gruppo. Un processo solitamente eredita il suo gruppo dal proprio padre, ma la chiamata di sistema **setpgrp** permette ad un processo di cambiare il proprio gruppo.

I gruppi di processi sono usati dalla shell del C per controllare il funzionamento di job multipli. Solo un gruppo di processi può utilizzare, in un qualunque momento, un dispositivo terminale per I/O. Questo job in *foreground* ha l'attenzione dell'utente su quel terminale mentre tutti gli altri job non

attaccati al terminale (job in *background*) effettuano la loro esecuzione senza interagire con l'utente. L'accesso al terminale è controllato dai segnali del gruppo di processi: ogni job ha un *terminale di controllo* (ancora, ereditato dal padre); se il gruppo di processi del terminale di controllo coincide con il gruppo di un processo, quel processo è in foreground e gli è permesso di effettuare operazioni di I/O. Se un processo non appartiene al gruppo (background) e tenta di fare la stessa cosa, viene trasmesso al suo gruppo di processi un segnale SIGTTOU o SIGTTIN, che solitamente provoca il congelamento del gruppo di processi fino a che non sia posto in foreground dall'utente; a quel punto riceve un segnale SIGCONT, per indicare che il processo può effettuare un'operazione di I/O. In modo simile, SIGSTOP può essere mandato al gruppo di processi in foreground per congelarlo.

3.5 Informazioni per la gestione del sistema

Esistono chiamate di sistema per configurare e leggere sia un timer (**getitimer/setitimer**) che il tempo corrente (**gettimeofday/settimeofday**) in microsecondi. In più, i processi possono chiedere il proprio identificativo (**getpid**), quello di gruppo (**getgid**), il nome della macchina su cui stanno eseguendo (**gethostname**) e molti altri valori.

3.6 Procedure di libreria

L'interfaccia delle chiamate di sistema verso UNIX è supportata da un'ampia raccolta di procedure di libreria di e dai file di intestazione. I file dell'intestazione forniscono la definizione delle complesse strutture dati usate nelle chiamate di sistema. Inoltre, una grande libreria di funzioni fornisce un ulteriore supporto ai programmi.

Per esempio, le chiamate di sistema UNIX di I/O forniscono la possibilità di leggere e scrivere dei blocchi di byte. Alcune applicazioni possono voler leggere e scrivere solo un byte alla volta. Anche se è possibile, questo richiederebbe una chiamata di sistema per ogni byte: un grande overhead (sovraccarico). Invece, un insieme di procedure standard di libreria (il pacchetto standard di I/O a cui si accede attraverso il file di intestazione <*stdio.h*>) fornisce un'altra interfaccia, che legge e scrive parecchie migliaia di byte alla volta usando buffer locali e trasferendoli fra questi buffer (nella memoria utente) quando si vuole fare un'operazione di I/O. È pure supportato l'I/O formattato dal pacchetto standard di I/O.

Un supporto supplementare è fornito dalla libreria delle funzioni matematiche, dall'accesso alla rete, dalla conversione dei dati e così via. Il kernel FreeBSD supporta oltre 300 chiamate di sistema; la libreria di programma in C ha oltre 300 funzioni di libreria. Le funzioni di libreria possono provocare chiamate di sistema (per esempio, la procedura di libreria *getchar* provocherà una chiamata di sistema **read** se il buffer del file è vuoto). Comunque, il programmatore generalmente non deve distinguere fra l'insieme base delle chiamate di sistema al kernel e le funzioni addizionali fornite dalle funzioni di libreria.

4 Interfaccia utente

Sia il programmatore che l'utente di un sistema UNIX utilizzano principalmente un'insieme di programmi di sistema che sono stati scritti e sono disponibili per l'esecuzione. Questi programmi

eseguono le necessarie chiamate di sistema per supportare le proprie funzioni, e le chiamate stesse di sistema sono contenute nel programma, ma ciò non deve essere evidente all'utente.

I comuni programmi di sistema possono essere raggruppati in parecchie categorie; la maggior parte di essi è orientata alla gestione di file o direttori. Per esempio, i programmi di sistema per gestire i direttori sono *mkdir* per creare un nuovo direttorio, *rmdir* per rimuovere un direttorio, *cd* per passare dal direttorio corrente ad un altro e *pwd* per stampare il path name assoluto del direttorio (di lavoro) corrente.

Il programma *ls* elenca i nomi dei file nel direttorio corrente. Inoltre ci sono 28 opzioni per chiedere la visualizzazione delle proprietà dei file. Per esempio, con l'opzione *-l* si chiede un elenco esteso, che mostra il nome del file, il proprietario, la protezione, la data e l'ora di creazione e la dimensione. Il programma *cp* crea un nuovo file che è una copia di un file esistente. Il programma *mv* sposta un file da un posto ad un altro nell'albero del direttorio. Nella maggior parte dei casi, questo spostamento è fatto per ottenere un cambiamento di nome del file; se necessario, tuttavia, il file è copiato nella nuova posizione e la vecchia copia viene cancellata. Un file è cancellato con il programma *rm* (che fa una chiamata di sistema **unlink**).

Per visualizzare un file sul terminale, un utente può lanciare *cat*. Il programma *cat* prende una lista di file e li concatena, copiando il risultato sull'output standard, che di solito è il terminale. Naturalmente su un display ad alta velocità, come un monitor con il tubo a raggi catodici (CRT), il file può scorrere troppo velocemente per essere letto. Il programma *more* visualizza una schermata alla volta del file, facendo una pausa fino a che l'utente non preme un tasto per proseguire con lo schermo successivo. Il programma *head* mostra solo le prime righe di un file, mentre *tail* mostra le ultime righe.
Questi sono i programmi di sistema di base ampiamente usati in UNIX. Inoltre, sono disponibili un certo numero di editor (*ed*, *sed*, *emacs*, *vi* e così via), di compilatori (C, Pascal, Fortran, ecc) e di formattatori di testo (troff, TEX, scribe e così via). Ci sono anche programmi di ordinamento (*sort*) e di confronto dei file (*cmp, diff*), programmi per ricercare pattern (modelli) di testo (*grep, awk*), programmi per spedire posta elettronica (mail) agli altri utenti e molte altre attività.

4.1 Shell e comandi

Sia i programmi scritti dall'utente che i programmi di sistema sono normalmente eseguiti da un interprete dei comandi. In UNIX, l'interprete dei comandi è un processo utente come qualsiasi altro. È chiamato *shell*, poiché circonda il kernel del sistema operativo. Gli utenti possono scrivere la propria shell e, infatti, parecchie shell sono di uso comune. La *Bourne shell*, scritta da Steve Bourne, è probabilmente la più usata, o almeno, la più ampiamente disponibile. La *C shell*, lavoro principalmente dovuto a Bill Joy, uno dei fondatori di Sun Microsystems, è la più popolare nei sistemi BSD. La Korn shell, di Dave Korn, è diventata popolare poiché unisce le caratteristiche della Bourne shell e della C shell.

Le shell comuni condividono molta della sintassi del proprio linguaggio di comando. UNIX è normalmente un sistema interattivo. Le shell indicano che sono pronte ad accettare un altro comando mostrando un cursore (prompt) e l'utente può scrivere un comando su una singola riga. Per esempio, nella riga

% *ls -l*

il segno percento è il normale cursore della C shell e *ls -l* (scritto dall'utente) è il comando per avere il listato (lungo) del direttorio. I comandi possono prendere argomenti, che l'utente scrive dopo il nome del comando sulla stessa riga, separata da uno spazio vuoto (spazi o tab).

Anche se alcuni comandi sono incorporati nelle shell (come *cd*), un comando tipico è un file oggetto binario eseguibile. Un elenco di molti direttori, il *path* (percorso) *di ricerca*, è memorizzato dalla shell. Per ogni comando, ciascuno dei direttori nel path di ricerca viene scandagliato, nell'ordine, alla ricerca di un file con lo stesso nome. Se il file è trovato, viene caricato ed eseguito. Il path di ricerca può essere configurato dall'utente. I direttori */bin* e */usr/bin* sono quasi sempre nel path di ricerca e un path di ricerca tipico in un sistema FreeBSD potrebbe essere

(. */usr/avi/bin /usr/local/bin /bin /usr/bin*)

Il file oggetto del comando *ls* è */bin/ls* e la shell stessa è */bin/sh* (Bourne shell) o */bin/csh* (C shell).

L'esecuzione di un comando è fatta per mezzo di una chiamata di sistema **fork** seguita da un **execve** del file oggetto. La shell esegue solitamente una **wait** per sospendere la propria esecuzione fino a che il comando non termina (Figura 4). È prevista una semplice sintassi (il carattere "ampersand" [&] alla fine della riga di comando) per indicare che la shell non deve aspettare il completamento del comando. Un comando lasciato funzionare in questo modo mentre la shell continua ad interpretare ulteriori comandi è detto essere un comando in *background*, o che gira in background. I processi per cui la shell aspetta si dice che girano in *foreground*.

La C shell, nei sistemi FreeBSD, fornisce una funzione denominata *controllo del job* (job control) (parzialmente inserita nel kernel), come accennato in precedenza. Il controllo del job permette ai processi di essere portati dalla modalità in background a quella in foreground e viceversa. I processi possono essere interrotti e fatti ripartire in varie condizioni, quale un job in background che desidera l'input dal terminale dell'utente. Questo schema permette la maggior parte del controllo dei processi fornito dai sistemi grafici a finestre o da interfacce stratificate, ma non richiede hardware speciale. Il controllo del job è pure utile nei sistemi a finestre, come il sistema X Window sviluppato presso l'MIT. Ogni finestra è trattata come un terminale, permettendo che processi multipli siano in foreground (uno per finestra) in un qualsiasi momento. Naturalmente, i processi in background possono esistere in qualsiasi finestra. Anche la shell Korn supporta il controllo del job e tale controllo (e di gruppi di processi) sarà probabilmente standard nelle future versioni di UNIX.

4.2 Standard I/O

I processi possono aprire file come desiderano, ma la maggior parte dei processi si aspetta che tre descrittori di file (numeri 0, 1 e 2) vengano aperti quando partono. Questi descrittori di file sono ereditati da una **fork** (e possibilmente da **execve**) che ha generato il processo. Sono conosciuti come *standard input* (0), *standard output* (1) ed *standard error* (2). Tutti e tre sono spesso aperti verso il terminale utente. Così, il programma può leggere ciò che l'utente scrive, leggendo l'input standard ed il programma può mandare un output sullo schermo dell'utente scrivendo sull'output standard. Anche il descrittore di file standard error è pure aperto per la scrittura ed è usato per visualizzare errori; lo standard output è usato per l'output ordinario. La maggior parte dei programmi può anche accettare un file (invece che un terminale) come standard input e standard output. Il programma non si preoccupa da dove proviene l'input e dove sta andando l'output. Questa è una delle caratteristiche eleganti di UNIX.

Le comuni shell hanno una sintassi semplice per cambiare i file aperti per i flussi standard di I/O di un processo. Cambiare un file standard è detto *redirezione di I/O*. La sintassi per la redirezione di I/O è

indicata nella Figura 5. In questo esempio, il comando *ls* produce un elenco di nomi di file nel direttorio corrente, il comando *pr* formatta quel listato in pagine adatte alla stampante e il comando *lpr* manda l'output formattato allo spool della stampante, come */dev/lp0*. Il comando successivo forza la redirezione di tutto l'output e di tutti i messaggi di errore su un file. Senza il segno &, i messaggi di errore compaiono sul terminale.

Comando	significato del comando
% *ls* > *filea*	dirige l'output di *ls* al file *filea*
% *pr* < *filea* > *fileb*	input dal *filea* e output al *fileb*
% *lpr* < *fileb*	input dal *fileb*
% % make program > & errs	salva sia lo standard output che lo standard error in un file

Figura 5. Redirezione standard I/O.

4.3 Pipeline, filtri e script di shell

I primi tre comandi della Figura 5 potrebbero essere compattati in un comando unico:

% ls | pr | lpr

Ogni barra verticale dice alla shell di fare in modo che l'uscita del comando precedente sia passato come input al comando successivo. Una pipe è utilizzata per trasportare i dati da un processo all'altro. Un processo scrive ad un'estremità della pipe e un altro processo legge dall'altra estremità. Nell'esempio, l'estremità di scrittura di una pipe sarà posizionata dalla shell in modo da essere l'output standard di *ls* e l'estremità in lettura della pipe sarà l'input standard di *pr*; un'altra pipe si troverà fra *pr* e *lpr*.

Un comando come *pr* che passa il proprio suo input standard all'output, effettuando alcune elaborazioni su di esso, è chiamato *filtro*. Molti comandi UNIX possono essere usati come filtri. Funzioni complicate possono essere assemblate come pipeline di comandi comuni. Inoltre, funzioni comuni, come la formattazione di output, non devono essere composte da numerosi comandi, perché l'output di qualsiasi programma può essere messo in pipe a *pr* (o a qualche altro filtro opportuno). Entrambe le shell più comuni di UNIX sono anche linguaggi di programmazione, con variabili della shell e costrutti comuni di controllo propri dei linguaggi di programmazione di livello più elevato (cicli, ed espressioni condizionali). L'esecuzione di un comando è analoga ad una chiamata di una procedura. Un file di comandi della shell, uno *script della shell*, può essere eseguito come qualsiasi altro comando, invocando automaticamente l'opportuna shell per leggerlo. La *programmazione della shell* può così essere usata per combinare opportunamente programmi comuni per applicazioni sofisticate senza la necessità di programmazione tramite linguaggi convenzionali.

Questa interfaccia di presentazione all'utente è comunemente associata alla definizione di UNIX, inoltre è la definizione più facilmente cambiata. La scrittura di nuove shell con una sintassi e una semantica abbastanza differenti potrebbe cambiare notevolmente il punto di vista dell'utente, senza cambiare il kernel e pure l'interfaccia di programmazione. Attualmente esistono per UNIX parecchie interfacce organizzate a menu ed icone; il sistema X Window sta divenendo velocemente uno standard. Il cuore di UNIX è, naturalmente, il kernel che è molto più difficile da cambiare dell'interfaccia utente, poiché tutti i programmi dipendono dalle chiamate di sistema che fornisce per rimanere consistente. Naturalmente, nuove chiamate di sistema si possono aggiungere per incrementare le funzionalità, ma i programmi devono essere modificati per poter usare le nuove chiamate.

5 Amministrazione dei processi

Il problema principale riguardo il progetto di sistemi operativi è la rappresentazione dei processi. Una differenza sostanziale fra UNIX e molti altri sistemi è la facilità con cui processi multipli possono essere creati e manipolati. In UNIX, questi processi sono rappresentati da vari blocchi di controllo. Non ci sono blocchi di controllo del sistema accessibili nello spazio di indirizzamento virtuale di un processo utente; i blocchi di controllo associati ad un processo sono memorizzati nel kernel. Il kernel usa le informazioni in questi blocchi per controllare i processi e la schedulazione della CPU.

5.1 I blocchi di controllo del processo

La struttura dati più, profondamente connessa con i processi, è la *struttura di processo*. Una struttura di processo contiene tutto ciò che il sistema deve conoscere circa un processo quando il processo viene cambiato, come ad esempio il suo unico identificatore di processo, le informazioni di schedulazione (quali la priorità del processo) e i puntatori ad altri blocchi di controllo. C'è un array delle strutture del processo la cui lunghezza viene definita al momento del link del sistema. Le strutture di processo relative ai processi pronti sono mantenute collegate insieme dallo schedulatore in una lista a doppio collegamento (la coda dei processi pronti) e ci sono puntatori da ogni struttura di processo a quella del processo padre, a quella dei figli più giovani che sono ancora in vita e a vari altri parenti che possono interessare, come una lista di processi che condividono lo stesso codice di programma (testo).

Lo *spazio d'indirizzamento virtuale* di un processo utente è diviso in testo (codice del programma), dati e segmenti dello stack. I dati ed i segmenti dello stack sono sempre nello stesso spazio di indirizzamento, ma possono crescere separatamente e solitamente in direzioni opposte: più frequentemente, lo stack decresce mentre si sviluppa la parte dati. Il segmento di testo è talvolta (come nell'Intel 8086 con lo spazio separato per i dati e le istruzioni) in uno spazio d'indirizzamento differente per i dati e lo stack ed è solitamente a sola lettura. Il debugger mette un segmento di testo in modalità lettura-scrittura per potere permettere l'inserimento di breakpoint (punti di arresto dell'esecuzione).
Ogni processo con testo in condivisione (quasi tutti, in FreeBSD) ha un puntatore dalla propria struttura di processo ad una *struttura di testo*.

La struttura di testo registra come molti processi usano il segmento di testo, compreso un puntatore ad una lista delle loro strutture di processo, inoltre memorizza dove si trova su disco la tabella di pagina per il segmento di testo quando è sottoposto a swapping. La struttura di testo stessa è sempre residente nella memoria centrale: un array di tali strutture è allocata nella fase di link del sistema. Il testo, i dati ed i segmenti di stack per i processi possono essere sottoposti a swapping. Quando i segmenti sono soggetti a swapping, vengono paginati.

Le *tabelle di pagina* registrano informazioni sulla mappatura dalla memoria virtuale dei processi alla memoria fisica. La struttura del processo contiene i puntatori alla tabella di pagina, che viene usata quando il processo è residente nella memoria centrale, o per memorizzare l'indirizzo del processo nel dispositivo di swap, quando il processo viene sottoposto a swapping. Non vi è una speciale tabella di pagina separata per il segmento di testo condiviso; ogni processo che condivide il segmento di testo ha degli indici per le proprie pagine nella tabella di pagina dei processi.

Le informazioni sul processo che devono essere disponibili solo quando il processo è residente (cioè non è stato scaricato nell'area di swap) sono mantenute in una *struttura utente* (o *struttura u*), piuttosto che nella struttura del processo. La struttura *u* è mappata a sola lettura nello spazio di indirizzamento virtuale dell' utente, in modo che i processi utente possano leggerne il contenuto, ed è scrivibile dal kernel. La struttura *u* contiene una copia del blocco di controllo del processo o PCB che viene qui mantenuta per salvare i registri generali dei processi, il puntatore dello stack, il contatore di programma ed i registri base della tabella di pagina quando il processo non è in funzione. C'è uno spazio per tenere i parametri delle chiamate di sistema ed i valori di ritorno. Qui sono memorizzati tutti gli identificativi degli utenti e di gruppo associati al processo (non solo l'identificativo utente effettivo mantenuto nella struttura del processo). Qui i segnali, i timer e le quote hanno le loro strutture dati. Di importanza più evidente per l'utente ordinario, è il direttorio corrente e la tabella del file aperti che sono mantenuti nella struttura utente.

Ogni processo ha sia la modalità utente che quella di sistema. Il lavoro più ordinario avviene in *modalità utente*, ma quando si esegue una chiamata di sistema, essa avviene in *modalità di sistema*. Le fasi di sistema e utente di un processo non si eseguono mai simultaneamente. Quando un processo è in esecuzione in modalità di sistema, viene usato uno stack del kernel per quel processo, invece dello stack utente di quel processo. Lo *stack del kernel* del processo segue immediatamente la struttura utente: lo stack del kernel e la struttura utente costituiscono insieme il *segmento dati di sistema* del processo. Il kernel ha il proprio stack che viene usato quando non sta lavorando per un processo (per esempio, per la gestione di un interrupt).

La Figura 6 illustra come la struttura di processo venga usata per trovare le varie parti di un processo. La chiamata di sistema **fork** alloca una nuova struttura di processo (con un nuovo identificatore del processo) per il processo figlio e copia la struttura utente. Non c'è di solito l'esigenza di una nuova struttura di testo, poiché i processi condividono il proprio testo; i contatori appropriati e le liste sono solo aggiornati. Viene creata una nuova tabella di pagina e viene allocata nuova memoria centrale per i dati ed i segmenti dello stack del processo figlio. La copia della struttura utente preserva i descrittori dei file aperti, gli identificatori dell'utente e del gruppo, la gestione dei segnali e la maggior parte delle proprietà simili di un processo.

La chiamata di sistema **vfork** *non* copia i dati e lo stack nel nuovo processo; invero, il nuovo processo condivide semplicemente la tabella di pagina di quello vecchio. Viene creata una nuova struttura utente e una nuova struttura del processo. Un uso comune di questa chiamata di sistema avviene tramite la shell per eseguire un comando e per aspettare il relativo completamento. Il processo padre usa **vfork** per produrre il processo figlio. Poiché il processo figlio desidera usare immediatamente **execve** per cambiare completamente il proprio spazio di indirizzamento virtuale, non c'è alcun bisogno di una copia completa del processo padre. Tali strutture dati, che sono necessarie per gestire oggetti come pipe, possono essere mantenute in registri tra **vfork** e **execve**. I file possono essere chiusi in un processo senza interessare l'altro processo, poiché le strutture dati del kernel coinvolte dipendono dalla struttura utente, che non è condivisa. Il padre viene sospeso quando chiama **vfork** finché il figlio chiama **execve** o termina, in modo che il padre non cambi la memoria di cui il figlio ha bisogno.

Quando il processo padre è vasto, **vfork** può produrre un notevole risparmio nel tempo di uso della CPU di sistema. Tuttavia, è una chiamata di sistema abbastanza pericolosa, poiché un qualsiasi

cambiamento di memoria si presenta in entrambi i processi fino a che non viene eseguita una **execve**. Un'alternativa è di condividere tutte le pagine duplicando la tabella di pagina, solo per contrassegnare le voci di entrambe le tabelle di pagina come *copy-on-write*. I bit di protezione hardware sono configurati per intrappolare qualsiasi tentativo di scrivere in queste pagine condivise. Se avviene una trap, viene allocato un nuovo frame e la pagina condivisa viene copiata nel nuovo frame. Le tabelle di pagina vengono aggiustate per indicare che questa pagina non è più condivisa (e quindi non necessità più di essere protetta in scrittura) e l'esecuzione può riprendere.

Una chiamata di sistema **execve** non crea alcun nuovo processo o struttura utente; piuttosto sono rimpiazzati il testo ed i dati del processo. I file aperti sono preservati (anche se c'è un modo per specificare che certi descrittori di file devono essere chiusi su una **execve**). La maggior parte delle funzioni di gestione dei segnali sono mantenute, ma per ovvi motivi sono annullate le disposizioni per chiamare una specifica procedura utente su un segnale. Resta immutato l'identificatore del processo e la maggior parte delle altre proprietà del processo.

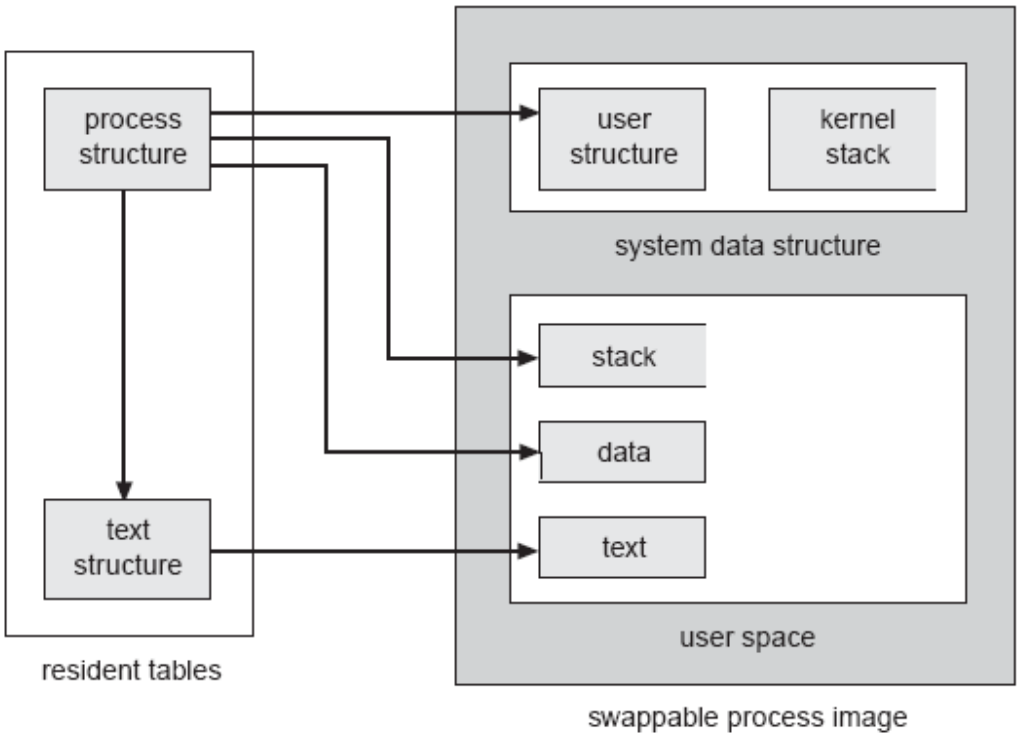

Figura 6. Modo in cui vengono trovate le parti di un processo usando la struttura del processo.

process structure = struttura di processo; user structure = struttura utente
kernel stack = stack del kernel
system data structure = struttura dati di sistema
stack; data = dati; text = testo
text structure = struttura di testo
user space = spazio utente; resident tables = tabelle residenti
swappable process image = immagine di processo swappabile

5.2 Schedulazione della CPU

In UNIX la *schedulazione della CPU* è progettata per avvantaggiare i processi interattivi. Ai processi sono assegnate piccole fette di tempo della CPU da un algoritmo di priorità che si riduce ad una schedulazione del tipo round-robin per i job legati alla CPU.

Ogni processo ha una *priorità di schedulazione* associata con esso; numeri maggiori indicano una priorità inferiore. I processi che fanno I/O su disco o altri compiti importanti hanno priorità inferiore a "pzero" e non possono essere uccisi da segnali. Gli ordinari processi utente hanno priorità positive e così hanno una probabilità inferiore di venir fatti funzionare rispetto a qualsiasi processo di sistema, anche se i processi utente possono regolare le rispettive precedenze con il comando *nice*.

Più tempo di CPU un processo accumula, più bassa (più positiva) diventa la sua priorità e viceversa. Questa retroazione negativa nella schedulazione della CPU rende difficile ad un singolo processo di prendersi tutto il tempo della CPU. L'invecchiamento di un processo serve per impedire la starvation.

I più vecchi sistemi UNIX usavano un quanto di 1 secondo per la schedulazione round-robin. FreeBSD rischedula i processi ogni 0.1 secondi e ricalcola le priorità ogni secondo. La schedulazione round-robin è realizzata dal meccanismo *timeout*, che dice al driver di interrupt del clock di chiamare una sottoprocedura del kernel dopo un intervallo di tempo specificato; la procedura che in questo caso viene chiamata, causa la rischedulazione e quindi setta un *timeout* per essere richiamata. Il ricalcolo della priorità è schedulato da una procedura che anch'essa setta un *timeout* per la nuova esecuzione.

Nel kernel non c'è la prelazione di un processo su un altro. Un processo può cedere la CPU perché sta attendendo un'operazione di I/O o perché la sua fetta di tempo è scaduta. Quando un processo sceglie di cedere la CPU, va in *sleep* in presenza di un evento. La primitiva del kernel usata per questo scopo è chiamata *sleep* (da non confondersi con la procedura di libreria a livello utente con lo stesso nome). Prende una argomento, che, per convenzione, è l'indirizzo di una struttura dati del kernel collegata ad un *evento* che il processo desidera che accada prima che il processo venga risvegliato. Quando accade l'evento, il processo di sistema, che conosce tutto questo, chiama *wakeup* con l'indirizzo corrispondente all'evento e *tutti* i processi che avevano eseguito una *sleep* allo stesso indirizzo sono messi nella coda di attesa pronta per l'esecuzione.

Per esempio, un processo che attende il completamento di un'operazione di I/O da disco eseguirà una *sleep* all'indirizzo dell'intestazione del buffer che corrisponde ai dati in trasferimento. Quando la routine di interrupt per il driver del disco nota che il trasferimento è completo, chiama *wakeup* all'intestazione del buffer. L'interrupt usa lo stack del kernel per qualsiasi processo stia girando in quel momento e *wakeup* viene eseguito da quel processo di sistema.

Il processo che funziona realmente viene scelto dallo schedulatore. *Sleep* prende un secondo argomento, che è la priorità di schedulazione da usare per questo scopo. Questo argomento di priorità, se vale meno di "pzero", impedisce che il processo sia risvegliato prematuramente da un certo evento eccezionale, quale un *segnale*.

Quando viene generato un segnale, è lasciato in sospeso fino alla prossima esecuzione della parte di sistema del processo coinvolto. Questo evento solitamente accade subito, poiché normalmente il segnale causa il risveglio del processo, se stava aspettando che accadesse una qualche altra condizione.

Nessuna parte della memoria è associata agli eventi. Il chiamante della procedura che esegue *sleep* su un evento deve essere preparato per gestire un ritorno prematuro, compresa la possibilità che il motivo dell'attesa sia venuto meno.

Le *corse critiche* (race conditions) sono coinvolte nel meccanismo dell'evento. Se un processo decide (per esempio a causa del controllo di un flag in memoria) di andare in sleep per un evento e l'evento accade prima che il processo possa eseguire la primitiva che fa effettivamente la *sleep* sull'evento, il processo in *sleep* può rimanere in tale stato per sempre. Questa situazione viene impedita alzando la priorità hardware del processore durante la sezione critica, in modo che non possa accadere nessun interrupt e solo il processo che attende l'evento possa girare fino a che non vada in sleep. La priorità hardware del processore è usata in questo modo per proteggere le regioni critiche durante il lavoro del kernel ed è l'ostacolo maggiore riguardante la portabilità di UNIX su macchine multiprocessore. Tuttavia, questo problema non ha impedito che tali operazioni siano state fatte più volte.

Molti processi quali gli editor di testo sono legati all'I/O e solitamente saranno schedulati principalmente in base alle attese di I/O. L'esperienza suggerisce che lo schedulatore di UNIX lavora meglio con job legati a I/O, come si può osservare quando stanno girando parecchi job legati alla CPU, quali i formattatori di testo o interpreti di linguaggio.
Il termine *schedulazione della CPU* corrisponde molto bene a quanto illustrato nel Capitolo 4 riguardo alla *schedulazione a breve termine*. Tuttavia, la proprietà di retroazione negativa dello schema di priorità fornisce una certa schedulazione a lungo termine in quanto in gran parte determina il *miscuglio di job* a lungo termine. La schedulazione a medio termine è realizzata dal meccanismo di swap, descritto nel Capitolo 6.

6. Gestione della memoria

Molto dello sviluppo iniziale di UNIX è stato fatto su un PDP-11. Il PDP-11 ha soltanto otto segmenti nel proprio spazio di indirizzamento virtuale e ognuno di questi è al massimo di 8192 byte. Le macchine più grandi, tipo il PDP-11/70, permettono spazi separati per indirizzi e istruzioni, che effettivamente raddoppiano lo spazio di indirizzamento e il numero di segmenti, ma questo spazio di indirizzamento è ancora relativamente piccolo. Inoltre, il kernel era ancor più severamente sacrificato a causa di un solo segmento dati dedicato ai vettori di interrupt, un altro per puntare al segmento di dati del sistema per il processo, ed un altro ancora per i registri UNIBUS (I/O bus di sistema). Inoltre, sul più piccolo dei PDP-11, il totale della memoria fisica totale era limitata a 256 KB. Le risorse totali di memoria erano insufficienti per giustificare o supportare complessi algoritmi di amministrazione della memoria. Pertanto, UNIX eseguiva lo swap delle immagini della memoria dell'intero processo.

6.1 Paginazione

Berkeley ha introdotto la paginazione su UNIX con 3BSD. VAX 4.2BSD è un sistema con paginazione su richiesta della memoria virtuale. La paginazione elimina la frammentazione esterna della memoria (la frammentazione interna avviene ancora, ma è trascurabile con una dimensione ragionevolmente piccola di pagina). Poiché la paginazione permette l'esecuzione con solo parti di ogni processo in memoria, più job possono essere mantenuti nella memoria centrale e l'attività di swapping può essere ridotta al minimo. La *paginazione su domanda* è fatta in modo diretto. Quando un processo ha bisogno di una pagina e la pagina non è presente, nel kernel accade un errore di pagina, viene allocato un frame della memoria centrale e la pagina voluta è letta dal disco nel frame. Ci sono poche ottimizzazioni. Se la pagina necessaria è ancora nella tabella di pagina per il processo, ma è stata contrassegnata come non valida dal processo di rimpiazzo della pagina, può essere contrassegnata

come valida ed usata senza alcun trasferimento di I/O. In modo simile le pagine possono essere richiamate dalla lista dei frame liberi. Quando la maggior parte dei processi viene avviata, molte delle loro pagine sono prepaginate e messe nella lista di quelle libere per poterle recuperare con questo meccanismo. Possono anche essere fatte configurazioni che consentano ad un processo di non essere sottoposto a prepaginazione all'avvio, ma ciò avviene raramente, poiché provoca un overhead in termini di errori di pagina, essendo più vicino alla pura paginazione su domanda. FreeBSD realizza la colorazione di pagina con le code di paginazione. Le code sono organizzate secondo la dimensione delle cache L1 e L2 del processore e quando una nuova pagina deve essere allocata, FreeBSD prova ad ottenerne una che può essere ottimamente adattata alla cache.

Se la pagina deve essere caricata da disco, deve essere bloccata in memoria per la durata del trasferimento; questa operazione garantisce che la pagina non sarà selezionata per il rimpiazzo di pagina. Una volta che la pagina è caricata e mappata correttamente, deve rimanere bloccata se si sta facendo su essa un I/O fisico di basso livello.

L'algoritmo di *sostituzione della pagina* è più interessante. 4.2BSD usa una modifica dell'algoritmo di *second chance* (clock) descritto nel Paragrafo 10.4.5. La mappa di tutta la memoria centrale, non appartenente al kernel (il *core map* o *cmap*), è scandita linearmente e ripetutamente da una *lancetta dell'orologio* software. Quando il ciclo dell'orologio raggiunge un dato frame, se il frame è marcato come in uso da qualche condizione software (per esempio, è in progresso un I/O fisico che lo usa), o il frame è già libero, esso viene lasciato intatto e la lancetta dell'orologio passa al frame successivo. Altrimenti viene individuato il testo corrispondente o la voce relativa alla tabella di pagina del processo per questo frame. Se la voce è già non valida, il frame viene aggiunto alla lista di quelli liberi; altrimenti, la voce relativa alla tabella di pagina è resa non valida ma reclamabile (cioè, se non viene tolta dalla memoria e rimessa su disco prima della successiva richiesta, può essere resa nuovamente valida).

BSD Tahoe ha aggiunto il supporto per i sistemi che utilizzano il bit di riferimento. In tali sistemi, un passaggio della lancetta dell'orologio pone il bit di riferimento su off, e un secondo passaggio mette quelle pagine, in cui bit di riferimento resta su off, nella lista libera per il rimpiazzo. Naturalmente, se la pagina è sporca, la si deve in primo luogo scrivere su disco prima di aggiungerla alla lista delle pagine libere. Lo scarico delle pagine è fatto a gruppi per migliorare le prestazioni.

Ci sono controlli per assicurarsi che il numero di pagine di dati valide per un processo non diventi troppo basso, e per far sì che il dispositivo di paginazione non sia inondato da richieste. Esiste anche un meccanismo per cui un processo può limitare la quantità che usa di memoria centrale.

Lo schema della lancetta dell'orologio LRU è realizzato nel *pagedaemon*, che è il processo 2. (si ricordi che *swapper* è il processo 0 e *init* è il processo 1). Questo processo utilizza la maggior parte del suo tempo in uno stato dormiente, ma viene fatto un controllo parecchie volte al secondo (schedulato da un *timeout*) per vedere se è necessario agire; in caso affermativo, il processo 2 viene svegliato. Ogni volta che il numero dei frame liberi cade sotto una soglia, *lotsfree*, il *pagedaemon* viene svegliato; quindi, se è sempre disponibile una grande quantità di memoria libera, il *pagedaemon* non impone alcun carico sul sistema, poiché non funziona mai.

Ogni volta che il processo *pagedaemon* viene svegliato (cioè il numero dei frame controllati, che è solitamente maggiore del numero di quelli cancellati dalla memoria), la *scansione* della lancetta dell'orologio è determinata sia dal numero di frame che mancano per raggiungere *lotsfree* che dal numero di frame che lo *scheduler* ha stabilito essere necessario per vari motivi (più frame sono necessari, più lunga è la scansione). Se il numero di frame liberi supera il *lotsfree* prima che la *scansione* prevista sia completata, il ciclo si arresta ed il processo *pagedaemon* va in sleep. I parametri che determinano l'intervallo della *scansione* delle lancette dell'orologio sono determinati alla partenza del sistema secondo la quantità di memoria centrale, tale che il *pagedaemon* non usi più del 10 per cento di tutto il tempo della CPU.

Se lo *schedulatore* decide che il sistema paginante è sovraccaricato, i processi saranno scaricati fuori interamente finché il sovraccarico non diminuisce. Solitamente questo tipo di swap accade soltanto se si verificano parecchie circostanze: il carico medio è alto, la memoria libera è caduta sotto la soglia, *minfree*; e la media di memoria disponibile in tempi recenti è meno di una certa quantità desiderabile, *desfree*, ove *lotsfree* > *desfree* > *minfree*. In altre parole soltanto una scarsità cronica di memoria con parecchi processi che cercano di funzionare causerà lo swap, e anche allora la memoria libera deve essere estremamente bassa. (Un tasso di paginazione eccessivo o un'esigenza di memoria dal kernel stesso può anche diventare determinante, in rare occasioni). I processi possono essere sottoposti a swap dallo *scheduler*, naturalmente, per altri motivi (tipo non essere semplicemente in funzione da lungo tempo).

Il parametro *lotsfree* è solitamente un quarto della memoria nella mappa che la lancetta dell'orologio esegue e *desfree* e *minfree* hanno solitamente gli stessi valori in sistemi differenti, ma sono limitati a frazioni della memoria disponibile. FreeBSD regola dinamicamente le proprie code di paginazione in modo che questi parametri della memoria virtuale verranno aggiustati raramente. Le pagine *minfree* devono essere mantenute libere per fornire qualsiasi pagine potrebbe venire richiesta al momento di un interrupt.

Ogni segmento di testo del processo è per default condiviso e in sola lettura. Questo schema è pratico con la paginazione, perché non c'è frammentazione esterna e lo spazio di swap guadagnato con la condivisione sorpassa la quantità trascurabile di sovraccarico comportato, poiché lo spazio virtuale del kernel è grande.

La schedulazione della CPU, lo swap di memoria e la paginazione interagiscono tra loro: più è bassa la priorità di un processo e più è probabile che le sue pagine saranno eliminate dalla memoria e con più probabilità saranno sottoposte a swap nella loro interezza, ma così la paginazione è più efficace. Idealmente, i processi non saranno sottoposti a swap a meno che non siano in uno stato idle, poiché ogni processo avrà bisogno in qualsiasi momento soltanto di un piccolo insieme di pagine in memoria centrale ed il *pagedaemon* reclamerà le pagine inutilizzate per l'uso da parte di altri processi.
La quantità di memoria di cui il processo avrà bisogno è una qualche frazione della memoria virtuale totale del processo, fino a metà se quel processo è stato messo in swap per un lungo periodo.

7 File system

Il file system di UNIX supporta due oggetti principali: i file e i direttori che non sono altro che file con un formato speciale, e quindi la rappresentazione di un file è il concetto di base di UNIX.

7.1 Blocchi e frammenti

La maggior parte del file system è fatto di *blocchi di dati*, che contiene qualunque cosa gli utenti hanno messo nei loro file. Vediamo come questi blocchi di dati sono memorizzati su disco.
Il settore fisico del disco è solitamente di 512 byte. Per aumentare la velocità è desiderabile che la dimensione del blocco sia maggiore di 512 byte; tuttavia, poiché i file system di UNIX contengono solitamente un gran numero di piccoli file, blocchi di dimensioni maggiori potrebbero causare una eccessiva frammentazione interna. Ecco perché i primi file system di 4.1BSD sono stati limitati alla dimensione di 1.024 byte (1 Kb). La soluzione di 4.2BSD è di usare *due* dimensioni di *blocco* per i file che non hanno blocchi indiretti: tutti i blocchi di un file sono di grandi dimensioni (da 8 Kb), tranne l'ultimo. L'ultimo *blocco* è un opportuno multiplo adatto alla più piccola dimensione del *frammento*

(per esempio, 1.024) per completare il file. Pertanto, un file con dimensione di 18.000 byte avrebbe due blocchi da 8 Kb ed un *frammento* di 2 Kb (che non verrà riempito completamente).

Le dimensioni del *blocco* e del *frammento* sono configurate durante la creazione del file-system in base all'uso del file system: se sono previsti molti file piccoli, la dimensione del *frammento* dovrebbe essere piccola; se sono previsti ripetuti trasferimenti di file grandi, la dimensione del *blocco* base dovrebbe essere grande. I dettagli realizzativi forzano un rapporto massimo di blocco - frammento di 8:1 e un *blocco* minimo di 4 Kb, in modo che la scelta tipica sia 4.096:512 nel primo caso, e 8.192:1.024 nell'ultimo caso.

Supponiamo che i dati siano scritti su un file con dimensioni di trasferimento da 1 Kbyte e che le dimensioni del *blocco* e del *frammento* del file system siano di 4 KB e di 512 byte. Il file system allocherà un frammento da 1 KB per contenere i dati del primo trasferimento.

Il trasferimento successivo farà sì che venga allocato un nuovo frammento da 2 KB. I dati del frammento originale devono essere copiati in questo nuovo frammento, seguito dal secondo trasferimento di 1 KB. Le procedure di allocazione tentano di trovare lo spazio richiesto su disco, subito dopo il frammento attuale in modo che non ci sia bisogno di alcuna copiatura; ma se ciò non è possibile, possono venire richieste fino a sette copie prima che il frammento diventi un blocco. Si fa in modo che i programmi scoprano la dimensione del blocco per un file in modo che i trasferimenti, di quella dimensione, possano avvenire evitando di ricopiare il frammento.

7.2 Inode

Un file è rappresentato da un inode (Figura 11.7). Un inode è un record che memorizza la maggior parte delle informazioni riguardanti un file specifico su disco. Il nome inode (pronunciato EYE node) è derivato da "index-node" ed era originalmente scandito con "i-node"; il trattino è andato in disuso nel corso degli anni. Il termine è a volte indicato con "I node".

Inode contiene gli identificativi del gruppo e dell'utente del file, l'ora dell'ultima modifica e accesso al file, un contatore del numero di hard link (ingressi al direttorio) al file ed il tipo di file (file normale, direttorio, link simbolico, dispositivo a caratteri, dispositivo di blocco, o socket). Inoltre, l'inode contiene 15 puntatori ai blocchi del disco che contengono i dati del file; i primi 12 puntatori puntano a *blocchi diretti:* contengono cioè gli indirizzi di blocchi che contengono i dati del file. Pertanto, i dati per file piccoli (non più di 12 blocchi) possono essere referenziati immediatamente, perché una copia dell'inode è mantenuta nella memoria centrale mentre il file è aperto. Se la dimensione del blocco è di 4 KB, allora fino a 48 KB di dati possono essere raggiunti direttamente dall'inode.

I tre puntatori successivi dell'inode puntano a *blocchi indiretti*. Se il file è abbastanza grande per usare i blocchi indiretti, i blocchi indiretti hanno, ognuno di essi, la dimensione del blocco maggiore e la dimensione del frammento si applica solo ai blocchi di dati. Il primo puntatore indiretto del blocco è l'indirizzo del *singolo blocco indiretto*. Il singolo blocco indiretto è un blocco di indici, contenente non i dati, ma gli indirizzi dei blocchi che contengono i dati. Quindi, c'è un *puntatore di doppio blocco indiretto*, l'indirizzo di un blocco che contiene gli indirizzi dei blocchi che contengono i puntatori ai dati attuali. L'ultimo puntatore conterrà l'indirizzo di un *triplo blocco indiretto*; tuttavia, non c'è alcun bisogno di esso.

La dimensione di blocco minima per un file system in 4.2BSD è di 4 KB, così da archiviare file fino a 2^{32} byte usando soltanto la doppia indirezione, non la triplice. Cioè, poiché ogni puntatore del blocco occupa 4 byte, abbiamo 49.152 byte accessibili nei blocchi diretti, 4.194.304 byte accessibili tramite una singola indirezione e 4.294.967.296 byte raggiungibili con doppia indirezione, per un totale

di 4.299.210.752 byte, che è maggiore di 2^{32} byte. Il numero 2^{32} è significativo perché lo spiazzamento del file nella struttura del file nella memoria centrale è mantenuta in word da 32-bit; quindi i file non possono essere maggiori di 2^{32} byte. Poiché i puntatori del file sono numeri interi dotati di segno (per poter cercare all'indietro e in avanti in un file), l'attuale dimensione massima del file è 2^{32-1} byte. Due gigabyte è una dimensione abbastanza grande per la maggior parte degli scopi.

7.3 Direttori

A questo livello, i normali file non sono distinti dai direttori; i contenuti di un direttorio sono mantenuti nei blocchi dati e i direttori sono rappresentati da un inode come se fossero file normali. Solo il tipo di campo inode fa distinzione fra file normali e direttori. Per i file normali non si presume che abbiano una struttura, mentre i direttori hanno una struttura specifica. Nella versione 7, i nomi dei file erano limitati a 14 caratteri, in modo che i direttori erano una lista di voci da 16-byte: 2 byte per un numero di inode e 14 byte per il nome di file.

In FreeBSD i nomi dei file sono di lunghezza variabile, fino a 255 byte, così che le voci dei direttori sono di lunghezza variabile. Ogni voce contiene in primo luogo la lunghezza dell'elemento, poi il nome del file ed il numero di inode. Questo elemento di lunghezza variabile rende la gestione del direttorio e le procedure di ricerca più complesse, ma permette agli utenti di scegliere nomi molto più espressivi per i loro file e direttori. I primi due nomi in ogni direttorio sono "." e "..". Le nuove voci del direttorio vengono aggiunte al direttorio nel primo spazio disponibile, generalmente dopo i file esistenti. Si usa una ricerca lineare.

L'utente si riferisce ad un file mediante un path name, mentre il file system usa l'inode come propria definizione di un file. Pertanto, il kernel deve mappare il path name fornito dall'utente in un inode. Per questa operazione si usano i direttori.

In primo luogo, è individuato un direttorio di partenza. Se il primo carattere del path ame è "/", il direttorio di partenza è quello di root. Se il path name comincia con qualunque carattere tranne lo slash, il direttorio di partenza è quello corrente del processo attivo. Il direttorio di partenza viene controllato per stabilire se il tipo è adeguato e se ci sono i permessi di accesso al file e, se necessario, viene restituito un errore. L'inode del direttorio di partenza è sempre disponibile.

L'elemento successivo del path name, fino al prossimo "/", o alla fine del path name, è un nome di file. Si cerca questo nome nel direttorio di partenza e, se non si trova il nome, vene restituito un errore. Se il path name ha ancora un altro elemento, l'inode corrente deve riferirsi ad un direttorio; se non è così, o se l'accesso è negato, viene restituito un errore. In questo direttorio si cerca come nel precedente. Questo processo continua fino a raggiungere la fine del path name e viene restituito l'inode voluto. Questo processo graduale è necessario perché in ogni direttorio si può incontrare un punto di montaggio (o link simbolico, si veda sotto), costringendo a muoversi ad una nuova struttura di direttori per continuare.

Gli hard link sono semplicemente elementi di direttorio come qualsiasi altro. Maneggiamo i link simbolici per la maggior parte iniziando la ricerca sul path name preso dal contenuto del link simbolico. Impediamo cicli infiniti contando il numero di link simbolici incontrati durante la ricerca del path name e restituendo un errore quando viene oltrepassata la oglia (otto).

I file che non sono del disco (quali i dispositivi) non hanno blocchi dati allocati su disco. Il kernel si accorge di questi tipi di file (come indicato nell'inode) e richiama driver adatti per gestire gli I/O.

Una volta che viene trovato l'inode, per esempio la chiamata di sistema **open**, viene allocata una *struttura di file* per puntare all'inode. Il descrittore del file assegnato all'utente si riferisce a questa

struttura di file. FreeBSD ha un *cache del nome del direttorio* per memorizzare le traduzioni recenti del direttorio in inode, ciò aumenta notevolmente le prestazioni del file system.

7.4 Mappaggio di un Descrittore di File in un Inode

Le chiamate di sistema che si riferiscono a file aperti indicano il file passando come argomento un descrittore di file. Il descrittore di file è usato dal kernel per indicizzare una tabella di file aperti per il processo corrente. Ogni elemento della tabella contiene un puntatore ad una struttura del file che a sua volta punta all'inode; si veda la Figura 7. La tabella dei file aperti ha una lunghezza fissa che è configurabile durante l'avvio del sistema. Di conseguenza, c'è un limite fisso sul numero di file simultaneamente aperti in un sistema.

Le chiamate di sistema **read** e **write** non prendono come argomento una posizione nel file. Piuttosto, il kernel mantiene un indice, che è aggiornato dopo una lettura o scrittura di una quantità opportuna in base al numero di dati realmente trasferiti. Lo spiazzamento può essere cambiato direttamente con la chiamata di sistema **lseek**. Se il descrittore di file indicizzava un array di puntatori di inode invece dei puntatori del file, tale spiazzamento dovrà essere mantenuto nell'inode. Poiché più di un processo può aprire lo stesso file ed ognuno dei processi ha bisogno del proprio spiazzamento per il file, è inadeguato mantenere lo spiazzamento nell'inode. Quindi, la struttura del file è usata per contenere lo spiazzamento.

Le strutture dei file sono ereditate dal processo figlio dopo una **fork**, in modo che parecchi processi possano condividere lo *stesso* spiazzamento di posizione per un file.

La *struttura di inode* cui punta la struttura del file è una copia interna dell'inode su disco. L'inode interno ha pochi campi supplementari, come un contatore di riferimento di quante strutture di file stanno puntando ad esso e la struttura del file ha un contatore simile di riferimento in base a quanti descrittori di file si riferiscono ad esso. Quando un contatore arriva a zero, la voce non è più necessaria e può essere reclamata e riutilizzata.

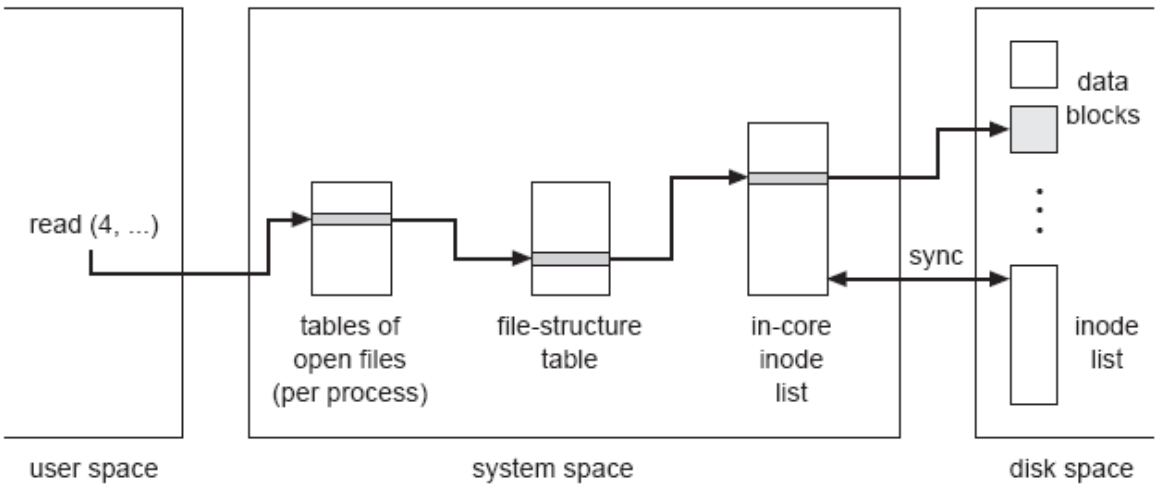

Figura 7. Blocchi di controllo del file system.

read = leggi(4...); tables of open files (per process) = tabella dei file aperti (per processo)

file–structure table = tabella delle strutture di file; list inode in–core = lista di inode interna;
sync
data blocks = blocchi dati; inode list = lista di inode; user space = spazio utente
system space = spazio di sistema; disk space = spazio di disco

7.5 Strutture dei dischi

Il file system che l'utente vede è supportato dai dati su un dispositivo di memoria di massa, solitamente un disco. L'utente di solito conosce solo un file system, ma questo file system logico può realmente consistere di parecchi file system *fisici*, ciascuno su di un dispositivo differente. Poiché le caratteristiche del dispositivo differiscono, ogni dispositivo hardware separato definisce il proprio file system fisico. Infatti, generalmente desideriamo partizionare grandi dispositivi fisici, quali i dischi, in dispositivi *logici* multipli. Ogni dispositivo logico definisce un file system fisico. La Figura 8 illustra come una struttura di direttorio è partizionata in file system, che sono mappati su dispositivi logici, che sono partizioni di dispositivi fisici.

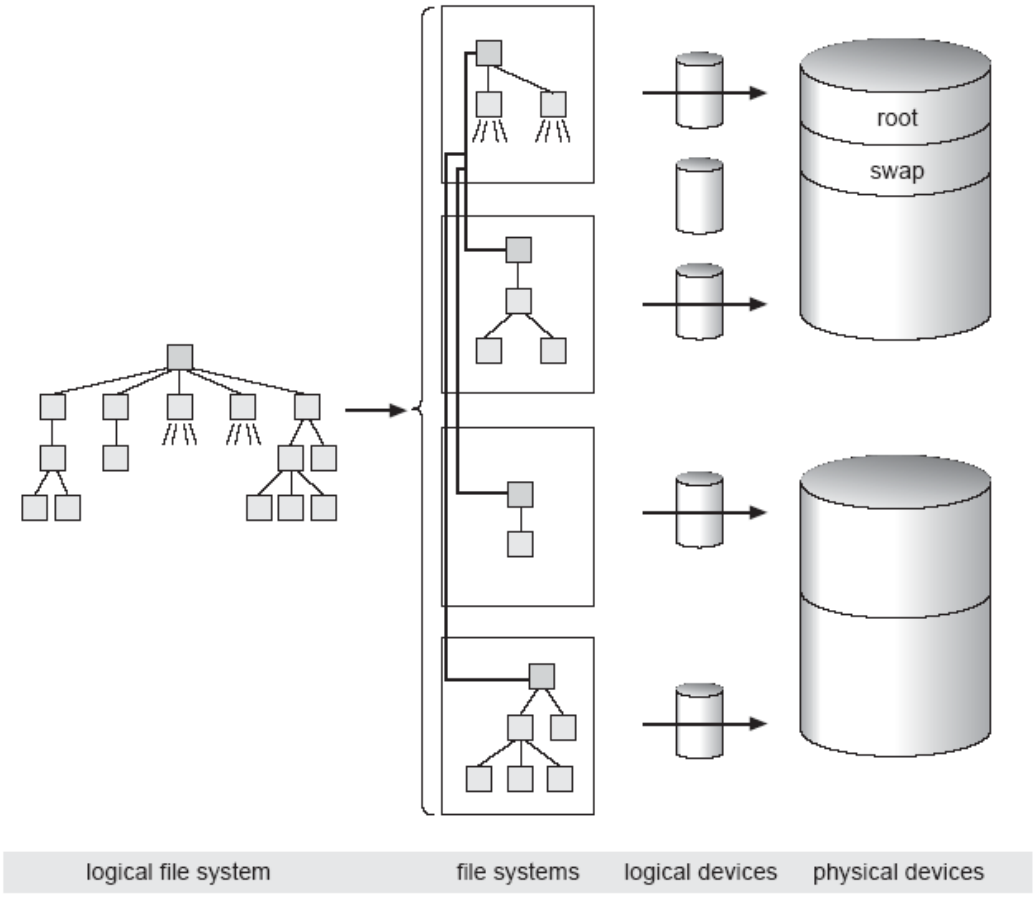

Figura 8. Mappatura di un file system logico nei dispositivi fisici.

Root = radice; swap; logical file system = file system logico; file system; logical devices = dispositivi logici
physical devices = dispositivi fisici

Le dimensioni e le posizioni di queste partizioni sono state codificate nei driver del dispositivo nei primi sistemi, ma sono mantenute su disco da FreeBSD.
Partizionare un dispositivo fisico in file system multipli ha parecchi benefici. File system differenti possono supportare usi differenti. Sebbene la maggior parte delle partizioni sarà usata dal file system, sarà necessario dedicare almeno una di esse alla zona di swap per il software della memoria virtuale. L'affidabilità viene migliorata, perché il danno che può essere provocato dal software è generalmente limitato solo ad un file system. Si può migliorare l'efficienza variando, per ogni partizione, i parametri del file system (quali le dimensioni del blocco e del frammento). Inoltre, file system separati impediscono ad un programma di usare tutto lo spazio disponibile per un grande file, poiché i file non possono essere spezzati fra i file system. In conclusione, i backup su disco vengono eseguiti in base alla partizione ed è più veloce cercare un file in un nastro di backup, se la partizione è piccola. Inoltre il ripristino completo della partizione dal nastro è pure più veloce.

Il numero di file system su un drive varia a seconda delle dimensioni del disco e le finalità del sistema di elaborazione. Un file system, il *root file system*, è sempre disponibile. Si possono montare altri file system, cioè integrarli nella gerarchia del direttorio del root file system.

Un bit nella struttura dell'inode indica che l'inode stesso ha un file system montato su di esso. Un riferimento a questo file causa una ricerca nella *tabella di montaggio* per trovare il numero del dispositivo montato. Il numero del dispositivo viene usato per trovare l'inode del direttorio di root del file system montato e quell'inode viene poi utilizzato. Per contro, se un elemento del path name ha come nome ".." e il direttorio cercato è quello di root di un file system montato, viene analizzata la tabella di montaggio per trovare l'inode su cui è montato che poi viene usato.

Ogni file system è una risorsa di sistema separata e rappresenta un gruppo di file. Il primo settore sul dispositivo logico è il *boot block* (blocco di avvio), che dovrebbe contenere il programma di bootstrap primario da usare per chiamare un programma di bootstrap secondario che risiede nei 7.5 KB successivi. Un sistema ha bisogno solo di una partizione di dati di bootstrap che contiene i dati del boot block, ma il gestore del sistema può installare duplicati con programmi privilegiati per consentire di eseguire il boot di sistema quando la copia primaria risulta danneggiata. Il *superblocco* (superblock) contiene parametri statici del file system. Questi parametri includono la dimensione totale del file system, le dimensioni del blocco e del frammento dei blocchi di dati e altri vari parametri che interessano le politiche di allocazione.

7.6 Implementazioni

L'interfaccia utente del file system è semplice e ben definita, permettendo che l'implementazione del file system sia cambiata senza alcun effetto significativo sull'utente. Il file system è stato cambiato fra la versione 6 e la versione 7 ed ancora fra le versioni 7 e 4BSD. Nella versione 7, le dimensioni degli inode sono raddoppiate, le dimensioni massime dei file e del file-system sono aumentate e sono pure cambiati i dettagli di gestione della lista dei blocchi liberi e le informazioni del superblocco. A quel temp, inoltre, *seek* (con uno spiazzamento di 16 bit) è diventata *lseek* (con uno spiazzamento di 32-bit), per permettere la specifica di spiazzamenti in file più grandi, ma pochi altri cambiamenti furono visibili al di fuori del kernel.

In 4.0BSD, la dimensione dei blocchi usati nel file system è stata aumentata da 512 a 1.024 byte. Sebbene questa dimensione incrementata abbia prodotto un aumento di frammentazione interna su disco, ciò ha raddoppiato la velocità di trasferimento, grazie al maggior numero di dati letti ad ogni trasferimento da disco. Questa modifica è stata successivamente adottata dal System V, con un certo numero di altre modifiche, driver di dispositivi e programmi.

4.2BSD ha aggiunto il Fast File System di Berkeley, che ha aumentato la velocità ed è stato accompagnato da nuove caratteristiche. I link simbolici hanno richiesto nuove chiamate di sistema. I nomi lunghi dei file hanno reso necessarie nuove chiamate di sistema per i direttori per attraversare l'attuale struttura complessa interna dei direttori. Infine, è stata aggiunta la chiamata **truncate**. Il Fast File System fu un successo ed ora si trova nella maggior parte dei sistemi UNIX. Le sue prestazioni sono permesse dalle politiche di allocazione e disposizione, che discuteremo più avanti. Nel Paragrafo 12.4.4, abbiamo discusso i cambiamenti operati in SunOS per aumentare ulteriormente la velocità di trasferimento del disco.

7.7 Politiche di disposizione e di allocazione

Il kernel usa una coppia <*numero di dispositivo logico, numero di inode*> per identificare un file. Il numero di dispositivo logico definisce il file system in questione. Gli inode nel file system sono numerati in sequenza. Nel file system della versione 7, tutti gli inode sono in un array che segue un singolo superblocco all'inizio del dispositivo logico, con i blocchi dati che seguono gli inode. Il *numero di inode* è un prorio un indice nell'array.

Con la versione 7 del file system, un blocco di un file può essere ovunque sul disco fra la fine dell'array di inode e la fine del file system. I blocchi liberi sono mantenuti in una lista collegata nel superblocco. I blocchi sono spinti verso la cima della lista libera e sono rimossi, in base alle necessità, per prestare servizio ai nuovi file o per estendere i file esistenti. Quindi, i blocchi di un file possono essere arbitrariamente lontani sia dall'inode che da un altro blocco. Inoltre, più viene usato un file system di questo tipo, più disorganizzati diventano i blocchi in un file. Si può invertire questo processo solo reinizializzando e ristabilendo l'intero file system, ma non è un'operazione conveniente da effettuare. Questo processo è stato descritto nel Paragrafo 12.7.2.

Un'altra difficoltà è che l'affidabilità del file system viene minacciata. Per aumentare la velocità, viene mantenuto in memoria il superblocco di ogni file system montato. Mantenere il superblocco in memoria permette al kernel di accedere rapidamente ad un superblocco, in particolare per usare la lista libera. Il superblocco viene scritto su disco ogni 30 secondi, , per mantenere sincronizzate le copie su disco e in memoria (mediante il programma di aggiornamento, tramite la chiamata di sistema **sync**). Tuttavia, i bachi del sistema o i guasti hardware possono causare un arresto al sistema che distrugge il superblocco in memoria tra un aggiornamento e l'altro. Quindi, la lista libera su disco non riflette esattamente lo stato del disco; per ricostruirla, si deve eseguire un lungo esame di tutti i blocchi nel file system. Questo problema permane nel nuovo file system.

La costruzione del file system di 4.2BSD è radicalmente differente da quella della versione 7. Questa ricostruzione è stata fatta principalmente per migliorarne l'efficienza e la robustezza, e la maggior parte di tali cambiamenti è invisibile fuori del kernel. Contemporaneamente sono stati introdotti altri cambiamenti, quali i link simbolici ed i nomi di file lunghi (fino a 255 caratteri), che sono entrambi visibili sia a livello di chiamata di sistema che a livello utente. Tuttavia la maggior parte dei cambiamenti richiesti per queste funzionalità non erano nel kernel, ma nei programmi che li usavano.

L'allocazione dello spazio è particolarmente differente. In FreeBSD, la nuova funzionalità principale è il *gruppo di cilindri*. Il gruppo di cilindri è stato introdotto per permettere la localizzazione dei blocchi in un file. Il gruppo di cilindri fu introdotto per permettere la localizzazione dei blocchi in un file. Ogni gruppo di cilindri occupa uno o più cilindri consecutivi del disco, in modo che gli accessi al gruppo di cilindri richiedono un minimo movimento della testina del disco. Ogni gruppo di cilindri ha un superblocco, un blocco di cilindri, un array di inode e alcuni blocchi di dati (Figura 9).

Il superblocco è identico in ogni gruppo di cilindri, in modo da poter essere recuperato da qualsiasi di essi in caso di corruzione del disco. Il *blocco di cilindri* contiene parametri dinamici del particolare gruppo di cilindri; essi includono una mappa di bit dei blocchi di dati e frammenti e degli inode liberi; inoltre sono qui memorizzate le statistiche su sviluppi recenti delle strategie di allocazione.

Le informazioni di intestazione in un gruppo di cilindri (il superblocco, il blocco di cilindri e gli inode) non sono sempre all'inizio del gruppo di cilindri. Se così fosse, le informazioni di intestazione per ogni gruppo di cilindri potrebbero essere sullo stesso piatto del disco; un singolo arresto della testina su un solo disco potrebbe eliminarle tutte. Pertanto, ogni gruppo di cilindri ha proprie informazioni di intestazione con un diverso spiazzamento dall'inizio del gruppo.
Il comando *ls,* che fornisce l'elenco del direttorio, legge normalmente tutti gli inode di ogni file del direttorio, rendendo desiderabile che tutti gli inode siano tra loro vicini su disco. Per questo motivo, l'inode di un file è normalmente allocato dallo stesso gruppo di cilindri poiché è l'inode del direttorio padre del file. Tuttavia, non tutto può essere localizzato, quindi un inode per un nuovo direttorio è messo in un gruppo di cilindri *diverso* da quello del suo direttorio padre. Il gruppo di cilindri scelto per il nuovo inode del direttorio è quello con il maggior numero di inode inutilizzati.

Figura 9. Gruppo di cilindri in 4.3BSD.

Per ridurre le ricerche della testina del disco nell'accesso ai blocchi di dati di un file, si allocano i blocchi, provenienti dallo stesso gruppo di cilindri, il più spesso possibile. Poiché non può essere permesso ad un singolo file di prendere tutti i blocchi in un gruppo di cilindri, un file che eccede un determinata dimensione (tipo 2 MB) ha un'ulteriore allocazione di blocchi redirezionata ad un differente gruppo di cilindri; il nuovo gruppo viene scelto fra da quelli che hanno lo spazio libero superiore alla media. Se il file continua a crescere, l'allocazione è redirezionata di nuovo (per ogni megabyte) ancora ad un altro gruppo di cilindri. Pertanto, tutti i blocchi di file piccoli sono probabilmente nello stesso gruppo di cilindri, e il numero di ricerche della testina per accedere ad un file grande è mantenuto piccolo.

Ci sono due livelli di procedure per l'allocazione di blocchi su disco. Le procedure di politica globale selezionano un blocco desiderato del disco secondo le considerazioni già discusse. Le procedure di politica locale usano le informazioni specifiche registrate nei blocchi del cilindro per scegliere un blocco vicino a quello richiesto; se tale blocco non è in uso, viene restituito. Altrimenti è selezionato il blocco più vicino per rotazione a quello chiesto nello stesso cilindro, o un blocco in un differente cilindro ma appartenente allo stesso gruppo. Se non ci sono più blocchi nel gruppo di

cilindri, viene eseguito di nuovo un hash quadratico fra tutti gli altri gruppi di cilindri per trovare un blocco e, se non viene trovato, viene fatta una ricerca esaustiva. Se viene lasciato abbastanza spazio libero (in generale il 10 per cento) nel file system, i blocchi sono solitamente trovati dove si desidera; in tal caso non si usa l'hash quadratico e la ricerca esaustiva e le prestazioni del file system non degradano con l'uso.

A causa dell'efficienza aumentata del Fast File System, i tipici dischi sono ora utilizzati al 30 per cento della loro capacità grezza di trasferimento. Questa percentuale è un notevole miglioramento di quella realizzata con il file system versione 7, che usava circa il 3 per cento della larghezza di banda.

BSD Tahoe ha introdotto il Fat Fast File System, che permette di configurare il numero di inode per gruppo di cilindri, il numero di cilindri per gruppo di cilindri ed il numero di posizioni di rotazione distinte, quando viene creato il file system. FreeBSD è solito regolare questi parametri secondo il tipo di hardware del disco.

8 Sistema I/O

Uno degli scopi di un sistema operativo è di nascondere all'utente le peculiarità di dispositivi hardware specifici. Per esempio, il file system presenta una semplice consistente funzionalità di memorizzazione (il file) indipendente dall'hardware sottostante del disco. In UNIX, le peculiarità dei dispositivi di I/O inoltre sono pure nascoste ad una gran parte del kernel dal *sistema di I/O* stesso. Il sistema di I/O consiste in un sistema di buffer dotato di cache, in un codice generale per i dispositivi driver e in driver per dispositivi hardware specifici. Solo il dispositivo del driver conosce le peculiarità di uno specifico dispositivo. Le parti principali del sistema I/O sono visualizzate in Figura 10.

In FreeBSD, ci sono tre generi principali di I/O: dispositivi a blocchi, dispositivi a carattere e interfaccia del *socket*. L'interfaccia del socket, con i propri protocolli e le interfacce di rete, saranno trattati nel Paragrafo 9.1 di questo capitolo.

I *dispositivi a blocchi* includono dischi e nastri. La loro caratteristica distintiva è che sono direttamente indirizzabili a blocchi di dimensione fissa, di solito 512 byte. Un dispositivo driver a blocchi è richiesto per isolare i dettagli delle tracce e dei cilindri dal resto del kernel. I dispositivi a blocchi sono accessibili direttamente attraverso appropriati file speciali del dispositivo (come */dev/rp0*), ma più comunemente indirettamente attraverso il file system. In entrambe i casi, i trasferimenti sono bufferizzati tramite la *cache del buffer del blocco*, che ha effetto notevole sull'efficienza.

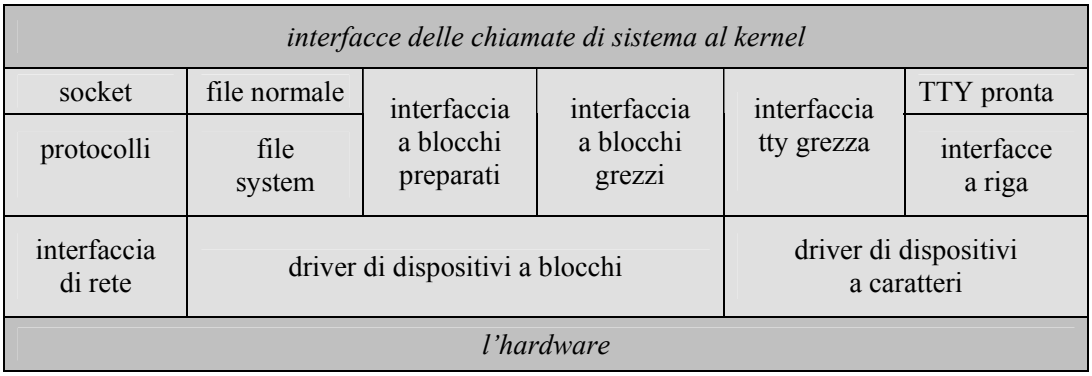

Figura 10. La struttura di I/O del kernel 4.3BSD.

I *dispositivi a carattere* includono i terminali e le stampanti in linea, ma anche quasi ogni altra cosa (tranne le interfacce di rete) che non usa la cache del buffer per i blocchi. Per esempio, */dev/mem* è un'interfaccia verso la memoria fisica centrale e */dev/null* è una voragine senza fondo per i dati e una fonte senza fine di marcatori di fine file (end-of-file). Alcuni dispositivi, quali le interfacce grafiche ad alta velocità, possono avere dei buffer propri o possono eseguire chiamate I/O direttamente nello spazio dati dell'utente; poiché non usano la cache del blocco del buffer e sono classificati come dispositivi a carattere. I terminali e dispositivi simili usano le C-list, che sono buffer più piccoli di quelli della cache del blocco del buffer.

I dispositivi a *blocchi* ed i dispositivi a *carattere* sono le due classi del dispositivo principale. Ai driver del dispositivo si accede tramite uno di due array formato dai punti di ingresso. Un array è per i dispositivi a blocchi; l'altro è per i dispositivi a carattere. Un dispositivo è contraddistinto da una classe (blocco o carattere) e da un *numero di dispositivo*. Il numero di dispositivo consta di due parti. Il *numero di dispositivo maggiore* è usato per indicizzare l'array per dispositivi a blocchi o a carattere, per trovare gli indirizzi nel driver di dispositivo adatto. *Il numero di dispositivo minore* è interpretato dal driver del dispositivo come, per esempio, una partizione logica del disco o una riga del terminale.

Un driver del dispositivo è collegato al resto del kernel solo dai punti di ingresso registrati nell'array per la sua classe e tramite l'uso di sistemi comuni di buffering. Questa separazione è importante per la portabilità e per la configurazione del sistema.

8.1 La cache dei buffer dei blocchi

I dispositivi a blocchi usano una cache dei buffer dei blocchi. La cache del buffer consiste di un certo numero di intestazioni del buffer, ciascuno delle quali può puntare ad una porzione della memoria fisica, come pure ad un numero del dispositivo e ad un numero del blocco nel dispositivo. Le intestazioni del buffer per i blocchi non correntemente in uso sono tenute in parecchie liste collegate, una per ogni buffer:

- buffer recentemente usati, collegati con ordinamento LRU (la lista LRU)
- buffer non usati recentemente, o senza valido contenuto (la lista AGE)
- buffer VUOTI senza memoria fisica associata ad essi

I buffer in queste liste sono inoltre indicizzati dal numero del dispositivo e del blocco per rendere la ricerca efficiente. Quando un blocco è richiesto da un dispositivo (una read), si ricerca nella cache. Se il blocco viene trovato, viene usato e non è necessario alcun trasferimento di I/O. Se non viene trovato, si sceglie un buffer nella lista AGE, o nella lista LRU se AGE è vuota. Vengono poi aggiornati il numero di dispositivo ed il numero di blocco associato, se necessario viene trovata la memoria ed i nuovi dati vengono trasferiti in essa dal dispositivo. Se non ci sono buffer vuoti, il buffer LRU viene scritto nel proprio dispositivo (se è modificato) e il buffer è riutilizzato.

Durante una scrittura, se il blocco in questione è già nella cache del buffer, i nuovi dati vengono messi nel buffer (sovrascrivendo qualsiasi dato precedente); l'intestazione del buffer è contrassegnata per indicare che il buffer è stato modificato e che nessuna operazione di I/O è immediatamente necessaria. I dati saranno scritti quando il buffer sarà necessario per altri dati. Se non si trova il blocco nella cache del buffer, si sceglie un buffer vuoto (come avviene nella read) e si esegue un trasferimento in questo buffer.

Periodicamente vengono forzate delle scritture per i blocchi sporchi del buffer per minimizzare potenziali inconsistenze del file system dopo un guasto del sistema.

In FreeBSD, il numero di dati in un buffer è variabile, fino ad un massimo, per tutti i file system, solitamente di 8 KB. La dimensione minima coincide con la dimensione del gruppo di paginazione, solitamente 1024 byte. I buffer sono gruppi di pagine allineate e qualsiasi gruppo di pagine può essere mappato soltanto in un buffer alla volta, proprio come ogni blocco del disco può essere mappato solo in un buffer alla volta. La lista di quelli VUOTI contiene le intestazioni del buffer che vengono usate se un blocco di memoria fisica di 8 KB è frammentato per contenere molti blocchi più piccoli. Le intestazioni sono necessarie per questi blocchi e sono recuperate dalla lista di quelli VUOTI.
Il numero di dati in un buffer può crescere quando un processo utente scrive più dati di quelli già nel buffer. Quando accade questo aumento di dati, viene allocato un nuovo buffer, sufficientemente grande, per contenere tutti i dati, e i dati originali vengono copiati in esso, seguiti dai nuovi dati. Se un buffer si restringe, dalla coda vuota viene tolto un buffer, le pagine in eccesso sono messe in esso e quel buffer viene rilasciato per essere scritto su disco.

Alcuni dispositivi, come i nastri magnetici, richiedono che i blocchi siano scritti in un certo ordine. In questo caso sono fornite funzionalità per forzare una scrittura sincrona dei buffer su questi dispositivi, nell'ordine corretto.

I blocchi dei direttori sono pure scritti in modo sincrono, per prevenire inconsistenze dovute ai guasti. Si pensi al caos che potrebbe accadere se molti cambiamenti fossero fatti ad un direttorio, senza aggiornare gli elementi dei direttori stessi.

La dimensione della cache del buffer può avere un effetto notevole sulle prestazioni di un sistema, poiché, se è abbastanza grande, la percentuale degli utilizzi di cache può essere alta e basso il numero di trasferimenti I/O reali. FreeBSD ottimizza continuamente la cache del buffer registrando la quantità di memoria usata dai programmi e la cache del disco.

Ci sono alcune interazioni interessanti fra la cache del buffer, il file system ed i driver del disco. Quando si scrivono i dati in un file su disco, essi vngono bufferizzati nella cache ed il driver del disco riordina la propria coda di output in base all'indirizzo su disco. Queste due azioni permettono al driver del disco di minimizzare le ricerche della testina e di scrivere i dati in momenti ottimizzati secondo la rotazione del disco. A meno che non siano richieste scritture sincrone, un processo di scrittura su disco scrive semplicemente nella cache del buffer ed il sistema scrive in modo asincrono i dati su disco nel momento opportuno. Il processo utente vede scritture molto veloci. Quando i dati sono letti da un file su disco, il sistema di blocco di I/O esegue una qualche lettura anticipata; tuttavia, le scritture sono molto più prossime ad essere asincrone rispetto alle letture. Pertanto, l'output su disco, tramite il file system, è spesso più veloce dell’input per grandi trasferimenti, al contrario di quanto si possa credere.

8.2 Interfacce di dispositivi grezzi (raw)

Quasi ogni dispositivo a blocchi ha anche un'interfaccia a caratteri che è chiamata *interfaccia di dispositivo grezzo* (raw). Una tale interfaccia differisce dall'*interfaccia a blocchi* in quanto la cache del buffer per i blocchi viene scavalcata.

Ogni driver del disco mantiene una coda di trasferimenti pendenti. Ogni elemento nella coda specifica se è una lettura o una scrittura, l’indirizzo della memoria centrale per il trasferimento (solitamente in incrementi da 512 byte), l’indirizzo del dispositivo per il trasferimento (solitamente l'indirizzo di un settore del disco) e la dimensione del trasferimento (in settori). È semplice mappare le informazioni da un buffer del blocco a ciò che è richiesto per questa coda.

È quasi semplice come mappare una porzione della memoria centrale, che corrisponde alla porzione dello spazio dell'indirizzo virtuale di un processo utente. Tale mappatura è ciò che esegue, per esempio, un'interfaccia grezza (raw) del disco. Sono così permessi trasferimenti non bufferizzati direttamente verso o dallo spazio di indirizzamento virtuale dell'utente. La dimensione del trasferimento è limitata dai dispositivi fisici, alcuni dei quali richiedono un numero pari di byte.

Il kernel compie trasferimenti per operazioni di swap e paginazione, inserendo semplicemente l'appropriata richiesta nella coda del dispositivo adatto. Non è necessaria alcuna operazione di swap o di paginazione da parte del driver del dispositivo.

La realizzazione del file system 4.2BSD è stata attualmente scritta ed in gran parte controllata come un processo utente che usava un'interfaccia grezza (raw) del disco, prima che il codice fosse trasferito nel kernel. In una contrapposizione interessante, il sistema operativo Mach lo ha realizzato come task a livello utente.

8.3 Le c-list

I driver dei terminali usano un sistema di bufferizzazione a caratteri (byte in liste collegate). Ci sono procedure per accodare e togliere dalla coda i caratteri per tali liste. Sebbene tutti i buffer liberi a carattere siano mantenuti in una singola lista libera, la maggior parte dei driver del dispositivo che li usa limita il numero di caratteri che possono essere accodati contemporaneamente per una qualsiasi linea del terminale.

Una chiamata di sistema **write** verso un terminale accoda i caratteri in una lista per il dispositivo. Viene avviato un trasferimento iniziale, e gli interrupt causano un de-accodamento dei caratteri e ulteriori trasferimenti.

In modo simile anche l'input è guidato dagli interrupt. I driver di terminale tipicamente supportano *due* code di input e la conversione dalla prima coda (coda grezza) all'altra (coda canonica) è innescata dalla routine di interrupt che pone un carattere di fine riga nella coda grezza. Il processo che esegue una read sul dispositivo viene poi risvegliato e la propria fase di sistema esegue la conversione; i caratteri così posti nella coda canonica sono poi disponibili per essere ritornati al processo utente dalla read.
Il driver del dispositivo può evitare la coda canonica e restituire i caratteri direttamente dalla coda grezza. Questo modo di operare è conosciuto come *modo grezzo* (raw mode). Gli editor a schermo pieno ed altri programmi che devono reagire ad ogni input da tastiera, usano questa modalità.

9 Comunicazione tra processi

Molti compiti possono essere svolti in processi isolati, ma molti altri richiedono la comunicazione tra processi. Sistemi di elaborazione isolati sono stati a lungo usati per molte applicazioni, ma la rete sta diventando sempre più importante. Con l'uso crescente di workstation personali, la condivisione delle risorse sta diventando più comune. La comunicazione tra processi non è stata tradizionalmente uno dei punti forti di UNIX.

9.1 Socket

La *pipe* (discussa nel Paragrafo 4.3 di questo capitolo) è il meccanismo IPC più caratteristico di UNIX. Una pipe consente un flusso unidirezionale affidabile di byte fra due processi. È tradizionalmente

realizzato come un file ordinario, con poche eccezioni. Non ha un nome nel file system, essendo, a differenza dagli altri file, creata dalla chiamata di sistema **pipe**. La sua dimensione è fissata e quando un processo tenta di scrivere in una pipe piena, tale processo viene sospeso. Una volta che tutti i dati precedentemente scritti nella pipe sono stati letti, la scrittura prosegue all'inizio del file (le pipe non sono veri buffer circolari). Un beneficio riguardo la piccola dimensione (solitamente 4096 byte) delle pipe è che i dati della pipe vengono raramente scritti realmente sul disco; sono solitamente mantenuti in memoria dalla normale cache dei buffer dei blocchi.

In FreeBSD le pipe sono realizzate come caso speciale del meccanismo di *socket*, che fornisce un'interfaccia generale non solo per funzionalità quali le pipe, che sono locali in una macchina, ma anche per le funzionalità di rete. Anche sulla stessa macchina, una pipe può essere usata soltanto da due processi collegati dall'uso della chiamata di sistema **fork**. Il meccanismo di socket può essere usato da processi indipendenti.

Un socket è un punto terminale di comunicazione, che ha normalmente un *indirizzo* collegato ad esso. La natura dell'indirizzo dipende dal *dominio di comunicazione* del socket. Una proprietà caratteristica di un dominio è che i processi, che comunicano nello stesso dominio, usano lo stesso *formato dell'indirizzo*. Un singolo socket può comunicare in solo un dominio.

I tre domini attualmente realizzati in FreeBSD sono il dominio UNIX (AF_UNIX), il dominio Internet (AF_INET) ed il dominio di servizi di rete XEROX (AF_NS). Il formato dell'indirizzo del dominio UNIX è quello dei nomi ordinari di percorso del filesystem, come */alpha/beta/gamma*. I processi che comunicano nel dominio Internet usano i protocolli di comunicazioni di DARPA Internet (quali TCP/IP) e gli indirizzi Internet, che consistono di un numero di host a 32 bit e di un numero di porta a 32 bit (che rappresenta un punto d'incontro sull'host).

Ci sono parecchi **tipi di socket**, che rappresentano classi di servizi; ogni tipo di servizio può o non può essere abilitato in un qualsiasi dominio di comunicazione. Se un tipo è abilitato in un dato dominio, ciò può avvenire mediante uno o più protocolli, che possono essere selezionati dall'utente:

- **Stream sockets** (socket di flusso)**:** Questi socket forniscono degli stream affidabili, full-duplex e con dati in sequenza. Nessun dato va perso o duplicato nella consegna, e non ci sono limiti di struttura. Questo tipo è supportato nel dominio Internet dal protocollo TCP. Nel dominio UNIX, le pipe sono realizzate come una coppia di socket di flusso comunicanti.

- **Socket a sequenza di pacchetti:** Questi socket forniscono flussi di dati analoghi ai socket di flusso, tranne che sono forniti anche i confini del record. Questo tipo è usato nel protocollo XEROX AF_NS.

- **Socket a datagramma**: Questi socket trasferiscono messaggi di dimensioni variabili in entrambe le direzioni. Non c'è alcuna garanzia che tali messaggi arrivino nello stesso ordine con cui sono stati trasmessi, o che non saranno duplicati, o che non arriveranno affatto, ma la dimensione originale del messaggio (o record) è preservata in qualsiasi datagramma che arriva. Questo tipo è supportato nel dominio Internet dal protocollo UDP.

- **Socket a consegna affidabile del messaggio**: Questi socket trasferiscono i messaggi con la garanzia di arrivare, e che altrimenti sono simili ai messaggi trasferiti per mezzo di socket a datagramma. Questo tipo non è attualmente supportato

- **Raw socket (sochet grezzi)**: Questi socket permettono l'accesso diretto dei processi ai protocolli che supportano gli altri tipi di socket. I protocolli accessibili includono non solo quelli a livello più alto, ma anche protocolli a livello più basso.

Per esempio, nel dominio Internet, è possibile raggiungere il TCP, l'IP sottostante, o un protocollo Ethernet sotto di esso. Questa possibilità è utile per sviluppare nuovi protocolli.

Le funzionalità dei socket hanno un insieme di chiamate di sistema specifiche. La chiamata di sistema `socket` crea un socket. Prende come argomenti specifiche del dominio di comunicazione, il tipo di socket e il protocollo da usare per supportare quel tipo. Il valore restituito dalla chiamata è un piccolo numero intero chiamato **descrittore del socket**, che si trova nello stesso spazio del nome come i descrittori dei file. Il descrittore del socket indicizza l'array dei *file* aperti nella struttura *u* nel kernel che ha una struttura di file allocata per esso. La struttura del file in FreeBSD può puntare ad una struttura di socket anziché ad un inode. In questo caso, certe informazioni del socket (quali, il tipo di socket, il conteggio dei messaggi ed i dati nelle proprie di input e output) sono mantenute direttamente nella struttura del socket.

Affinché un altro processo possa indirizzare un socket, questi deve avere un nome collegato alla chiamata di sistema `bind`, che prende il descrittore del socket, un puntatore al nome e la lunghezza del nome come stringa di byte. Il contenuto e la lunghezza della stringa di byte dipendono dal formato dell'indirizzo. La chiamata di sistema `connect` viene usata per iniziare una connessione. Gli argomenti sono sintatticamente gli stessi di quelli della `bind`; il descrittore del socket rappresenta il socket locale e l'indirizzo è quello del socket remoto verso cui si esegue il tentativo di collegamento.

Molti processi che comunicano per mezzo dei socket IPC seguono il modello client-server, in cui il processo *server* fornisce un servizio al processo *client*. Quando il servizio è disponibile, il processo server ascolta ad un indirizzo ben noto, e il processo client usa `connect` per raggiungere il server.

Un processo server utilizza la chiamata `socket` per creare un socket e `bind` per collegare l'indirizzo ben noto del proprio servizio a quel socket. Poi, usa la chiamata di sistema `listen` per comunicare al kernel che è pronto ad accettare connessioni dai client e per specificare quante connessioni in attesa il kernel dovrà accodare fino a quando il server potrà servirle. Infine, il server usa la chiamata di sistema `accept` per accettare connessioni individuali. Sia `listen` che `accept` prendono come argomento il descrittore del socket originale. La chiamata di sistema `accept` restituisce un nuovo descrittore di socket corrispondente alla nuova connessione; il descrittore originale è ancora aperto per ulteriori connessioni. Il server solitamente usa `fork` per produrre un nuovo processo dopo la chiamata `accept` per servire il client mentre il processo originale del server continua a rimanere in ascolto per altre connessioni.

Ci sono anche chiamate di sistema per configurare i parametri di una connessione e per restituire l'indirizzo del socket remoto dopo una chiamata `accept`.

Quando viene stabilita una connessione per un tipo di socket quale un socket di flusso, gli indirizzi di entrambi i punti terminali sono noti e non sono necessarie ulteriori informazioni di indirizzo per trasferire i dati. Le comuni chiamate di sistema ~~read~~ e ~~write~~ possono quindi essere usate per trasferire i dati.

Il modo più semplice per terminare una connessione, e distruggere il socket associato, è di usare la chiamata di sistema `close` sul proprio descrittore del socket. Si può anche voler terminare solo la comunicazione in un senso in una connessione duplex; a questo fine può essere usata la chiamata di sistema `shutdown`.

Alcuni tipi di socket, quali i datagrammi socket, non supportano connessioni; invece scambiano i datagrammi che devono essere indirizzati individualmente; per tali connessioni vengono usate le chiamate di sistema `sendto` e `recvfrom`. Entrambe prendono come argomenti un descrittore del socket, un puntatore del buffer e la lunghezza, e un puntatore dell'indirizzo del buffer e la lunghezza. Il buffer dell'indirizzo contiene l'indirizzo a cui inviare con la `sendto` ed è riempito con l'indirizzo del datagramma appena ricevuto da `recvfrom`. Il numero di dati attualmente trasferiti è restituito da entrambe le chiamate di sistema.

La chiamata di sistema `select` può essere usata per effettuare il multiplex dei trasferimenti di dati su parecchi descrittori del file e/o descrittori del socket. Può anche essere usata per permettere a un processo server di ascoltare le connessioni client di molti servizi e di eseguire una `fork` di un processo per ogni connessione quando si instaura la connessione. Il server esegue `socket`, `bind` e `listen` per ogni servizio e poi una `select` su tutti i descrittori del socket. Quando `select` indica attività su di un descrittore, il server esegue `accept` su di esso ed esegue la fork di un processo sul nuovo descrittore restituito da `accept`, lasciando che il processo padre esegua di nuovo una `select`.

9.2 Supporto di rete

Quasi tutti gli attuali sistemi UNIX supportano i servizi di rete UUCP, che sono principalmente usati su linee telefoniche per supportare la rete della posta UUCP e la rete USENET delle news (notizie). Queste sono, tuttavia, funzionalità rudimentali di rete: non supportano neppure il login remoto, e tanto meno la remote procedure call o i file-system distribuiti. Queste funzionalità sono quasi completamente realizzate come processi utente e non fanno parte del sistema operativo stesso.

FreeBSD supporta i protocolli DARPA di internet: UDP, TCP, IP, e ICMP su una vasta gamma di reti Ethernet, token-ring e su interfacce ARPANET. L'ossatura del kernel per supportare queste funzionalità si propone di facilitare la realizzazione di ulteriori protocolli, e tutti i protocolli saranno accessibili tramite l'interfaccia del socket. Rob Gurwitz di BBN ha scritto la prima versione del codice come un pacchetto aggiuntivo per 4.1BSD.

Il modello di riferimento per la rete dell'International Standards Organization's (ISO) Open System Interconnection (OSI) prescrive sette strati di protocolli di rete e metodi rigorosi di comunicazione fra di essi. Una realizzazione di un protocollo può comunicare solo con un'entità paritetica che parla con lo stesso protocollo nello stesso strato, o con l'interfaccia protocollo-protocollo di un protocollo nello strato immediatamente sopra o sotto nello stesso sistema. Il modello della rete ISO è realizzato in FreeBSD Reno e 4.4BSD.

La realizzazione della rete di FreeBSD e, in una certa misura la funzionalità del socket, è più orientata verso il modello di riferimento di ARPANET (ARM). ARPANET nella sua forma originale è servito come base concettuale per i molti concetti di rete come la commutazione a pacchetti e stratificazione del protocollo.

ARPANET è stato ritirato nel 1988 perché l'hardware che lo supportava era più allo stato dell'arte. I suoi successori, quali NSFNET e Internet, sono ancora più estesi e servono come utilità di comunicazione per i ricercatori e come banco di prova per ricerche sul gateway di Internet. ARM ha monopolizzato il modello ISO che è stato in grande parte ispirato dalla ricerca di ARPANET.

Anche se il modello ISO spesso viene interpretato come richiesta di un limite di un protocollo di comunicazione per lo strato, ARM permette parecchi protocolli nello stesso strato. In ARM, ci sono solo quattro strati di protocollo:

- **Processo/Applicazioni:** questo strato include l'applicazione, la presentazione e gli strati di sessione del modello ISO. I programmi a livello utente come il file transfer protocol (FTP), Telnet (remote login) operano a questo livello.

- **Host-Host**: questo strato corrisponde allo strato di trasporto ISO ed alla parte superiore dei propri strati di rete. Sia transmission control protocol (TCP) che internet protocol (IP) sono in questo strato, con TCP sopra IP. TCP corrisponde ad un protocollo di trasporto ISO e IP realizza le funzioni di indirizzamento dello strato di rete ISO.

- **Interfaccia di rete**: questo strato copre la parte inferiore dello strato di rete ISO e l'intero strato data-link. I protocolli qui coinvolti dipendono dal tipo fisico di rete. ARPANET usa protocolli IMP-Host, mentre Ethernet usa i protocolli Ethernet.

- **Hardware di rete**: ARM è soprattutto coinvolto con il software, così non c'è uno strato esplicito per l'hardware di rete; tuttavia, una qualsiasi rete attuale avrà l'hardware corrispondente allo strato fisico ISO.

L'ossatura della rete in FreeBSD è più generalizzata di quella del modello ISO o ARM, sebbene sia più strettamente collegata a quella ARM (Figura 11).

I processi utente comunicano con i protocolli di rete (e quindi con altri processi su altre macchine) tramite le funzionalità del socket, che corrispondono allo strato di sessione ISO, poiché sono responsabili dell'inizializzazione e del controllo delle comunicazioni.

I socket sono supportati dai protocolli, possibilmente da parecchi, stratificati uno sull'altro. Un protocollo può fornire servizi come la consegna affidabile, la soppressione di trasmissioni duplicate, il controllo di flusso, o di indirizzamento, a seconda del tipo di socket supportato, e i servizi richiesti da qualsiasi protocollo di livello più elevato.

Un protocollo può comunicare con un altro protocollo o con l'interfaccia di rete, adatta per l'hardware di rete. C'è una piccola limitazione nell'impostazione generale su quali protocolli possono comunicare con altri protocolli, o su come i protocolli possono essere stratificatai l'uno sull'altro. Il processo utente può, per mezzo del tipo di socket grezzo (raw), accedere direttamente a qualsiasi strato del protocollo dal più elevato, usato per supportare uno degli altri tipi di socket, come i flussi, giù fino all'interfaccia di rete grezza. Questa capacità è usata dai processi di routing e pure per lo sviluppo di nuovi protocolli.

Si tende a costruire un **driver di interfaccia di rete** per tipo di controller di rete. L'interfaccia di rete è responsabile della gestione delle caratteristiche specifiche della rete locale che viene indirizzata. Questa disposizione assicura che i protocolli che usano l'interfaccia non hanno bisogno di preoccuparsi di queste caratteristiche.

Le funzioni dell'interfaccia della rete dipendono in gran parte dall'**hardware della rete**, che è qualsiasi cosa necessaria per la rete a cui è connessa. Alcune reti possono supportare, a questo livello,trasmissioni affidabili ma la maggior parte non lo fa. Alcune reti forniscono indirizzi broadcast, ma molte non li forniscono.

Le funzionalità dei socket e l'ossatura di rete usano un gruppo comune di buffer di memoria, o *mbuf*. Questi hanno una dimensione intermedia fra i grandi buffer usati dal sistema di blocco I/O e le C-list usate dai dispositivi a carattere. Un *mbuf* è lungo 128 byte, di cui 112 byte possono essere usati per i dati; il resto è usato per puntatori per collegare *mbuf* nelle code e per gli indicatori dell'ampiezza della zona dei dati attualmente in uso.

I dati sono solitamente passati fra strati negli *mbuf*: protocollo-socket, protocollo-protocollo, o protocollo-interfaccia di rete,. Questa capacità di passare i buffer che contengono i dati elimina alcune operazioni di copia dei dati, ma c'è ancora frequentemente necessità di rimuovere o aggiungere intestazioni del protocollo. per molti scopi è pure conveniente ed efficiente poter tenere dati che occupano un'area della dimensione della pagina di gestione della memoria. Quindi, i dati di *mbuf* possono risiedere non in mbuf stesso, ma altrove in memoria. A questo scopo esiste una tabella di pagina di *mbuf*, come pure un gruppo di pagine dedicate all'uso di mbuf.

modello di riferimento	modello di riferimento	strati di	esempio di stratificazione

<table>
<tr><th>ISO</th><th>ARPANET</th><th>4.2BSD</th><th></th></tr>
<tr><td>Applicazione</td><td rowspan="3">Processi
di applicazioni</td><td rowspan="2">Programmi utente
e librerie</td><td rowspan="2">telnet</td></tr>
<tr><td>Presentazione</td></tr>
<tr><td>Sessione</td><td>Socket</td><td>Sock_stream</td></tr>
<tr><td>Trasporto</td><td rowspan="2">Host-host</td><td rowspan="2">Protocollo</td><td>TCP</td></tr>
<tr><td rowspan="2">Collegamento
di rete</td><td>IP</td></tr>
<tr><td rowspan="2">Interfaccia di rete</td><td rowspan="2">Interfaccie di rete</td><td rowspan="2">Driver Ethernet</td></tr>
<tr><td>Hardware</td></tr>
<tr><td></td><td>Hardware di rete</td><td>Hardware di rete</td><td>Controller interlan</td></tr>
</table>

Figura 11. Modelli di riferimento di rete e stratificazioni.

Sommario

I vantaggi iniziali di UNIX erano che questo sistema fu scritto in un linguaggio al alto livello, distribuito sotto forma di sorgenti e fornito di potenti primitive del sistema operativo su di una piattaforma economica. Questi vantaggi hanno condotto alla popolarità di UNIX nel mondo dell'istruzione, della ricerca, delle istituzioni governative e talvolta nel mondo commerciale. Questa popolarità in primo luogo ha prodotto molti sforzi su UNIX con varianti e funzionalità migliorate.

UNIX fornisce un file system con direttori strutturati ad albero. Il kernel supporta file come sequenze non strutturate di byte. L'accesso diretto e sequenziale sono supportati con chiamate di sistema e procedure di libreria.

I file sono memorizzati come un array di blocchi dati di misura fissata con al più un frammento finale. I blocchi di dati si recuperano per mezzo di puntatori negli inode. Gli ingressi del direttorio puntano agli inode. Lo spazio su disco è allocato da gruppi di cilindri al fine di minimizzare il movimento della testina e di migliorarne le prestazioni.

UNIX è un sistema multiprogrammato. I processi possono facilmente generare nuovi processi con la chiamata di sistema `fork`. I processi possono comunicare con pipe o più generalmente con i socket e possono essere raggruppati in job che possono essere controllati con i segnali.

I processi sono rappresentati da due strutture: la struttura di processo e la struttura utente. La schedulazione della CPU è un algoritmo di priorità che calcola dinamicamente le priorità e che si riduce alla schedulazione round-robin nel caso estremo. La gestione della memoria in FreeBSD è basata sullo swap supportata dalla paginazione. Un processo *pagedaemon* usa una procedura modificata di rimpiazzo della pagina di seconda-possibilità per mantenere abbastanza frame liberi per supportare i processi in esecuzione.

L'I/O di pagina e di file usa una cache del buffer del blocco per minimizzare la quantità attuale di I/O. I dispositivi a terminale usano un sistema separato di buffer per i caratteri.

Il supporto di rete è una delle caratteristiche più importanti in FreeBSD. Il concetto del socket fornisce al meccanismo di programmazione la possibilità di accedere ad altri processi, anche attraverso una rete. I socket forniscono un'interfaccia a parecchi protocolli.

Esercizi

1. In che modo gli obiettivi architetturali di UNIX erano differenti da quelli di altri sistemi operativi durante le fasi iniziali di sviluppo di UNIX?

2. Perché attualmente sono disponibili molte versioni differenti di UNIX? In quale modo questa differenza è un vantaggio per UNIX? In quale modo è uno svantaggio?

3. Quali sono i vantaggi e gli svantaggi di scrivere un sistema operativo in un linguaggio ad alto livello, come il C?

4. In quali circostanze la sequenza di chiamate di sistema `fork execve` è più appropriata? Quando è preferibile `vfork`?

5. FreeBSD assegna le priorità di schedulazione ai processi legati all'I/O o alla CPU? Per quale motivo fa differenza fra queste categorie e perché ad una viene data una priorità sull'altra? Come fa a sapere quale di queste categorie si adatta ad un dato processo?

6. I primi sistemi UNIX usavano lo swap per la gestione della memoria, ma FreeBSD usa la paginazione e lo swap. Discutere vantaggi e svantaggi dei due metodi di gestione della memoria.

7. Descrivere le modifiche ad un file system, che introduce FreeBSD, quando un processo richiede la creazione di nuovo file */tmp/foo* e scrive sequenzialmente in quel file fino a raggiungere la dimensione di 20 KB.

8. I blocchi dei direttori in FreeBSD sono scritti in modo sincrono quando vengono cambiati. Si consideri cosa potrebbe accadere se fossero scritti in modo asincrono. Descrivere lo stato del file system se accadesse un arresto dopo che tutti i file in un direttorio sono stati cancellati, ma prima dell'aggiornamento della voce del direttorio su disco.

9. Descrivere il processo per ricreare la lista libera dopo che un arresto in 4.1BSD.

10. Quali effetti sulle prestazioni del sistema avrebbero i seguenti cambiamenti in FreeBSD? Spiegate le risposte.

a. Raggruppamento di I/O su disco in gruppi più grandi
b. Realizzazione ed uso della memoria condivisa per passare i dati fra i processi, invece che tramite RPC o i socket
c. Usando il modello a sette strati della rete ISO, invece del modello di rete ARM

11. Quale tipo di socket si dovrebbe usare per realizzare un programma di trasferimento di file tra computer? Quale tipo si dovrebbe usare in un programma che controlla periodicamente se un altro calcolatore è attivo nella rete? Spiegare la risposta.

Note bibliografiche

La migliore descrizione generale delle caratteristiche peculiari di UNIX è ancora quella presentata da Ritchie e Thompson [1974]. Gran parte della storia di UNIX è stata data in Ritchie [1979]. Una critica a UNIX si trova in Blair et al. [1985].

Probabilmente il miglior libro sulla programmazione generale in UNIX, in particolare sull'uso della shell e di funzionalità quali `yacc` e `sed`, è quello di Kernighan e Pike [1984]. La programmazione dei sistemi è stata trattata da Stevens [1992]. Un altro testo interessante è Bourne [1983]. Il linguaggio di programmazione scelto sotto UNIX è il C, Kernighan e Ritchie [1988]. C'è pure il linguaggio di implementazione del sistema. La shell di Bourne è stata descritta in Bourne [1978]. La shell di Korn è stata descritte in Korn [1983]. La programmazione della rete UNIX è stata descritta in modo esteso da Stevens [1997].

La documentazione che viene fornita con i sistemi UNIX è chiamata *UNIX Programmer's Manual:UPM* ed è organizzata in due volumi. Il volume 1 contiene brevi voci per ogni comando, chiamate di sistema e la libreria di sottoprocedure, che nel sistema sono disponibili dalla linea di comando `man`. Il volume 2, *Supplementary Documents* (solitamente diviso nei volumi 2A e 2B per comodità di rilegatura), contiene vari importanti lavori sul sistema e manuali per quei comandi o pacchetti troppo complessi da essere descritti in una o due pagine. I sistemi Berkeley aggiungono il volume 2C che contiene documenti riguardanti caratteristiche specifiche di Berkeley. FreeBSD è descritto in The freebsd Handbook FreeBSD [1999] e può essere scaricato da http://www.freebsd.org/.