Capitolo 2

Strutture dei sistemi operativi

OBIETTIVI

- **Descrizione dei servizi messi a disposizione dal sistema operativo a utenti, processi e altri sistemi.**
- **Esame delle possibili strutture dei sistemi operativi.**
- **Installazione e adattamento dei sistemi operativi; descrizione delle operazioni da eseguire all'avvio.**

I sistemi operativi forniscono l'ambiente in cui si eseguono i programmi. Essendo organizzati secondo criteri che possono essere assai diversi, lo può essere anche la struttura interna che li caratterizza. La progettazione di un nuovo sistema operativo è un compito complesso, perciò è necessario definirne in modo chiaro gli scopi. Il tipo di sistema desiderato definisce i criteri di scelta dei metodi e degli algoritmi necessari.

Un sistema operativo si può considerare da diverse angolazioni: secondo i servizi che esso fornisce o l'interfaccia messa a disposizione degli utenti e dei programmatori, oppure secondo i suoi componenti e le relative interconnessioni. In questo capitolo vengono analizzati questi tre aspetti, mostrando il punto di vista dell'utente, del programmatore e del progettista. Si esaminano i servizi offerti da un sistema operativo e la modalità e i metodi da adottare per la sua progettazione. Infine se ne descrive creazione e avvio.

2.1 Servizi di un sistema operativo

Un sistema operativo offre un ambiente in cui eseguire i programmi e fornire servizi. Naturalmente, i servizi specifici variano secondo il sistema operativo, ma si possono identificare alcune classi di servizi comuni. Il loro scopo è facilitare il compito dei programmatori di applicazioni. La Figura 2.1 fornisce una panoramica dei servizi del sistema operativo e delle loro correlazioni.

Ogni insieme di servizi offre funzionalità utili all'utente.

- **Interfaccia con l'utente**. Quasi tutti i sistemi operativi hanno un'**interfaccia con l'utente** (UI). Essa può assumere diverse forme. Un'**interfaccia a riga di comando** (CLI) è basata su stringhe che codificano i comandi, insieme a un metodo per inserirli e modificarli come ad esempio un programma apposito. Un'**interfaccia a lotti**, invece, prevede che comandi e relative direttive siano codificati nei file, eseguiti successivamente

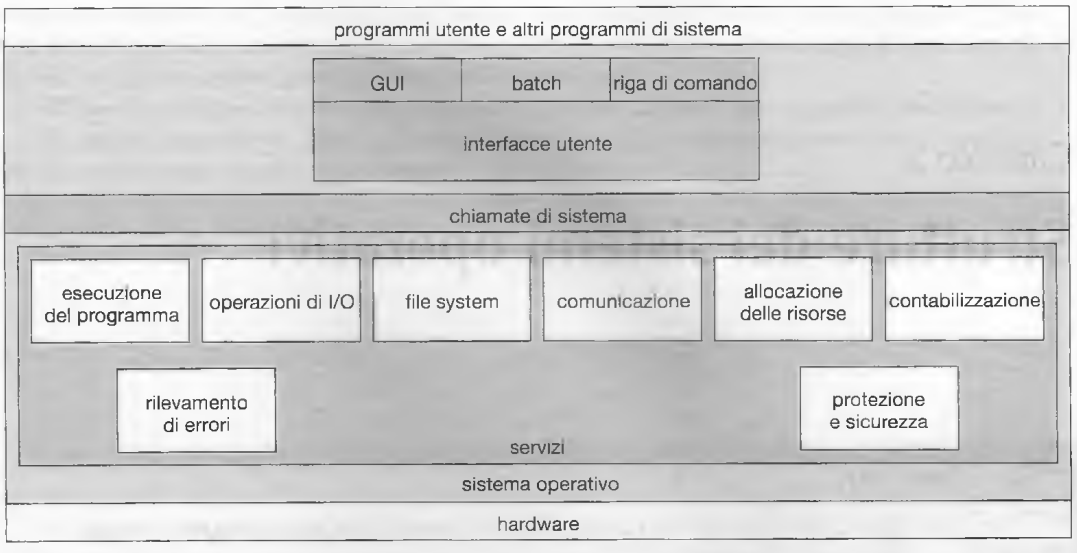

Figura 2.1 Panoramica dei servizi del sistema operativo.

a lotti. La forma senz'altro più diffusa è l'**interfaccia grafica con l'utente** (GUI), ossia un sistema grafico a finestre dotato di un dispositivo puntatore (per esempio, il mouse) per comandare operazioni di I/O e selezionare opzioni dai menu, insieme a una tastiera per inserire del testo. Certi sistemi offrono alcune o anche tutte queste soluzioni.

- **Esecuzione di un programma.** Il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anormale (indicando l'errore).
- **Operazioni di I/O.** Un programma in esecuzione può richiedere un'operazione di I/O che implica l'uso di un file o di un dispositivo di I/O. Per particolari dispositivi possono essere necessarie funzioni speciali, come il riavvolgimento di un'unità a nastro, oppure la cancellazione dello schermo di un tubo a raggi catodici (CRT). Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di I/O, quindi il sistema operativo deve offrire mezzi adeguati.
- **Gestione del file system.** Il file system riveste un interesse particolare. I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file con un certo nome, e disporre di informazioni relative al file stesso. Alcuni programmi, infine, devono poter gestire i permessi di accesso ai file sulla base della proprietà del file interessato. Molti sistemi operativi offrono all'utente la scelta di file system diversi con funzionalità e prestazioni specifiche.
- **Comunicazioni.** In molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una **memoria condivisa** o attraverso lo **scambio di messaggi**, in questo caso il sistema operativo trasferisce pacchetti d'informazioni tra i vari processi.

- **Rilevamento d'errori.** Il sistema operativo deve essere sempre capace di rilevare eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria, quali un errore di memoria o un guasto all'alimentazione elettrica; nei dispositivi di I/O, come un errore di parità in un nastro, il guasto di una connessione di rete, la mancanza di carta nella stampante; in un programma utente, come una divisione per zero, un tentativo d'accesso a una locazione di memoria illegale, un uso eccessivo del tempo di CPU. Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore. Naturalmente sistemi operativi differenti reagiscono agli errori e vi pongono riparo in modi diversi. Eventuali funzionalità di debug – cioè, degli strumenti che permettano l'analisi, per esempio, del software che ha causato un errore – aumentano di molto le possibilità dei programmatori e degli utenti di usare il sistema efficientemente.

Esiste anche un'altra serie di funzioni del sistema operativo che non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso. Sistemi con più utenti possono guadagnare in efficienza condividendo le risorse del calcolatore tra i diversi utenti.

- **Assegnazione delle risorse.** Se sono in corso più sessioni di lavoro di utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di CPU, la memoria centrale e la registrazione in file, possono avere un codice di assegnazione speciale, mentre altre, come i dispositivi di I/O, possono avere un codice di richiesta e di rilascio più generale. Per esempio, per determinare come utilizzare al meglio la CPU, i sistemi operativi impiegano le procedure di scheduling della CPU, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili e di altri fattori. Esistono anche procedure per l'assegnazione di risorse a uso di un processo, quali stampanti, modem, driver di memorizzazione USB e altri dispositivi periferici.
- **Contabilizzazione dell'uso delle risorse.** È possibile registrare quali utenti usino il calcolatore, segnalando quali e quante risorse impieghino. Questo tipo di registrazione si può usare per contabilizzare l'uso delle risorse, per addebitare il costo agli utenti, oppure per redigere statistiche; queste ultime possono essere un valido strumento per i ricercatori che desiderano riconfigurare il sistema per migliorarne i servizi di calcolo.
- **Protezione e sicurezza.** I proprietari di informazioni memorizzate in un sistema di calcolo multiutente o in rete possono voler controllare l'uso di tali informazioni. Più processi non correlati e in esecuzione concorrente non devono influenzarsi o interferire con il sistema operativo. La protezione assicura che l'accesso alle risorse del sistema sia controllato. È importante anche proteggere il sistema dagli estranei. La **sicurezza** di un sistema comincia con l'obbligo d'identificazione da parte di ciascun utente, di solito attraverso parole d'ordine che permettono l'accesso alle risorse; si estende alla 'difesa' dei dispositivi di I/O (compresi i modem e gli adattatori di rete) dai tentativi d'accesso illegali e provvede al loro rilevamento. Se un sistema deve essere protetto e sicuro, al suo interno devono esistere precauzioni ovunque. La forza di una catena è esattamente quella del suo anello più debole.

2.2 Interfaccia con l'utente del sistema operativo

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo. Uno si basa su un'interfaccia a riga di comando o **interprete dei comandi**, e lascia inserire direttamente agli utenti le istruzioni che il sistema deve eseguire. L'altro sfrutta un'interfaccia grafica con l'utente o GUI, che serve da tramite tra utente e sistema.

2.2.1 Interprete dei comandi

Talvolta l'interprete dei comandi è una funzionalità compresa nel kernel dei sistemi operativi. In altri ambienti, come Windows XP e UNIX, l'interprete dei comandi è considerato un programma speciale, che si innesca all'avvio di un processo o allorché un utente si collega per la prima volta (nel caso di sistemi interattivi). Quando i sistemi consentono la scelta tra molteplici interpreti dei comandi, questi vengono definiti **shell**. In UNIX e Linux, per esempio, l'utente può scegliere tra svariate shell differenti, come la *Bourne*, la *C*, la *Bourne-again*, la *Korn*, e così via. Sono anche disponibili shell di terze parti e shell gratuite scritte dagli utenti. Nella maggior parte dei casi, le shell forniscono funzionalità simili e la scelta di un utente è solitamente dovuta alle preferenze personali. La Figura 2.2 illustra la shell Bourne, l'interprete dei comandi utilizzato da Solaris 10.

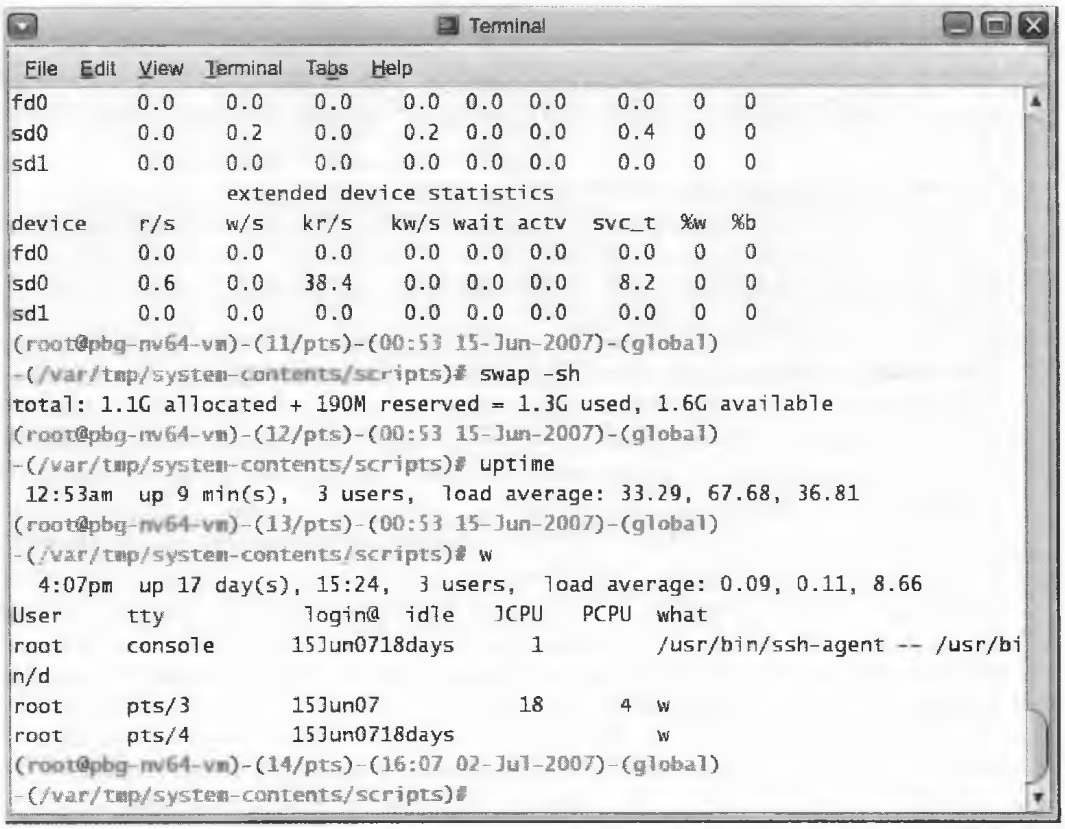

Figura 2.2 Shell Bourne, l'interprete dei comandi utilizzato da Solaris 10.

La funzione principale dell'interprete dei comandi consiste nel prelevare ed eseguire il successivo comando impartito dall'utente. A questo livello, si usano molti comandi per la gestione dei file, vale a dire per creazione, cancellazione, elenchi, stampe, copie, esecuzioni di file, e così via. Le shell dell' MS-DOS e di UNIX funzionano in questo modo.

I comandi si possono implementare in due modi. Nel primo, lo stesso interprete dei comandi contiene il codice per l'esecuzione del comando. Il comando di cancellazione di un file, per esempio, può causare un salto dell'interprete dei comandi a una sezione del suo stesso codice che imposta i parametri e invoca le idonee chiamate di sistema; in questo caso, poiché ogni comando richiede il proprio segmento di codice, il numero dei comandi che si possono impartire determina le dimensioni dell'interprete dei comandi.

L'altro metodo, usato per esempio nel sistema operativo UNIX, implementa la maggior parte dei comandi per mezzo di programmi speciali del sistema; in questo caso l'interprete dei comandi non 'capisce' il significato del comando, ma ne impiega semplicemente il nome per identificare un file da caricare in memoria per l'esecuzione. Quindi, il comando UNIX

```
rm file.txt
```

cerca un file chiamato `rm`, lo carica in memoria e lo esegue con il parametro `file.txt`. La funzione corrispondente al comando `rm` è interamente definita dal codice del file `rm`. In questo modo i programmatori possono aggiungere nuovi comandi al sistema, semplicemente creando nuovi file con il nome appropriato. Il programma dell'interprete dei comandi, che può quindi essere abbastanza piccolo, non necessita di alcuna modifica quando s'introducono nuovi comandi.

2.2.2 Interfaccia grafica con l'utente

Un'interfaccia grafica con l'utente o GUI rappresenta la seconda modalità di comunicazione con il sistema operativo. Si tratta di uno strumento più intuitivo, o, come si dice, *user-friendly*, dell'interfaccia a riga di comando. Infatti, invece di obbligare gli utenti a digitare direttamente i comandi, nella GUI l'interfaccia è costituita da una o più finestre e dai relativi menu, entro cui muoversi con il mouse. La GUI è l'equivalente di una scrivania da lavoro (**desktop**) in cui, spostando il puntatore con il mouse, si indicano le immagini o **icone** sullo schermo (la scrivania): queste rappresentano programmi, file, directory e funzioni del sistema. A seconda della posizione del puntatore, cliccando un pulsante del mouse si può invocare un programma, selezionare un file o una directory – nota in questo contesto come **cartella** – o far apparire un menu a tendina contenente comandi.

Le interfacce grafiche con l'utente si affacciarono sulla scena, da principio, per effetto delle ricerche condotte a Palo Alto nei primi anni '70 dai laboratori di ricerca Xerox PARC. La prima GUI apparve sul computer Xerox Alto nel 1973. Tuttavia, una diffusione più consistente delle interfacce grafiche si ebbe con l'avvento dei computer Apple Macintosh negli anni '80. L'interfaccia utente con il sistema operativo Macintosh (Mac OS) ha subito, nel corso degli anni, diverse modifiche, la più significativa delle quali è stata l'adozione dell'interfaccia *Aqua* per il Mac OS X. La prima versione del Windows di Microsoft, cioè la 1.0, era costruita su di un'interfaccia GUI al sistema operativo MS-DOS. I vari sistemi Windows che si rifacevano a questa versione iniziale hanno apportato ritocchi cosmetici all'aspetto della GUI e una serie di miglioramenti sul piano della funzionalità, compresa l'adozione di Windows Explorer.

Nei sistemi UNIX, tradizionalmente, le interfacce a riga di comando hanno avuto un ruolo preponderante, quantunque vi sia disponibilità di alcune interfacce GUI, come il CDE (*common desktop environment*, ambiente da scrivania comune) e i sistemi X-Windows, che so-

no diffusi fra le versioni commerciali di UNIX quali Solaris e il sistema AIX di IBM. Nella creazione di interfacce, tuttavia, un impulso determinante è giunto da vari progetti **open-source** come il KDE (*K desktop environment*, ambiente da scrivania K) e la scrivania GNOME del progetto GNU. Ambedue le scrivanie, KDE e GNOME, sono compatibili con Linux e con vari sistemi UNIX, e sono regolate da licenza open-source, vale a dire che il loro codice sorgente è reso disponibile per consultazioni e per modifiche soggette a specifiche condizioni di licenza.

La scelta di un'interfaccia GUI piuttosto che di una testuale dipende in buona misura dalle preferenze personali. In linea di massima, gli utenti di UNIX optano per le interfacce a riga di comando, dato che esse offrono shell dotate di caratteristiche potenti. Molti utenti Windows, d'altro canto, sono soddisfatti dell'ambiente Windows GUI, e per questo motivo non usano quasi mai la shell dell'interfaccia MS-DOS. I sistemi operativi Macintosh, con i molti cambiamenti attraversati, costituiscono un utile caso di studio da confrontare alla situazione appena descritta per UNIX e Windows. Fino a tempi recenti, il Mac OS non disponeva di un'interfaccia a riga di comando, e vincolava l'interazione degli utenti con il sistema alla propria interfaccia GUI. Tuttavia, con l'introduzione del Mac OS X (realizzato, in parte, sfruttando il kernel UNIX), il sistema operativo contiene ora sia la nuova interfaccia grafica Aqua sia un'interfaccia testuale a riga di comando. La Figura 2.3 mostra una schermata dell'interfaccia grafica di Mac OS X.

L'interfaccia con l'utente può cambiare da sistema a sistema e persino da utente a utente all'interno dello stesso sistema; in genere è ben distinta dalla struttura portante del sistema. La progettazione di un'interfaccia utile e intuitiva per l'utente non è, pertanto, intrinseca-

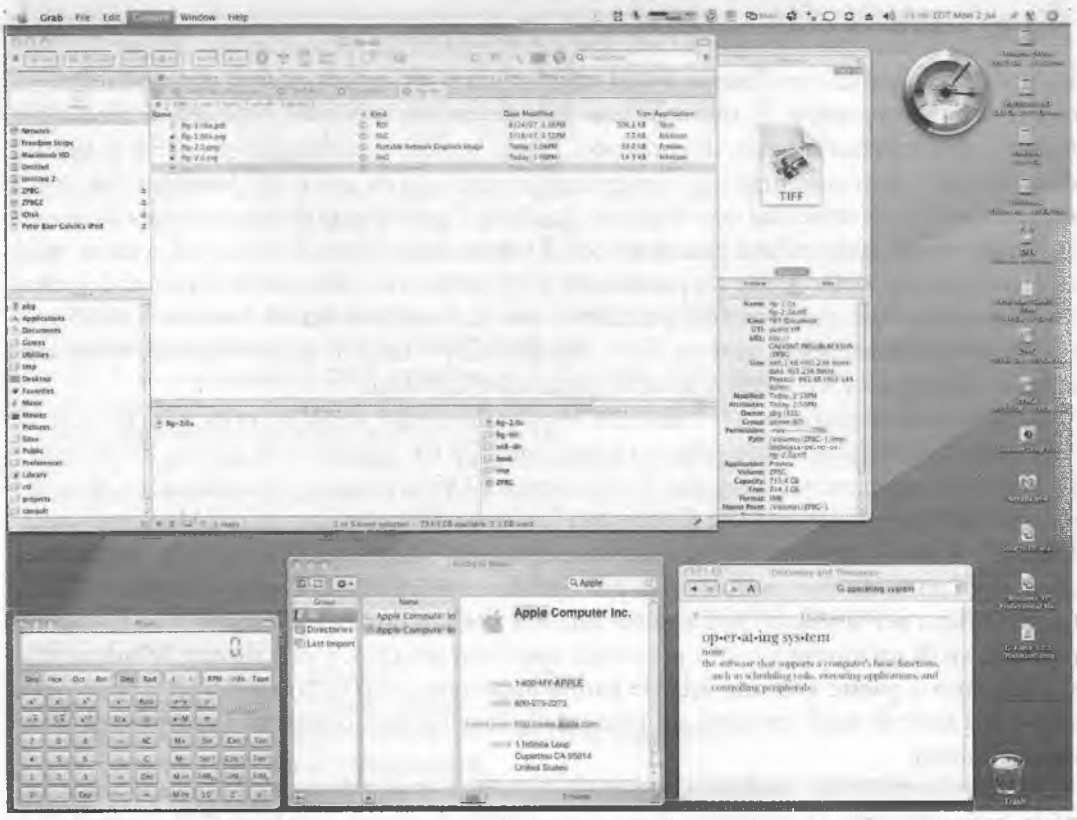

Figura 2.3 Interfaccia grafica di Mac OS X.

mente legata al sistema operativo. In questo libro vengono evidenziati i problemi correlati alla prestazione di un servizio adeguato ai programmi utenti: dal punto di vista del sistema operativo non si applicherà alcuna distinzione tra programmi utenti e programmi del sistema.

2.3 Chiamate di sistema

Le **chiamate di sistema** costituiscono l'interfaccia tra un processo e il sistema operativo. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in C o C++, sebbene per alcuni compiti di basso livello, come quelli che comportano un accesso diretto all'hardware, sarebbe necessario il ricorso a istruzioni in linguaggio assembly.

Prima di illustrare come le chiamate di sistema vengano rese disponibili da parte del sistema operativo, consideriamo come esempio la scrittura di un semplice programma che legga i dati da un file e li trascriva in un altro. La prima informazione di cui il programma necessita è costituita dai nomi dei due file: il file in ingresso e il file in uscita. Questi file si possono indicare in molti modi diversi, secondo la struttura del sistema operativo. Un primo metodo consiste nel richiedere i nomi dei due file all'utente del programma. In un sistema interattivo questa operazione necessita di una sequenza di chiamate di sistema, innanzitutto per scrivere un messaggio di richiesta sullo schermo e quindi per leggere dalla tastiera i caratteri che compongono i nomi dei due file. Nei sistemi basati su mouse e finestre in genere appare in una finestra un menu contenente i nomi dei file. L'utente può usare il mouse per scegliere il nome del file di origine, dopodiché è possibile aprire una finestra simile alla precedente in cui specificare il nome del file di destinazione. Come vedremo, questa sequenza richiede molte chiamate di sistema.

Una volta ottenuti i nomi, il programma deve aprire il file in ingresso e creare il file di destinazione. Ciascuna di queste operazioni richiede un'altra chiamata di sistema e può andare incontro a condizioni d'errore. Per esempio, quando il programma tenta di aprire il file in ingresso, può scoprire che non esiste alcun file con quel nome, oppure che l'accesso al file è negato. In questi casi il programma deve scrivere un messaggio nello schermo della console (altra sequenza di chiamate di sistema) e quindi terminare in maniera anormale la propria elaborazione (ulteriore chiamata di sistema). Se il file in ingresso esiste, è necessario creare il file di destinazione. È possibile che esista già un file col nome indicato per il file di destinazione; questa situazione potrebbe causare l'interruzione del programma (una chiamata di sistema) o la cancellazione del file esistente (un'altra chiamata di sistema) e la creazione di uno nuovo. Un'ulteriore possibilità, in un sistema interattivo, prevede di richiedere all'utente (attraverso una sequenza di chiamate di sistema per emettere il messaggio di richiesta e per leggere la risposta dal terminale) se si debba sostituire il file già esistente o terminare l'esecuzione del programma.

Una volta predisposti i due file, si entra in un ciclo che legge dal file in ingresso (una chiamata di sistema) e scrive nel file di destinazione (altra chiamata di sistema). Ciascuna lettura (`read`) e ogni scrittura (`write`) deve riportare informazioni di stato relative alle possibili condizioni d'errore. Nel file in ingresso il programma può rilevare che è stata raggiunta la fine del file, oppure che nella lettura si è riscontrato un errore del dispositivo, per esempio un errore di parità. Nella fase di scrittura si possono verificare vari errori, la cui natura dipende dal dispositivo impiegato. Esempi tipici sono l'esaurimento dello spazio nei dischi, mancanza della carta in una stampante, e così via.

Infine, una volta copiato tutto il file, il programma può chiuderli entrambi (altre chiamate di sistema), inviare un messaggio alla console (più chiamate di sistema) e infine termi-

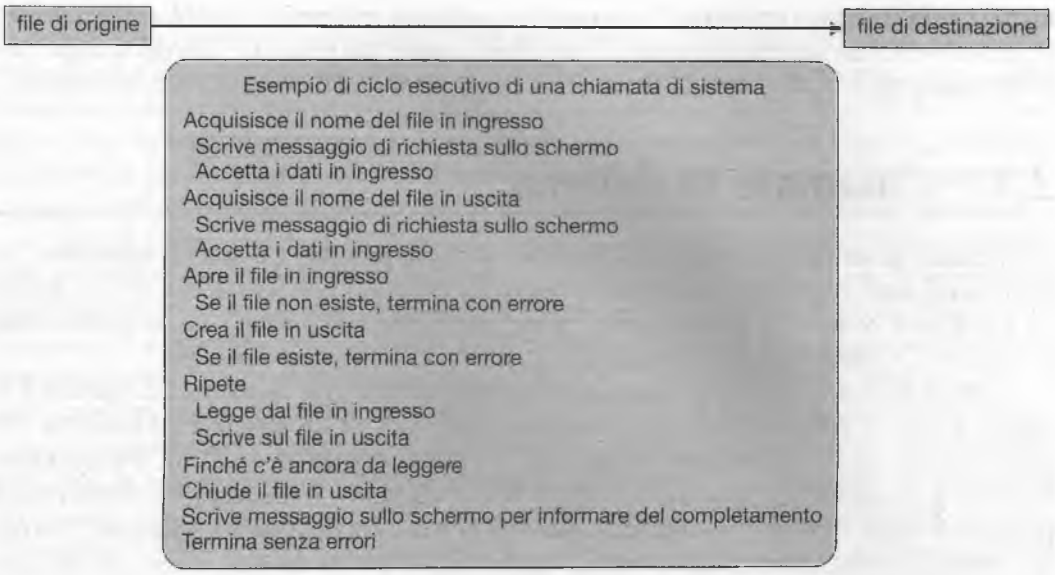

Figura 2.4 Esempio d'uso delle chiamate di sistema.

nare normalmente (ultima chiamata di sistema). Come s'è visto, anche programmi molto semplici possono fare un intenso uso del sistema operativo. Non è raro che un sistema esegua migliaia di chiamate di sistema al secondo. Il ciclo di esecuzione di una chiamata di sistema è illustrato nella Figura 2.4.

La maggior parte dei programmatori, tuttavia, non si dovrà mai preoccupare di questi dettagli: infatti, gli sviluppatori usano in genere un'**interfaccia per la programmazione di applicazioni** (API, *application programming interface*). Essa specifica un insieme di funzioni a disposizione dei programmatori, e dettaglia i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti. Tre delle interfacce più diffuse sono la API Win32 per i sistemi Windows, la API POSIX per i sistemi basati sullo standard POSIX (il che include essenzialmente tutte le versioni di UNIX, Linux e Mac OS X), e la API Java per la progettazione di applicazioni eseguite dalla macchina virtuale Java.

Si noti che (se non diversamente specificato) i nomi delle chiamate di sistema che ricorrono in questo libro sono esempi generici; un dato sistema operativo adotterà nomi suoi propri per funzioni analoghe.

Dietro le quinte, le funzioni fornite da un API invocano solitamente le chiamate di sistema per conto del programmatore. La funzione Win32 `CreateProcess()`, per esempio, che serve a generare un nuovo processo, invoca in effetti `NTCreateProcess()`, una chiamata di sistema del kernel di Windows. Ci sono molte ragioni per cui è preferibile, per un programmatore, sfruttare l'intermediazione della API piuttosto che invocare direttamente le chiamate di sistema. Una di loro è legata alla portabilità delle applicazioni: a grandi linee, un programma sviluppato sulla base di una certa API girerà su qualunque sistema che la metta a disposizione, anche se le differenze architetturali possono rendere la transizione non del tutto indolore. Inoltre, le chiamate di sistema sono spesso più dettagliate e difficili da usare dell'interfaccia API. Bisogna però dire che vi è spesso una stretta correlazione tra le funzioni di una API e le associate chiamate di sistema all'interno del kernel. In effetti, molte funzioni delle API POSIX e WIN32 sono simili alle chiamate di sistema fornite dai sistemi operativi UNIX, Linux e Windows.

TIPICO ESEMPIO DI API

Come esempio tipico di una API, si consideri la funzione `ReadFile()` della API Win32, che serve a leggere da file (Figura 2.2).

Ecco una descrizione dei parametri passati a `ReadFile()`.

- HANDLE file – file da cui leggere.
- LPVOID buffer – buffer da cui leggere i dati provenienti dal file.
- DWORD bytesToRead – numero di byte da trasferire dal file al buffer.
- LPDWORD bytesRead – numero di byte effettivamente letti nell'ultima invocazione.
- LPOVERLAPPED ovl – indica l'uso dello spooling (overlapped I/O).

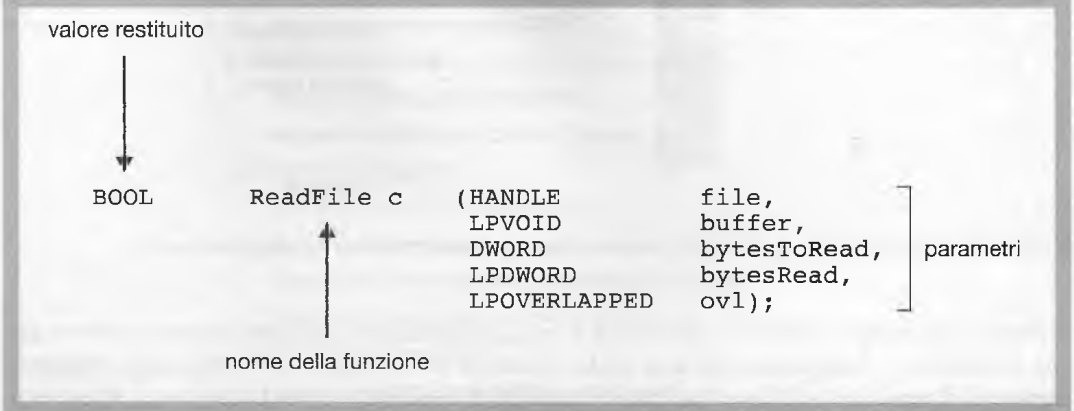

Figura 2.5 API per la funzione `ReadFile()`.

Il cosiddetto sistema di supporto all'esecuzione (*run-time support system*) di un linguaggio di programmazione, ossia l'insieme di funzioni strutturate in librerie incluse nel compilatore, fornisce nella gran parte dei casi un'**interfaccia alle chiamate di sistema** rese disponibili dal sistema operativo, che funge da raccordo tra il linguaggio e il sistema stesso. L'interfaccia intercetta le chiamate di sistema invocate dalla API, e richiede effettivamente la chiamata necessaria. Di solito, ogni chiamata di sistema è codificata da un numero; il compilatore mantiene una tabella delle chiamate di sistema, cui si accede usando questi numeri come indici. L'interfaccia alle chiamate di sistema invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, e passa al chiamante i valori restituiti dalla chiamata di sistema, inclusi quelli di stato.

Il chiamante non ha alcuna necessità di conoscere alcunché sull'implementazione della chiamata di sistema o del suo ciclo esecutivo: gli è sufficiente riconoscere la API e il risultato dell'esecuzione della chiamata di sistema operativo. Ne consegue che la gran parte dei dettagli relativi alle chiamate di sistema sono nascosti al programmatore dalla API, e gestiti dal sistema di supporto all'esecuzione. Le relazioni fra la API, l'interfaccia alle chiamate di sistema e il sistema operativo sono illustrati nella Figura 2.6, ove si mostra come il sistema operativo tratti l'invocazione della chiamata di sistema `open()` da parte di un'applicazione.

Le chiamate di sistema si presentano in modi diversi, secondo il calcolatore in uso. Spesso sono richieste maggiori informazioni oltre alla semplice identità della chiamata di sistema desiderata. Il tipo e l'entità esatti delle informazioni variano secondo lo specifico sistema operativo e la specifica chiamata di sistema. Per ottenere l'immissione di un dato, per

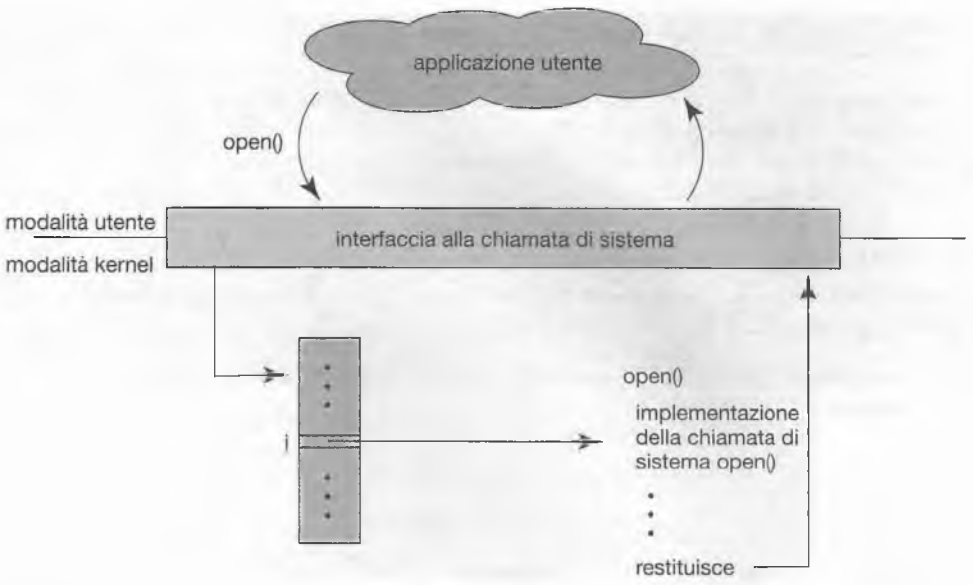

Figura 2.6 Gestione della chiamata di sistema `open()` invocata da un'applicazione utente.

esempio, può essere necessario specificare il file o il dispositivo da usare come sorgente e anche l'indirizzo e l'ampiezza dell'area della memoria in cui depositare i dati letti. Naturalmente, il dispositivo o il file e l'ampiezza possono essere impliciti nella chiamata di sistema.

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in *registri*; si possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un *blocco* o tabella di memoria e si passa l'indirizzo del blocco, in forma di parametro, in un registro (Figura 2.7). È il metodo seguito dal sistema operativo Linux. Il programma può anche collocare (*push*) i parametri in una pila da cui sono prelevati (*pop*) dal sistema operativo. Alcuni sistemi operativi preferiscono i metodi del blocco o della pila, poiché non limitano il numero o la lunghezza dei parametri da passare.

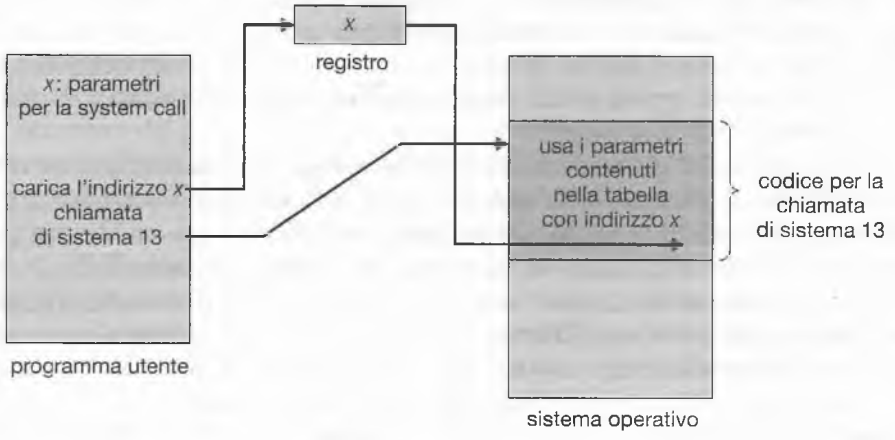

Figura 2.7 Passaggio di parametri in forma di tabella.

- Controllo dei processi
  - terminazione normale e anormale
  - caricamento, esecuzione
  - creazione e arresto di un processo
  - esame e impostazione degli attributi di un processo
  - attesa per il tempo indicato
  - attesa e segnalazione di un evento
  - assegnazione e rilascio di memoria
- Gestione dei file
  - creazione e cancellazione di file
  - apertura, chiusura
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
  - richiesta e rilascio di un dispositivo
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un dispositivo
  - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
  - esame e impostazione dell'ora e della data
  - esame e impostazione dei dati del sistema
  - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
  - creazione e chiusura di una connessione
  - invio e ricezione di messaggi
  - informazioni sullo stato di un trasferimento
  - inserimento ed esclusione di dispositivi remoti

Figura 2.8 Tipi di chiamate di sistema.

2.4 Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in cinque categorie principali: **controllo dei processi**, **gestione dei file**, **gestione dei dispositivi**, **gestione delle informazioni** e **comunicazioni**. Nei Paragrafi dal 2.4.1 al 2.4.6 sono illustrati brevemente i tipi di chiamate di sistema forniti da un sistema operativo. La maggior parte di queste chiamate di sistema implica o presuppone concetti e funzioni trattati in capitoli successivi. La Figura 2.8 riassume i tipi di chiamate di sistema forniti normalmente da un sistema operativo.

ESEMPIO DI CHIAMATE DI SISTEMA DI WINDOWS E UNIX

	Windows	UNIX
Controllo dei processi	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
Gestione dei file	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Gestione dei dispositivi	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Gestione delle informazioni	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Comunicazione	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shmget()`
	`MapViewOfFile()`	`mmap()`
Protezione	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

2.4.1 Controllo dei processi

Un programma in esecuzione deve potersi fermare in modo sia normale (in tal caso si parla di `end`) sia anormale (`abort`). Se si ricorre a una chiamata di sistema per terminare in modo anormale un programma in esecuzione, oppure se il programma incontra difficoltà e causa l'emissione di un segnale di eccezione, talvolta si ha la registrazione in un file di un'immagine del contenuto della memoria (*dump*) e l'emissione di un messaggio d'errore. Uno specifico programma di ricerca e correzione degli errori (*debugger*) può esaminare tali informazioni per determinare le cause del problema. Sia in condizioni normali sia anormali il sistema operativo deve trasferire il controllo all'interprete dei comandi che legge il comando successivo. In un sistema interattivo l'interprete dei comandi continua semplicemente a interpretare il comando successivo; si suppone che l'utente invii un comando idoneo per rispondere a qualsiasi errore. In un sistema a interfaccia GUI una finestra avverte l'utente dell'errore e richiede chiarimenti. In un sistema a lotti l'interprete dei comandi generalmente termina il lavoro corrente e prosegue con il successivo. Quando si presenta un errore, alcuni sistemi permettono alle schede di controllo di indicare le specifiche azioni di recupero da intraprendere. La **scheda di controllo** è un concetto proveniente dai sistemi a lotti: è in sostanza un comando per gestire l'esecuzione di un processo. Se il programma scopre un errore nei dati ricevuti e intende terminare in modo anormale, può anche definire un livello d'errore. Più grave è l'errore, più alto è il livello del parametro che lo individua. Sono quindi possibili una terminazione normale e una terminazione anormale, definendo la termina-

zione normale come errore di livello 0. L'interprete dei comandi o un programma successivo possono usare questo livello d'errore per determinare l'azione da intraprendere.

Un processo che esegue un programma può richiedere di caricare (`load`) ed eseguire (`execute`) un altro programma. In questo caso l'interprete dei comandi esegue un programma come se tale richiesta fosse stata impartita, per esempio, da un comando utente, oppure dal clic di un mouse. È interessante chiedersi dove si debba restituire il controllo una volta terminato il programma caricato. La questione è legata alle eventualità che il programma attuale sia andato perso, salvato oppure che abbia continuato l'esecuzione in modo concorrente con il nuovo programma.

Se al termine del nuovo programma il controllo rientra nel programma esistente, si deve salvare l'immagine della memoria del programma attuale, creando così effettivamente un meccanismo con cui un programma può richiamare un altro programma. Se entrambi i programmi continuano l'esecuzione in modo concorrente, si è creato un nuovo processo da sottoporre a multiprogrammazione. A questo scopo spesso si fornisce una chiamata di sistema specifica, e precisamente `create process` oppure `submit job`).

Quando si crea un nuovo processo, o anche un insieme di processi, è necessario mantenerne il controllo; ciò richiede la capacità di determinare e reimpostare gli attributi di un pro-

ESEMPIO DI LIBRERIA STANDARD DEL LINGUAGGIO C

La libreria standard del linguaggio C fornisce una parte dell'interfaccia alle chiamate di sistema per molte versioni di UNIX e Linux. Come esempio, supponiamo che un programma C invochi la funzione `printf()`. La libreria C intercetta la funzione e invoca le necessarie chiamate di sistema: in questo caso, la chiamata `write()`. La libreria riceve il valore restituito da `write()` e lo passa al programma utente. Il meccanismo è illustrato nella Figura 2.6.

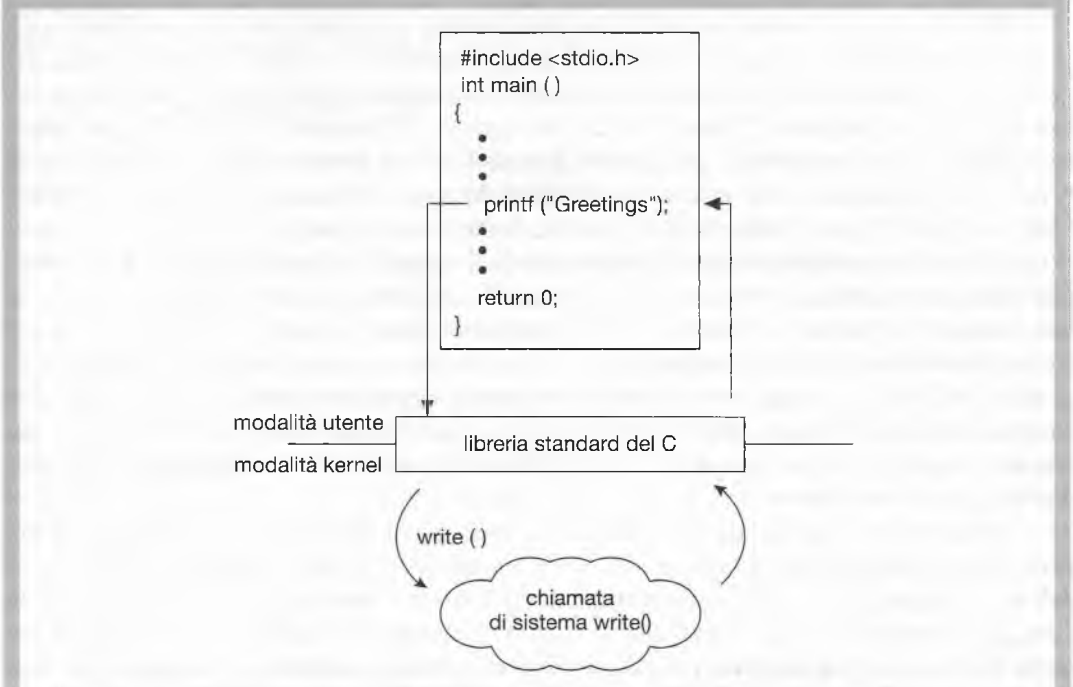

Figura 2.9 Gestione di `write()` della libreria standard del C.

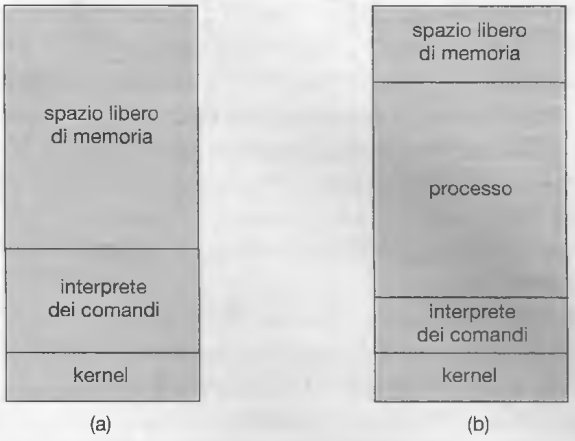

Figura 2.10 Esecuzione nell'MS-DOS. (a) All'avviamento del sistema. (b) Durante l'esecuzione di un programma.

cesso, compresi la sua priorità, il suo tempo massimo d'esecuzione e così via (`get process attributes` e `set process attributes`). Inoltre, può essere necessario terminare un processo creato, se si riscontra che non è corretto o se non serve (`terminate process`).

Una volta creati, può essere necessario attendere che i processi terminino la loro esecuzione. Quest'attesa si può impostare per un certo periodo di tempo (`wait` *`tempo`*), ma è più probabile che si preferisca attendere che si verifichi un dato evento (`wait` *`evento`*). I processi devono quindi segnalare il verificarsi di quell'evento (`signal` *`evento`*). Chiamate di sistema di questo tipo, che trattano cioè il coordinamento di processi concorrenti, sono esaminate in profondità nel Capitolo 6.

Il controllo dei processi presenta così tanti aspetti e varianti: per chiarire questi concetti conviene ricorrere ad alcuni esempi. Il sistema operativo MS-DOS è un sistema che dispone di un interprete di comandi, attivato all'avviamento del calcolatore, e che esegue un solo programma alla volta (Figura 2.10.a), impiegando un metodo semplice e senza creare alcun nuovo processo; carica il programma in memoria, riscrivendo anche la maggior parte della memoria che esso stesso occupa, in modo da lasciare al programma quanta più memoria è possibile (Figura 2.10.b); quindi imposta il contatore di programma alla prima istruzione del programma da eseguire. A questo punto si esegue il programma e si possono verificare due situazioni: un errore causa un segnale di eccezione, oppure il programma esegue una chiamata di sistema per terminare la propria esecuzione. In entrambi i casi si registra il codice d'errore in memoria di sistema per un eventuale uso successivo, quindi quella piccola parte dell'interprete che non era stata sovrascritta riprende l'esecuzione e il suo primo compito consiste nel ricaricare dal disco la parte rimanente dell'interprete stesso. Eseguito questo compito, quest'ultimo mette a disposizione dell'utente, o del programma successivo, il codice d'errore registrato.

Nel FreeBSD (derivato da UNIX Berkeley), quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (in quest'ambiente chiamato *shell*) scelto dall'utente. Quest'interprete è simile a quello dell'MS-DOS nell'accettare i comandi e nell'eseguire programmi richiesti dall'utente. Tuttavia, poiché il FreeBSD è un sistema a partizione del tempo, può eseguire più programmi contemporaneamente, l'interprete dei comandi può continuare l'esecuzione mentre si esegue un altro programma (Figura 2.11).

Per avviare un nuovo processo, l'interprete dei comandi esegue la chiamata di sistema `fork()`; si carica il programma selezionato in memoria tramite la chiamata di sistema

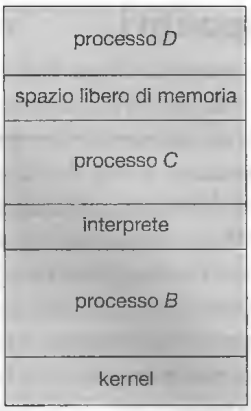

Figura 2.11 Esecuzione di più programmi nel sistema operativo FreeBSD.

`exec()` e infine si esegue il programma. Secondo come il comando è stato impartito, l'interprete dei comandi attende il termine del processo, oppure esegue il processo *background*. In quest'ultimo caso l'interprete dei comandi richiede immediatamente un altro comando. Se un processo è eseguito in background, non può ricevere dati direttamente dalla tastiera, giacché anche l'interprete dei comandi sta usando tale risorsa. L'eventuale operazione di I/O è dunque eseguita tramite un file o tramite un'interfaccia GUI. Nel frattempo l'utente è libero di richiedere all'interprete dei comandi l'esecuzione di altri, di controllare lo svolgimento del processo in esecuzione, di modificare la priorità di quel programma e così via. Completato il proprio compito, il processo esegue una chiamata di sistema `exit()` per terminare la propria esecuzione, riportando al processo chiamante un codice di stato 0 oppure un codice d'errore diverso da 0. Questo codice di stato (o d'errore) rimane disponibile per l'interprete dei comandi o per altri programmi. I processi sono trattati nel Capitolo 3, dove si illustra un esempio di programma che utilizza le chiamate di sistema `fork()` ed `exec()`.

2.4.2 Gestione dei file

Il file system è esaminato più in profondità nei Capitoli 10 e 11, tuttavia possiamo già identificare diverse chiamate di sistema riguardanti i file.

Innanzitutto è necessario poter creare (`create`) e cancellare (`delete`) i file. Ogni chiamata di sistema richiede il nome del file e probabilmente anche altri attributi. Una volta creato il file è necessario aprirlo (`open`) e usarlo. Si può anche leggere (`read`), scrivere (`write`) o riposizionare (`reposition`), per esempio riavvolgendo e saltando alla fine del file. Si deve infine poter chiudere (`close`) un file per indicare che non è più in uso.

Queste stesse operazioni possono essere necessarie anche per le directory, nel caso in cui il file system sia strutturato in directory. È inoltre necessario poter determinare i valori degli attributi dei file o delle directory ed eventualmente modificarli. Tra gli attributi dei file figurano: nome, tipo, codici di protezione, informazioni di contabilizzazione, e così via. Per questa funzione sono richieste almeno due chiamate di sistema, e precisamente `get file attribute` e `set file attribute`. Alcuni sistemi operativi forniscono molte più chiamate di sistema per spostare (`move`) e copiare (`copy`) file, per esempio. Altri forniscono API che eseguono operazioni di questo tipo tramite codice appropriato combinato e chiamate di sistema, e altri ancora mettono semplicemente a disposizione programmi di sistema che le eseguono. Se i programmi di sistema sono invocabili da altri programmi, ognuno di loro funge da API per altri programmi.

2.4.3 Gestione dei dispositivi

Per essere eseguito un programma necessita di parecchie risorse: spazio in memoria, driver, unità a nastro, accesso a file, e così via. Se le risorse sono disponibili, si possono concedere, e il controllo può ritornare al processo utente, altrimenti il processo deve attendere finché non siano disponibili risorse sufficienti.

Le diverse risorse controllate dal sistema operativo si possono concepire come dei dispositivi, alcuni dei quali sono in effetti dispositivi fisici (per esempio nastri), mentre altre sono da considerarsi dispositivi astratti o virtuali (i file, per esempio). In presenza di utenti multipli, il sistema potrebbe prescrivere la richiesta (tramite `request`) del dispositivo, al fine di assicurarne l'uso esclusivo. Dopo l'uso, avviene il rilascio (tramite `release`). Si tratta di funzioni analoghe alle chiamate `open` e `close` per i file. Altri sistemi operativi permettono l'accesso incontrollato ai dispositivi, con il rischio che la competizione per l'accesso provochi lo stallo (Capitolo 7).

Una volta richiesto e assegnato il dispositivo, è possibile leggervi (`read`), scrivervi (`write`) ed eventualmente procedere a un riposizionamento (`reposition`), esattamente come nei file ordinari; la somiglianza tra file e dispositivi di I/O è infatti tale che molti sistemi operativi, tra cui UNIX, li combinano in un'unica struttura file-dispositivi. A volte, i dispositivi di I/O sono identificati da nomi particolari, attributi speciali, o dal collocamento in certe directory.

L'interfaccia con l'utente può anche far apparire simili i file e i dispositivi, nonostante le chiamate di sistema sottostanti non lo siano: è un altro esempio di una delle scelte che il progettista deve intraprendere nella costruzione del sistema operativo e relativa interfaccia utente.

2.4.4 Gestione delle informazioni

Molte chiamate di sistema hanno semplicemente lo scopo di trasferire le informazioni tra il programma utente e il sistema operativo. La maggior parte dei sistemi, per esempio, ha una chiamata di sistema per ottenere l'ora (`time`) e la data attuali (`date`). Altre chiamate di sistema possono ottenere informazioni sul sistema, come il numero degli utenti collegati, il numero della versione del sistema operativo, la quantità di memoria disponibile o di spazio nei dischi, e così via.

Un altro insieme di chiamate di sistema è utile per il debugging di programmi. Molti sistemi operativi forniscono chiamate di sistema per ottenere un'immagine della memoria (effettuare il `dump`). Questa funzionalità è utile per il debugging. Un programma di tracciamento (`trace`) fornisce un elenco delle chiamate di sistema in esecuzione. Anche i microprocessori offrono una modalità conosciuta come *a singolo passo* (*single step*) nella quale viene eseguita una *trap* dopo ogni istruzione. La *trap* viene solitamente catturata dal debugger.

Molti sistemi operativi possono effettuare un'analisi del tempo utilizzato da un programma e indicare la quantità di tempo in cui il programma rimane in esecuzione in una particolare locazione o in un insieme di locazioni. Un'analisi del tempo richiede un'utilità di tracciamento o delle interruzioni regolari da parte del timer. A ogni occorrenza di un'interruzione del timer il valore del contatore di programma viene memorizzato. Con una frequenza di interruzioni sufficientemente alta è possibile ottenere una statistica del tempo trascorso nelle varie parti di un programma.

Il sistema operativo contiene inoltre informazioni su tutti i propri processi; a queste informazioni si può accedere tramite alcune chiamate di sistema. In genere esistono anche chiamate di sistema per modificare le informazioni sui processi (`get process attributes` e `set process attributes`). Nel Paragrafo 3.1.3 si spiega quali sono tali informazioni.

2.4.5 Comunicazione

Esistono due modelli molto diffusi di comunicazione tra processi: il modello a scambio di messaggi e quello a memoria condivisa. Nel modello **a scambio di messaggi** i processi comunicanti si scambiano messaggi per il trasferimento delle informazioni sia direttamente sia indirettamente attraverso una casella di posta comune. Prima di effettuare una comunicazione occorre aprire un collegamento. Il nome dell'altro comunicante deve essere noto, sia che si tratti di un altro processo nello stesso calcolatore, sia di un processo in un altro calcolatore collegato attraverso una rete di comunicazione. Tutti i calcolatori di una rete hanno un **nome di macchina** (*host name*), per esempio un nome IP, con il quale sono individuati. Analogamente, ogni processo ha un **nome di processo**, che si converte in un identificatore equivalente che il sistema operativo impiega per farvi riferimento. La conversione nell'identificatore si compie con le chiamate di sistema `get hostid` e `get processid`. Questi identificatori sono quindi passati alle chiamate di sistema d'uso generale `open` e `close` messe a disposizione dal file system, oppure, secondo il modello di comunicazione del sistema, alle chiamate di sistema specifiche `open connection` e `close connection`. Generalmente il processo ricevente deve acconsentire alla comunicazione con una chiamata di sistema `accept connection`. Nella maggior parte dei casi i processi che gestiscono la comunicazione sono **demoni** specifici, cioè programmi di sistema realizzati esplicitamente per questo scopo. Questi programmi eseguono una chiamata di sistema `wait for connection` e sono chiamati in causa quando si stabilisce un collegamento. L'origine della comunicazione, nota come *client*, e il demone ricevente, noto come *server*, possono quindi scambiarsi i messaggi per mezzo delle chiamate di sistema `read message` e `write message`; la chiamata di sistema `close connection` pone fine alla comunicazione.

Nel modello a **memoria condivisa**, invece, i processi usano chiamate di sistema `shared memory create` e `shared memory attach` per creare e accedere alle aree di memoria possedute da altri processi. Occorre ricordare che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo. Il modello a memoria condivisa richiede che più processi concordino nel superare tale limite; a questo punto tali processi possono scambiarsi le informazioni leggendo e scrivendo i dati nelle aree di memoria condivise. La forma e la posizione dei dati sono determinate esclusivamente da questi processi e non sono sotto il controllo del sistema operativo. I processi sono dunque responsabili del rispetto della condizione di non scrivere contemporaneamente nella stessa posizione. Questi meccanismi sono discussi nel Capitolo 6. In questo testo, precisamente nel Capitolo 4, si illustra anche una variante del modello di processo, detto *thread*, che prevede che la condivisione della memoria sia predefinita.

Entrambi i metodi sono assai comuni e in certi sistemi operativi sono presenti contemporaneamente. Lo scambio di messaggi è utile soprattutto quando è necessario trasferire una piccola quantità di dati, poiché, in questo caso, non sussiste la necessità di evitare conflitti; è inoltre più facile da realizzare rispetto alla condivisione della memoria per la comunicazione tra calcolatori diversi. La condivisione della memoria permette la massima velocità e convenienza nelle comunicazioni, poiché queste ultime, se avvengono all'interno del calcolatore, si possono svolgere alla velocità della memoria. Sussistono, in ogni caso, problemi per quel che riguarda la protezione e la sincronizzazione tra processi che condividono la memoria.

2.4.6 Protezione

La protezione fornisce un meccanismo per controllore l'accesso alle risorse di un calcolatore. Storicamente ci si preoccupava della protezione solo su calcolatori multiprogrammati e con numerosi utenti. Ora, con l'avvento delle reti e di Internet, tutti i calcolatori, dai server ai dispositivi palmari, tengono conto della protezione.

Tra le chiamate di sistema che offrono meccanismi di protezione vi sono solitamente la `set permission` e la `get permission`, che permettono di modificare i permessi di accesso a risorse come file e dischi. Le chiamate di sistema `allow user` e `deny user` specificano se un particolare utente abbia il permesso di accesso a determinate risorse.

La protezione è trattata nel Capitolo 14. Il Capitolo 15 esamina la sicurezza in maniera più estesa.

2.5 Programmi di sistema

Un'altra caratteristica importante di un sistema moderno è quella che riguarda la serie di programmi di sistema. Facendo riferimento alla Figura 1.1, in cui s'illustra la gerarchia logica di un calcolatore, si può notare che il livello più basso è occupato dai dispositivi fisici. Seguono, nell'ordine, il sistema operativo, i programmi di sistema e i programmi applicativi. I **programmi di sistema**, conosciuti anche come **utilità di sistema**, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le chiamate di sistema, altri sono considerevolmente più complessi; in generale si possono classificare nelle seguenti categorie.

- **Gestione dei file.** Questi programmi creano, cancellano, copiano, ridenominano, stampano, elencano e in genere compiono operazioni sui file e le directory.
- **Informazioni di stato.** Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti o informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni tramite terminale, o tramite altri dispositivi per l'uscita dei dati, o, ancora, all'interno di una finestra della GUI. Alcuni sistemi comprendono anche **registri** (*registry*), al fine di archiviare e poter poi consultare informazioni sulla configurazione del sistema.
- **Modifica dei file.** Diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.
- **Ambienti d'ausilio alla programmazione.** Compilatori, assemblatori, programmi per la correzione degli errori e interpreti dei comuni linguaggi di programmazione, come C, C++, Java, Visual Basic e PERL, sono spesso forniti insieme con il sistema operativo.
- **Caricamento ed esecuzione dei programmi.** Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*). Sono necessari anche i sistemi d'ausilio all'individuazione e correzione degli errori (*debugger*) per i linguaggi d'alto livello o per il linguaggio macchina.
- **Comunicazioni.** Questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi. Permettono agli utenti d'invia-

re messaggi agli schermi d'altri utenti, di consultare il Web, d'inviare messaggi di posta elettronica, di accedere a calcolatori remoti, di trasferire file da un calcolatore a un altro.

Oltre ai programmi di sistema, con la maggior parte dei sistemi operativi sono forniti programmi che risolvono problemi comuni o che eseguono operazioni comuni, quali quelli di consultazione del Web, elaboratori di testi, fogli di calcolo, sistemi di basi di dati, compilatori, programmi di disegno, programmi per analisi statistiche e videogiochi.

L'immagine che gli utenti si fanno di un sistema è influenzata principalmente dalle applicazioni e dai programmi di sistema, più che dalle chiamate di sistema. Quando un utente di Mac OS X usa la GUI del sistema, si trova di fronte un insieme di finestre e il puntatore del mouse; quando, invece, usa la riga di comando, si trova di fronte a una shell in stile UNIX, magari in una delle finestre della GUI. In entrambi i casi, l'insieme di chiamate di sistema sottostanti è lo stesso, ma l'aspetto e il modo d'operare del sistema sono ben diversi. Come esempio dell'ulteriore confusione che si può generare, si consideri un sistema in cui viene effettuato un dual boot da Mac OS X a Windows Vista. In questo caso lo stesso utente sulla stessa macchina ha due differenti interfacce e due differenti insiemi di applicazioni che usano le stesse risorse fisiche. Con lo stesso hardware un utente può quindi utilizzare diverse interfacce, sequenzialmente o in modo concorrente.

2.6 Progettazione e realizzazione di un sistema operativo

Nei seguenti paragrafi si trattano i problemi riguardanti la progettazione e la realizzazione di un sistema. Naturalmente non si dispone di soluzioni complete, ma alcuni metodi si sono dimostrati efficaci.

2.6.1 Scopi della progettazione

Il primo problema che s'incontra nella progettazione di un sistema riguarda la definizione degli scopi e delle caratteristiche del sistema stesso. Al più alto livello, la progettazione del sistema è influenzata in modo decisivo dalla scelta dell'architettura fisica e del tipo di sistema: a lotti o a partizione del tempo, mono o multiutente, distribuito, per elaborazioni in tempo reale o d'uso generale.

D'altra parte, oltre questo livello di progettazione, i requisiti possono essere molto difficili da specificare; anche se in generale si possono distinguere in due gruppi fondamentali: obiettivi degli *utenti* e obiettivi del *sistema*.

Gli utenti desiderano che un sistema abbia alcune caratteristiche ovvie: deve essere utile, facile da imparare e usare, affidabile, sicuro e veloce; queste caratteristiche non sono particolarmente utili nella progettazione di un sistema, poiché non tutti concordano sui metodi da applicare per raggiungere questi scopi.

Requisiti analoghi sono richiesti da chi deve progettare, creare e operare con il sistema: il sistema operativo deve essere di facile progettazione, realizzazione e manutenzione; deve essere flessibile, affidabile, senza errori ed efficiente. Anche in questo caso si tratta di requisiti vaghi, interpretabili in vari modi.

Non esiste una soluzione unica al problema della definizione dei requisiti di un sistema operativo. L'ampia gamma di sistemi mostra che da requisiti diversi possono risultare le soluzioni più varie per ambienti diversi. Per esempio, i requisiti VxWorks, un sistema operativo real-time per sistemi integrati, sono assai diversi da MVS, il sistema operativo multiaccesso e multiutente per mainframe IBM.

2.6.2 Meccanismi e criteri

Un principio molto importante è quello che riguarda la distinzione tra **meccanismi** e **criteri** o **politiche** (*policy*). I meccanismi determinano *come* eseguire qualcosa; i criteri, invece, stabiliscono *che cosa* si debba fare. Il timer del sistema (Paragrafo 1.5.2), per esempio, è un meccanismo che assicura la protezione della CPU, ma la decisione riguardante la quantità di tempo da impostare nel timer per un utente specifico riguarda i criteri.

La distinzione tra meccanismi e criteri è molto importante ai fini della flessibilità. I criteri sono soggetti a cambiamenti di luogo o di tempo. Nei casi peggiori il cambiamento di un criterio può richiedere il cambiamento del meccanismo sottostante. Sarebbe preferibile disporre di meccanismi generali: in questo caso un cambiamento di criterio implicherebbe solo la ridefinizione di alcuni parametri del sistema. Per esempio, consideriamo un meccanismo che assegni priorità a certe categorie di programmi rispetto ad altre. Se tale meccanismo è debitamente separato dal criterio con cui si procede, esso è utilizzabile per far sì che programmi che impiegano intensamente operazioni di I/O abbiano priorità su quelli che impiegano intensamente la CPU, oppure viceversa.

I sistemi operativi basati su microkernel (Paragrafo 2.7.3) portano alle estreme conseguenze la separazione dei meccanismi dai criteri, fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi indipendenti dai criteri, consentono l'aggiunta di meccanismi e criteri più complessi tramite moduli del kernel creati dagli utenti o anche tramite programmi utenti. Per un esempio, si consideri l'evoluzione di UNIX. In principio, si trattava di un sistema basato sulla ripartizione del tempo. Nelle ultime versioni di Solaris, lo scheduling è controllato da tabelle caricabili: a seconda della tabella corrente, il sistema può essere a tempo ripartito, a lotti, in tempo reale, fair share, o adottare una combinazione delle strategie precedenti. Tale parametrizzazione dei meccanismi di scheduling permette di attuare cambiamenti di vasta portata ai criteri del sistema con l'esecuzione di un singolo comando di caricamento (`load-new-table`) di una nuova tabella. All'altro estremo si trovano sistemi come Windows, nei quali sia i criteri sia i meccanismi sono fissati a priori e cablati nel sistema, al fine di fornire agli utenti un'unica immagine globale. In questi casi, infatti, tutte le applicazioni hanno interfacce simili, perché l'interfaccia stessa fa parte del kernel e delle librerie del sistema. Il sistema Mac OS X è di tipo analogo.

Le decisioni relative ai criteri sono importanti per tutti i problemi di assegnazione delle risorse e di scheduling. Invece, ogni volta che un problema riguarda il *come* piuttosto che il *che cosa* occorre definire un meccanismo.

2.6.3 Realizzazione

Una volta progettato, un sistema operativo va realizzato. Tradizionalmente i sistemi operativi si scrivevano in un linguaggio assembly, attualmente si scrivono spesso in linguaggi di alto livello come il C o il C++.

Il primo sistema scritto in un linguaggio di alto livello fu probabilmente il Master Control Program (MCP) per i calcolatori Burroughs; l'MCP fu scritto infatti in una variante del linguaggio ALGOL; il MULTICS, sviluppato al MIT, fu scritto prevalentemente in PL/1; Linux e Windows XP sono scritti per lo più in C, sebbene si trovino alcune brevi sezioni di codice assembly che riguardano i driver dei dispositivi e il salvataggio e il ripristino dei registri del sistema.

I vantaggi derivanti dall'uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, sono gli stessi che si ottengono quando il linguaggio si usa per i programmi applicativi: il codice si scrive più rapidamente, è più compatto ed è più facile da capire e mettere a punto. Inoltre il perfezionamento delle tecniche di compilazione consente di migliorare il codice generato per l'intero sistema ope-

rativo con una semplice ricompilazione. Infine, un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un'altra architettura (*porting*). L'MS-DOS, per esempio, fu scritto nel linguaggio assembly dell'Intel 8088, quindi è disponibile solo per la famiglia di CPU Intel. Il sistema operativo Linux, scritto prevalentemente in C, è invece disponibile su diversi tipi di CPU, tra cui Intel 80*x*86, Motorola 680*x*0, SPARC e MIPS RX000.

I soli eventuali svantaggi che possono presentarsi nella realizzazione di un sistema operativo in un linguaggio di alto livello sono una minore velocità d'esecuzione e una maggiore occupazione di spazio di memoria, una questione ormai superata nei sistemi moderni. D'altra parte, benché un programmatore esperto di un linguaggio assembly possa produrre piccole procedure di grande efficienza, per quel che riguarda i programmi molto estesi, un moderno compilatore può eseguire complesse analisi e applicare raffinate ottimizzazioni che producono un codice eccellente. Le moderne CPU sono organizzate in numerose fasi d'elaborazione concatenate (*pipelining*) e parecchie unità, le cui complesse interdipendenze possono sopraffare la limitata capacità della mente umana di tener traccia dei dettagli.

Come in altri sistemi, i miglioramenti principali nel rendimento sono dovuti più a strutture dati e algoritmi migliori che a un ottimo codice in linguaggio assembly. Inoltre, sebbene i sistemi operativi siano molto grandi, solo una piccola parte del codice assume un'importanza critica riguardo al rendimento: il gestore della memoria e lo scheduler della CPU sono probabilmente le procedure più critiche. Una volta scritto il sistema e verificato il suo corretto funzionamento, è possibile identificare le procedure che possono costituire colli di bottiglia e sostituirle con procedure equivalenti scritte in linguaggio assembly.

2.7 Struttura del sistema operativo

Affinché possa funzionare correttamente ed essere facilmente modificato, un sistema vasto e complesso come un sistema operativo moderno va progettato con estrema attenzione. Anziché progettare un sistema monolitico, un orientamento diffuso prevede la sua suddivisione in piccoli componenti; ciascuno deve essere un modulo ben definito del sistema, con interfacce e funzioni definite con precisione. Nel Capitolo 1 sono stati brevemente illustrati i componenti comuni ai sistemi operativi; nel paragrafo seguente si descrive come questi componenti siano interconnessi e fusi in un kernel.

2.7.1 Struttura semplice

Molti sistemi commerciali non hanno una struttura ben definita; spesso sono nati come sistemi piccoli, semplici e limitati, e solo in un secondo tempo si sono accresciuti superando il loro scopo originale. Un sistema di questo tipo è l'MS-DOS, originariamente progettato e realizzato da persone che non avrebbero mai immaginato una simile diffusione. Non fu suddiviso attentamente in moduli poiché, a causa dei limiti dell'architettura su cui era eseguito, lo scopo prioritario era fornire la massima funzionalità nel minimo spazio. La Figura 2.12 riporta la sua struttura.

In MS-DOS non vi è una netta separazione fra le interfacce e i livelli di funzionalità, tanto che, per esempio, le applicazioni accedono direttamente alle routine di sistema per l'I/O, scrivendo direttamente sul video e sui dischi. Libertà di questo genere rendono MS-DOS vulnerabile agli errori e agli attacchi dei programmi utenti, fino al blocco totale del sistema. Naturalmente, le limitazioni di MS-DOS riflettono quelle dell'hardware disponibile ai tempi della sua progettazione. Il processore Intel 8088, per il quale fu scritto, non distingueva fra modalità utente e di sistema, e non offriva protezione hardware, ciò che non lasciava altra scelta ai progettisti di MS-DOS se non permettere accesso incondizionato all'hardware sottostante.

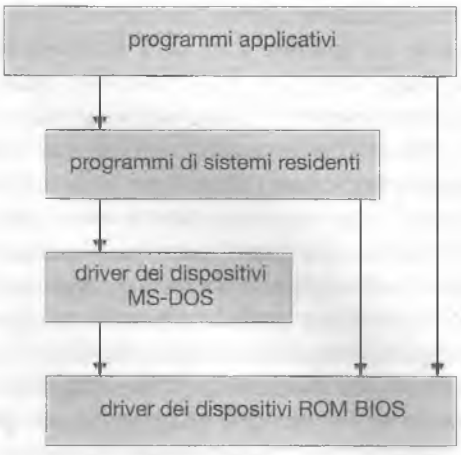

Figura 2.12 Struttura degli strati dell'MS-DOS.

Anche il sistema UNIX originale è poco strutturato, a causa delle limitazioni dell'hardware disponibile al tempo della sua progettazione. Il sistema consiste di due parti separate, il kernel e i programmi di sistema. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi, aggiunti ed espansi nel corso dell'evoluzione di UNIX. La Figura 2.13 mostra i diversi livelli di UNIX: tutto ciò che sta al di sotto dell'interfaccia alle chiamate di sistema e al di sopra dell'hardware costituisce il kernel. Esso comprende il file system, lo scheduling della CPU, la gestione della memoria, e le altre funzionalità del sistema rese disponibili tramite chiamate di sistema. Tutto sommato, si tratta di un'enorme massa di funzionalità diverse combinate in un solo livello. È questa struttura monolitica che rendeva difficile l'implementazione e la manutenzione.

2.7.2 Metodo stratificato

In presenza di hardware appropriato, i sistemi operativi possono essere suddivisi in moduli più piccoli e gestibili di quanto non fosse possibile nelle prime versioni di MS-DOS e UNIX.

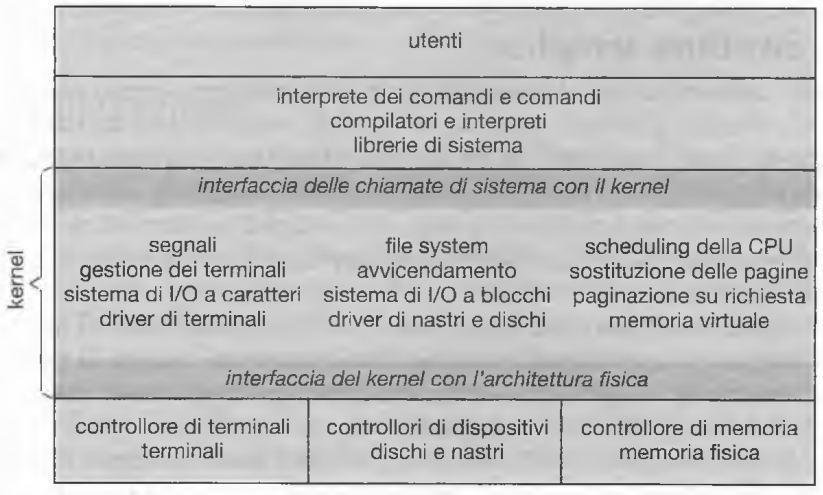

Figura 2.13 Struttura del sistema UNIX.

Ciò permette al sistema operativo di mantenere un controllo molto più stretto delle applicazioni che girano sulla macchina. Inoltre, gli sviluppatori del sistema godono di maggiore libertà nel modificare i meccanismi interni del sistema e nel suddividere il sistema in moduli. Secondo i dettami della metodologia di progettazione dall'alto verso il basso (*top-down approach*), le specifiche del sistema partono dall'individuazione delle sue funzionalità e delle sue caratteristiche complessive, che sono poi suddivise in componenti distinte. L'attenzione all'incapsulamento delle informazioni è anche importante, in quanto dà libertà agli sviluppatori di implementare le routine di basso livello nel modo che ritengono più opportuno, fintanto che l'interfaccia esterna alla routine rimanga invariata, e la routine stessa esegua il compito per cui è stata prevista.

Vi sono molti modi per rendere modulare un sistema operativo. Uno di loro è il **metodo stratificato**, secondo il quale il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N). Si veda la Figura 2.14.

Lo strato di un sistema operativo è una realizzazione di un oggetto astratto, che incapsula i dati e le operazioni che trattano tali dati. Un tipico strato di sistema operativo (chiamato M) è composto da strutture dati e da un insieme di routine richiamabili dagli strati di livello più alto. Lo strato M, a sua volta, è in grado di invocare operazioni dagli strati di livello inferiore.

Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e dalle funzionalità di debug. Gli strati sono composti in modo che ciascuno usi solo funzioni (o operazioni) e servizi che appartengono a strati di livello inferiore. Questo metodo semplifica il debugging e la verifica del sistema. Il primo strato si può mettere a punto senza intaccare il resto del sistema, poiché per realizzare le proprie funzioni usa, per definizione, solo lo strato fisico, che si presuppone sia corretto. Passando alla lavorazione del secondo strato si presume, dopo la messa a punto, la correttezza del primo. Il procedimento si ripete per ogni strato. Se si riscontra un errore, questo deve trovarsi in quello strato, poiché gli strati inferiori sono già stati corretti; quindi la suddivisione in strati semplifica la progettazione e la realizzazione di un sistema.

Ogni strato si realizza impiegando unicamente le operazioni messe a disposizione dagli strati inferiori, considerando soltanto le azioni che compiono, senza entrare nel merito di

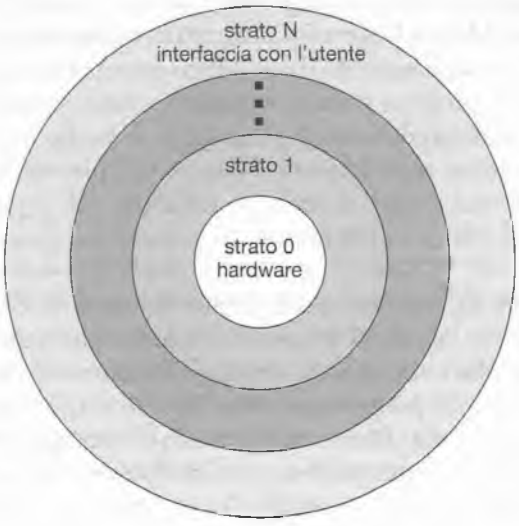

Figura 2.14 Struttura a strati di un sistema operativo.

come queste sono realizzate. Di conseguenza ogni strato nasconde a quelli superiori l'esistenza di determinate strutture dati, operazioni ed elementi fisici.

La principale difficoltà del metodo stratificato risiede nella definizione appropriata dei diversi strati. È necessaria una progettazione accurata, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia. Il dispositivo che controlla lo spazio sul disco usato dagli algoritmi che implementano la memoria virtuale deve risiedere in uno strato che si trovi sotto le routine per la gestione della memoria, perché essa richiede l'utilizzo di quello spazio.

Altri requisiti possono non essere così ovvi. Normalmente il driver della memoria ausiliaria (*backing store*) dovrebbe trovarsi sopra lo scheduler della CPU, poiché può accadere che il driver debba attendere un'istruzione di I/O, e in questo periodo la CPU si può sottoporre a scheduling. Tuttavia, in un grande sistema, lo scheduler della CPU può avere più informazioni su tutti i processi attivi di quante se ne possano contenere in memoria; perciò è probabile che queste informazioni si debbano caricare e scaricare dalla memoria, quindi il driver della memoria ausiliaria dovrebbe trovarsi sotto lo scheduler della CPU.

Un ulteriore problema che si pone con la struttura stratificata è che essa tende a essere meno efficiente delle altre; per esempio, per eseguire un'operazione di I/O un programma utente invoca una chiamata di sistema che è intercettata dallo strato di I/O che, a sua volta, esegue una chiamata allo strato di gestione della memoria, che a sua volta richiama lo strato di scheduling della CPU e che quindi è passata all'opportuno dispositivo di I/O. In ciascuno strato i parametri sono modificabili, può rendersi necessario il passaggio di dati, e così via; ciascuno strato aggiunge un carico alla chiamata di sistema. Ne risulta una chiamata di sistema che richiede molto più tempo di una chiamata di sistema corrispondente in un sistema non stratificato.

Negli ultimi anni questi limiti hanno causato una piccola battuta d'arresto allo sviluppo della stratificazione. Attualmente si progettano sistemi basati su un numero inferiore di strati con più funzioni, che offrono la maggior parte dei vantaggi del codice modulare, evitando i difficili problemi connessi alla definizione e all'interazione degli strati.

2.7.3 Microkernel

A mano a mano che il sistema operativo UNIX è stato esteso, il kernel è cresciuto notevolmente, diventando sempre più difficile da gestire. Verso la metà degli anni '80 un gruppo di ricercatori della Carnegie Mellon University progettò e realizzò un sistema operativo, **Mach**, col kernel strutturato in moduli secondo il cosiddetto orientamento a **microkernel**. Seguendo questo orientamento si progetta il sistema operativo rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema. Ne risulta un kernel di dimensioni assai inferiori. Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione.

Lo scopo principale del microkernel è fornire funzioni di comunicazione tra i programmi client e i vari servizi, anch'essi in esecuzione nello spazio utente. La comunicazione si realizza secondo il modello a scambio di messaggi, descritto nel Paragrafo 2.4.5. Per accedere a un file, per esempio, un programma client deve interagire con il file server; ciò non avviene mai in modo diretto, ma tramite uno scambio di messaggi con il microkernel.

Uno dei vantaggi del microkernel è la facilità di estensione del sistema operativo: i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel. Poiché è ridotto all'essenziale, se il kernel deve essere modificato, i cambiamenti da fare sono

ben circoscritti, e il sistema operativo risultante è più semplice da adattare alle diverse architetture. Inoltre offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto.

L'orientamento a microkernel è stato adottato per molti sistemi operativi moderni: il sistema Tru64 UNIX (in origine Digital UNIX), per esempio, offre all'utente un'interfaccia di tipo UNIX, ma il suo kernel è il Mach (il kernel traduce le chiamate di sistema di UNIX in messaggi ai corrispondenti servizi del livello utente). Anche il kernel di Mac OS X (conosciuto con il nome di *Darwin*) è basato sul microkernel Mach.

Un altro esempio è costituito da QNX, un sistema operativo real-time basato su un microkernel che fornisce i sevizi di scheduling e di consegna dei messaggi, oltre a gestire le interruzioni e la comunicazione lungo la rete a basso livello. Tutti gli altri servizi necessari sono forniti da processi ordinari eseguiti al di fuori del kernel in modalità utente.

Purtroppo i microkernel possono incorrere in cali di prestazioni dovuti al sovraccarico indotto dall'esecuzione di processi utente con funzionalità di sistema. La prima versione di Windows NT, basata su un microkernel stratificato, aveva prestazioni inferiori rispetto a Windows 95. La versione 4.0 di Windows NT risolse parzialmente il problema, spostando strati dal livello utente al livello kernel, e integrandoli più strettamente. Questa tendenza fece sì che, al tempo della progettazione di Windows XP, l'architettura di NT era ormai monolitica.

2.7.4 Moduli

Forse il miglior approccio attualmente disponibile per la progettazione dei sistemi operativi si fonda su tecniche della programmazione orientata agli oggetti per implementare un kernel modulare. In questo contesto, il kernel è costituito da un insieme di componenti fondamentali, integrati poi da funzionalità aggiunte dinamicamente durante l'avvio o l'esecuzione. Questa strategia, che impiega moduli caricati dinamicamente, è comune nelle implementazioni moderne di UNIX, come Solaris, Linux e Mac OS X. La struttura di Solaris, per esempio, illustrata dalla Figura 2.15, si incentra su un kernel dotato dei seguenti sette tipi di moduli caricabili.

1. Classi di scheduling.
2. File system.
3. Chiamate di sistema caricabili.
4. Formati eseguibili.
5. Moduli STREAMS.
6. Varie.
7. Driver dei dispositivi e del bus.

Questa organizzazione lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente certe caratteristiche. È possibile aggiungere driver per dispositivi specifici, per esempio, o gestire file system diversi tramite moduli caricabili. Il risultato complessivo somiglia ai sistemi a strati, perché ogni parte del kernel ha interfacce ben definite e protette. È tuttavia più flessibile dei sistemi a strati tradizionali, perché ciascun modulo può invocare funzionalità di un qualunque altro modulo. Inoltre, come nei sistemi basati su microkernel, il modulo principale gestisce solo i servizi essenziali, oltre a poter caricare altri moduli e comunicare con loro. E tuttavia, rispetto ai sistemi orientati a un microkernel, l'efficienza è superiore, perché i moduli sono in grado di comunicare senza invocare le funzionalità di trasmissione dei messaggi.

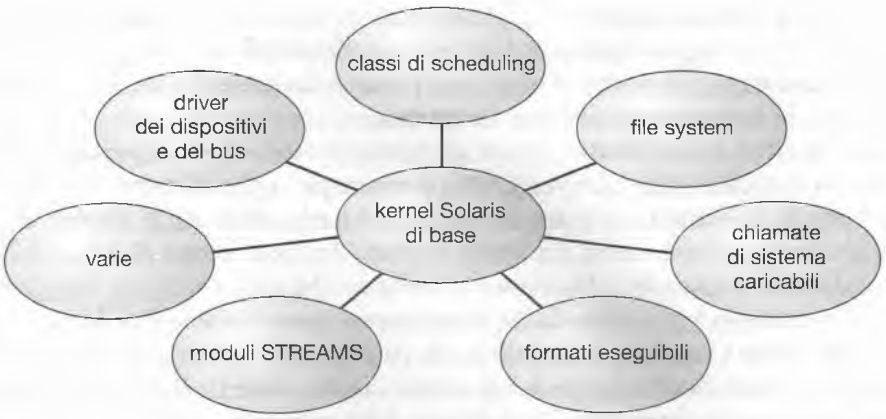

Figura 2.15 Moduli caricabili di Solaris.

Il sistema operativo Mac OS X di Apple adotta una struttura ibrida; è organizzato in strati, uno dei quali contiene il microkernel Mach. Si veda la Figura 2.16.

Gli strati superiori comprendono gli ambienti esecutivi delle applicazioni e un insieme di servizi che offre un'interfaccia grafica con le applicazioni. Il kernel si trova in uno strato sottostante, ed è costituito dal microkernel Mach e dal kernel BSD. Il primo cura la gestione della memoria, le chiamate di procedure remote (RPC), la comunicazione tra processi (IPC) – compreso lo scambio di messaggi – e lo scheduling dei thread. Il secondo mette a disposizione un'interfaccia BSD a riga di comando, i servizi legati al file system e alla rete, e la API POSIX, compreso Pthreads. Oltre a Mach e BSD, il kernel possiede un kit di strumenti connessi all' I/O per lo sviluppo di driver dei dispositivi e di moduli dinamicamente caricabili, detti **estensioni del kernel** nel gergo di Mac OS X. Come si vede dalla figura, le applicazioni e i servizi comuni possono accedere direttamente sia ai servizi di Mach sia a quelli di BSD.

2.8 Macchine virtuali

La metodologia di progettazione per strati descritta nel Paragrafo 2.7.2 porta nella sua naturale conclusione al concetto di **macchina virtuale** (*virtual machine*). L'idea alla base delle macchine virtuali è di astrarre dalle unità hardware del singolo computer (CPU, memoria,

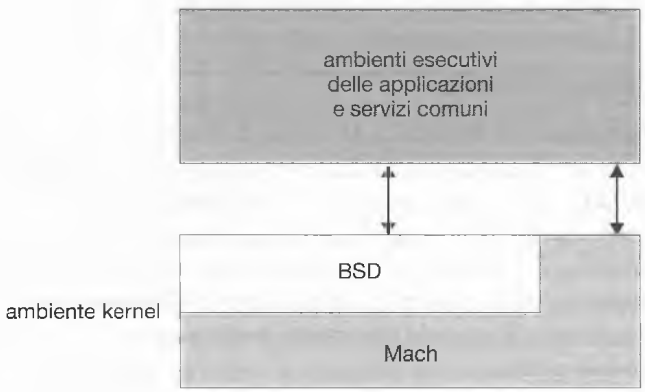

Figura 2.16 Struttura di Mac OS X.

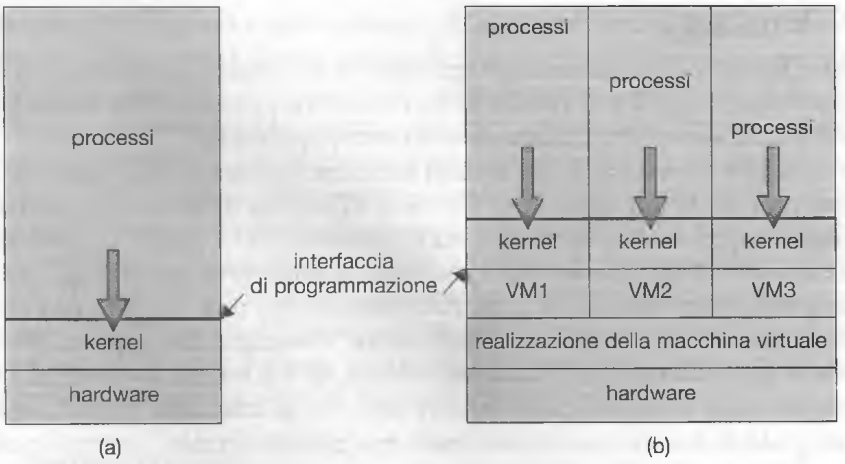

Figura 2.17 Modelli di sistema: (a) Semplice. (b) Macchina virtuale.

dispositivi periferici, e così via), progettando per ciascuna unità un ambiente esecutivo software diverso, così da dare l'impressione che ognuno di loro giri sulla propria macchina.

Tramite lo scheduling della CPU (Capitolo 5) e la memoria virtuale (Capitolo 9), il sistema operativo può dare l'impressione che ogni processo sia dotato del proprio processore e del proprio spazio di memoria (virtuale). La macchina virtuale fornisce un'interfaccia *coincidente* con il nudo hardware. Ogni **processo ospite** può usufruire di una copia (virtuale) del calcolatore sottostante (Figura 2.17). Solitamente il processo ospite è un sistema operativo. In questo modo una singola macchina fisica può far girare più sistemi operativi concorrentemente, ciacuno sulla sua macchina virtuale.

2.8.1 Storia

Le macchine virtuali hanno fatto la loro prima apparizione commerciale nel 1972 sui mainframe IBM, grazie al sistema operativo VM. Questo sistema ha subìto un'evoluzione ed è ancora oggi disponibile. Molte delle idee introdotte da VM si possono ritrovare in altri sistemi e dunque vale la pena esplorare questo strumento.

IBM VM370 suddivideva un mainframe in diverse macchine virtuali, ognuna dotata del proprio sistema operativo. Una tra le difficoltà maggiori che si incontrarono con le macchine virtuali di VM riguardò i sistemi di dischi. Supponiamo che la macchina fisica contenga tre dischi, ma voglia supportare sette macchine virtuali. Chiaramente, non era possibile allocare un disco fisso a ogni macchina virtuale, perché il solo software della macchina virtuale richiedeva una notevole quantità di spazio su disco per memoria virtuale e spooling. La soluzione fu quella di fornire dischi virtuali (chiamati *minidisk* nel sistema operativo VM di IBM) fra loro identici in tutto tranne che nelle dimensioni. Il sistema implementava ciacun minidisk allocando la quantità necessaria di tracce sul disco fisso.

Una volta create queste macchine virtuali, l'utente poteva eseguire tutti i sistemi operativi e i pacchetti software disponibili sulla macchina sottostante. Nel caso di VM, l'utente eseguiva solitamente CMS, un sistema operativo interattivo a singolo utente.

2.8.2 Vantaggi

Sono diverse le ragioni che portano alla creazione di una macchina virtuale, ma la maggior parte è legata alla possibilità di condividere l'utilizzo concorrente dello stesso hardware in diversi ambienti di esecuzione (ovvero diversi sistemi operativi).

Un vantaggio importante è che sistema ospitante è protetto dalle macchine virtuali, e queste sono protette le une dalle altre. Un virus all'interno di un sistema operativo ospite può danneggiare quel sistema operativo, ma è improbabile che colpisca il sistema ospitante o altri sistemi ospiti. Non ci sono quindi problemi di protezione, perché ogni macchina virtuale è completamente isolata dalle altre. Allo stesso tempo non vi è però una condivisione diretta delle risorse. Per la condivisione delle risorse si seguono due approcci diversi. Il primo prevede la possibilità di condividere un volume del file system, e quindi di condividere file. Il secondo offre la possibilità di definire una rete di macchine virtuali, ognuna delle quali sia in grado di inviare informazioni sulla rete privata virtuale. La rete è modellata come una rete fisica, ma è implementata via software.

Una macchina virtuale è uno strumento perfetto per condurre ricerche sui sistemi operativi e per svilupparne di nuovi. La modifica di un sistema operativo è solitamente un compito difficile. I sistemi operativi sono programmi lunghi e complessi ed è difficile essere sicuri che un cambiamento in una parte del sistema non causi dei malfunzionamenti nascosti in altre parti. La potenza dei sistemi operativi rende le modifiche particolarmente pericolose. Essi, infatti, sono in esecuzione in modalità kernel, e un'errata modifica a un puntatore potrebbe causare un errore che cancella l'intero file system. È quindi necessario un test scrupoloso dopo ogni modifica effettuata.

I sistemi operativi gestiscono e controllano l'intera macchina. È quindi necessario, dopo eventuali modifiche e durante i test, arrestare il sistema e renderlo inutilizzabile dagli utenti. Questo periodo di tempo è solitamente chiamato **periodo di sviluppo del sistema** (*system-development time*). Il periodo di sviluppo del sistema è usualmente pianificato la notte o nel fine settimana, quando il carico del sistema è basso, proprio perché in questo periodo il sistema operativo non è utilizzabile dagli utenti.

Una macchina virtuale può eliminare molti di questi problemi. Ai programmatori di sistema è messa a disposizione una macchina virtuale dedicata e lo sviluppo di sistema è effettuato sulla macchina virtuale invece che sulla macchina fisica. Di rado le normali operazioni di sistema hanno bisogno di essere disturbate a causa dello sviluppo.

Un altro vantaggio che le macchine virtuali offrono agli sviluppatori è dato dal fatto che diversi sistemi operativi possono lavorare in concorrenza sulla stessa macchina. Una workstation così configurata permette una rapida portabilità delle applicazioni su differenti piattaforme e facilita la fase di test. In maniera del tutto simile, gli ingegneri addetti alla qualità possono provare le applicazioni su piattaforme multiple senza dover comprare, né aggiornare, né mantenere un computer diverso per ogni diversa piattaforma.

Un grande vantaggio delle macchine virtuali, sfruttato nei centri di trattamento dati, è il **consolidamento** del sistema, ovvero l'esecuzione su uno stesso sistema di due o più sistemi diversi originariamente installati su macchine fisiche distinte. Questo passaggio dai sistemi fisici ai sistemi virtuali permette un'efficace ottimizzazione delle risorse, in quanto diversi sistemi scarsamente utilizzati possono essere combinati su un'unica macchina utilizzata più intensamente.

Se l'utilizzo di macchine virtuali continuasse a diffondersi, lo sviluppo di applicazioni dovrebbe subire una conseguente evoluzione. Infatti, se un sistema può facilmente aggiungere, rimuovere o spostare macchine virtuali, perché installare un'applicazione direttamente su quel sistema? Piuttosto, gli sviluppatori installerebbero le applicazioni su un sistema operati-

vo correttamente configurato e personalizzato all'interno di una macchina virtuale. Questa piattaforma virtuale diventerà il meccanismo di distribuzione dell'applicazione. Questo metodo costituirebbe una notevole miglioria per gli sviluppatori: la gestione delle applicazioni diventerebbe più semplice, sarebbe richiesta una minor messa a punto (*tuning*) e il supporto tecnico sarebbe semplificato. Anche gli amministratori di sistema vedrebbero semplificato il loro compito. L'installazione sarebbe infatti più semplice e la ricompilazione di un'applicazione su un nuovo sistema sarebbe facilitata, non richiedendo più gli usuali passi di disinstallazione e reinstallazione. Affinché sia possibile un'adozione di massa della metodologia descritta, è però necessaria la definizione di uno standard del formato delle macchine virtuali, in modo che ogni macchina virtuale possa girare su ogni piattaforma di virtualizzazione. Il progetto "Open Virtual Machine Format" è un tentativo di definire questo standard e il suo successo porterebbe a una unificazione del formato delle macchine virtuali.

2.8.3 Simulazione

La virtualizzazione, descritta fin qui, è soltanto uno dei numerosi metodi per emulare un sistema. È il metodo più comune, perché fa in modo che il sistema operativo ospite e le applicazioni "credano" di essere in esecuzione su un hardware nativo. Visto che solo le risorse di sistema devono essere virtualizzate, i processi ospitati possono girare quasi a piena velocità.

Un altro metodo di emulazione è la **simulazione**. In questo caso si ha un sistema ospitante con una propria architettura e un sistema ospite compilato per un'architettura diversa. Supponiamo per esempio che un'azienda abbia sostituito i vecchi sistemi con sistemi più nuovi, ma che voglia continuare a utilizzare alcuni importanti programmi compilati per i vecchi sistemi. I programmi potrebbero essere mandati in esecuzione su un emulatore in grado di tradurre le istruzioni del vecchio sistema in istruzioni per il nuovo sistema. L'emulazione permette quindi di incrementare la vita dei programmi e inoltre aiuta a studiare vecchie architetture di sistema. Il grosso limite sono le prestazioni. Le istruzioni emulate vengono eseguite infatti molto più lentamente rispetto alle istruzioni native. Succede dunque che, anche se si dispone di una macchina 10 volte più potente rispetto alla vecchia, il programma in esecuzione sulla nuova macchina sarà ancora più lento di quanto lo era sull'hardware per il quale era stato progettato. Un'ulteriore difficoltà si incontra nella creazione di un emulatore che funzioni correttamente, perché, in sostanza, progettare un emulatore significa riscrivere un processore via software.

2.8.4 Paravirtualizzazione

Un'altra variazione sul tema è la **paravirtualizzazione**. Piuttosto che provare a offrire al sistema ospite una piattaforma identica a quella da esso desiderata, con la paravirtualizzazione si cerca di rendere disponibile al sistema ospite un sistema simile, ma non identico, alle sue preferenze. L'ospite deve allora essere modificato prima di essere eseguito su un hardware paravirtualizzato. Il lavoro addizionale richiesto da queste modifiche offre in cambio un guadagno in termini di uso più efficiente delle risorse e snellezza del livello paravirtuale.

Solaris 10 include dei **contenitori** (*containers*), anche detti **zone**, che creano un livello virtuale tra il sistema operativo e le applicazioni. In questo sistema è installato un unico kernel e l'hardware non è virtualizzato. A essere virtualizzato è il sistema operativo con i suoi dispositivi, in modo da dare ai processi un contenitore in cui vi sia l'impressione di essere l'unico processo in esecuzione sul sistema. Possono essere creati uno o più contenitori, ognuno con le proprie applicazioni, il proprio indirizzo di rete, le proprie porte, i propri account utente, e così via. Le risorse del processore possono ripartite tra i contenitori e i pro-

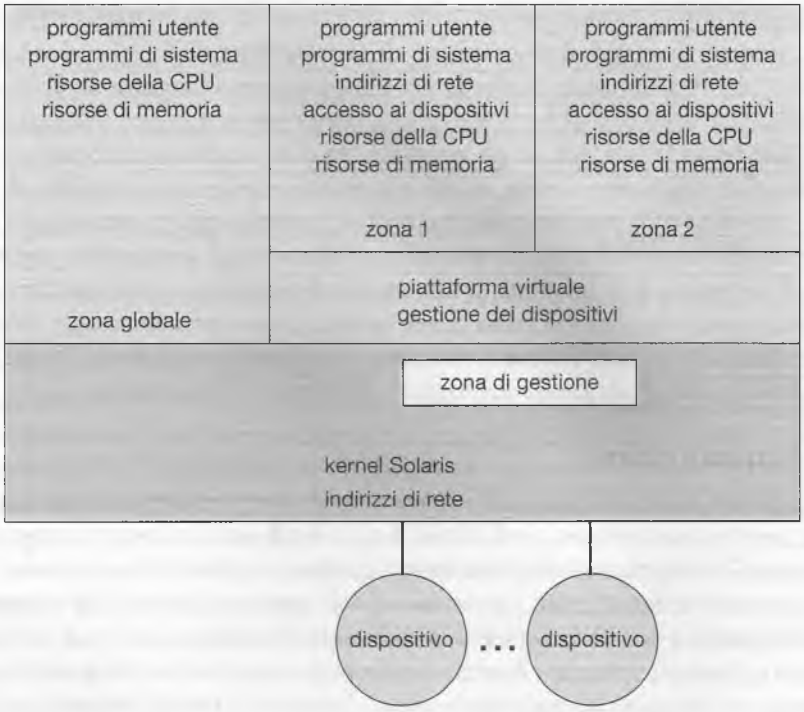

Figura 2.18 Solaris 10 con due contenitori.

cessi di sistema. La Figura 2.18 mostra un sistema Solaris 10 con due contenitori e lo spazio utente standard, chiamato "globale".

2.8.5 Realizzazione

Benché utile, il concetto di macchina virtuale è difficile da realizzare; è difficile ottenere un *esatto* duplicato della macchina sottostante. Si ricordi che la macchina sottostante ha due modalità di funzionamento, utente e di sistema. I programmi che realizzano il sistema di macchine virtuali si possono eseguire nella modalità di sistema, poiché costituiscono il sistema operativo, mentre ciascuna macchina virtuale può funzionare solo nella modalità utente. Quindi, proprio come la macchina fisica, anche quella virtuale deve avere due modalità; di conseguenza si devono avere modalità utente virtuale e modalità di sistema virtuale, entrambe operanti in modalità utente fisico. Le azioni che causano un trasferimento dalla modalità utente a quella di sistema su una macchina reale, come una chiamata di sistema o un tentativo d'esecuzione di un'istruzione privilegiata, in una macchina virtuale devono causare un analogo trasferimento dalla modalità utente virtuale a quella di sistema virtuale.

Generalmente questo trasferimento è eseguibile in modo abbastanza semplice. Se, per esempio, un programma in esecuzione su una macchina virtuale in modalità utente virtuale esegue una chiamata di sistema, si passa alla modalità di sistema della macchina virtuale all'interno della macchina reale. Quando il sistema della macchina virtuale acquisisce il controllo, può modificare il contenuto dei registri e il contatore di programma della macchina virtuale, in modo da simulare l'effetto della chiamata di sistema; quindi può riavviare la macchina virtuale, che si trova in modalità di sistema virtuale.

La differenza più rilevante consiste, naturalmente, nel tempo. Mentre l'I/O reale potrebbe richiedere 100 millisecondi, quello virtuale potrebbe comportare un tempo inferiore (poiché le operazioni di I/O avvengono su file contenuti in dischi, che simulano i dispositivi periferici di I/O), o superiore, poiché è interpretato. Inoltre la condivisione della CPU rallenta ulteriormente in modo imprevedibile le macchine virtuali. In un caso limite, per offrire una vera macchina virtuale, può essere necessario simulare tutte le istruzioni. Il sistema VM funziona su calcolatori IBM, poiché le normali istruzioni per le macchine virtuali sono eseguibili direttamente dalla macchina reale. Soltanto le istruzioni privilegiate, richieste soprattutto per le operazioni di I/O, devono essere simulate e quindi eseguite più lentamente.

Senza qualche supporto hardware la virtualizzazione non sarebbe possibile. Più supporto hardware fornirà un sistema, più le macchine virtuali saranno stabili, performanti e ricche di funzionalità. Le più diffuse CPU general-purpose offrono un supporto hardware dedicato alla virtualizzazione. La tecnologia di virtualizzazione AMD, presente su diversi processori AMD, ne è un esempio. Tale tecnologia definisce due nuove modalità per le operazioni: modalità host (ospitante) e modalità guest (ospitato). Il software per la macchina virtuale può abilitare la modalità host, definire le caratteristiche di ogni macchina virtuale e quindi cambiare la modalità in guest e passare il controllo del sistema al processo ospite in esecuzione sulla macchina virtuale. In modalità guest il sistema operativo virtualizzato crede di essere in esecuzione su hardware nativo e può vedere alcuni dei dispositivi (quelli inclusi nella definizione delle caratteristiche dell'ospite fatta da parte dell'ospitante). Se il processo ospite prova ad accedere a una risorsa il controllo passa all'ospitante, che renderà possibile l'interazione.

2.8.6 Esempi

Nonostante i vantaggi offerti dalle macchine virtuali, dopo il loro primo sviluppo esse sono state oggetto di poche attenzioni per diversi anni. Ciononostante, oggi le macchine virtuali sono diventate di moda come strumento per risolvere problemi di compatibilità di sistemi. In questo paragrafo analizziamo due popolari macchine virtuali: VMware workstation e la Java virtual machine. Come vedremo, queste macchine virtuali possono essere solitamente eseguite su ognuno dei sistemi operativi descritti in precedenza. Le metodologie di disegno dei sistemi operativi (livelli semplici, microkernel, moduli e macchine virtuali) non sono quindi mutuamente esclusive.

2.8.6.1 VMware

La maggior parte delle tecniche di virtualizzazione discusse in questo paragrafo richiede il supporto del kernel alla virtualizzazione. Un altro metodo richiede che gli strumenti di virtualizzazione siano scritti per essere eseguiti in modalità utente, come applicazione eseguita dal sistema operativo. Le macchine virtuali in esecuzione tramite questi strumenti credono di essere in esecuzione su un hardware dedicato, ma in effetti sono eseguite all'interno di un'applicazione a livello utente.

VMware è una nota applicazione commerciale che converte l'apparato fisico Intel 80x86 in macchine virtuali distinte. VMware funge da applicazione in un sistema operativo ospitante, quale Windows o Linux, che può così installare alcuni **sistemi operativi ospiti** a titolo di macchine virtuali indipendenti.

Si può osservare l'architettura di un sistema siffatto nella Figura 2.19. In questo esempio, Linux è il sistema operativo residente, mentre FreeBSD, Windows NT e Windows XP sono i sistemi ospiti. Lo strato incaricato della virtualizzazione è il fulcro di VMware, poiché grazie a esso i dispositivi fisici sono trasformati in macchine virtuali a sé stanti che fungono

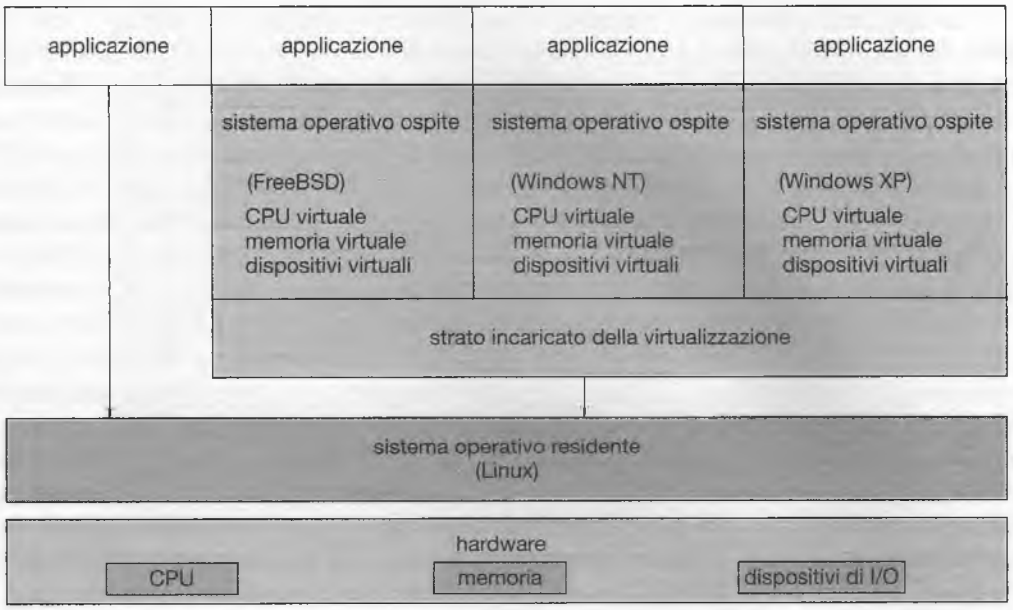

Figura 2.19 Architettura VMware.

da sistemi operativi ospiti. Ciascuna macchina virtuale, oltre a possedere una CPU virtuale, può contare su elementi virtuali propri, quali la memoria, i drive del disco, le interfacce di rete e via di seguito.

Il disco fisico di cui l'ospite dispone è in realtà semplicemente un file all'interno del file system del sistema operativo ospitante. Per creare un'istanza identica all'ospite è sufficiente copiare il file. Copiare il file in una nuova posizione protegge l'istanza dell'ospite da possibili danneggiamenti alla posizione originale. Spostando il file in una nuova posizione viene spostato il sistema ospite. Questi scenari mostrano come la virtualizzazione può effettivamente aumentare l'efficienza nell'amministrazione di un sistema e nell'uso delle risorse di sistema.

2.8.6.2 Macchina virtuale Java

Il linguaggio di programmazione Java, introdotto dalla Sun Microsystems alla fine del 1995, è un linguaggio orientato agli oggetti molto diffuso e fornisce, oltre a una descrizione analitica della sintassi del linguaggio e a una vasta libreria API, anche la definizione della **macchina virtuale Java** (*Java virtual machine*, JVM).

Gli oggetti si specificano con il costrutto `class` e un programma consiste di una o più classi. Per ognuna di queste, il compilatore produce un file (`.class`) contenente il cosiddetto *bytecode*; si tratta di codice nel linguaggio di macchina della JVM, indipendente dall'architettura soggiacente, che viene per l'appunto eseguito dalla JVM.

La JVM è un calcolatore astratto che consiste di un **caricatore delle classi** e di un interprete del linguaggio che esegue il bytecode (Figura 2.20). Il caricatore delle classi carica i file `.class`, sia del programma scritto in Java sia dalla libreria API, affinché l'interprete possa eseguirli. Dopo che una classe è stata caricata, il verificatore delle classi controlla la correttezza sintattica del codice bytecode, che il codice non produca accessi oltre i limiti della

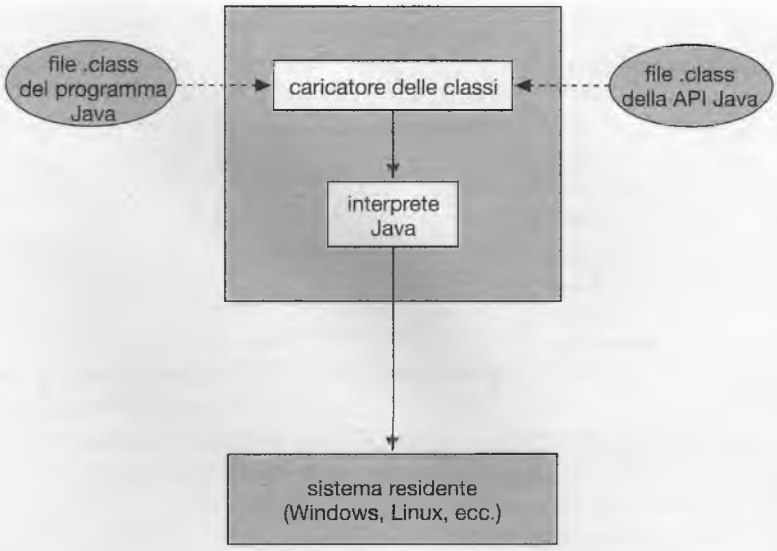

Figura 2.20 Macchina virtuale Java.

pila e che non esegua operazioni aritmetiche sui puntatori, che potrebbero generare accessi illegali alla memoria. Se il controllo ha un esito positivo, la classe viene eseguita dall'interprete. La JVM gestisce la memoria in modo automatico procedendo alla sua "ripulitura" (*garbage collection*) che consiste nel recupero delle aree della memoria assegnate a oggetti non più in uso per restituirla al sistema. Al fine di incrementare le prestazioni dei programmi eseguiti dalla macchina virtuale una notevole attività di ricerca è focalizzata allo studio degli algoritmi di ripulitura della memoria.

La JVM può essere implementata come software ospitato da un sistema operativo residente, per esempio Windows, Linux o Mac OS X, oppure all'interno di un browser web. In alternativa, può essere cablata in un circuito integrato espressamente progettato per l'esecuzione di programmi Java. Nel primo caso, l'interprete Java *interpreta* le istruzioni bytecode una alla volta. Una soluzione più efficiente consiste nell'uso di un **compilatore istantaneo** o **just-in-time** (JIT). Alla prima invocazione di un metodo Java, il bytecode relativo è tradotto in linguaggio macchina comprensibile dalla macchina fisica ospitante. Il codice macchina relativo è poi salvato appropriatamente, in modo da essere direttamente riutilizzabile a una successiva invocazione del metodo Java, evitando la lenta interpretazione delle istruzioni bytecode. Una soluzione potenzialmente ancora più veloce è cablare la JVM in un circuito integrato, come detto poc'anzi, che esegua le istruzioni bytecode come codice macchina primitivo, eliminando del tutto la necessità di interpreti e compilatori.

2.9 Debugging dei sistemi operativi

Il **debugging** può essere genericamente definito come l'attività di individuare e risolvere errori nel sistema, i cosiddetti **bachi** (*bugs*). Questa operazione viene effettuata sia sull'hardware sia sul software. I problemi che condizionano le prestazioni sono considerati bachi, quindi il debugging può comprendere anche una **regolazione delle prestazioni** (*performance tuning*), che ha lo scopo di migliorare le prestazioni eliminando i **colli di bottiglia** (*bottle-*

L'AMBIENTE .NET

L'ambiente .NET può essere definito come la sinergia tra un gruppo di risorse (per esempio, le librerie delle classi) e un ambiente di esecuzione che, combinati insieme, formano una piattaforma idonea allo sviluppo di programmi. La piattaforma permette di realizzare programmi concepiti espressamente per l'ambiente .NET anziché per altre specifiche architetture. Un programma scritto per .NET non deve preoccuparsi dei dettagli legati all'hardware o al sistema operativo che lo eseguirà. Infatti, ogni sistema che implementi .NET, sarà in grado di eseguire con successo il programma. Ciò è reso possibile dall'ambiente di esecuzione, che ricava i dettagli necessari e fornisce una macchina virtuale che opera come intermediario tra il programma in esecuzione e l'architettura sottostante.

Il Linguaggio Comune Runtime (CLR) è ciò che dà vita alla cornice .NET. Il CLR è l'implementazione della macchina virtuale .NET, che permette l'esecuzione di programmi scritti nei linguaggi compatibili con l'ambiente .NET. I programmi scritti con linguaggi come il C# (pronunciato, in inglese, "C-sharp") e il VB.NET sono compilati in un linguaggio intermedio indipendente dall'architettura, denominato Linguaggio Intermedio Microsoft (MS-IL). Questi file compilati, detti assemblati, comprendono le istruzioni MS-IL e i metadati. Hanno l'estensione .EXE oppure .DLL. Al momento dell'esecuzione di un programma, il CLR carica i file assemblati in ciò che è noto come **dominio dell'applicazione**. Mentre il programma in esecuzione richiede le istruzioni, il CLR converte, tramite compilazione istantanea, le istruzioni MS-IL nei file assemblati nel codice nativo utilizzato dall'architettura sottostante. Una volta che le istruzioni siano state convertite in codice nativo, sono appropriatamente salvate e continueranno a essere eseguite, all'occorrenza, dalla CPU, in forma di codice macchina. L'architettura del CLR per la cornice .NET è rappresentata nella Figura 2.21.

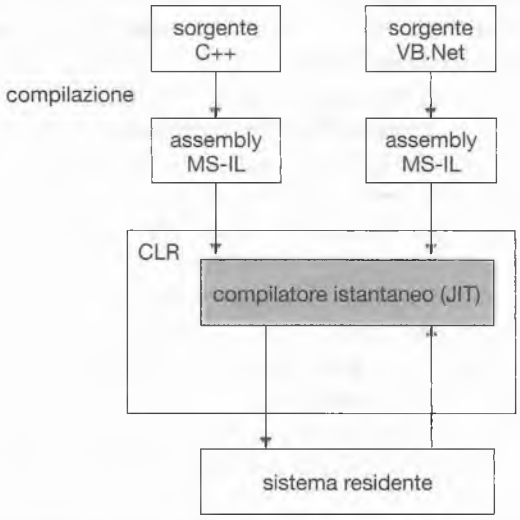

Figura 2.21 Architettura di CLR nell'ambiente .NET.

neck) che hanno luogo nei processi di sistema. In questo paragrafo tratteremo del debugging del kernel, degli errori di processo e dei problemi di prestazioni; il tema del debugging dell'hardware esula invece dagli scopi di questo testo.

2.9.1 Analisi dei guasti

Se un processo fallisce, la maggior parte dei sistemi operativi scrive le informazioni relative all'errore avvenuto in un **file di log** (*log file*), in modo da rendere noto agli operatori o agli utenti del sistema di ciò che è avvenuto. Il sistema operativo può anche acquisire un'immagine del contenuto della memoria utilizzata dal processo, chiamata **core dump**, in quanto la memoria ai primordi dell'era informatica era chiamata "nucleo" (*core*). L'immagine della memoria viene conservata in un file per un'analisi successiva. Il **debugger**, uno strumento che permette al programmatore di esplorare il codice e la memoria di un processo, è incaricato di esaminare i programmi in esecuzione e i core dump.

Se il debugging di processi a livello utente è una sfida, a livello del kernel del sistema operativo esso è un'attività ancora più difficile a causa della dimensione e della complessità del kernel, del suo controllo dell'hardware e della mancanza di strumenti per eseguire il debugging a livello utente. Un guasto nel kernel viene chiamato **crash**. Come accade per i guasti dei processi, anche in questo caso dell'informazione riguardante l'errore viene salvata in un file di log, mentre lo stato della memoria viene salvato in un'immagine del contenuto della memoria al momento del crash (**crash dump**).

Il debugging del sistema operativo usa spesso strumenti e tecniche differenti rispetto al debugging dei processi, perché la natura delle due attività è molto diversa. Teniamo in considerazione il fatto che un guasto del kernel nel codice relativo al file system renderebbe rischioso per il kernel provare a salvare il suo stato in un file all'interno prima del riavvio. Una tecnica comune consiste nel salvare lo stato di memoria del kernel in una sezione del disco adibita esclusivamente a questo scopo. Quando il kernel rileva un errore irrecuperabile scrive l'intero contenuto della memoria, o per lo meno delle parti della memoria di sistema possedute dal kernel, nell'area di disco a ciò destinata. Nel momento in cui il kernel si riavvia, viene eseguito un processo che raccoglie i dati da quest'area e li scrive in un file depositario dei guasti all'interno del file system per un'analisi.

2.9.2 Regolazione delle prestazioni

Per identificare eventuali colli di bottiglia dobbiamo essere in grado di monitorare le prestazioni del sistema. A tale scopo deve essere presente del codice che esegua misurazioni sul comportamento del sistema e mostri i risultati. In molti casi il sistema operativo produce degli elenchi che tracciano il comportamento del sistema; tutti gli eventi di rilievo sono descritti indicandone l'ora e i parametri importanti e sono scritti in un file. Successivamente, un programma di analisi può esaminare il file di log al fine di determinare le prestazioni del sistema e di identificarne ostacoli e inefficienze. Le stesse tracce possono essere utilizzate come input per la simulazione di una miglioria al sistema operativo e inoltre possono contribuire alla scoperta di errori nel comportamento dello stesso.

LA LEGGE DI KERNIGHAN

"Il debugging è due volte più difficile rispetto alla stesura del codice. Di conseguenza, chi scrive il codice nella maniera più intelligente possibile non è, per definizione, abbastanza intelligente per eseguirne il debugging."

Un altro approccio alla regolazione delle prestazioni consiste nell'includere nel sistema strumenti interattivi che permettano a utenti e amministratori di interrogare lo status dei vari componenti del sistema per individuare i colli di bottiglia. Il comando `top` in UNIX mostra le risorse di sistema impiegate, nonché un elenco ordinato dei principali processi che utilizzano le risorse. Altri strumenti mostrano lo stato del disco I/O, la memoria allocata e il traffico di rete. Gli autori di questi strumenti dotati di un'unica finalità provano a indovinare le necessità dell'utente che analizza il sistema e offrono queste informazioni.

Rendere i sistemi operativi esistenti più facili da comprendere e semplificarne il debugging e la regolazione delle prestazioni sono un'area attiva della ricerca e dell'implementazione dei sistemi. Il ciclo che comincia con il tracciare i problemi riscontrati nel sistema e prosegue con l'analisi delle tracce sta per essere interrotto da una nuova generazione di strumenti adibiti all'analisi delle prestazioni del kernel. Questi strumenti non hanno un unico scopo né sono destinati esclusivamente a sezioni del codice scritte appositamente per produrre informazioni sul debugging. Solaris 10 DTrace è un'utilità per il tracciamento dinamico che costituisce un esempio rilevante di tali strumenti.

2.9.3 DTrace

DTrace è un'utilità che aggiunge dinamicamente delle sonde al sistema operativo, sia nei processi utente sia nel kernel. Queste sonde possono essere interrogate attraverso il linguaggio di programmazione `D` per determinare una quantità stupefacente di informazioni sul kernel, sullo status del sistema e sulle attività di processo. Ad esempio, nella Figura 2.22 si segue il comportamento di un'applicazione mentre esegue una chiamata di sistema (`ioctl`)

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                                U
  0  -> _XEventsQueued                              U
  0   -> _X11TransBytesReadable                     U
  0   <- _X11TransBytesReadable                     U
  0   -> _X11TransSocketBytesReadable               U
  0   <- _X11TransSocketBytesreadable               U
  0   -> ioctl                                      U
  0    -> ioctl                                     K
  0     -> getf                                     K
  0      -> set_active_fd                           K
  0      <- set_active_fd                           K
  0     <- getf                                     K
  0     -> get_udatamodel                           K
  0     <- get_udatamodel                           K
...
  0     -> releasef                                 K
  0      -> clear_active_fd                         K
  0      <- clear_active_fd                         K
  0      -> cv_broadcast                            K
  0      <- cv_broadcast                            K
  0     <- releasef                                 K
  0    <- ioctl                                     K
  0   <- ioctl                                      U
  0  <- _XEventsQueued                              U
  0 <- XEventsQueued                                U
```

Figura 2.22 DTrace, su Solaris 10, segue una chiamata di sistema all'interno del kernel.

e si mostrano le chiamate di funzioni all'interno del kernel che vengono eseguite per completare la chiamata di sistema. Le linee che terminano con "U" sono eseguite in modalità utente, quelle che terminano con "K" in modalità kernel. È pressoché impossibile eseguire il debugging dell'interazione tra livello utente e codice kernel senza avere a disposizione un insieme di strumenti che capiscano entrambi i tipi di codice e possano tenere traccia di questa interazione con strumenti appropriati.

Affinché un tale gruppo di strumenti risulti veramente utile deve essere in grado di fare il debugging di tutte le aree del sistema, incluse quelle scritte originariamente senza prendere in considerazione il debugging, e deve poterlo fare senza condizionare l'affidabilità del sistema. Questo strumento deve avere inoltre un impatto minimo sulle prestazioni: idealmente, non dovrebbe avere alcun impatto mentre non è in funzione e un impatto minimo durante l'utilizzo. L'utilità DTrace soddisfa questi requisiti offrendo uno strumento di debugging dinamico, sicuro e a basso impatto.

Il framework e gli strumenti DTrace furono disponibili a partire dal Solaris 10. Fino a quel momento, il debugging del kernel era una sorta di oggetto misterioso; lo si eseguiva tramite codice e strumenti arcaici e non sistematici. Ad esempio, i processori sono dotati di una funzionalità di breakpoint che ferma l'esecuzione e permette al debugger di esaminare lo stato del sistema. L'esecuzione può poi continuare fino al breakpoint o alla terminazione successivi. Questo metodo non può essere utilizzato in un kernel multiutente senza condizionare negativamente tutti gli utenti del sistema. La **profilazione** o **profiling**, che saggia periodicamente il puntatore alle istruzioni per verificare qual è il codice in esecuzione, mostra le tendenze statistiche, ma non le attività individuali. Può essere incluso nel kernel un codice destinato a produrre dati specifici in specifiche circostanze, ma quel codice rallenta il kernel e tende a non essere incluso in quella parte del kernel dove si è verificato il problema che ha richiesto il debugging.

Al contrario, DTrace funziona su sistemi di produzione (sistemi che stanno mettendo in funzione applicazioni importanti o critiche) e non è dannoso al sistema. Pur rallentando le attività quando è in esecuzione, dopo l'esecuzione riporta il sistema allo stato precedente il debugging. Si tratta inoltre di uno strumento che agisce ampiamente e in profondità, in quanto può eseguire il debugging di tutto ciò che sta accadendo nel sistema (a livello utente e a livello del kernel, comprese le interazioni tra i due livelli) e può scavare profondamente nel codice, mostrando le singole istruzioni del processore e le attività del kernel.

DTrace è composto da un compilatore, un framework, diversi **provider di sonde** (*providers of probes*), scritti all'interno di quel framework, e **consumer delle sonde**. I provider di DTrace creano le sonde, delle quali viene tenuta traccia in apposite strutture del kernel. Le sonde vengono immagazzinate in una tabella hash, dove sono suddivise per nome e indicizzate secondo identificatori univoci di sonde. Quando una sonda viene abilitata, una porzione di codice nell'area da sondare è riscritta per eseguire la chiamata `dtrace_probe` (`probe identifier`) per poi proseguire con il normale flusso del codice. Provider differenti danno origine a differenti tipi di sonde. Ad esempio, una sonda che esegue una chiamata di sistema del kernel lavora in modo differente da una sonda in un processo utente, che è a sua volta diversa da una sonda I/O.

DTrace fornisce un compilatore che genera un byte code eseguito nel kernel; il compilatore stesso garantisce la sicurezza del codice che ha creato. Ad esempio, non sono permessi cicli e vengono permesse solo specifiche modifiche allo stato del kernel ed esclusivamente su richiesta. Solamente gli utenti di DTrace che godono di privilegi (ovvero gli amministratori, o *utenti root*) possono usare DTrace, dal momento che questo può recuperare dal kernel dati privati (e modificarli se richiesto). Il codice generato funziona nel kernel e attiva le sonde, oltre ad attivare i consumer in modalità utente e a permettere la comunicazione tra i due.

Un consumer di DTrace è un codice interessato a una sonda e ai suoi risultati. Esso richiede al provider di creare una o più sonde. Quando una sonda si attiva, produce dei dati che sono gestiti dal kernel. Nel momento dell'attivazione all'interno del kernel vengono eseguite delle azioni di **abilitazione dei blocchi di controllo** (*enabling control blocks*, ECB). Una sonda può provocare l'esecuzione di diversi ECB se più consumer sono interessati a essa. Ogni ECB contiene un predicato ("if statement", ovvero istruzione if) che può o escludere l'ECB in questione, oppure eseguire la lista di azioni nell'ECB. L'azione più frequente è quella di catturare alcuni gruppi di dati, come il valore di una variabile a quel punto dell'esecuzione della sonda. Raccogliendo tali dati, può essere costruita un'immagine completa dell'azione dell'utente o del kernel. Inoltre, l'attivazione delle sonde dall'area utente e dal kernel può mostrare come un'azione a livello utente abbia causato reazioni a livello kernel. Questi dati hanno un valore inestimabile per il monitoraggio delle prestazioni e l'ottimizzazione del codice.

Una volta che il consumer delle sonde abbia terminato le sue operazioni vengono rimossi i rispettivi ECB. Nel caso in cui non ci siano ECB che utilizzano una sonda viene rimossa anche la sonda. Il codice viene quindi riscritto per rimuovere la chiamata `dtrace_probe` e ripristinare il codice originario. In questo modo il sistema è esattamente lo stesso di prima della creazione della sonda e dopo la sua distruzione è proprio come se l'attività di quella sonda non fosse mai esistita.

Per evitare danni al sistema, DTrace si preoccupa di fare in modo che le sonde non utilizzino troppa memoria né troppa capacità del processore. I buffer utilizzati per conservare i risultati delle analisi (le attività della sonda) vengono monitorati per verificare che non eccedano la dimensione massima constentita. Anche il tempo che il processore impiega a eseguire l'analisi è monitorato. Se si superano i limiti, il consumer e le sonde dannose vengono eliminate. Per evitare conflitti e perdite di dati, si allocano buffer dedicati per ogni processore.

Un esempio di codice D e del suo output ne illustra alcuni vantaggi. Il programma seguente mostra il codice DTrace che attiva sonde dello scheduler e registra il tempo di CPU utilizzato dai processi aventi ID utente 101 mentre le sonde sono attive (ossia mentre il programma è in esecuzione):

```
sched:::on-cpu
uid == 101
{
    self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
    @time[execname] = sum(timestamp - self->ts);
    self->ts = 0 };
}
```

La Figura 2.23 mostra il risultato del programma, cioè quali sono i processi e per quanto tempo (in nanosecondi) occupano il processore.

Poiché DTrace fa parte del sistema operativo open-source Solaris 10, lo si può incorporare in altri sistemi operativi qualora non vi siano conflitti fra le licenze. Ad esempio,

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
    gnome-setting-d                142354
    gnome-vfs-deamon               158243
    dsdm                           189804
    wnck-applet                    200030
    gnome-panel                    277864
    clock-applet                   374916
    mapping-deamon                 385475
    xscreensaver                   514177
    metacity                       539281
    Xorg                          2579646
    gnome-terminal                5007269
    mixer_applet2                 7388447
    java                         10769137
```

Figura 2.23 Risultato del codice D.

DTrace è già stato aggiunto a Mac OS X 10.5 e a FreeBSD. È probabile che esso si diffonderà ulteriormente in futuro per merito delle sue capacità uniche. Anche altri sistemi operativi, in special modo i derivati da Linux, si stanno attrezzando per incorporare funzionalità di tracciamento del kernel. Altri sistemi stanno cominciando a includere strumenti di monitoraggio delle prestazioni e di tracciamento promossi dalla ricerca in vari istituti, come nel caso del progetto Paradyn.

2.10 Generazione di sistemi operativi

Un sistema operativo si può progettare, codificare e realizzare specificamente per una singola macchina; tuttavia, è più diffusa la pratica di progettare sistemi operativi da impiegare in macchine di una stessa classe con configurazioni diverse. Il sistema si deve quindi configurare o generare per ciascuna situazione specifica, un processo talvolta noto come **generazione di sistemi** (SYSGEN).

I sistemi operativi sono normalmente distribuiti per mezzo di dischi o CD-ROM. Per generare un sistema è necessario usare un programma speciale che può leggere da un file o richiedere all'operatore le informazioni riguardanti la configurazione specifica del sistema o anche esplorare il sistema di calcolo per determinarne i componenti. A questo scopo sono necessarie le seguenti informazioni.

- La CPU che si deve impiegare e le opzioni installate (serie di istruzioni estese, aritmetica in virgola mobile, e così via). Nel caso di sistemi multiprocessore occorre anche descrivere ciascuna di loro.
- Come verrà formattato il disco d'avvio? In quante sezioni o "partizioni" verrà suddiviso e che cosa ci sarà in ciascuna partizione?
- La quantità di memoria disponibile. Alcuni sistemi determinano autonomamente questi valori, accedendo a tutte le locazioni della memoria, fino alla generazione di un errore di indirizzo illegale. Questa procedura definisce l'indirizzo legale finale e quindi la quantità di memoria disponibile.

- I dispositivi disponibili. Il sistema deve conoscere gli indirizzi di ogni dispositivo e sapere come farvi riferimento (numero di dispositivo), deve conoscere il numero del segnale d'interruzione del dispositivo, il tipo e il modello del dispositivo e tutte le sue caratteristiche specifiche.
- Le opzioni del sistema operativo richieste o i valori dei parametri che è necessario usare. Queste informazioni possono contenere il numero delle aree di memoria da usare per le operazioni di I/O e la loro dimensione, l'algoritmo di scheduling della CPU richiesto, il numero massimo di processi da sostenere, e così via.

Una volta ottenute, queste informazioni si possono impiegare in modi diversi. In un caso limite, un amministratore di sistema le potrebbe usare per modificare una copia del codice sorgente del sistema operativo, che si dovrebbe poi ricompilare. Dichiarazioni di dati, inizializzazioni e costanti, insieme con una compilazione condizionale, produrrebbero una versione in codice di macchina del sistema operativo specifica per il sistema desiderato.

Per ottenere una versione meno specifica ma comunque adatta al sistema desiderato, la descrizione del sistema può determinare la creazione di tabelle e la selezione di moduli da una libreria precompilata, che si collegano per formare il sistema operativo richiesto. La selezione permette alla libreria di contenere i driver di tutti i dispositivi di I/O previsti, sebbene solo quelli effettivamente necessari siano inclusi nel sistema operativo richiesto. Poiché non si ricompila il sistema, la sua generazione risulta più rapida, ma il sistema ottenuto può essere inutilmente generale.

Un altro caso limite è rappresentato dalla costruzione di un sistema completamente controllato mediante tabelle. In questo caso tutto il codice è sempre parte del sistema e la selezione si effettua al momento dell'esecuzione del sistema stesso, anziché nella fase della compilazione o del collegamento. La generazione del sistema implica semplicemente la creazione di tabelle idonee a descrivere il sistema stesso. Le differenze più rilevanti tra questi metodi riguardano la dimensione e la generalità del sistema ottenuto, oltre alla facilità di apportarvi modifiche in seguito a cambiamenti della sua struttura fisica. Si consideri, per esempio, il costo delle modifiche da apportare al sistema per consentirgli la gestione di un nuovo terminale grafico o di un'altra unità a disco. I costi vanno ovviamente bilanciati con la frequenza delle modifiche.

2.11 Avvio del sistema

Dopo che un sistema operativo è stato scritto, bisogna predisporlo all'uso da parte dei dispositivi fisici. Ma come fa l'apparato fisico dell'elaboratore a sapere dove si trova il kernel e a caricarlo? La procedura d'avviamento di un calcolatore attraverso il caricamento del kernel è nota come **avviamento** (*booting*) del sistema: nella maggior parte dei sistemi di calcolo c'è un piccolo segmento di codice, noto come **programma d'avvio** (*bootstrap program*) o **caricatore d'avvio** (*bootstrap loader*), che individua il kernel, lo carica in memoria e ne avvia l'esecuzione. Alcuni sistemi, come i PC, eseguono tale compito in due fasi: un caricatore d'avvio molto semplice preleva dal disco un più complesso programma d'avvio, che a sua volta carica il kernel.

Quando una CPU sta per entrare in funzione – per esempio, quando l'elaboratore viene acceso o riavviato – il registro delle istruzioni è caricato con una locazione di memoria predefinita, da cui ha inizio l'esecuzione. Il programma di avvio inizia da questa locazione.

Esso è contenuto in una **memoria a sola lettura** (**read-only memory,** ROM), poiché non si conosce lo stato della RAM all'avvio del sistema; inoltre, la ROM presenta il vantaggio di non dover essere inizializzata e di essere immune ai virus.

Il programma di avvio può effettuare operazioni di vario genere. Una di queste, solitamente, sottopone a diagnosi la macchina per ottenere informazioni sul suo stato. Se la diagnostica dà esito positivo, il programma è in grado di proseguire con le altre fasi di avvio. Esso può anche inizializzare il sistema in ogni suo elemento, dai registri della CPU ai controllori dei dispositivi, fino ai contenuti della memoria centrale. Presto o tardi, comunque, farà partire il sistema operativo.

Alcuni sistemi, come i telefoni cellulari, i PDA e le console per videogiochi, memorizzano l'intero sistema operativo nella ROM. La scelta di custodire nella ROM il sistema operativo si addice a sistemi di piccole dimensioni, con dispositivi fisici modesti e un funzionamento tutt'altro che sofisticato. Questa soluzione comporta un problema, cioè la necessità di modificare i circuiti ROM al fine di poter modificare il codice del programma di avvio. Alcuni sistemi ovviano a questo inconveniente utilizzando la **memoria a sola lettura programmabile e cancellabile** (EPROM), che è appunto a sola lettura, ma può diventare riscrivibile qualora riceva un comando apposito. Tutte le forme di ROM sono anche dette **firmware,** in considerazione delle loro caratteristiche, che sono un ibrido tra hardware e software. Un problema sollevato dal firmware, in genere, concerne l'esecuzione del codice, più lenta di quanto avvenga con la RAM. Taluni sistemi memorizzano il sistema operativo nel firmware e lo copiano nella RAM per eseguirlo rapidamente. Ancora, una pecca del firmware è il suo essere relativamente costoso, una circostanza per cui, di solito, è disponibile in piccole quantità.

Per sistemi operativi di grandi dimensioni (tra cui Windows, Mac OS X, UNIX e quasi tutti quelli a carattere generale) o per sistemi che cambiano di frequente, il caricatore di avvio è memorizzato nel firmware e il sistema operativo risiede su disco. In questo caso, il programma di avvio applica gli strumenti di diagnosi e utilizza una parte di codice per la lettura di un blocco singolo che occupa una locazione fissa del disco (per esempio, il blocco zero); quindi, lo trasferisce in memoria per eseguire il codice da quel **blocco di avvio** (*boot block*). Il programma custodito dal blocco di avvio è a volte abbastanza complesso per caricare in memoria l'intero sistema operativo e dare avvio alla sua esecuzione. Più spesso, è un programma semplice (deve risiedere in un singolo blocco del disco) che conosce unicamente la lunghezza e l'indirizzo sul disco del codice residuo di cui è composto l'intero programma d'avvio. Tutto il codice di avviamento destinato al disco, e il sistema operativo stesso, può essere sostituito facilmente, scrivendone nuove versioni sul disco. Un disco che contenga una partizione di avvio è chiamato **disco di avvio** (*boot disk*) o **disco di sistema;** si veda il Paragrafo 12.5.1 sull'argomento.

Una volta caricato il programma di avvio completo, esso può addentrarsi nel file system per localizzare il kernel, così da caricarlo in memoria e dare inizio alla sua esecuzione. È solo a questo punto che il sistema può essere considerato in funzione (**running**).

2.12 Sommario

I sistemi operativi offrono diversi servizi: al livello più basso, le chiamate di sistema permettono al programma in esecuzione di fare richieste direttamente al sistema operativo; a un livello superiore, l'interprete dei comandi (in alcuni ambiti noto come *shell*) mette a disposizione un meccanismo che consente a un utente di impartire una richiesta senza scrivere un

programma. I comandi possono provenire da file, in un'esecuzione a lotti (*batch*) oppure direttamente da una tastiera, in modo interattivo o a partizione del tempo. I programmi di sistema offrono agli utenti i servizi più comuni.

I tipi di richieste variano secondo il livello delle stesse richieste. Il livello cui appartengono le chiamate di sistema deve offrire le funzioni di base, come quelle di controllo dei processi e gestione di file e dispositivi. Le richieste di livello superiore, soddisfatte dall'interprete dei comandi o dai programmi di sistema, sono tradotte in una sequenza di chiamate di sistema. I servizi di sistema si possono classificare in diverse categorie: controllo dei programmi, richieste di stato e richieste di I/O. Gli errori dei programmi si possono considerare richieste di servizio implicite.

Una volta definiti i servizi del sistema è possibile passare allo sviluppo della struttura del sistema operativo. Per registrare le informazioni che definiscono lo stato del calcolatore e lo stato dei processi del sistema occorrono diverse tabelle.

La progettazione di un nuovo sistema operativo è un compito molto difficile. Gli scopi del sistema si devono definire chiaramente prima di iniziare la progettazione; costituiscono la base da cui partire per poter scegliere tra le varie strategie e i vari algoritmi necessari.

Poiché un sistema operativo è di grandi dimensioni, la modularità è un altro fattore importante. La progettazione di un sistema come una sequenza di strati o l'uso di un microkernel sono considerati buone tecniche. Il concetto di macchina virtuale tiene in grande considerazione il metodo basato sulla stratificazione e tratta il kernel del sistema operativo come se facesse parte della macchina fisica. Su questa macchina virtuale si possono caricare persino altri sistemi operativi.

In tutto il ciclo di progettazione del sistema operativo, occorre prestare attenzione alla distinzione tra la scelta dei criteri e i dettagli dei meccanismi adottati. In questo modo si ottiene la massima flessibilità, che all'occorrenza consente di modificare più facilmente i criteri adottati.

Ormai i sistemi operativi sono quasi tutti scritti in un linguaggio per lo sviluppo di sistemi o in un linguaggio di alto livello; questa caratteristica facilita realizzazione, manutenzione e adattabilità a sistemi diversi.

Il processo di debugging e i guasti nel kernel possono essere studiati grazie all'utilizzo di debugger e di altri strumenti in grado di analizzare un'immagine dello stato della memoria. Strumenti quali DTrace analizzano i sistemi di produzione per trovare colli di bottiglia e capire altri comportamenti del sistema.

All'avvio di un calcolatore, la CPU deve eseguire il programma d'avvio residente nel firmware. Se l'intero sistema operativo risiede nel firmware, all'accensione l'intero sistema è eseguibile direttamente; altrimenti, il ciclo di avvio della macchina procede per fasi progressive, a ognuna delle quali si caricano in memoria, dal firmware e dal disco, porzioni sempre più potenti del sistema operativo, fino a eseguire l'intero sistema stesso.

Esercizi pratici

2.1 Qual è lo scopo delle chiamate di sistema?

2.2 Quali sono le cinque attività principali di un sistema operativo dal punto di vista della gestione dei processi?

2.3 Quali sono le tre attività principali di un sistema operativo dal punto di vista della gestione della memoria?

2.4 Quali sono le tre attività principali di un sistema operativo dal punto di vista della memoria secondaria?

2.5 Qual è lo scopo dell'interprete dei comandi? Perché è solitamente separato dal kernel?

2.6 Quali chiamate di sistema devono essere eseguite dall'interprete dei comandi, o shell, per avviare un nuovo processo?

2.7 Qual è lo scopo dei programmi di sistema?

2.8 Qual è il vantaggio principale dell'approccio a strati (layer) all'architettura di sistema? Quali sono invece i suoi svantaggi?

2.9 Elencate cinque servizi forniti da un sistema operativo e spiegate la convenienza per l'utente di ciascuno. In quali casi sarebbe impossibile per i programmi a livello utente offrire questi servizi? Argomentate la vostra risposta.

2.10 Perché alcuni sistemi memorizzano il sistema operativo nel firmware mentre altri lo memorizzano su disco?

2.11 Come potrebbe essere progettato un sistema perché offra la possibilità di scegliere quale sistema operativo avviare? Che cosa dovrebbe fare in questo caso il bootstrap?

Esercizi

2.12 I servizi e le funzioni offerti da un sistema operativo possono essere divisi in due categorie. Procedete a una loro breve descrizione, analizzandone le differenze.

2.13 Descrivete tre metodi generali per passare parametri al sistema operativo.

2.14 Descrivete come si possa ottenere un profilo statistico del tempo consumato da un programma per eseguire le differenti parti del proprio codice. Argomentate l'importanza di simili profili statistici.

2.15 Quali sono le cinque attività principali di un sistema operativo relative alla gestione dei file?

2.16 Quali sono i vantaggi e gli svantaggi di usare la medesima interfaccia alle chiamate di sistema sia per i file sia per i dispositivi?

2.17 Sarebbe possibile per l'utente sviluppare un nuovo interprete dei comandi utilizzando le chiamate di sistema offerte dal sistema operativo?

2.18 Quali sono i due modelli della comunicazione tra processi? Quali i loro punti di forza e di debolezza?

2.19 Perché è auspicabile separare i meccanismi dai criteri o politiche?

2.20 Talvolta è difficile realizzare un'architettura a strati se due componenti del sistema operativo dipendono l'uno dall'altro. Identificate una situazione in cui non risulti immediatamente evidente come stratificare due componenti del sistema e nel contempo mantenere strettamente connesse le rispettive funzionalità.

2.21 Quale vantaggio si riscontra nell'architettura orientata al microkernel? In che modo interagiscono i programmi utenti e i servizi del sistema in tale architettura? Quali sono gli svantaggi?

2.22 Per quali versi la strategia del kernel modulare è simile alla strategia stratificata? Per quali aspetti la prima si differenzia dalla seconda?

2.23 Dite qual è il vantaggio principale che ottengono i progettisti di un sistema operativo che impiega un'architettura a macchine virtuali, e qual è il vantaggio principale per gli utenti.

2.24 Spiegate perché un compilatore istantaneo (*just-in-time*) è utile per l'esecuzione dei programmi scritti in Java.

2.25 Che tipo di relazione sussiste tra un sistema operativo ospite e un sistema operativo residente in un ambiente quale VMware? Quali fattori devono essere valutati nella scelta del sistema operativo residente?

2.26 Il sistema operativo sperimentale Synthesis ha un assemblatore incorporato nel kernel. Per ottimizzare le prestazioni delle chiamate di sistema, il kernel assembla le procedure nello spazio del kernel, al fine di ridurre al minimo il percorso che le chiamate di sistema devono seguire attraverso il kernel. Tale metodo è in antitesi al metodo stratificato che, pur rendendo più semplice la costruzione di un sistema operativo, determina un prolungamento del percorso delle chiamate di sistema attraverso il kernel. Valutate i pro e i contro di tale metodo nella progettazione di un kernel e nell'ottimizzazione delle prestazioni di un sistema.

Problemi di programmazione

2.27 Nel Paragrafo 2.3 si è descritto un programma che copia i contenuti di un file di origine in un file di destinazione. Questo programma esordisce con la richiesta, indirizzata all'utente, dei nomi dei file di origine e di destinazione. Si scriva un tale programma usando Windows 32 o mediante la API del POSIX. Prestate particolare attenzione alla gestione degli errori, assicurandovi che il file di origine esista. Fatto ciò, eseguite il programma insieme a un'applicazione per la tracciatura delle chiamate di sistema, se si dispone di un sistema operativo con tale funzionalità. In ambiente Linux è disponibile l'applicazione `ptrace`, mentre i sistemi Solaris ricorrono ai comandi `truss` o `dtrace`. Una funzionalità simile è fornita, per il Mac OS X, dall'istruzione `ktrace`. Dato che i sistemi Windows non offrono queste caratteristiche, è necessario eseguire il tracciamento della versione Win32 del programma utilizzando un debugger.

Progetti di programmazione

2.28 **Introduzione di una chiamata di sistema nel kernel di Linux**

In questo progetto si studierà l'interfaccia alle chiamate di sistema fornita da Linux, analizzando come i programmi utenti, attraverso questa interfaccia, possano comunicare con il kernel del sistema operativo. Il compito del lettore è incorporare una nuova chiamata di sistema all'interno del kernel, aumentando così le funzionalità del sistema operativo.

Parte 1: Per cominciare

Una chiamata di procedura in modalità utente si implementa passando gli argomenti alla funzione chiamata o tramite la pila (*stack*) oppure tramite registri, salvando lo stato corrente e il valore nel registro contatore di programma (*program counter*) e saltando all'inizio del codice che corrisponde alla procedura chiamata. Il processo continua a mantenere i privilegi di cui godeva in precedenza.

Le chiamate di sistema appaiono come chiamate di procedura rivolte a programmi utenti, ma finiscono per modificare i privilegi, determinando un diverso contesto di esecuzione. Se Linux gira su Intel 386, l'attuazione di una chiamata di sistema consiste nel memorizzare il suo numero identificativo nel registro EAX, collocare gli argomenti della chiamata in altri registri, e nell'eseguire un'istruzione di eccezione (che è l'istruzione `0x80` dell'assembly INT). Una volta sollevata l'eccezione, il numero della chiamata di sistema è usato come indice in una tabella di puntatori al codice, per ottenere l'indirizzo iniziale del codice che implementa il gestore della chiamata di sistema. Il processo, allora, salta a questo indirizzo, e i suoi privilegi subiscono una mutazione, dalla modalità utente alla modalità di sistema. In seguito all'ampliamento dei privilegi, il processo può ora eseguire il codice del kernel, che potrebbe contenere istruzioni privilegiate altrimenti non eseguibili. Il codice del kernel può dunque attuare i servizi richiesti come, per esempio, l'interazione con i dispositivi di I/O e la gestione dei processi, e può svolgere altre attività di questo genere che, nella modalità utente, sarebbero precluse.

I numeri delle chiamate di sistema relativi alle ultime versioni del kernel Linux sono riportati in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (Per esempio `__NR_close`, che corrisponde alla chiamata di sistema `close()` per la chiusura di un descrittore di file, ha il numero 6.) L'elenco dei puntatori ai gestori delle chiamate di sistema è memorizzato, in genere, nel file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` sotto l'intestazione `ENTRY(sys_call_table)`. Si noti come `sys_close` sia memorizzato nella posizione numero 6 della tabella per essere coerente con il numero della chiamata di sistema definito nel file `unistd.h` (La parola chiave `.long` denota che l'elemento occuperà lo stesso numero di byte di un dato di tipo `long`.)

Parte 2: Costruzione di un nuovo kernel

Prima di poter aggiungere una chiamata di sistema al kernel, il lettore dovrà acquisire dimestichezza con la traduzione del codice sorgente in codice binario, e con la procedura di riavvio della macchina con il nuovo kernel. Queste attività prevedono le seguenti operazioni, alcune delle quali dipendono dalla specifica distribuzione di Linux.

- Procurarsi il sorgente del kernel per la distribuzione di Linux in questione. Se il pacchetto con il codice sorgente è stato già installato sulla macchina del lettore, i file corrispondenti potrebbero essere disponibili in `usr/src/linux` o in `/usr/src/linux-2.x` (dove il suffisso identifica la versione del kernel). Qualora il pacchetto non sia stato precedentemente installato, può essere scaricato dal fornitore della vostra versione di Linux o da: http://www.kernel.org.
- Apprendere come si configura, compila e installa il codice binario del kernel. A causa delle differenze tra le varie versioni, queste operazioni non sono generalizzabili, ma alcuni tipici comandi per costruire il kernel (dalla directory in cui si trova il codice sorgente) sono:
  - `make xconfig`
  - `make dep`
  - `make bzImage`
- Aggiungere un nuovo elemento alla serie di kernel avviabili presenti nel sistema. In genere il sistema operativo Linux si avvale di istruzioni quali `lilo` e `grub` per mantenere una lista di kernel avviabili, da cui l'utente può selezionare durante la fase di avvio della macchina. Se il vostro sistema prevede `lilo`, aggiungete un elemento a `lilo.conf`, quale:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

dove `/boot/bzImage.mykernel` è l'immagine del kernel, mentre `mykernel` è il nome simbolico associato al nuovo kernel, da selezionare durante la fase di avviamento. In questo modo si avrà la possibilità di avviare il nuovo kernel o, in alternativa, il vecchio kernel non sottoposto a modifiche, nel caso in cui il nuovo non funzioni come dovrebbe.

Parte 3: Estensione del sorgente del kernel

A questo punto il lettore potrà sperimentare l'inserimento di un nuovo file nel gruppo dei file sorgente utilizzati per la compilazione del kernel. Il codice sorgente è spesso memorizzato nella directory `/usr/src/linux-2.x/kernel`, sebbene questa collocazione potrebbe cambiare da una versione all'altra di Linux. Per aggiungere una chiamata di sistema vi sono due opzioni. Una è di annettere la chiamata a un file sorgente preesistente di questa directory. L'altra consiste nella creazione di un file inedito nella directory, modificando `/usr/src/linux-2.x/kernel/Makefile` di modo che includa questo nuovo file nel processo di compilazione. Il vantaggio della prima soluzione è che, agendo su un file esistente, e dunque già incluso nel processo di compilazione, non è necessario modificare `Makefile`.

Parte 4: Introduzione di una chiamata di sistema al kernel

Dopo aver preso confidenza con le varie operazioni propedeutiche alla costruzione e all'avvio dei kernel in ambiente Linux, il lettore potrà ora dedicarsi all'introduzione di una nuova chiamata di sistema nel kernel di Linux. In questo progetto la chiamata di sistema avrà funzionalità contenute; si limiterà semplicemente a passare dalla modalità utente a quella di sistema, stampare un messaggio che viene conservato nell'apposito spazio previsto dal kernel, e ritornare quindi alla modalità utente. Chiameremo questa chiamata di sistema *helloworld*. Pur avendo effetti limitati, essa illustra la dinamica delle chiamate di sistema, facendo luce sull'interazione tra i programmi utenti e il kernel.

- Create un nuovo file denominato `helloworld.c` per definire la chiamata di sistema. Includete i file di intestazione `linux/linkage.h` e `linux/kernel.h`. Si aggiunga a questo file il codice seguente:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys helloworld() {
   printk(KERN EMERG "hello world!");

   return 1;
}
```

Otterrete come risultato una chiamata di sistema dal nome `sys_helloworld()`. Se deciderete di aggregare questa chiamata di sistema a un file già esistente nella directory con il codice sorgente, sarà sufficiente aggiungere la funzione `sys_helloworld()` al file seleziona-

to. La presenza di `asmlinkage`, risalente all'epoca in cui Linux adoperava non solo il C++ ma anche il C, segnala che il codice è scritto in C. La funzione `printk()` serve per stampare messaggi su un file di log gestito dal kernel, e pertanto può unicamente essere invocata dal kernel stesso. I messaggi del kernel specificati nel parametro di `printk()` sono registrati nel file di log `/var/log/kernel/warnings`. Il prototipo della funzione `printk()` è definito in `/usr/include/linux/kernel.h`.

- Definite un nuovo numero della chiamata di sistema per `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Un programma utente può servirsi di tale numero per identificare la nuova chiamata inserita. Accertatevi, inoltre, di aumentare di uno il valore di `__NR_syscalls`, anch'esso custodito nello stesso file, e volto a tenere traccia del numero di chiamate di sistema presenti nel kernel.
- Introducete l'elemento `.long sys_helloworld.c` nella `sys_call_table` definita nel file `/usr/src/linux-2.x/arch/i386/kernel/entry.S`. Come già chiarito, il numero della chiamata di sistema funge da indice in una tabella per individuare la posizione del codice gestore della chiamata di sistema invocata.
- Aggiungete il vostro file `helloworld.c` al `Makefile` (qualora si sia creato un nuovo file per la chiamata di sistema). Salvate una copia dell'immagine binaria del vecchio kernel (come precauzione per eventuali problemi con il nuovo). Procedete ora alla costruzione del nuovo kernel; rinominatelo per distinguerlo dal kernel originario e aggiungete una posizione ai file di configurazione del caricatore (per esempio, a `lilo.conf`). Dopo aver completato questi passaggi, si potrà finalmente avviare o il kernel originario oppure il nuovo, che ospita la nuova chiamata di sistema.

Parte 5: Uso della chiamata di sistema da un programma utente

All'avvio della macchina con il nuovo kernel, la nuova chiamata di sistema sarà pronta all'uso; ora si tratta semplicemente di invocarla da un programma utente. La libreria standard del C prevede normalmente un'interfaccia alle chiamate di sistema Linux. Poiché la nuova funzione non è collegata alla libreria standard del C, per invocare la chiamata di sistema si dovrà ricorrere a un intervento manuale.

Come notavamo prima, invocare una chiamata di sistema equivale a memorizzare il valore appropriato in un registro hardware e sollevare un'eccezione. Queste operazioni di basso livello, tuttavia, non si prestano a essere compiute con la sintassi del linguaggio C, richiedendo invece istruzioni assembly. Fortunatamente Linux dispone di macro per istanziare funzioni che racchiudono le istruzioni assembly appropriate. Per esempio, il seguente programma in C utilizza la macro `_syscall0()` per invocare la nuova chiamata di sistema:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>
_syscall0(int, helloworld);
main()
{
   helloworld();
}
```

- La macro `_syscall0` accetta due argomenti. Il primo specifica il tipo del valore restituito dalla chiamata di sistema: il secondo è il nome della chiamata di sistema. Il nome viene utilizzato per identificare il numero della chiamata di sistema, memorizzato nel registro prima che sia sollevata l'eccezione.

  Se la chiamata di sistema richiede più argomenti, si potrà impiegare una macro differente (come `_syscall1`, il cui suffisso indica il numero di argomenti) per istanziare il codice assembly necessario a effettuare la chiamata di sistema.

- Compilate ed eseguite il programma con il kernel di nuova creazione. Il messaggio "`hello,world!`" nel file di log `/var/log/kernel/warnings` del kernel indicherà l'avvenuta esecuzione della chiamata di sistema.

Come possibile prosieguo del progetto, si rifletta su come ampliare le funzionalità della nuova chiamata di sistema. Come si potrebbe passare un valore intero o una costante carattere alla chiamata di sistema per farlo successivamente stampare nel file di log del kernel? Quali implicazioni comporterebbe il passaggio di puntatori ai dati memorizzati nello spazio degli indirizzi del programma utente rispetto al passaggio di un semplice valore intero dal programma utente al kernel usando i registri hardware?

2.13 Note bibliografiche

L'orientamento stratificato alla progettazione dei sistemi operativi è stato sostenuto da [Dijkstra 1968]. [Brinch-Hansen 1970] è stato uno dei primi sostenitori della costruzione di un sistema operativo intorno a un nucleo (il kernel), sulla base del quale sviluppare sistemi più completi.

La strumentazione del sistema e la tracciatura dinamica sono descritti in [Tamches e Miller 1999]. DTrace è trattato in [Cantrill et al. 2004]. [Cheung e Loong 1995] affrontano la questione della strutturazione di un sistema operativo, dal microkernel ai sistemi estendibili.

L'MS-DOS, Versione 3.1, è descritto in [Microsoft 1986]. Windows NT e Windows 2000 sono esaminati in [Solomon 1998], nonché in [Solomon e Russinovich 2000] Il Berkeley UNIX (BSD) è descritto in [McKusick et al. 1996]. [Bovet e Cesati 2002] dissertano ampiamente sul kernel di Linux. Diversi sistemi UNIX – tra cui Mach – sono esaminati in dettaglio da [Vahalia 1996]. Il Mac OS X è presentato all'indirizzo http://www.apple.com/macosx. Il sistema sperimentale Synthesis è presentato in [Massalin e Pu 1989]. Solaris è dettagliatamente descritto da [Mauro e McDougall 2001].

Il primo sistema operativo a offrire una macchina virtuale è stato il CP/67 su un IBM 360/67. Il sistema operativo IBM VM/370 disponibile in commercio deriva dal CP/67. Dettagli sul sistema operativo Mach, basato su microkernel, sono reperibili in [Young et al. 1987]. [Kaashoek et al. 1997] riferiscono in merito ai sistemi operativi basati sul cosiddetto esokernel, la cui architettura tiene separate le questioni di protezione dalla gestione ordinaria, tollerando così il controllo indiscriminato delle risorse da parte di programmi inaffidabili.

Le specifiche del linguaggio Java e della relativa macchina virtuale sono presentate in [Gosling et al. 1996] e in [Lindholm e Yellin 1999], rispettivamente. Il funzionamento interno della JVM è descritto in [Venners 1998]. [Golm et al. 2002] si soffermano sul sistema operativo JX; [Back et al. 2000] evidenziano numerose tematiche di rilievo per la progettazione dei sistemi operativi Java. Ulteriori informazioni sul linguaggio Java sono disponibili all'indirizzo http://www.javasoft.com. Un approfondimento sull'implementazione di VMware è reperibile in [Sugerman et al. 2001]. All'indirizzo http://www.wmware.com/appliances/learn/ovf.html si possono trovare informazioni sul progetto Open Virtual Machine Format.