Capitolo e12

Progettazione di un sistema operativo

Negli undici capitoli precedenti abbiamo percorso un lungo cammino, considerato molti argomenti e accennato a molti concetti ed esempi relativi ai sistemi operativi. Studiare i sistemi operativi esistenti è tuttavia diverso dal progettarne uno nuovo. In questo capitolo diamo un rapido sguardo ad alcune questioni e compromessi che chi progetta i sistemi operativi deve considerare nella progettazione e implementazione di un nuovo sistema.

Nella comunità dei sistemi operativi circola ogni sorta di idee bizzarre a proposito di che cosa sia bene e di che cosa sia male, ma per iscritto è stato messo sorprendentemente poco. Probabilmente il libro principale è il classico *The Mythical Man Month* di Fred Brooks, nel quale egli racconta la sua esperienza nel progettare e implementare IBM OS/360. L'edizione del ventesimo anniversario rivede una parte del materiale e aggiunge quattro nuovi capitoli (Brooks, 1995). Tre pubblicazioni classiche sulla progettazione dei sistemi operativi sono "Hints for Computer System Design" (Lampson, 1984), "On Building Systems That Will Fail" (Corbató, 1991) ed "End-to-End Arguments in System" (Saltzer et al., 1984). Come il libro di Brooks, tutte queste tre pubblicazioni sono sopravvissute estremamente bene al trascorrere degli anni: la maggior parte delle loro opinioni è ancora molto valida come lo era quando esse furono pubblicate la prima volta. Questo capitolo attinge da tali fonti, unitamente all'esperienza personale dell'autore come progettista o coprogettista di due sistemi operativi: Amoeba (Tanenbaum et al., 1990) e MINIX (Tanenbaum e Woodhull, 2006). Dato che non esiste una convergenza di idee fra gli ideatori di sistemi operativi circa la migliore progettazione possibile, questo capitolo risulterà più personale, speculativo e indubbiamente più controverso dei precedenti.

12.1 Natura del problema della progettazione

Il progetto di un sistema operativo è più un problema di ingegneria che una scienza esatta. È molto più difficile definire degli obiettivi che raggiungerli. Partiamo proprio da questi punti.

12.1.1 Obiettivi

Per progettare un sistema operativo di successo, i progettisti devono avere un'idea chiara di quello che vogliono. La mancanza di obiettivi rende molto difficile prendere successivamente

decisioni. Per chiarire questo punto, è istruttivo dare uno sguardo a due linguaggi di programmazione, PL/I e C. PL/I è stato progettato da IBM negli anni Sessanta, in quanto era una seccatura dover supportare sia FORTRAN sia COBOL ed era imbarazzante che gli accademici cianciassero del fatto che Algol fosse superiore a entrambi. Fu quindi creato un comitato per produrre un linguaggio che accontentasse tutti: PL/I. Aveva un po' di FORTRAN, un po' di COBOL e un po' di Algol. Fallì perché perse qualsiasi visione d'insieme. Era semplicemente una raccolta di caratteristiche in contraddizione l'una con l'altra e troppo ingombrante per poter essere compilato e usato in modo efficiente.

Consideriamo adesso il linguaggio C. È stato progettato da una sola persona (Dennis Ritchie) con uno scopo (la programmazione dei sistemi). Il suo enorme successo, non in piccola parte, è dovuto al fatto che Ritchie sapeva ciò che voleva e ciò che non voleva. Il risultato è che è ancora usato diffusamente a più di quarant'anni dalla sua comparsa. Avere una visione chiara di ciò che si vuole è cruciale.

Che cosa vogliono i progettisti di sistemi operativi? Ovviamente la risposta varia da sistema a sistema, essendo diversa per sistemi integrati (*embedded*) rispetto a sistemi server. Tuttavia, relativamente ai sistemi operativi general-purpose, vengono in mente quattro punti:

1. definizione delle astrazioni;
2. disponibilità delle operazioni primitive;
3. assicurazione dell'isolamento;
4. gestione dell'hardware.

Ognuno di questi punti sarà trattato nel seguito.

L'attività principale, probabilmente la più difficile per un sistema operativo, è quella di definire le giuste astrazioni. Alcune di esse, come i processi, gli spazi degli indirizzi e i file, sono in circolazione da così tanto tempo da poter sembrare ovvie. Altre, come i thread, sono più recenti e meno mature. Per esempio, se un processo multithread che ha un thread bloccato in attesa di input da tastiera esegue una fork, nel nuovo processo esisterà un thread anch'esso in attesa dell'input della tastiera? Altre astrazioni sono relative alla sincronizzazione, ai segnali, al modello della memoria, alla modellazione dell'I/O e a molte altre aree.

Ognuna delle astrazioni può essere istanziata sotto forma di strutture dati concrete. Le operazioni primitive manipolano queste strutture dati. Per esempio, gli utenti possono leggere e scrivere i file. Le operazioni primitive sono implementate sotto forma di chiamate di sistema. Dal punto di vista dell'utente, il cuore del sistema operativo è costituito dalle astrazioni e dalle operazioni su di esse disponibili attraverso le chiamate di sistema.

Considerato che più utenti possono essere collegati a un computer nello stesso momento, il sistema operativo ha bisogno di fornire dei meccanismi per tenerli separati. Un utente non deve interferire con un altro. Il concetto di processo è utilizzato ampiamente per raggruppare risorse ai fini di protezione; i file e le altre strutture dati sono generalmente protetti allo stesso modo. Un obiettivo fondamentale della progettazione del sistema operativo è accertarsi che ogni utente possa eseguire solo operazioni autorizzate su dati autorizzati. Tuttavia, gli utenti vogliono anche condividere dati e risorse, quindi l'isolamento deve essere selettivo e sotto il controllo dell'utente, il che lo rende più difficile da mettere in pratica. Il programma di posta elettronica non dovrebbe causare problemi al browser di rete. Anche quando esiste un solo utente, i processi differenti necessitano di essere isolati.

Strettamente correlata a questo punto è la necessità di isolare gli errori. Se una qualche parte del sistema si blocca, più comunemente un processo utente, non dovrebbe essere in grado di trascinare con sé il resto del sistema. Il progetto del sistema dovrebbe assicurare che le varie parti siano ben isolate l'una dall'altra. Idealmente, le parti del sistema operativo dovrebbero essere isolate anche per consentire dei guasti indipendenti l'uno dall'altro.

Infine, il sistema operativo deve gestire l'hardware. In particolare, deve tenere conto di tutti i chip di basso livello, come i controller degli interrupt e dei bus. Deve anche fornire una struttura per permettere ai driver dei dispositivi di gestire i dispositivi di I/O più grandi, come dischi, stampanti e schermo.

12.1.2 Perché è difficile progettare un sistema operativo?

La Legge di Moore afferma che l'hardware dei computer migliora di un fattore 100 ogni decennio. Nessuno ha scritto una legge che affermi lo stesso per i sistemi operativi, o che dica almeno che migliorino in qualche modo. In effetti, gli aspetti chiave (come l'affidabilità) di alcuni di loro risultano peggiorati rispetto alla versione 7 di UNIX negli anni Settanta.

Perché? L'inerzia e la compatibilità con il passato hanno spesso la maggior parte della colpa, così come il disattendere i buoni princìpi di progettazione. Ma c'è di più. I sistemi operativi sono fondamentalmente differenti in un certo modo dai piccoli programmi applicativi venduti a 49 euro. Esaminiamo otto fra i problemi che rendono la progettazione di un sistema operativo più difficile rispetto a quella di un programma applicativo.

Primo, i sistemi operativi sono diventati programmi estremamente grandi. Nessuno può sedersi davanti a un PC e realizzare di getto un sistema operativo serio in pochi mesi, e nemmeno in pochi anni. Tutte le versioni attuali di UNIX sono composti da milioni di righe di codice; Linux ha toccato i 15 milioni, per esempio. Windows 8 è probabilmente fra i 50 e i 100 milioni di righe, secondo come le si conta (Windows Vista era intorno ai 70 milioni, ma i cambiamenti apportati hanno levato molte righe e ne hanno introdotte di nuove). Nessuno è in grado di comprendere un milione di righe di codice, figuriamoci 50 o 100. Quando si ha un prodotto che nessuno dei progettisti può sperare di comprendere del tutto, non dovrebbe sorprendere che il risultato sia spesso lontano dall'ottimale.

I sistemi operativi non sono i sistemi più complessi esistenti. Le portaerei sono di gran lunga più complicate, per esempio, ma si suddividono meglio in sottosistemi isolati. Chi progetta i servizi igienici di una portaerei non si deve preoccupare del sistema radar: i due sottosistemi non interagiscono più di tanto. In un sistema operativo, il file system spesso interagisce con il sistema della memoria in modi inattesi e imprevedibili.

Secondo, i sistemi operativi devono gestire la concorrenza. Ci sono più utenti e più dispositivi di I/O attivi tutti nello stesso momento. La gestione della concorrenza è inerentemente più difficile della gestione di singole attività sequenziali. Le condizioni competitive e i deadlock sono solo due fra i problemi che si presentano.

Terzo, i sistemi operativi devono rapportarsi a utenti potenzialmente ostili, utenti cioè che vogliono interferire con le operazioni del sistema o compiere azioni vietate, come rubare i file di altri utenti. Il sistema operativo deve prendere precauzioni per impedire agli utenti di comportarsi in modo improprio. I programmi di videoscrittura o di elaborazione di immagini non hanno questo problema.

Quarto, a dispetto del fatto che non tutti gli utenti si fidano l'uno dell'altro, molti vogliono condividere alcune loro informazioni e risorse con altri utenti selezionati. Il sistema

operativo deve renderlo possibile, ma in una modalità tale per cui gli utenti malintenzionati non possano interferire. Di nuovo, i programmi applicativi non devono affrontare nulla di simile a questa problematica.

Quinto, i sistemi operativi sopravvivono per periodi molto lunghi. UNIX è in circolazione da quarant'anni; Windows è in circolazione da più trent'anni e non mostra segni di cedimento. Di conseguenza, i progettisti devono pensare a come l'hardware e le applicazioni possano cambiare in un futuro a lungo termine e a come si dovrebbero preparare a ciò. I sistemi arroccati su una visione del mondo troppo ristretta spesso scompaiono.

Sesto, i progettisti di sistemi operativi non hanno realmente idea di come i loro sistemi saranno utilizzati, quindi devono prevedere sistemi assolutamente generici. Né UNIX né Windows sono stati progettati tenendo conto di e-mail o browser di rete, eppure molti computer che utilizzano questi sistemi fanno ben poco altro. Nessuno richiede a un progettista di navi di costruirne una senza specificare se si voglia un battello da pesca, una nave da crociera o un incrociatore da guerra, né cambia idea dopo che il prodotto è stato consegnato.

Settimo, i sistemi operativi moderni sono generalmente progettati per essere portabili, nel senso che devono funzionare su più piattaforme hardware. Devono anche supportare migliaia di dispositivi di I/O, tutti progettati in modo indipendente e senza alcuna considerazione degli altri. Un esempio in cui questa diversità causa problemi è rappresentato dal bisogno di un sistema operativo di funzionare sia su macchine *little-endian* che su macchine *big-endian*. Un secondo esempio era molto comune nell'MS-DOS quando gli utenti tentavano di installare, diciamo, una scheda audio e un modem che utilizzavano la medesima porta di I/O o lo stesso IRQ. Pochi programmi, oltre ai sistemi operativi, devono occuparsi della risoluzione dei problemi causati da componenti hardware in conflitto.

Ottavo, e ultimo di questo elenco, è la necessità frequente di essere compatibili a ritroso con alcuni sistemi operativi precedenti. Un determinato sistema può avere restrizioni sulla lunghezza delle parole, sui nomi dei file o altri aspetti che ora i progettisti considerano datati, ma che sono rimasti legati a quel sistema. È come convertire una fabbrica alla produzione di autovetture dell'anno successivo al posto delle autovetture dell'anno precedente, ma continuando a produrre quelle dell'anno in corso a pieno regime.

12.2 Progettazione delle interfacce

Dovrebbe essere chiaro fin da ora che scrivere un sistema operativo moderno non è semplice. Ma da dove si comincia? Probabilmente il miglior punto di partenza è pensare alle interfacce fornite dal sistema. Un sistema operativo fornisce un insieme di astrazioni, la maggior parte implementate da tipologie di dati (per esempio, file) e operazioni su di essi (per esempio, `read`). Insieme costituiscono l'interfaccia per gli utenti del sistema. Notate che in questo contesto gli utenti del sistema operativo sono programmatori che scrivono del codice utilizzando le chiamate di sistema, non persone che eseguono programmi applicativi.

Oltre all'interfaccia principale per le chiamate di sistema, la maggior parte dei sistemi operativi possiede interfacce aggiuntive. Per esempio, alcuni programmatori hanno bisogno di scrivere driver dei dispositivi da inserire nel sistema operativo. Questi driver vedono certe caratteristiche e possono fare determinate chiamate di procedure. Anche queste caratteristiche e queste chiamate definiscono un'interfaccia, ma molto diversa da quella che vede un

programmatore di applicazioni. Tutte queste interfacce devono essere progettate attentamente per permettere il successo del sistema.

12.2.1 Princìpi guida

Esistono princìpi che possono guidare la progettazione delle interfacce? Noi crediamo di sì. Riassunti brevemente, essi sono la semplicità, la completezza e la capacità di essere implementate efficientemente.

Principio 1: Semplicità

Un'interfaccia semplice è facile da capire e da implementare senza errori. Tutti i progettisti di sistemi dovrebbero tenere in mente questa famosa citazione dell'avviatore e scrittore francese Antoine de St.Exupéry:

> *La perfezione non si ottiene quando non c'è più nulla da aggiungere, bensì quando non c'è più nulla da togliere.*

Se vogliamo essere precisi, non ha detto esattamente così, ma:

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*

L'idea, comunque, è quella. Mandatela a memoria.

Questo principio afferma che meno è di più, almeno per quanto riguarda il sistema operativo. Un altro modo di dirlo è il principio KISS: *Keep It Simple, Stupid*.

Principio 2: Completezza

L'interfaccia deve naturalmente rendere possibile l'esecuzione di ogni cosa sia necessaria agli utenti, cioè, deve essere completa. Questo ci porta a un'altra famosa citazione, stavolta di Albert Einstein:

> *Ogni cosa dovrebbe essere semplice quanto possibile, ma non più semplice.*

In altre parole, il sistema operativo dovrebbe fare esattamente ciò che gli viene richiesto e niente di più. Se gli utenti hanno bisogno di memorizzare dei dati, il sistema deve fornire alcuni meccanismi allo scopo. Se gli utenti necessitano di comunicare fra loro, il sistema operativo deve fornire meccanismi di comunicazione e così via. Nella sua lezione successiva al conferimento del Turing Award del 1991, Fernando Corbató, uno dei progettisti di CTSS e di MULTICS, fuse tra loro i concetti di semplicità e completezza e affermò:

> *Innanzitutto è importante enfatizzare il valore della semplicità e dell'eleganza, perché la complessità crea le difficoltà e, come abbiamo visto, crea errori. La mia definizione di eleganza è il raggiungimento di una determinata funzionalità con il meccanismo minimo e il massimo della chiarezza.*

L'idea fondamentale in questo caso è il *meccanismo minimo*. In altre parole, ogni caratteristica, funzione e chiamata di sistema dovrebbe sostenersi autonomamente. Dovrebbe fare una cosa e farla bene. Quando un membro del team di progettazione propone di estendere una chiamata di sistema o di aggiungere alcune caratteristiche, gli altri membri dovrebbero chiedersi

se è così terribile decidere di non farlo. Se la risposta è: "No, ma qualcuno potrebbe trovare questa caratteristica utile in futuro," mettetela in una libreria utente, non nel sistema operativo, anche se in questo modo sarà più lenta. Non tutte le caratteristiche devono essere più veloci di un proiettile. L'obiettivo è quello di mantenere ciò che Corbató chiama *minimo dei meccanismi*.

Consideriamo brevemente due esempi attinti dalla mia personale esperienza: MINIX (Tanenbaum e Woodhull, 2006) e Amoeba (Tanenbaum et al., 1990). Per qualsiasi intento e scopo, MINIX aveva fino a non molto tempo fa soltanto tre chiamate di sistema: `send`, `receive` e `sendrec`. Il sistema è strutturato come una raccolta di processi, in cui il gestore della memoria, il file system e tutti i driver dei dispositivi sono processi separati schedulabili. In prima approssimazione, tutto quello che fa il kernel è schedulare i processi e gestire il passaggio di messaggi fra di loro. Di conseguenza, sono necessarie due sole chiamate di sistema: `send`, per inviare un messaggio, e `receive`, per riceverne. La terza chiamata, `sendrec`, è semplicemente un'ottimizzazione per ragioni di efficienza che permette a un messaggio di essere spedito e alla risposta di essere richiesta con un solo trap nel kernel. Ogni altra cosa è svolta richiedendo a un altro processo (per esempio, il processo del file system o il driver del disco) di compiere il lavoro.

La più recente versione di MINIX aggiunge due chiamate, entrambe per le comunicazioni asincrone. La chiamata `senda` invia un messaggio asincrono. Il kernel cerca di inviare il messaggio, ma l'applicazione non resta in attesa della sua ricezione: continua semplicemente l'esecuzione. Allo stesso modo, per inviare brevi notifiche il sistema utilizza una chiamata `notify`. Per esempio, il kernel può notificare a un driver di dispositivo nello spazio utente che è accaduto qualcosa, in modo molto simile a un interrupt. Non esiste un messaggio associato a una notifica: quando il kernel invia una notifica da elaborare, l'unica operazione che compie è attivare un bit in una bitmap di processo a indicare che è successo qualcosa. Poiché è molto semplice, può essere veloce e il kernel non si deve preoccupare di quale messaggio inviare se il processo riceve due volte la stessa notifica. Vale la pena osservare che, anche se il numero di chiamate è estremamente ridotto, sta aumentando. La lievitazione è inevitabile. La resistenza è inutile.

Ovviamente, queste sono solo le chiamate di sistema. L'esecuzione di un sistema compatibile POSIX su MINIX richiede l'implementazione di molte chiamate di sistema POSIX; ma il bello di questo sistema è che tutte queste chiamate sono mappate a un insieme molto piccolo di chiamate nel kernel. Con un sistema (ancora) così semplice, esiste la possibilità di farlo per bene.

Amoeba è ancora più semplice. Ha una sola chiamata di sistema: esegue chiamate di procedure remote. Questa chiamata invia un messaggio e aspetta una risposta. È essenzialmente come `sendrec` di MINIX. Tutto il resto è costruito su quest'unica chiamata. Che poi la comunicazione sincrona sia il modo più giusto per procedere è poi una questione totalmente diversa, che affronteremo nel Paragrafo 12.3.

Principio 3: Efficienza

La terza linea guida è l'efficienza dell'implementazione. Se una caratteristica o chiamata di sistema non può essere implementata efficientemente, probabilmente non vale la pena averla. Dovrebbe anche essere intuitivamente ovvio per i programmatori sapere quanto costa una chiamata di sistema. Per esempio, i programmatori UNIX si aspettano che la chiamata di

sistema `lseek` sia più economica della chiamata `read`, perché la prima cambia solo un puntatore in memoria, mentre la seconda esegue un I/O dal disco. Se i costi intuitivi sono sbagliati, i programmatori scriveranno programmi non efficienti.

12.2.2 Paradigmi

Una volta stabiliti gli obiettivi la progettazione può iniziare. Un buon punto di partenza è considerare come i fruitori vedranno il sistema. Uno dei problemi principali è come tenere bene insieme tutte le proprietà del sistema e presentare quella che è spesso denominata la **coerenza architetturale**. A questo riguardo, è importante distinguere due tipologie di "clienti" dei sistemi operativi. Da una parte ci sono gli *utenti*, che interagiscono con programmi applicativi; dall'altra parte ci sono i *programmatori*, che scrivono tali programmi. I primi hanno maggiormente a che fare con la GUI; i secondi con l'interfaccia delle chiamate di sistema. Se l'intenzione è di avere una singola GUI che pervade l'intero sistema, come in Macintosh, il progetto dovrebbe partire da lì. Se invece l'intenzione è di supportare molte possibili GUI, come in UNIX, l'interfaccia delle chiamate di sistema dovrebbe essere progettata per prima. Creare la GUI per prima è essenzialmente una progettazione dall'alto verso il basso (*top-down*).

Le questioni da prendere in considerazione riguardano le caratteristiche che dovrà avere, come l'utente interagirà con essa e come il sistema dovrà essere progettato per supportarla. Per esempio, se la maggior parte dei programmi visualizza icone sullo schermo e poi aspetta che l'utente ne selezioni una, questo suggerisce un modello guidato dagli eventi (*event-driven*) per la GUI e probabilmente anche per il sistema operativo. D'altra parte, se lo schermo è composto per lo più da finestre di testo, probabilmente un modello in cui i processi leggono dalla tastiera risulterà migliore. Creare per prima l'interfaccia per le chiamate di sistema è un progetto dal basso verso l'alto (*bottom-up*). In questo caso, i problemi riguardano la tipologia delle caratteristiche di cui i programmatori hanno in generale bisogno. Effettivamente, non servono caratteristiche speciali per supportare una GUI. Per esempio, il sistema a finestre di UNIX, X, è solo un grosso programma C che esegue `read` e `write` sulla tastiera, sul mouse e sullo schermo. X è stato sviluppato parecchio dopo UNIX e non ha richiesto molti cambiamenti al sistema operativo per farlo funzionare. Questa esperienza conferma il fatto che UNIX era sufficientemente completo.

Paradigmi dell'interfaccia utente

Sia per l'interfaccia a livello della GUI sia per l'interfaccia a livello di chiamate di sistema, l'aspetto principale è avere un buon paradigma (talvolta chiamato metafora) per comprendere l'interfaccia. Molte GUI per i computer desktop utilizzano il paradigma WIMP cui si è parlato nel Capitolo 5. Questo paradigma utilizza il *punta-e-clicca*, il *punta-e-clicca due volte*, il trascinamento e altre modalità di interazione in tutta l'interfaccia per fornire un'architettura d'insieme coerente. Spesso esistono delle richieste ulteriori per i programmi, come il presentare una barra del menu con le voci File, Modifica e altre, ognuna delle quali ha determinati elementi di menu noti. In questo modo, gli utenti che conoscono bene un programma possono apprenderne velocemente un altro.

Tuttavia, l'interfaccia utente WIMP non è la sola possibile. Tablet, smartphone e alcuni portatili usano schermi touch che consentono un'interazione più diretta e intuitiva. Alcuni computer palmari utilizzano un'interfaccia con scrittura manuale stilizzata. Dispositivi mul-

timediali dedicati possono utilizzare un'interfaccia simile ai VCR. Naturalmente l'input vocale ha un paradigma completamente diverso. L'importante non è tanto la scelta del paradigma, ma il fatto che esista un singolo paradigma predominante che unifichi l'intera interfaccia utente.

Qualunque paradigma sia scelto, è importante che tutti i programmi applicativi lo utilizzino. Di conseguenza, è necessario che i progettisti del sistema forniscano ai programmatori di applicazioni librerie e kit di strumenti che diano loro accesso alle procedure, per produrre un *look-and-feel* uniforme. Il progetto dell'interfaccia utente è molto importante, ma non è argomento di questo libro, quindi ritorniamo all'argomento dell'interfaccia del sistema operativo.

Paradigmi di esecuzione

La coerenza architetturale è importante a livello utente, ma lo è ugualmente a livello dell'interfaccia delle chiamate di sistema. A questo punto è spesso utile distinguere fra il paradigma di esecuzione e il paradigma dei dati, per cui li affronteremo entrambi a partire dal primo.

Sono diffusi due paradigmi di esecuzione: algoritmico e guidato dagli eventi. Il **paradigma algoritmico** è basato sull'idea che un programma viene eseguito perché svolga delle funzioni che conosce in anticipo o ricava dai suoi parametri. Questa funzione potrebbe essere quella di compilare un programma, fare il calcolo delle paghe o far volare un aeroplano a San Francisco. La logica di base è direttamente codificata nel codice, con il programma che fa chiamate di sistema, di tanto in tanto, per ricevere l'input dell'utente, per ottenere i servizi del sistema operativo e così via. Questo approccio è schematizzato nella Figura 12.1(a).

L'altro paradigma di esecuzione è il **paradigma guidato dagli eventi** della Figura 12.1(b). In questo caso il programma esegue un certo tipo di inizializzazione, per esempio mostrando una certa schermata, quindi attende che il sistema operativo indichi quale sia il primo evento. L'evento è spesso la pressione di un tasto o un movimento del mouse. Questo schema è utile nel caso di programmi altamente interattivi.

Ognuna di queste due modalità di agire genera il proprio stile di programmazione. Nel paradigma algoritmico, gli algoritmi sono centrali e il sistema operativo è considerato come

```
main( )
{
      int ... ;

      init( );
      do_something( );
      read(...);
      do_something_else( );
      write(...);
      keep_going( );
      exit(0);
      }
}
```

(a)

```
main( )

{
      mess_t msg;

      init( );
      while (get_message(&msg)) {
            switch (msg.type) {
                  case 1: ... ;
                  case 2: ... ;
                  case 3: ... ;
            }
      }
}
```

(b)

Figura 12.1 (a) Codice algoritmico. (b) Codice guidato dagli eventi.

un fornitore di servizi. Nel paradigma guidato dagli eventi, il sistema operativo fornisce anche dei servizi, ma questo ruolo è meno importante del ruolo di coordinatore delle attività dell'utente e di generatore di eventi consumati dai processi.

Paradigmi dei dati

Il paradigma di esecuzione non è l'unico esportato dal sistema operativo. Un altro ugualmente importante è il paradigma dei dati. La questione chiave in questo caso è come sono presentate al programmatore le strutture e i dispositivi del sistema. Nei primi sistemi batch FORTRAN, tutto era modellato come un nastro magnetico sequenziale. I mazzi di schede da leggere erano trattati come nastri di input, i mazzi di schede da perforare erano trattati come nastri di output e l'output per la stampante era trattato come un nastro di output. Anche i file del disco erano gestiti come nastri. L'accesso casuale a un file era possibile solo riavvolgendo il nastro corrispondente a quel file e rileggendolo.

La mappatura era eseguita utilizzando schede di controllo dei job come queste:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

La prima scheda indicava all'operatore di andare a prendere la bobina del nastro 781 dal raccoglitore dei nastri e di montarla sull'unità a nastro 8. La seconda scheda istruiva il sistema operativo ad avviare il programma FORTRAN appena compilato, mappando *INPUT* (intendendo il lettore di schede) sul nastro logico 1, il file del disco *MYDATA* sul nastro logico 2, la stampante (chiamata *OUTPUT*) sul nastro logico 3, il perforatore di schede (chiamato *PUNCH*) sul nastro logico 4 e l'unità nastro fisica 8 sul nastro logico 5.

Il FORTRAN aveva una sintassi per la lettura e la scrittura delle unità nastro logiche. Leggendo dal nastro logico 1, il programma riceveva l'input delle schede. Scrivendo sul nastro logico 3, più tardi sarebbe apparso un output sulla stampante. Leggendo dal nastro logico 5, poteva essere letta la bobina del nastro 781 e così via. Notate che l'idea del nastro era solo un paradigma per integrare il lettore delle schede, la stampante, il perforatore, i file su disco e i nastri. In questo esempio, solo il nastro logico 5 era un nastro fisico; il resto erano normali file su disco. Era un paradigma primitivo, ma ha costituito un primo passo nella giusta direzione.

In seguito è arrivato UNIX, che va ben oltre utilizzando il modello "tutto è un file". Sfruttando questo paradigma, tutti i dispositivi di I/O sono trattati come file e possono essere aperti e manipolati come file normali. L'istruzione C

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

apre un file reale su disco e apre il terminale dell'utente (tastiera + video). Le istruzioni seguenti possono utilizzare *fd1* e *fd2* rispettivamente per leggervi e scrivervi. Da quel punto in poi, non c'è differenza tra l'accesso a un file e l'accesso al terminale, eccetto per il fatto che sul terminale non sono permesse le ricerche.

UNIX non solo unifica i file e i dispositivi di I/O, ma permette anche che si possa avere accesso ad altri processi tramite le *pipe* come se fossero dei file. Inoltre, quando sono supportati i file mappati, un processo può accedere alla propria memoria virtuale come fosse un file. Infine, in versioni di UNIX che supportano il file system */proc*, l'istruzione C

```
fd3 = open("/proc/501", O_RDWR);
```

permette al processo di (provare ad) accedere alla memoria del processo 501 per leggere e scrivere utilizzando il descrittore del file *fd3*, cosa utile per esempio per un debugger.

Ovviamente, il fatto che qualcuno dica che tutto è un file non significa che sia vero sempre e comunque. Per esempio, i socket di rete di UNIX possono assomigliare a file, ma hanno la propria API, che è piuttosto diversa. Un altro sistema operativo, Plan 9 di Bell Labs, non ha accettato compromessi e non fornisce alcuna interfaccia specializzata per i socket di rete. Il risultato è che la sua progettazione è considerevolmente più chiara.

Windows tenta di far sì che tutto appaia come un oggetto. Una volta che un processo ha acquisito un handle valido per un file, un processo, un semaforo, una mailbox o un altro oggetto del kernel, può eseguirci delle operazioni. Questo paradigma è ancora più generale di quello di UNIX e molto più generale di quello di FORTRAN.

Anche in altri contesti vi sono dei paradigmi unificanti. Vale la pena menzionarne uno in questo caso: il Web. Il paradigma che sta dietro al Web è che il cyberspazio è pieno di documenti, ognuno dei quali ha un URL. Digitando un URL o cliccando su una voce dietro cui sta un URL, si ottiene il documento. In realtà molti "documenti" non sono veri e propri documenti, ma sono generati da programmi o da script di shell quando arriva una richiesta. Per esempio, quando un utente richiede a un venditore online un elenco di CD di un artista particolare, il documento è generato al volo da un programma; di sicuro non esisteva prima che venisse fatta l'interrogazione.

Abbiamo visto finora quattro casi: ossia, tutto è un nastro, un file, un oggetto o un documento. In tutti e quattro i casi l'intenzione è unificare i dati, i dispositivi e le altre risorse per facilitarne l'impiego. Ogni sistema operativo dovrebbe avere un paradigma unificante concepito in questo modo.

12.2.3 Interfaccia delle chiamate di sistema

Se crediamo nell'affermazione di Corbató del minimo meccanismo, allora il sistema operativo dovrebbe fornire quelle poche chiamate di sistema che riesce a gestire e ognuna dovrebbe essere il più semplice possibile (ma non più semplice). Un paradigma dei dati unificante, in questo contesto, può essere molto d'aiuto. Per esempio, se file, processi, dispositivi di I/O e molto altro sono tutti simili a file o ad oggetti, allora possono tutti essere letti con una singola chiamata di sistema `read`, altrimenti potrebbe essere necessario avere chiamate separate per `read_file`, `read_proc`, e `read_tty`, fra le altre.

In alcuni casi può sembrare che le chiamate di sistema necessitino di molte varianti, ma spesso è meglio avere una chiamata di sistema che gestisce la situazione generale, con differenti procedure di libreria che nascondano questo fatto ai programmatori. Per esempio, UNIX ha una chiamata di sistema per coprire lo spazio degli indirizzi virtuali di un processo, `exec`. La chiamata più generica è

```
exec(name, argp, envp);
```

che carica il file eseguibile *name* e gli passa i parametri puntati da *argp* e le variabili di ambiente puntate da *envp*. A volte è comodo elencare i parametri in modo esplicito, così la libreria contiene delle procedure chiamate come segue:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

Tutto quello che queste procedure fanno è mettere i parametri in un array e poi chiamare `exec` per eseguire il lavoro. Questo adattamento è il migliore possibile: una singola e chiara chiamata di sistema mantiene il sistema operativo semplice, eppure il programmatore ha la comodità di avere varie modalità di chiamare `exec`.

Naturalmente, tentare di avere una sola procedura per gestire tutti i casi possibili può facilmente sfuggire di mano. In UNIX la creazione di un processo richiede due chiamate: `fork` seguito da `exec`. La prima non ha parametri; la seconda ne ha tre. Diversamente la chiamata delle API di Win32 per la creazione un processo, `CreateProcess`, ha 10 parametri, ognuno dei quali è un puntatore a una struttura con altri 18 parametri.

Molto tempo fa, qualcuno avrebbe chiesto se sarebbe accaduto qualcosa di orribile nel caso si fosse tralasciato qualcuno di loro. La verità è che in determinati casi i programmatori dovrebbero fare più lavoro per ottenere un particolare risultato, ma il risultato finale sarebbe quello di avere un sistema operativo più semplice, più piccolo e più affidabile. Naturalmente, chi proponeva la versione a 10 + 18 parametri avrebbe ribadito: "Gli utenti apprezzano tutte queste caratteristiche". La successiva replica data loro sarebbe che agli utenti piacciono anche i sistemi che impiegano poca memoria e che non si bloccano in continuazione. I compromessi fra migliori funzionalità al prezzo di maggior memoria sono quanto meno visibili e può essergli attribuito un prezzo (in quanto il costo della memoria è noto). Tuttavia, è difficile stimare il numero di crash del sistema in più all'anno dovuti ad alcune funzionalità aggiuntive e se gli utenti avrebbero fatto la medesima scelta se fossero stati a conoscenza di questo prezzo nascosto da pagare. Un tale effetto può essere riassunto dalla "prima legge del software" di Tanenbaum:

Aggiungere più codice incrementa gli errori.

L'incremento delle funzionalità determina l'incremento di codice e quindi più errori. I programmatori che credono che l'aggiunta di nuove funzionalità non provochi nuovi errori sono o dei principianti del computer o degli ingenui.

La semplicità non è la sola questione che si presenta quando si progettano le chiamate di sistema. Una considerazione importante si trova nello slogan di Lampson (1984):

Non nascondete la potenza.

Se l'hardware ha un modo per svolgere un compito con estrema efficacia, dovrebbe essere esposto ai programmatori in modo semplice e non nascosto all'interno di qualche altra astrazione. L'intento delle astrazioni è quello di nascondere proprietà indesiderabili, non quelle appetibili. Per esempio, supponete che l'hardware abbia una particolare modalità per muovere grandi bitmap nello schermo (cioè, la RAM del video) a una velocità elevata. Sarebbe giustificato avere una nuova chiamata di sistema per accedere a questo meccanismo, piuttosto che fornire solo modi per leggere la RAM del video nella memoria principale e poi riscriverla nuovamente. La nuova chiamata muoverebbe solamente dei bit e niente altro. Se una chiamata di sistema è veloce, gli utenti possono sempre costruire su di essa interfacce più adatte. Se è lenta, nessuno la utilizzerà.

Un'altra questione di progettazione è quella delle chiamate orientate alla connessione rispetto a quelle senza connessione. Le chiamate di sistema standard di UNIX e Win32 per la lettura di un file sono orientate alla connessione, come l'utilizzo del telefono. Anzitutto si apre un file, poi lo si legge, alla fine lo si chiude. Anche alcuni protocolli di accesso a file remoti sono orientati alla connessione. Per esempio, per utilizzare FTP, l'utente prima si collega alla macchina remota, quindi legge i file e poi si scollega.

D'altra parte, alcuni protocolli di accesso a file remoti sono senza connessione. Il protocollo di rete (HTTP) è senza connessione, per esempio. Per leggere una pagina web si deve solo richiederla; non si deve impostare nulla prima (*è* richiesta una connessione TCP, ma si tratta di un protocollo a basso livello; il protocollo HTTP per l'accesso al Web è di per sé senza connessione).

Il compromesso tra qualsiasi meccanismo orientato alla connessione e uno senza connessione è il lavoro aggiuntivo richiesto per l'impostazione del meccanismo (per esempio, l'apertura del file) e il fatto di non doverlo fare sulle (probabilmente molte) chiamate successive. Nel caso di I/O di file su una singola macchina, dove il costo di impostazione è basso, probabilmente la modalità standard (prima apri, poi utilizza) è quella migliore. Per i file system remoti, si può fare in entrambi i modi.

Un'altra questione relativa all'interfaccia delle chiamate di sistema è la sua visibilità. L'elenco delle chiamate di sistema imposte da POSIX è facile da rintracciare. Tutti i sistemi UNIX le supportano, così come un piccolo numero di altre chiamate, ma la lista completa è sempre pubblica. Diversamente Microsoft non ha mai reso pubblica la lista delle chiamate di sistema di Windows. Invece le API di Win32 e le altre API sono state rese pubbliche, ma queste liste contengono un elevato numero di chiamate di libreria (più di 10.000), ma solo un piccolo numero sono vere chiamate di sistema. La motivazione per rendere pubbliche tutte le chiamate di sistema è rappresentato dal fatto che permette ai programmatori di conoscere che cosa sia economico (le funzioni effettuate nello spazio utente) e che cosa sia costoso (le chiamate nel kernel). La ragione per non renderle pubbliche è che questo dà a chi implementa la flessibilità di cambiare le effettive chiamate di sistema al fine di migliorarle senza far saltare i programmi dell'utente. Come si è visto nel Paragrafo 9.7.7, i progettisti originali hanno sbagliato completamente con la chiamata di sistema `access`, ma ormai le cose sono andate così e non si cambia più.

12.3 Implementazione

Allontanandoci dalle interfacce utente e delle chiamate di sistema, vediamo ora come implementare un sistema operativo: negli otto paragrafi successivi esamineremo alcune questioni concettuali generali relative alle strategie di implementazione, quindi analizzeremo alcune tecniche di basso livello che spesso sono utili.

12.3.1 Struttura dei sistemi

Probabilmente la prima decisione che gli implementatori devono intraprendere riguarda la struttura del sistema. Abbiamo esaminato le possibilità principali nel Paragrafo 1.7, ma le rivediamo qui. Un progetto monolitico non strutturato non è davvero una buona idea, eccezion fatta per un piccolo sistema operativo per un frigorifero, ma anche lì è discutibile.

Sistemi a livelli

Un approccio ragionevole che si è ben consolidato da anni è quello dei sistemi a livelli (*layered*). Il sistema THE di Dijkstra (Figura 1.25) fu il primo sistema operativo a livelli. Anche UNIX e Windows 8 hanno una struttura a livelli, ma in entrambi si tratta di un modo di provare a descrivere il sistema più che un reale principio guida nella costruzione del sistema.

Nel caso di un sistema nuovo, i progettisti che intraprendono questa strada dovrebbero *prima* scegliere molto accuratamente i livelli e definire le funzionalità di ciascuno. Il livello più in basso dovrebbe sempre cercare di nascondere le peggiori idiosincrasie dell'hardware, come nel caso di HAL della Figura 11.7. Probabilmente quello successivo dovrebbe gestire gli interrupt, lo scambio di contesto e l'MMU, per cui al di sopra di questo livello il codice è per la maggior parte indipendente dalla macchina. Sopra questo livello, progettisti diversi dimostrano gusti (e predilezioni) differenti. Una possibilità è quella che il livello 3 gestisca i thread, inclusa la sincronizzazione fra i thread e lo scheduling, come illustrato nella Figura 12.2. L'idea in questo caso è che a partire dal livello 4 si abbiano i thread normalmente schedulati e che si sincronizzino usando un meccanismo standard (per esempio, i mutex).

Al livello 4 dovremmo trovare i driver dei dispositivi, ciascuno eseguito come un thread separato, con il proprio stato, contatore di programma, insieme di registri e così via, eventualmente (ma non necessariamente) nello spazio degli indirizzi del kernel. Una simile progettazione semplifica enormemente la struttura di I/O perché, quando avviene un interrupt, può essere convertito in un'operazione di `unlock` su un mutex e una chiamata allo scheduler per (eventualmente) schedulare il nuovo thread divenuto pronto, prima bloccato sul mutex. MINIX 3 usa questo metodo, ma in UNIX, Linux e Windows 8 i gestori degli interrupt vengono eseguiti in una specie di terra di nessuno, invece che come veri e propri thread che possono essere schedulati, sospesi e simili. Dato che gran parte della complessità di un sistema operativo sta nell'I/O, vale la pensa considerare qualunque tecnica per renderlo più tracciabile e incapsulato.

Sopra il livello 4 ci dovremmo aspettare di trovare la memoria virtuale, uno o più file system e i gestori delle chiamate di sistema. Se la memoria virtuale sta a un livello inferiore

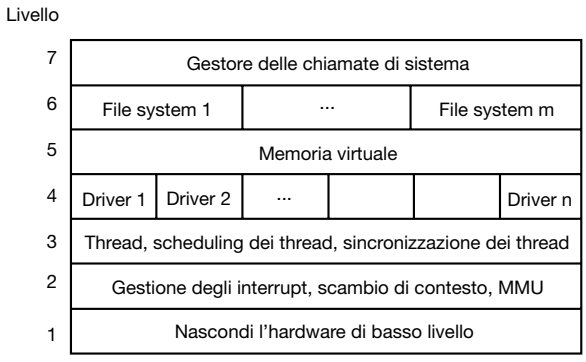

Figura 12.2 Possibile progetto di un moderno sistema operativo a livelli.

rispetto al file system, allora la cache dei blocchi può essere paginata, consentendo al gestore della memoria virtuale di determinare dinamicamente come la memoria virtuale dovrebbe essere suddivisa fra pagine utente e pagine del kernel, cache inclusa. Windows 8 funziona in questo modo.

Exokernel

Mentre l'approccio a livelli ha i suoi sostenitori fra i progettisti dei sistemi, c'è un altro fronte che ha una visione esattamente opposta (Engler et al., 1995), basata sul **principio end-to-end** (Saltzer et al., 1984), che afferma che, se qualcosa deve essere fatto dal programma utente stesso, è uno spreco farlo anche a un livello inferiore.

Considerate un'applicazione di questo principio all'accesso remoto ai file. Se un sistema si preoccupa del fatto che i dati in transito possano essere corrotti, dovrebbe fare in modo che per ciascun file sia calcolato un checksum al momento della scrittura e che tale checksum venga memorizzato insieme al file. Quando un file è trasferito sulla rete dal disco sorgente al processo di destinazione, viene anche trasferito il checksum, ricalcolato sul lato del ricevente. Se i due non combaciano, il file è scartato e ritrasferito.

Questo controllo è più accurato dell'utilizzo di un protocollo di rete affidabile, dato che rileva anche errori dei dischi, errori della memoria, errori software nei router e altri errori, oltre a quelli di trasmissione dei bit. Il principio end-to-end indica che non è necessario l'utilizzo di un protocollo di rete affidabile, dato che il punto di arrivo (il processo destinatario) ha abbastanza informazioni per verificare la correttezza del file. La sola ragione per l'utilizzo di un protocollo di rete affidabile in questa visione è l'efficienza, ossia poter intercettare e riparare prima gli errori di trasmissione.

Il principio end-to-end può essere esteso a quasi tutti i sistemi operativi. Sostiene che il sistema operativo non faccia nulla che il programma utente non possa fare da solo. Per esempio, perché avere un file system? Basta permettere che l'utente legga e scriva una porzione del disco a basso livello in modalità protetta. Naturalmente alla maggior parte degli utenti piace avere i file, ma il principio end-to-end indica che il file system dovrebbe essere una procedura di libreria collegata con qualunque programma che necessiti l'uso dei file. Questo approccio consente a programmi differenti di avere file system diversi. Tale linea di pensiero implica che tutto ciò che dovrebbe fare il sistema operativo è allocare le risorse in modo sicuro (per esempio la CPU e i dischi) fra gli utenti in competizione fra loro per utilizzarle. Exokernel è un sistema operativo costruito secondo i dettami del principio end-to-end (Engler et al., 1995).

Sistemi client-server basati su microkernel

Un compromesso fra un sistema operativo che fa tutto e uno che non fa nulla è un sistema operativo che fa qualcosa. Questo schema porta a un microkernel simile al sistema operativo eseguito come processi server a livello utente, come illustrato nella Figura 12.3. Si tratta del progetto più modulare e flessibile di tutti. L'estrema flessibilità è quella di avere anche ciascun driver dei dispositivi eseguito come processo utente, completamente protetto nei confronti del kernel e degli altri driver, ma anche l'avere i driver dei dispositivi eseguiti nel kernel è una forma di modularità.

Quando i driver dei dispositivi sono nel kernel, essi possono accedere ai registri dei dispositivi hardware direttamente. Quando non lo sono, serve un qualche meccanismo a tal

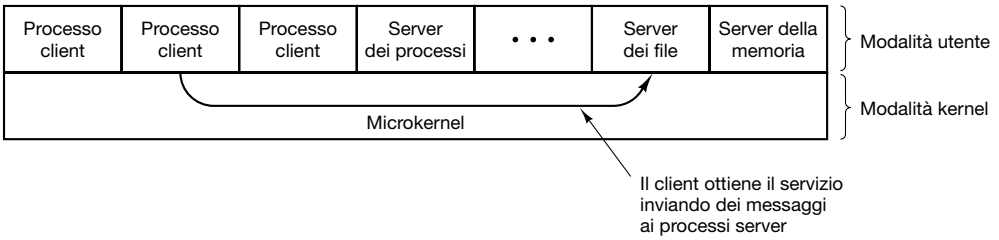

Figura 12.3 Elaborazione client-server basata su un microkernel.

fine. Se l'hardware lo permette, a ciascun processo dei driver potrebbe esser dato accesso solo a quei dispositivi di I/O di cui ha bisogno. Per esempio, con l'I/O a memoria mappata, ciascun processo dei driver potrebbe avere mappata la pagina per il suo dispositivo, ma nessun'altra pagina dei dispositivi. Se lo spazio della porta di I/O può essere parzialmente protetto, potrebbe essere disponibile a ciascun driver la corretta porzione di esso.

Anche in assenza di assistenza hardware, l'idea può comunque essere messa in pratica e resa funzionante. Quello che è necessario è una nuova chiamata di sistema, disponibile solo per i processi dei driver dei dispositivi, fornendo un elenco di coppie (porta, valore). Quello che fa il kernel è in primo luogo un controllo per vedere se il processo possegga tutte le porte dell'elenco. In caso affermativo, copia poi i valori corrispondenti nelle porte, per inizializzare l'I/O dei dispositivi. Una chiamata simile può essere usata per leggere le porte di I/O in modalità protetta.

Questo approccio evita che i driver dei dispositivi esaminino (e danneggino) le strutture dati del kernel, il che (nella maggior parte dei casi) è cosa buona e giusta. Un insieme analogo di chiamate potrebbe essere reso disponibile per consentire ai processi dei driver di leggere e scrivere le tabelle del kernel, ma soltanto in una modalità controllata e con l'approvazione del kernel.

Il principale problema di questo approccio, e del microkernel in generale, è il risultato prestazionale causato da tutti gli scambi di contesto extra. Tuttavia, l'intero lavoro sui microkernel fu praticamente svolto molti anni fa, quando le CPU erano molto più lente. Ai giorni nostri, le applicazioni che usano ogni goccia della potenza della CPU senza tollerare una piccola perdita nelle prestazioni sono rare. Dopo tutto, quando si esegue un editor di testi o un browser web, la CPU è con tutta probabilità inattiva per il 95% del tempo. Se un sistema operativo basato sul microkernel trasformasse un sistema inaffidabile a 3,5 GHz in un sistema affidabile a 3 GHz, probabilmente pochi utenti se ne lamenterebbero. Dopo tutto, la maggior parte di loro era abbastanza soddisfatta pochi anni fa del precedente PC alla, allora fantastica, velocità di 1 GHz. Non è chiaro, inoltre, se il costo della comunicazione fra processi è ancora un problema, visto che i core ormai non sono più una risorsa scarsa. Se ogni driver di dispositivo e ogni componente del sistema operativo ha il proprio core dedicato, non c'è scambio di contesto durante la comunicazione fra processi; inoltre, la cache, i predittori di branch e i TLB sono tutti pronti a partire velocemente. Alcuni lavori sperimentali su sistemi operativi ad alte prestazioni basati su un microkernel sono stati presentati da Hruby et al. (2013). È degno di nota il fatto che, mentre i microkernel non sono molto popolari sui computer desktop, sono invece usati diffusamente in telefoni cellulari,

sistemi industriali, sistemi embedded e sistemi militari, che richiedono un'alta affidabilità. Anche Apple OS X, il sistema dei Mac e dei Macbook, consiste di una versione modificata di FreeBSD in esecuzione sopra una versione modificata del microkernel Mach.

Sistemi estensibili

Con i sistemi client-server discussi in precedenza, l'idea era eliminare la maggior parte possibile del kernel. L'approccio opposto è quello di mettere più moduli nel kernel, ma in modo protetto. La parola chiave in questo caso è *protetto*, naturalmente. Abbiamo studiato alcuni meccanismi di protezione nel Paragrafo 9.5.6, inizialmente pensati per importare le applet su Internet; essi sono ugualmente applicabili all'inserimento di codice esterno nel kernel. I più importanti sono il *sandboxing* e la firma del codice, poiché l'uso di codice interpretato non è davvero pratico per il codice del kernel.

Naturalmente un sistema estensibile di per sé non è un modo per strutturare un sistema operativo. Tuttavia, partendo da un sistema minimale costituito da poco più di un meccanismo di protezione e con l'aggiunta di moduli protetti a quello del kernel, uno alla volta, sino a ottenere le funzionalità desiderate, si può costruire un sistema minimo per l'applicazione disponibile. In quest'ottica si può ritagliare un nuovo sistema operativo su ogni applicazione, includendo solo le parti richieste. Paramecium è un esempio di un sistema di questo genere (Van Doorn, 2001).

Thread del kernel

Un'altra importante questione, a prescindere dal modello di strutturazione scelto, è quella dei thread di sistema. Talvolta è conveniente permettere l'esistenza dei thread del kernel, separati da qualunque processo utente. Questi thread possono funzionare in background, scrivendo le pagine "sporche" sul disco, eseguendo lo swap dei processi fra la memoria principale e il disco, e così via. In effetti, il kernel stesso può essere strutturato interamente con questo tipo di thread, così quando un utente fa una chiamata di sistema, invece di avere il thread utente eseguito in modalità kernel, il thread dell'utente si blocca e passa il controllo al thread del kernel che si prende carico del lavoro. Oltre ai thread del kernel che funzionano in background, la maggior parte dei sistemi operativi avvia molti processi demoni in background. Sebbene questi non siano parte del sistema operativo, spesso eseguono delle attività "di sistema", come la ricezione e l'invio di posta elettronica e la gestione di richieste di vario genere per gli utenti remoti, come l'FTP e le pagine web.

12.3.2 Il meccanismo rispetto alla policy

Un altro principio che aiuta la coerenza architetturale, insieme al fatto di mantenere le cose piccole e ben strutturate, è la separazione del meccanismo dalla policy. Mettendo il meccanismo nel sistema operativo e lasciando la policy ai processi utente, il sistema stesso può non dover essere modificato anche in caso di necessità di un cambio di policy. Anche se il modulo della policy deve essere tenuto nel kernel, dovrebbe essere tenuto isolato dal meccanismo, se possibile, in modo che le modifiche al modulo della policy non abbiano effetti sul modulo del meccanismo.

Per rendere ancor più evidente questa divisione fra policy e meccanismo consideriamo due esempi del mondo reale. Come primo esempio considerate una grande società con un

Ufficio Paghe e Contributi, responsabile del pagamento degli stipendi dei dipendenti, con computer, software, libretti degli assegni, contratti con le banche e altri meccanismi per pagare le retribuzioni. Tuttavia la policy – chi è pagato e quanto – è completamente separata ed è decisa dai dirigenti. L'Ufficio Paghe e Contributi fa semplicemente quanto gli viene detto di fare.

Come secondo esempio considerate un ristorante. Possiede il meccanismo per servire i pasti, che include tavoli, piatti, camerieri, una cucina completa dell'equipaggiamento necessario, accordi con le società di carte di credito e così via. La policy è impostata dallo chef ed è di fatto rappresentata da quello che si trova sul menu. Se lo chef decide che il tofu non va bene e invece vanno bene delle grosse bistecche, la nuova policy può essere gestita dai meccanismi già esistenti.

Consideriamo adesso alcuni esempi relativi ai sistemi operativi. Consideriamo per primo lo scheduling dei thread. Il kernel potrebbe avere uno scheduler di priorità, con k livelli di priorità. Il meccanismo è un array, indicizzato secondo il livello di priorità, come nel caso di UNIX e Windows 8. Ogni voce è la testa di un elenco di thread pronti a quel livello di priorità. Lo scheduler semplicemente ricerca nell'array dalla priorità più alta a quella più bassa, selezionando il primo thread che trova. La policy imposta le priorità. Il sistema potrebbe avere differenti classi di utenti, ciascuno con un diverso livello di priorità, per esempio. Potrebbe anche consentire ai processi utente di impostare le relative classi di priorità dei suoi thread. Le priorità potrebbero essere aumentate dopo aver completato le operazioni di I/O o diminuite dopo aver utilizzato un quantum. Ci sono numerose altre policy che potrebbero essere seguite, ma l'idea in questa situazione è la separazione fra l'impostazione della policy e la sua realizzazione.

Un secondo esempio è la paginazione. Il meccanismo coinvolge la gestione dell'MMU, tenendo un elenco delle pagine occupate e delle pagine libere, e il codice per le pagine da scambiare da e verso il disco. La policy consiste nel decidere che cosa fare quando si verifica un errore di pagina. Potrebbe essere locale o globale, basato sull'LRU o sul FIFO o su qualcos'altro, ma questo algoritmo può (e dovrebbe) essere completamente separato dai meccanismi dell'effettiva gestione delle pagine.

Un terzo esempio è consentire che i moduli siano caricati nel kernel. Il meccanismo riguarda come sono inseriti, come sono collegati (linked), quali chiamate possono effettuare e quali possono esser fatte su di loro. La policy determina chi ha il permesso di caricare un modulo nel kernel e quali moduli. Può darsi sia solo il superuser che può caricarli, ma può darsi che qualunque utente possa caricare un modulo purché sia stato prima firmato digitalmente dall'autorità appropriata.

12.3.3 Ortogonalità

La progettazione di un buon sistema è fatta di concetti separati che possono essere combinati indipendentemente. In C, per esempio, vi sono dei tipi di dati primitivi che includono numeri interi, caratteri e numeri a virgola mobile. Ci sono anche dei meccanismi per combinare i tipi di dati, incluso vettori, strutture e unioni. Queste idee si combinano indipendentemente, consentendo vettori di interi, vettori di caratteri, membri di strutture e di unioni che sono numeri in virgola mobile e così via. Infatti, una volta definito un nuovo tipo di dati, come un array di interi, esso può essere usato come fosse un tipo di dati primitivo, per esempio come un membro di una struttura o di un'unione. La capacità di combinare dei

concetti separati indipendentemente è detta **ortogonalità**. È una conseguenza diretta dei princìpi di semplicità e completezza.

Il concetto di ortogonalità è presente anche nei sistemi operativi, travestito in vari modi. Un esempio è la chiamata di sistema `clone` di Linux, che crea un nuovo thread. La chiamata ha una bitmap come parametro, il che permette che lo spazio degli indirizzi, la directory di lavoro, i descrittori dei file e i segnali siano condivisi o copiati individualmente. Se è copiato tutto, si ha un nuovo processo, come con `fork`. Se non è copiato nulla, si crea un nuovo thread nel processo attuale. È tuttavia possibile creare delle forme intermedie di condivisione che non sono possibili nei sistemi UNIX tradizionali. Scomponendo le diverse caratteristiche e rendendole ortogonali è possibile ottenere un grado di controllo più preciso.

Un altro impiego dell'ortogonalità è quello della separazione del concetto di processo dal concetto di thread in Windows 8. Un processo è un contenitore per le risorse, niente di meno e niente di più. Un thread è un'entità schedulabile. Quando a un processo è dato un handle di un altro processo, non gli importa quanti thread abbia. Quando è schedulato un thread, non importa a quale processo appartenga. Questi concetti sono ortogonali.

Il nostro ultimo esempio di ortogonalità deriva da UNIX. In questo caso la creazione dei processi avviene in due passaggi: `fork` più `exec`. La creazione del nuovo spazio degli indirizzi e il caricamento in esso di una nuova immagine della memoria sono passaggi separati, rendendo così possibile che avvengano delle cose nel mezzo (come la manipolazione dei descrittori dei file). In Windows 8, questi due passaggi non possono essere separati, il che significa che in questo caso i concetti di creazione di un nuovo spazio degli indirizzi e il suo riempimento non sono ortogonali. La sequenza di Linux di `clone` più `exec` è ancor più ortogonale, dato che si possono costruire dei blocchi ancor più dettagliati. Come regola generale, avere a disposizione un piccolo numero di elementi ortogonali che possono essere combinati in molti modi porta a un sistema piccolo, semplice ed elegante.

12.3.4 Gestione dei nomi

La maggior parte delle strutture dati di lunga data utilizzate dai sistemi operativi ha un certo nome o identificatore tramite il quale ci si può loro riferire. Esempi ovvi sono il login, i nomi dei file, i nomi dei dispositivi, gli ID dei processi e così via. Il modo in cui questi nomi sono costruiti e gestiti rappresenta una questione importante nella progettazione e implementazione dei sistemi. I nomi progettati per l'uso da parte delle persone sono nomi di stringhe di caratteri in ASCII o in Unicode e sono solitamente gerarchici. I percorsi delle directory, come */usr/ast/books/mos2/chap-12,* sono chiaramente gerarchici, a indicare una serie di directory da ricercare partendo dalla root. Anche gli URL sono gerarchici. Per esempio *www.cs.vu.nl/~ast/* indica una specifica macchina (*www*), in uno specifico dipartimento (*cs*), a una specifica università (*vu*), in uno specifico stato (*nl*). La parte dopo la barra indica un determinato file sulla macchina indicata, in questo caso, per convenzione, *www/index.html* nella home directory di *ast*. Notate che gli URL (e in generale gli indirizzi DNS, inclusi gli indirizzi di posta elettronica) vanno a ritroso, partendo dal fondo dell'albero e salendo, diversamente dai nomi dei file, che partono dalla cima dell'albero e scendono. Un altro modo per vedere questa caratteristica è se l'albero è scritto dall'alto partendo da sinistra verso destra o partendo da destra verso sinistra.

Spesso il naming è fatto a due livelli: esterno e interno. Per esempio, i file hanno sempre un nome sotto forma di una stringa di caratteri per l'uso da parte delle persone. Oltre questo

c'è quasi sempre un nome interno utilizzato dal sistema. In UNIX il nome reale di un file è il suo numero di i-node; a livello interno il nome ASCII non è assolutamente utilizzato e infatti non è univoco, dato che un file può avere più collegamenti a esso. L'analogo nome interno in Windows 8 è l'indice del file nella MFT. Compito della directory è fornire una mappatura fra il nome esterno e il nome interno, come illustra la Figura 12.4.

In molti casi (come nell'esempio precedente dei nomi dei file), il nome interno è un intero senza segno che funge da indice in una tabella del kernel. Altri esempi di nomi indicizzati in tabelle sono i descrittori dei file in UNIX e gli handle degli oggetti in Windows 8. Notate che nessuno di questi ha una rappresentazione esterna. Essi sono usati dal sistema e dai processi in esecuzione. In generale è una buona idea usare indici di tabelle per quei nomi temporanei che sono persi al riavvio del sistema.

I sistemi operativi spesso supportano più spazi dei nomi, sia esterni sia interni. Per esempio, nel Capitolo 11 abbiamo preso in esame tre spazi dei nomi esterni supportati da Windows 8: i nomi dei file, i nomi degli oggetti e i nomi dei registri (e c'è anche lo spazio dei nomi di Active Directory, che non abbiamo considerato). Oltre a ciò ci sono innumerevoli spazi dei nomi interni che utilizzano numeri interi senza segno, per esempio gli handle degli oggetti e le voci della MFT. Sebbene i nomi negli spazi dei nomi esterni siano tutte stringhe Unicode, la ricerca di un nome di file nel registro non funzionerà, così come non funzionerà l'uso di un indice della MFT nella tabella degli oggetti. In un buon progetto, si deve pensare con attenzione a quanti spazi dei nomi siano necessari, quale sia la sintassi dei nomi in ciascuno, come possano essere distinti, se esistano dei nomi assoluti e relativi e così via.

12.3.5 Binding time

Come abbiamo appena visto, i sistemi operativi usano vari tipi di nomi per riferirsi agli oggetti. Talvolta la mappatura fra un nome e un oggetto è fissa, altre volte invece non lo è. In quest'ultimo caso, può essere importante il momento in cui il nome è legato all'oggetto. In generale l'**early binding** è semplice, ma non flessibile, mentre il **late binding** è più complicato ma più flessibile. Per chiarire il concetto di *binding time*, diamo uno sguardo ad alcuni esempi del mondo reale. Un esempio di early binding è la pratica di alcuni college

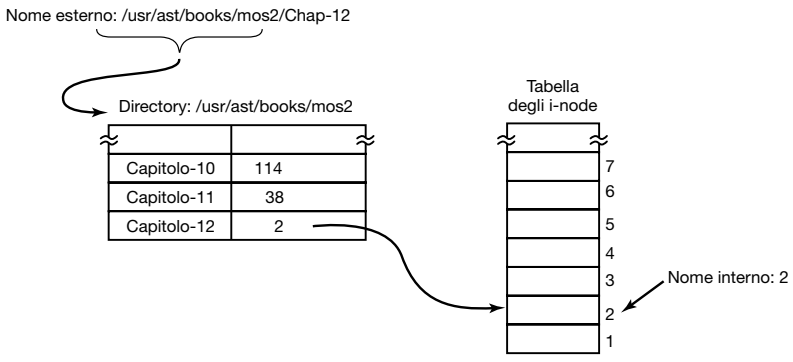

Figura 12.4 Le directory sono utilizzate per mappare i nomi esterni nei nomi interni.

di permettere ai genitori di iscrivere un bambino alla nascita e prepagare la retta attuale. Quando lo studente compare, 18 anni dopo, la retta è completamente pagata, non importa quanto alta possa essere in quel momento.

Nella industria manifatturiera, l'ordine di pezzi in anticipo e il mantenimento di un inventario è early binding. Al contrario, il just-in-time richiede che i fornitori siano in grado di fornire i pezzi sulla base della domanda puntuale, senza alcun avviso in anticipo. Questo è *late binding*.

I linguaggi di programmazione spesso supportano diversi binding time per le variabili. Le variabili globali sono legate a un particolare indirizzo virtuale dal compilatore, il che esemplifica l'early binding. Alle variabili locali di una procedura è assegnato un indirizzo virtuale (sullo stack) nell'istante in cui la procedura viene chiamata. Questo è *intermediate binding*. Le variabili memorizzate sullo heap (quelle allocate da *malloc* in C o *new* in Java) sono indirizzi virtuali assegnati solo al momento del loro utilizzo effettivo. In questo caso si ha un late binding.

I sistemi operativi spesso utilizzano l'early binding per la maggior parte delle loro strutture dati, ma occasionalmente utilizzano il late binding per avere maggiore flessibilità. L'allocazione della memoria è il caso in questione. I primi sistemi a multiprogrammazione su macchine che non avevano hardware per la riallocazione degli indirizzi dovevano caricare un programma a un certo indirizzo di memoria e rilocarlo per eseguirlo lì. Se ne veniva fatto lo swap su disco, doveva essere riportato indietro allo stesso indirizzo di memoria o sarebbe andato in errore. All'opposto, la memoria virtuale paginata è una forma di late binding. L'effettivo indirizzo fisico corrispondente a un dato indirizzo virtuale non è noto finché la pagina non è toccata ed effettivamente portata in memoria.

Un altro esempio di late binding è il posizionamento delle finestre in una GUI. In contrapposizione ai primi sistemi grafici, in cui il programmatore doveva specificare le coordinate assolute dello schermo per tutte le immagini sullo schermo, nelle moderne GUI il software utilizza le coordinate relative all'origine della finestra, che non è determinata finché la finestra non è messa sullo schermo e potrebbe anche essere cambiata più tardi.

12.3.6 Strutture statiche rispetto a strutture dinamiche

I progettisti di sistemi operativi sono costantemente costretti a scegliere fra strutture dati statiche e dinamiche. Quelle statiche sono sempre più semplici da capire, più facili da programmare e più veloci da usare; le dinamiche sono più flessibili. Un esempio ovvio è la tabella dei processi. I primi sistemi allocavano semplicemente un array fisso di strutture per processo. Se la tabella dei processi era di 256 voci, allora per ogni istante potevano esistere solo 256 processi. Un tentativo di creare il 257° sarebbe fallito per mancanza di spazio della tabella. Considerazioni analoghe valgono per la tabella dei file aperti (riguardo sia l'utente sia il sistema) e molte altre tabelle del kernel.

Una strategia alternativa è la costruzione della tabella dei processi come una lista collegata di mini-tabelle, inizialmente solo una. Se questa tabella si riempie, se ne alloca un'altra da un'area di memoria globale e la si collega poi alla prima. In questo modo la tabella dei processi non si riempie a meno di esaurire tutta la memoria del kernel.

D'altra parte il codice per la ricerca nelle tabelle diventa più complicato. Per esempio, il codice per la ricerca di un determinato PID, *pid*, in una tabella dei processi statica è illustrato nella Figura 13.5. Semplice ed efficiente. Fare la stessa cosa con un insieme di mini-tabelle collegate richiede molto lavoro in più.

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
            found = 1;
            break;
    }
}
```

Figura 12.5 Codice per cercare un dato PID nella tabella dei processi.

Le tabelle statiche sono migliori quando c'è abbondanza di memoria o gli utilizzi della tabella possono essere ipotizzati abbastanza accuratamente. Per esempio, in un sistema monoutente è improbabile che un utente faccia partire contemporaneamente 64 processi e non sarebbe poi un disastro irreparabile se il tentativo di far partire l'eventuale 65° dovesse fallire. Un'ulteriore alternativa è usare una tabella di dimensione fissa, ma qualora si riempia, allocarne un'altra sempre a dimensione fissa ma doppia rispetto alla precedente. Le voci attuali vengono quindi copiate nella nuova tabella e la vecchia tabella viene restituita al pool di memoria libera. In questo modo la tabella è sempre contigua, invece che collegata. Lo svantaggio di questa situazione è che serve una gestione dello spazio di memorizzazione e l'indirizzo della tabella diventa una variabile invece di una costante. Una questione simile si presenta per gli stack del kernel. Quando un thread passa in modalità kernel o un thread è eseguito in modalità kernel, ha bisogno di uno stack nello spazio del kernel. Per i thread utente, lo stack può essere inizializzato per correre lungo lo spazio degli indirizzi virtuali a partire dalla cima, per cui la dimensione non deve essere specificata in anticipo. Per i thread del kernel, la dimensione deve essere specificata in anticipo perché lo stack occupa parte dello spazio degli indirizzi virtuali del kernel e potrebbero esserci molti stack. La domanda è: quanto spazio occupa ciascuno? I compromessi sono in questo caso simili a quelli per la tabella dei processi. Rendere dinamiche strutture dati di questo tipo è possibile, ma è complicato.

Un altro compromesso fra statico e dinamico è lo scheduling dei processi. In alcuni sistemi, specialmente quelli real-time, lo scheduling può essere fatto staticamente in anticipo. Per esempio, una compagnia aerea conosce gli orari di partenza dei propri voli con settimane di anticipo. In modo simile i sistemi multimediali sanno quando schedulare i processi audio, video e altri in anticipo. Queste considerazioni non si adattano a un impiego generalista in cui lo scheduling deve essere dinamico.

Un'ulteriore questione dinamico-statica è la struttura del kernel: è molto più semplice se il kernel è costruito come un programma binario singolo e caricato in memoria per l'esecuzione. La conseguenza di questo schema, tuttavia, è che l'aggiunta di un nuovo dispositivo di I/O richiede che il kernel sia ricollegato (*relinked*) al nuovo driver del dispositivo. Le prime versioni di UNIX funzionavano in questo modo e ciò andava abbastanza bene quando in un ambiente di minicomputer l'aggiunta di un nuovo dispositivo era una occorrenza abbastanza rara. Ai giorni nostri la maggior parte dei sistemi operativi permette che il codice sia aggiunto al kernel dinamicamente, con l'ulteriore complessità che ne deriva.

12.3.7 Implementazione top-down e bottom-up

Sebbene da un lato sia meglio progettare un sistema secondo lo schema *top-down*, in teoria potrebbe essere implementato sia *top-down* che *bottom-up*. In un'implementazione top-down,

gli sviluppatori partono dai gestori delle chiamate di sistema e guardano che meccanismi e strutture dati servano per supportarli. Queste procedure sono quindi scritte e così via, fino ad arrivare all'hardware.

Il problema di questo approccio è che è difficile testare qualsiasi cosa con le sole procedure top-level a disposizione. Per questa ragione molti sviluppatori trovano sia più pratico costruire in realtà il sistema secondo il metodo bottom-up. Questo approccio comporta prima la scrittura del codice che nasconde l'hardware di basso livello, un po' come in HAL, nella Figura 11.4. La gestione degli interrupt e il driver del clock sono anch'essi necessari sin dal principio. Si può quindi affrontare il nodo della multiprogrammazione, insieme a un semplice scheduler (per esempio lo scheduling round-robin). A questo punto dovrebbe essere possibile provare il sistema per vedere se sia in grado di eseguire correttamente più processi alla volta. Se funziona, è il momento di cominciare la delicata definizione delle varie tabelle e strutture dati necessarie a tutto il sistema, specialmente quelle per la gestione dei processi e dei thread e più tardi della memoria. Il file system e l'I/O inizialmente possono attendere, fatta eccezione per un metodo primitivo che legga dalla tastiera e scriva sullo schermo per i test e il debug. In alcuni casi le strutture dati chiave di basso livello dovrebbero essere protette consentendo l'accesso solo tramite specifiche procedure – in effetti questa è programmazione orientata agli oggetti, a prescindere dal linguaggio di programmazione. Appena completati i livelli inferiori, possono essere testati accuratamente. In questo modo il sistema procede dal basso verso l'alto, come la costruzione di un edificio di tanti piani.

Se si lavora in un grande team, un metodo alternativo è quello di fare prima un progetto dettagliato dell'intero sistema, quindi assegnare a gruppi differenti la scrittura di moduli diversi. Ciascuno mette alla prova il proprio operato in condizioni di isolamento. Quando tutti i pezzi sono pronti, si procede a integrarli l'un l'altro e provarli. Il problema di questo metodo di attacco è che se in principio non funziona nulla può essere difficoltoso isolare quale sia il modulo o i moduli malfunzionanti o se un gruppo abbia capito male che cosa doveva attendersi dal funzionamento di qualche altro modulo. Tuttavia, con gruppi di lavoro di grandi dimensioni, questo metodo è usato spesso al fine di massimizzare la quantità di parallelismo nella programmazione.

12.3.8 Comunicazioni sincrone e asincrone

Un altro problema che spesso si presenta nelle conversazioni fra progettisti di sistemi operativi è se le interazioni fra i componenti del sistema debbano essere sincrone o asincrone (e, di conseguenza, se i thread siano meglio degli eventi). La questione porta spesso a furibondi litigi fra i sostenitori delle due tecniche, anche se non li lascia con la schiuma alla bocca tanto quanto decidere su questioni veramente importanti, come quale sia l'editor migliore, se *vi* o *emacs*.

Utilizziamo il termine "sincrono" nell'ampio senso del Paragrafo 8.2 per denotare chiamate che bloccano fino al completamento; "asincrone" sono invece le chiamate che il processo chiamante invia, continuando quindi la propria esecuzione. Entrambi i modelli presentano vantaggi e svantaggi.

Alcuni sistemi, come Amoeba, accolgono completamente la progettazione con chiamate sincrone e implementano la comunicazione fra processi come chiamate client/server bloccanti. Una comunicazione completamente sincrona è concettualmente molto semplice: un processo invia una richiesta e si blocca in attesa che arrivi la risposta; niente di più semplice. Diventa un po' più difficile quando ci sono più client che richiedono tutti l'attenzione del

server. Ogni singola richiesta può bloccare per molto tempo, in attesa che le altre richieste vengano evase prima di lei; questo problema può essere risolto rendendo il server multithread, in modo che ciascun thread possa gestire un unico client. Il modello è stato provato e testato in molte implementazioni reali, in sistemi operativi e applicazioni utente.

Il tutto diventa ancor più complesso se i thread leggono e scrivono di frequente strutture dati condivise; in questo caso, il lock diventa inevitabile. Purtroppo, non è facile fare in modo che i lock si comportino in maniera adeguata. La soluzione più semplice è mettere un unico grosso lock su tutte le strutture dati condivise (un po' come avviene nel grosso lock del kernel). Quando un thread desidera accedere alle strutture dati condivise, deve per prima cosa prendere possesso del lock. Per questioni di prestazioni, utilizzare un unico grosso lock non è una buona idea, perché i thread finiscono per aspettarsi continuamente l'un con l'altro anche se non entrano in conflitto. L'altro estremo, utilizzare numerosi micro lock per (parti) delle singole strutture dati, è molto più veloce, ma entra in conflitto con il principio guida numero 1: la semplicità.

Altri sistemi operativi realizzano la comunicazione fra processi utilizzando primitive asincrone. In un certo senso, la comunicazione asincrona è anche più semplice di quella sincrona: un processo cliente invia un messaggio a un server, ma invece di aspettare che il messaggio sia portato a destinazione o che venga restituita una risposta, continua tranquillamente la propria esecuzione. Ciò significa, ovviamente, che anche la risposta è inviata in modo asincrono e il processo deve quindi ricordarsi a quale richiesta corrispondeva quando la riceve. Il server solitamente elabora le richieste (eventi) come un unico thread in un ciclo di eventi. Quando la richiesta ha bisogno che il server contatti altri server per ulteriori elaborazioni, invia un proprio messaggio asincrono e, invece di bloccarsi, continua passando alla richiesta successiva. Non sono necessari più thread: con un singolo thread di elaborazione degli eventi, il problema di avere più thread che accedono a strutture dati condivise non si presenta. D'altro canto, un gestore di eventi in esecuzione da molto tempo rallenta la risposta del server a thread singolo.

Se sia migliore il modello di programmazione a thread o quello a eventi è una questione molto controversa, che ha turbato i cuori dei fanatici dell'una e dell'altra parte fin dalla pubblicazione del classico di John Ousterhout: "Perché i thread sono una pessima idea (per la maggior parte degli scopi)" (1996). Ousterhout sostiene che i thread rendono tutto inutilmente complicato: il lock, il debug, i callback, le prestazioni – qualsiasi cosa. Ovviamente, se tutti fossero d'accordo non ci sarebbero controversie. Qualche anno dopo la pubblicazione di Ousterhout, Von Behren et al. (2003) ha pubblicato un articolo intitolato "Perché gli eventi sono una pessima idea (per i server con elevata concorrenza)". Decidere quindi il modello di programmazione più adatto è difficile, ma è una decisione importante per i progettisti di sistemi operativi. Non c'è un vincitore assoluto. I server web come *apache* preferiscono la comunicazione sincrona e i thread, mentre altri come *lighttpd* sono basati sul paradigma orientato agli eventi. Entrambi sono molto diffusi. Secondo noi, gli eventi spesso sono più facili da capire e da correggere rispetto ai thread. Finché non c'è bisogno di gestire la concorrenza sui singoli core, sono probabilmente la scelta migliore.

12.3.9 Tecniche utili

Abbiamo appena preso in esame alcune idee astratte per la progettazione e l'implementazione dei sistemi. Esamineremo adesso un certo numero di tecniche concrete per l'implementazione dei sistemi.

Mascheramento dell'hardware

Gran parte dell'hardware è brutto. Deve essere nascosto in fretta (a meno che non esponga della potenza, cosa che la maggior parte dell'hardware non fa). Alcuni dei dettagli di bassissimo livello possono essere nascosti da un livello di tipo HAL come quello mostrato nella Figura 12.2. Tuttavia molti dettagli dell'hardware non possono essere mascherati in questo modo.

Un aspetto che merita attenzione fin dal principio è la gestione degli interrupt, che rendono la programmazione sgradevole, ma i sistemi operativi devono averci a che fare. Un approccio consiste nel trasformarli immediatamente in qualcosa di diverso. Per esempio ogni interrupt può essere immediatamente trasformato in un thread pop-up. A questo punto avremo a che fare con i thread, invece che con gli interrupt.

Un secondo approccio consiste nel convertire ciascun interrupt in una operazione di `unlock` su di un mutex su cui è in attesa il driver corrispondente. A questo punto il solo effetto di un interrupt è il provocare che un determinato thread diventi pronto (ready).

Un terzo metodo è quello della conversione dell'interrupt in un messaggio per qualche thread. Il codice di basso livello costruisce semplicemente un messaggio che descrive da dove proveniva l'interrupt, lo accoda e invoca lo scheduler per eseguire (eventualmente) il gestore (handler), che probabilmente era bloccato in attesa del messaggio. Tutte queste tecniche e altre simili provano a convertire gli interrupt in operazioni di sincronizzazione dei thread. Il fatto che ciascun interrupt sia gestito da un thread appropriato nel giusto contesto ne rende la gestione più facile rispetto all'esecuzione dell'handler nel contesto arbitrario in cui si trovi. Naturalmente tutto deve essere fatto efficacemente, ma nel profondo di un sistema operativo ogni cosa deve essere fatta efficacemente.

La maggior parte dei sistemi operativi è progettata per funzionare su piattaforme diverse. Queste piattaforme possono differenziarsi in termini di CPU, MMU, lunghezza delle parole, dimensioni della RAM e altre caratteristiche che non possono essere facilmente nascoste dall'HAL o da qualcosa di analogo. Tuttavia è altamente auspicabile avere un singolo insieme di file sorgente utilizzati per generare ciascuna versione; altrimenti ogni errore che dovesse verificarsi dovrebbe essere corretto più volte in più sorgenti, col pericolo che i sorgenti diventino diversi.

Alcune differenze hardware, come la dimensione della RAM, possono essere gestite dal sistema operativo che ne determina il valore al momento dell'avvio della macchina e lo tiene in una variabile. Gli strumenti di allocazione della memoria, per esempio, possono usare la variabile della dimensione della RAM per determinare le dimensioni dei blocchi della cache, delle tabelle delle pagine e cose simili. Anche le tabelle statiche, come la tabella dei processi, possono essere dimensionate sulla base della memoria totale disponibile.

Tuttavia vi sono altre differenze che non possono essere risolte con un singolo file binario che determina a run-time su quale CPU sia in esecuzione. Un modo per affrontare il problema di un sorgente e di più destinatari è l'utilizzo della compilazione condizionata. Nei file sorgente sono definiti alcuni flag compile-time per le diverse configurazioni e questi sono usati per suddividere in blocchi racchiusi tra parentesi il codice che dipende dal tipo di CPU, dalla lunghezza delle parole, dall'MMU e così via. Immaginate per esempio un sistema operativo che deve essere eseguito su chip IA32 della linea x386 (noti anche come X386-32) o UltraSPARC, che necessitano di un diverso codice di inizializzazione. La procedura *init* potrebbe essere scritta come illustrato nella Figura 12.6(a). A seconda del valore di CPU, definito nel file d'intestazione (header file) *config.h*, l'inizializzazione è diversa. Dato

```
#include "config.h"

init( )
{
#if (CPU == IA32)
/* Inizializzazione dell'IA32 qui. */
#endif

#if (CPU == ULTRASPARC)
/* Inizializzazione dell'UltraSPARC
qui. */
#endif


}
```

(a)

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

(b)

Figura 12.6 (a) Compilazione condizionata dipendente dalla CPU.
(b) Compilazione condizionata dipendente dalla lunghezza della parola.

che il binario reale contiene solo il codice specifico per la macchina di destinazione, non vi è alcuna perdita di efficienza con questo schema.

Come secondo esempio supponete vi sia un'esigenza per un tipo dato *Register*, che dovrebbe essere di 32 bit sull'IA32 e di 64 bit sull'UltraSPARC. Potrebbe essere gestito dal codice condizionato della Figura 12.6(b) (dando per assunto che il compilatore produca interi a 32 bit e interi lunghi a 64 bit). Una volta fatta la definizione (probabilmente in un file d'intestazione incluso ovunque) il programmatore dovrebbe solo dichiarare che le variabili sono del tipo *Register* perché siano in automatico della giusta lunghezza.

Il file d'intestazione, *config.h*, deve naturalmente essere definito correttamente. Per l'IA32 potrebbe essere qualcosa di simile a quanto segue:

```
#define CPU IA32
#define WORD_LENGTH 32
```

Nel compilare il sistema per l'UltraSPARC, dovrebbe essere usato un file *config.h* diverso, con i valori appropriati per l'UltraSPARC, probabilmente qualcosa del genere:

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Alcuni lettori potrebbero chiedersi perché *CPU* e *WORD_LENGTH* siano gestite da macro differenti. Avremmo potuto categorizzare la definizione di *Register* con un test su *CPU*, impostando quindi 32 bit per il Pentium e 64 bit per l'UltraSPARC. Non è tuttavia una buona idea. Considerate che cosa accadrebbe nel momento in cui dovessimo portare in seguito il sistema su un ARM a 32 bit. Avremmo dovuto aggiungere una terza condizione alla Figura 13.6(b) per l'ARM. Per farlo, la riga da includere sarebbe

```
#define WORD_LENGTH 32
```

nel file *config.h* per l'ARM.

Questo esempio illustra il principio di ortogonalità trattato in precedenza. Gli elementi che sono dipendenti dalla CPU dovrebbero essere compilati in modo condizionato basandosi sulla macro *CPU* e quelli dipendenti dalla lunghezza delle parole dovrebbero esserlo sulla base della macro *WORD_LENGTH*. Considerazioni simili valgono per molti altri parametri.

Indirezione

Si dice talvolta che non c'è in informatica un problema che non sia risolvibile introducendo un livello di indirezione. Come sempre in ogni esagerazione, anche in questo caso c'è in fondo un briciolo di verità. Consideriamo alcuni esempi. Sui sistemi basati su x86, quando viene premuto un tasto, l'hardware genera un interrupt e mette il numero del tasto, invece di un codice di carattere ASCII, in un registro del dispositivo. Inoltre, al rilascio di quel tasto è generato un altro interrupt, sempre con quel codice di tasto. Questa indirezione dà al sistema operativo la possibilità di usare il numero del tasto per indicizzare una tabella per prendere il carattere ASCII, il che semplifica la gestione di molte tastiere usate in giro per il mondo in paesi differenti. Il fatto di rilevare sia l'informazione di pressione sia di rilascio consente l'uso di qualunque tasto come tasto modificatore, poiché il sistema operativo conosce l'ordine in cui sono premuti e rilasciati i tasti.

L'indirezione è anche usata come output. I programmi possono scrivere caratteri ASCII sullo schermo, ma questi vengono poi interpretati come indici in una tabella per l'attuale font di output. La voce della tabella contiene la bitmap per il carattere. Questa indirezione rende possibile la separazione dei caratteri dal font. Un altro esempio di indirezione è l'utilizzo in UNIX dei major number dei dispositivi. All'interno del kernel c'è una tabella indicizzata col major number per i dispositivi a blocchi e un'altra per i dispositivi a caratteri. Quando un processo apre un file speciale come */dev/hd0*, il sistema estrae dall'i-node il tipo (a blocchi o a caratteri) e i major e minor number del dispositivo ed esegue una ricerca per indice nella tabella dei driver idonea, per ritrovare il driver. Questa indirezione semplifica la riconfigurazione del sistema, poiché i programmi hanno a che fare con nomi di dispositivi simbolici, non con i nomi dei dispositivi reali.

Ancora un altro esempio di indirezione avviene nei sistemi a passaggio di messaggi che definiscono una casella di posta piuttosto che un processo come destinazione del messaggio. Usando l'indirezione sulle caselle di posta elettronica (invece di definire un processo come destinazione) può essere ottenuta una flessibilità considerevole (per esempio, il fatto di avere una segretaria che gestisce i messaggi del suo capo).

In un certo senso, anche l'uso di macro del tipo:

```
#define PROC_TABLE_SIZE 256
```

è una forma di indirezione, poiché il programmatore può scrivere il codice senza dover conoscere la reale grandezza della tabella. È buona abitudine assegnare nomi simbolici a tutte le costanti (a eccezione talvolta di –1, 0 e 1), mettendole poi nei file d'intestazione con la spiegazione del loro utilizzo.

Riutilizzo del codice

Spesso è possibile riutilizzare lo stesso codice in contesti leggermente diversi. È una buona idea che riduce la dimensione del file binario e fa sì che si esegua il debug del codice una

volta sola. Per esempio, supponete che per tenere traccia dei blocchi liberi su disco si usino le bitmap. La gestione dei blocchi del disco può essere fatta facendo sì che le procedure *alloc* e *free* gestiscano le bitmap. Come condizione minima necessaria queste procedure dovrebbero funzionare su qualunque disco. Ma si può andare oltre. Le stesse procedure possono funzionare per gestire i blocchi di memoria, i blocchi nella cache dei blocchi del file system e gli i-node. Possono infatti essere utilizzate per allocare e liberare qualunque risorsa che possa essere numerata in modalità lineare.

Codice rientrante

Per codice rientrante si intende la capacità del codice di essere eseguito due o più volte simultaneamente. Su un multiprocessore esiste sempre il pericolo che mentre una CPU sta eseguendo una certa procedura, un'altra CPU cominci a eseguire la stessa procedura, prima che sia terminata la prima. In questo caso, potrebbero esservi due o più thread che eseguono il medesimo codice nello stesso momento. Ci si protegge da questa evenienza usando i mutex o qualche altro metodo per proteggere le zone critiche.

Il problema esiste tuttavia anche su un sistema monoprocessore. In particolare, la maggior parte dei sistemi operativi è in esecuzione con gli interrupt abilitati, altrimenti si perderebbero molti interrupt e i sistemi non sarebbero affidabili. Mentre il sistema operativo è impegnato nell'esecuzione di una qualche procedura, *P*, è del tutto possibile che si presenti un interrupt e che anche il gestore dell'interrupt chiami la procedura *P*. Se le strutture dati di *P* fossero in uno stato inconsistente al momento dell'interrupt, il gestore dell'interrupt si renderebbe conto di questo stato, generando un errore e fallendo.

Un ovvio esempio in cui ciò avviene è nel caso in cui *P* sia lo scheduler. Supponete che un processo abbia esaurito il suo quantum e che il sistema operativo lo stia muovendo al termine della coda. A un certo punto durante l'elaborazione della lista avviene l'interrupt, rende un certo processo *ready* ed esegue lo scheduler. Con le code in uno stato inconsistente, probabilmente il sistema andrà in errore, bloccandosi. Di conseguenza, anche su un monoprocessore, è meglio che la maggior parte del sistema operativo sia rientrante, che le strutture dati siano protette dai mutex e che gli interrupt siano disabilitati nel momento in cui non sia possibile tollerarli.

Forza bruta

Nel corso degli anni il ricorso alla forza bruta per risolvere un problema si è guadagnato una cattiva fama, ma spesso costituisce la migliore soluzione da adottare in nome della semplicità. Ogni sistema operativo ha molte procedure invocate raramente o che funzionano con così pochi dati da non valere la pena di ottimizzarli. Per esempio, è spesso necessario ricercare varie tabelle e array all'interno del sistema. L'algoritmo "forza bruta" consiste nel lasciare le tabelle nell'ordine in cui sono state inserite le voci ed eseguire una ricerca lineare al momento in cui serve. Se le voci sono poche (diciamo meno di mille), il guadagno che si ottiene ordinandole o facendone l'hash è minimo, ma il codice è molto più complicato ed è più probabile che vi possano essere errori.

Naturalmente, per funzioni che si trovano in passaggi critici, come gli scambi di contesto, deve essere fatto di tutto per renderle molto veloci, eventualmente anche scriverle in (il cielo non voglia) linguaggio assembly. La gran parte del sistema non si trova tuttavia in passaggi critici. Per esempio, molte chiamate di sistema sono invocate di rado. Se c'è una

`fork` ogni secondo e si impiega 1 ms per portarla a termine, anche ottimizzarla a 0 ms porta solamente a un guadagno dello 0,1%. Se il codice ottimizzato è più grosso e presenta più errori, è uno dei tipici casi in cui è meglio lasciar perdere l'ottimizzazione.

Controllo degli errori come prima mossa

Molte chiamate di sistema possono fallire, potenzialmente per una gran varietà di motivi: il file da aprire appartiene a qualcun altro; la creazione del processo fallisce a causa della tabella dei processi piena; un segnale non può essere inviato perché il processo di destinazione non esiste. Il sistema operativo deve controllare attentamente ogni possibile errore prima di intraprendere la chiamata. Molte chiamate di sistema richiedono inoltre l'acquisizione di risorse di sistema, come spazio nella tabella dei processi, spazio nella tabella degli i-node o descrittori dei file. Un buon consiglio generale che risparmia sgradevoli sorprese è controllare come prima cosa se il sistema è effettivamente in grado di portare a termine la chiamata prima dell'acquisizione delle risorse. Ciò significa mettere tutti i test in principio alla procedura che esegue la chiamata di sistema. Ciascun test dovrebbe essere nella forma

```
If (condizione_errore) return(ERROR_CODE);
```

Se la chiamata supera la serie di verifiche, certamente andrà a buon fine. A quel punto si acquisiscono le risorse.

Disseminare i test di acquisizione di risorse significa che, se un test fallisce, a quel punto le risorse acquisite devono essere restituite. Se si verifica un errore durante la fase di test e qualche risorsa non viene restituita, nell'immediato non viene arrecato alcun danno. Per esempio, una voce di una tabella dei processi può semplicemente diventare indisponibile in modo definitivo. Tuttavia, dopo un certo periodo di tempo, questo errore può essere rilevato più volte. Alla fine, la maggior parte o tutte le voci della tabella dei processi potrebbero diventare indisponibili, portando il sistema in crash, in una situazione estremamente imprevedibile e difficile da analizzare e correggere. Molti sistemi soffrono di questo problema riguardo le mancanze di memoria. Il programma chiama tipicamente *malloc* per allocare dello spazio, ma dimentica il successivo *free* per liberarlo. Per cui, gradualmente, la memoria si esaurisce sino al successivo riavvio del sistema. Per verificare questo genere di errori durante la fase di compilazione Engler et al. (2000) hanno proposto un metodo interessante. Hanno osservato che il programmatore conosce molte invarianti che il compilatore non conosce, come per esempio che quando si esegue un lock di un mutex, tutti i percorsi a partire dal lock devono contenere un unlock e nessun lock del medesimo mutex. Essi hanno quindi escogitato un modo per il programmatore di indicarlo al compilatore e di istruirlo affinché controlli tutti i percorsi compile-time in cerca di violazioni dell'invariante. Il programmatore può anche specificare che la memoria allocata deve essere rilasciata in ciascuno dei percorsi, così come molte altre condizioni.

12.4 Prestazioni

A parità di condizioni, un sistema operativo veloce è migliore di uno lento. Tuttavia, un sistema operativo veloce inaffidabile è peggio di uno lento ma affidabile. Considerato che

ottimizzazioni complesse spesso causano errori, è importante utilizzarle con parsimonia. A prescindere da ciò, ci sono punti in cui le prestazioni sono critiche e vale la pena porre in atto delle ottimizzazioni. Nei paragrafi successivi analizzeremo alcune tecniche generali utilizzabili per aumentare le prestazioni dove necessario.

12.4.1 Perché i sistemi operativi sono lenti?

Prima di parlare di tecniche di ottimizzazione, è giusto sottolineare che la lentezza di molti sistemi operativi è in gran parte autoinflitta. Per esempio, i più vecchi sistemi operativi, come MS-DOS e la versione 7 di UNIX, si avviavano in pochi secondi. I moderni sistemi UNIX e Windows 8 per avviarsi possono impiegare dei minuti, malgrado funzionino su hardware mille volte più veloci. Il motivo è che essi fanno molto di più rispetto ai precedenti, che si voglia o meno. Il plug-and-play è qualcosa che semplifica l'installazione di un nuovo dispositivo hardware, ma il prezzo pagato è che a *ogni* avvio il sistema operativo deve verificare tutto l'hardware per controllare se vi sia qualcosa di nuovo. Questa scansione dei bus richiede tempo.

Un approccio alternativo (e migliore, secondo l'opinione dell'autore) sarebbe cestinare tutto il plug-and-play e avere un'icona sullo schermo dal titolo "installa un nuovo hardware". Quando installa un nuovo dispositivo hardware, l'utente seleziona questa icona per far partire la scansione dei bus invece di farla in automatico a ogni avvio. I progettisti dei sistemi operativi erano ben consci di questa possibilità. Naturalmente l'hanno rifiutata poiché ritenevano che gli utenti fossero troppo stupidi per essere in grado di utilizzarla correttamente. Questo è solo un esempio, ma ce ne sono molti altri in cui l'obiettivo di rendere il sistema *user-friendly* (o "a prova di idiota", secondo il punto di vista) rallenta il sistema in continuazione per tutti indiscriminatamente.

Probabilmente la cosa migliore che i progettisti di sistemi possono fare per migliorare le prestazioni è essere molto più selettivi nell'aggiungere nuove caratteristiche. La domanda da porsi non è se all'utente può piacere l'opzione nuova, ma se ne vale la pena riguardo all'inevitabile prezzo in termini di dimensioni del codice, velocità, complessità e affidabilità. Dovrebbe essere inclusa solo nel caso in cui i vantaggi superino nettamente i problemi. I programmatori tendono a considerare che la dimensione del codice e il numero degli errori saranno 0 e che la velocità sarà infinita. L'esperienza dimostra che questa visione è un po' ottimistica.

Un altro fattore in gioco è il marketing del prodotto. Al momento in cui la versione 4 o 5 di un certo prodotto è sul mercato, probabilmente tutte le funzionalità effettivamente utili sono già state incluse e la maggior parte delle persone che ha bisogno di quel prodotto se lo è già procurato. Per far salire le vendite, molte aziende produttrici continuano a realizzare un flusso continuo di nuove versioni, con più funzionalità, solo per poter vendere i propri aggiornamenti ai clienti esistenti. Aggiungere nuove funzionalità solo per il gusto di farlo può aiutare le vendite, ma raramente aiuta le prestazioni.

12.4.2 Che cosa dovrebbe essere ottimizzato?

Come regola generale, la prima versione di un sistema dovrebbe essere la più semplice possibile. Le uniche ottimizzazioni dovrebbero riguardare aspetti che in maniera del tutto ovvia diventeranno un problema e sono quindi inevitabili. Un esempio: avere una cache dei blocchi per il file system. Quando il sistema è attivo e funzionante, dovrebbero essere prese

misure precise, in modo da verificare dove viene *realmente* speso il tempo. Basandosi su questi numeri, potrebbero essere realizzate delle ottimizzazioni dove sono più utili.

A questo proposito esiste una storia vera di come un'ottimizzazione è stata più un danno che un beneficio. Uno degli studenti dell'autore (di cui manterremo l'anonimato) ha scritto il programma *mkfs* originale di MINIX. Questo programma installa un file system nuovo di zecca su un disco appena formattato. Lo studente ha passato 6 mesi a ottimizzarlo, includendo la cache del disco. All'attivazione non funzionava e richiese molti mesi in più per il debug. Questo programma funziona in genere sull'hard disk una volta sola durante la vita del computer, quando è installato il sistema. Si avvia anche ogni volta che si formatta un floppy disk. Ogni esecuzione richiede circa 2 secondi. Se anche la versione non ottimizzata avesse richiesto 1 minuto per funzionare, sprecare così tanto tempo nell'ottimizzazione di un programma impiegato così poche volte è stato uno spreco di risorse.

Uno slogan che ha grande applicabilità all'ottimizzazione delle prestazioni è il seguente:

Abbastanza buono è abbastanza buono.

Con questo si intende che quando le prestazioni hanno raggiunto un livello ragionevole, probabilmente lo sforzo e la complessità di faticare dannatamente per ottenere la percentuale che manca al raggiungimento della perfezione del lavoro non valgono la pena. Se l'algoritmo di scheduling è abbastanza buono e tiene la CPU occupata il 90% del tempo, sta facendo il suo lavoro. Escogitarne uno molto più complesso, migliore del 5%, è probabilmente una cattiva idea. Analogamente, se la velocità di paginazione è abbastanza bassa da non essere un collo di bottiglia, solitamente non vale la pena fare i salti mortali per ottenere un rendimento ottimale. Evitare disastri è molto più importante di ottenere prestazioni ottimali, specialmente considerando che ciò che è ottimale per un carico di lavoro potrebbe non esserlo per un altro.

Un altro problema è quando ottimizzare il codice. Alcuni programmatori tendono a ottimizzare qualsiasi cosa sviluppino, non appena inizia a funzionare. Il problema è che, dopo l'ottimizzazione, il sistema è meno pulito ed è più difficile fare la manutenzione e il debug. Inoltre, diventa più difficile adattarlo; forse, rimandare l'ottimizzazione porta a risultati migliori. Il problema è noto come ottimizzazione prematura. Donald Knuth, citato a volte come padre dell'analisi degli algoritmi, disse una volta che "l'ottimizzazione prematura è la causa di tutti i mali".

12.4.3 Compromessi spazio-tempo

Un approccio generale per migliorare le prestazioni è trovare un compromesso fra il tempo e lo spazio. Capita spesso in informatica di dover scegliere fra un algoritmo che utilizza poca memoria ma è lento e un altro algoritmo che utilizza molta più memoria ma è più veloce. Quando si appronta un'ottimizzazione importante, è utile cercare algoritmi che aumentino la velocità utilizzando più memoria o che viceversa risparmino memoria preziosa facendo più calcoli.

Una tecnica che è spesso utile consiste nel sostituire le piccole procedure con delle macro. L'utilizzo di macro elimina il carico aggiuntivo normalmente associato con la chiamata della procedura. Il guadagno è significativo specialmente se la chiamata si verifica all'interno di un ciclo. Per esempio, supponete che utilizziamo delle bitmap per tenere traccia delle risorse e che abbiamo bisogno di sapere frequentemente quante unità sono libere in una certa porzione della bitmap. Per fare ciò abbiamo bisogno di una procedura, *bit_count*, che

conta il numero dei bit 1 in un byte. La semplice procedura è rappresentata nella Figura 12.7(a). Esegue ciclicamente il controllo dei bit all'interno del byte contandoli uno a uno.

Questa procedura è inefficiente per due motivi. Primo, deve essere chiamata, deve esserle assegnato uno spazio per lo stack e deve ritornare. Ogni chiamata della procedura ha questo carico aggiuntivo. Secondo, contiene un ciclo e questo comporta sempre un carico aggiuntivo associato.

Un approccio completamente diverso è quello di utilizzare la macro della Figura 12.7(b). È un'espressione inline che calcola la somma dei bit tramite spostamenti successivi del parametro, mascherando tutto eccetto il bit di livello più basso, e sommando gli otto termini. Difficilmente la macro è un'opera d'arte, ma compare nel codice una volta sola. Per esempio, quando la macro è chiamata da:

```
sum = bit_count(table[i]);
```

la chiamata della macro sembra identica alla chiamata della procedura. Quindi, invece di una qualsiasi definizione confusa, il codice non vede nulla di peggio nel caso della macro rispetto al caso della procedura, ma è molto più efficiente, in quanto elimina sia il carico aggiuntivo della chiamata di procedura sia quello del ciclo.

Possiamo prendere questo esempio e fare un passo avanti. Perché calcolare il conteggio dei bit? Perché non cercarlo in una tabella? Dopotutto ci sono solo 256 byte differenti, ognuno con un valore univoco da 0 a 8. Possiamo dichiarare una tabella di 256 voci, i bit, con ogni voce inizializzata (durante la compilazione) al numero di bit corrispondente a quel

```
#define BYTE_SIZE 8                          /* Un byte contiene 8 bit. */
int bit_count(int byte)
{                                            /* Conta i bit in un byte. */
      int i, count = 0;

      for (i = 0; i < BYTE_SIZE; i++)        /* Cicla sui bit di un byte. */
            if ((byte >> i) & 1) count++;    /* Se il bit vale 1 incrementa count. */
      return(count);                         /* Restituisce la somma. */
}
```

(a)

```
/* Macro per sommare i bit in un byte e restituire la somma */
#define bit_count(b) ((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                      ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))
```

(b)

```
/* Macro per contare il numero dei bit in una tabella */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

(c)

Figura 12.7 (a) Procedura per contare i bit in un byte. (b) Macro per contare i bit. (c) Macro che conta i bit tramite una ricerca in una tabella.

valore del byte. Con questo approccio non è necessario alcun calcolo durante il funzionamento, solo un'operazione d'indicizzazione. Nella Figura 12.7(c) è fornita una macro per svolgere questo compito.

Si tratta di un chiaro esempio in cui si sacrifica il tempo di calcolo in cambio di memoria. Tuttavia, potremmo andare oltre. Se sono necessari i conteggi dei bit per intere parole a 32 bit, utilizzando la nostra macro *bit_count*, abbiamo bisogno di eseguire quattro ricerche per parola. Se espandiamo la tabella a 65.536 voci, possiamo farci bastare due ricerche per parola, al prezzo di una tabella molto più grande.

La ricerca di risposte in una tabella può essere sfruttata in altre situazioni. Una tecnica di compressione molto nota, il formato GIF, utilizza una ricerca nelle tabelle per codificare pixel RGB a 24 bit. Tuttavia, il formato GIF funziona solo su immagini con al massimo 256 colori. Per ogni immagine da comprimere, si realizza una palette di 256 voci, ognuna contenente un valore RGB a 24 bit. L'immagine compressa consiste quindi di un indice a 8 bit per ogni pixel invece del valore RGB a 24 bit, con un guadagno di fattore tre. Questa idea è illustrata per una sezione 4 × 4 di un'immagine nella Figura 12.8. L'immagine originale compressa è mostrata nella Figura 12.8(a). Ogni valore è a 24 bit, rispettivamente con 8 bit per l'intensità del rosso, verde e blu. L'immagine GIF è mostrata nella Figura 12.8(b). Ogni valore è un indice a 8 bit nella palette dei colori. La palette dei colori è memorizzata come parte del file dell'immagine ed è rappresentata nella Figura 12.8(c). Effettivamente, ci sarebbe da dire altro sul formato GIF, ma l'idea principale rimane quella della ricerca nella tabella.

Esiste un altro sistema di riduzione della dimensione di un'immagine, che mostra un diverso compromesso. PostScript è un linguaggio di programmazione che può essere utilizzato per descrivere le immagini. (Effettivamente, qualunque linguaggio di programmazione può descrivere un'immagine, ma PostScript è fatto apposta.) Molte stampanti hanno un interprete PostScript incorporato che è in grado di eseguire i programmi PostScript che gli

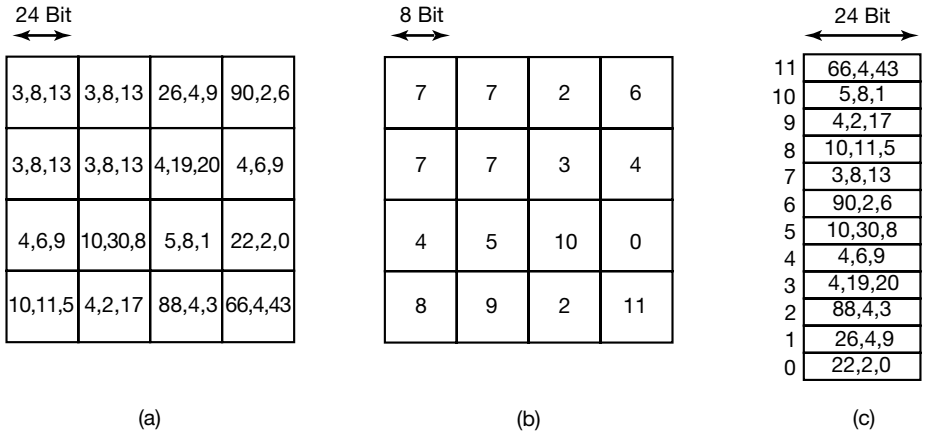

Figura 12.8 (a) Una parte di un'immagine non compressa con 24 bit per pixel. (b) La stessa parte compressa in GIF, con 8 bit per pixel. (c) La tavolozza (palette) dei colori.

vengono passati. Per esempio, se all'interno di un'immagine c'è un blocco rettangolare di pixel tutti dello stesso colore, un programma PostScript per l'immagine conterrebbe istruzioni per collocare un rettangolo in una determinata posizione e riempirlo con un certo colore. Per inviare questo comando serve solo una manciata di bit. Quando la stampante ha ricevuto l'immagine, un interprete avvia sulla stampante il programma per costruire l'immagine. Quindi PostScript ottiene una compressione di dati a spese di una maggior elaborazione, un differente bilanciamento rispetto alla ricerca in tabella, ma comunque valido quando la memoria o l'ampiezza della banda sono scarse.

Altri compromessi coinvolgono spesso le strutture dati. Liste doppiamente collegate utilizzano più memoria di quelle collegate singolarmente, ma spesso permettono un accesso più veloce agli elementi.

Le tabelle hash sono anche più dispendiose in termini di spazio, ma ancora più veloci. In breve, uno degli aspetti principali da considerare quando si ottimizza un pezzo di codice è valutare se l'utilizzo di una differente struttura dati potrebbe ottenere il miglior equilibrio spazio-tempo.

12.4.4 Uso della cache

Una tecnica molto conosciuta per aumentare le prestazioni è l'uso della cache. È applicabile ogni qual volta ci sia la probabilità che uno stesso risultato serva più volte. L'approccio generale è quello di fare l'intero lavoro la prima volta e poi salvare il risultato in una cache. Nei tentativi successivi, prima si controlla la cache. Se il risultato è al suo interno, è utilizzato, altrimenti l'intero lavoro è eseguito di nuovo.

Abbiamo già visto l'utilizzo della cache all'interno del file system per tenere memorizzato un certo numero di blocchi del disco utilizzati di recente, risparmiando una lettura del disco a ogni ricerca. Tuttavia, la cache può essere utilizzata anche per altri scopi. Per esempio, l'analisi dei nomi di percorso è estremamente onerosa. Considerate di nuovo l'esempio di UNIX della Figura 4.34. Per trovare */usr/ast/mbox* servono i seguenti accessi al disco.

1. Leggere l'i-node della directory root (i-node 1).
2. Leggere la directory root (blocco 1).
3. Leggere l'i-node per */usr* (i-node 6).
4. Leggere la directory */usr* (blocco 132).
5. Leggere l'i-node per */usr/ast* (i-node 26).
6. Leggere la directory */usr/ast* (blocco 406).

Solo per scoprire l'i-node del file servono sei accessi al disco. Poi l'i-node stesso deve essere letto per trovare i numeri dei blocchi del disco. Se il file è più piccolo delle dimensioni del blocco (per esempio, 1024 byte), servono 8 accessi per leggere i dati.

Alcuni sistemi ottimizzano l'analisi dei path name usando una cache per le combinazioni (percorso, i-node). Per l'esempio della Figura 4.34, la cache certamente manterrà le prime tre voci della Figura 12.9 dopo l'analisi di */usr/ast/mbox*. Le ultime tre voci vengono dall'analisi di altri percorsi.

Percorso	Numero dell'i-node
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

Figura 12.9 Una parte della cache dell'i-node della Figura 4.35.

Quando viene eseguita una ricerca in un percorso, il parser dei nomi prima consulta la cache e cerca in essa la sottostringa più lunga presente. Per esempio, se viene richiesto il percorso */usr/ast/grants/stw*, la cache restituisce il fatto che */usr/ast* è l'i-node 26, quindi la ricerca può partire da lì, eliminando quattro accessi al disco.

Un problema dell'utilizzo della cache per i percorsi è che la corrispondenza fra nome del file e numero dell'i-node non è fissa. Supponete che il file */usr/ast/mbox* sia rimosso dal sistema e il suo i-node sia riutilizzato per un file diverso, di proprietà di un diverso utente. In seguito, il file */usr/ast/mbox* è creato nuovamente e questa volta ottiene l'i-node 106. Se non viene fatto nulla per prevenirlo, la voce della cache stavolta sarà sbagliata e le ricerche successive restituiranno un numero di i-node errato. Per questa ragione, quando si cancella un file o una directory, la sua voce nella cache e (se si tratta di una directory) tutte le voci al di sotto devono essere eliminate dalla cache. I blocchi dei dischi e i nomi di percorso non sono gli unici elementi che possono essere messi nella cache. Anche gli i-node possono essere inseriti nella cache. Se sono utilizzati dei thread pop-up per gestire gli interrupt, ognuno di loro richiede uno stack e altri meccanismi aggiuntivi. Anche questi thread utilizzati in precedenza possono essere messi nella cache, dato che risistemarne uno usato è più semplice che crearne uno nuovo da zero (per evitare di dover allocare memoria). Qualsiasi cosa difficile da produrre può essere semplicemente messa nella cache.

12.4.5 Suggerimenti

Le voci della cache sono sempre corrette. Una ricerca nella cache può fallire, ma se viene trovata una voce, è garantito che sia corretta e che possa essere utilizzata senza timori. In alcuni sistemi, è comodo avere una **tabella di suggerimenti**. Si tratta di suggerimenti riguardanti la soluzione, ma non è garantito che siano corretti. Il chiamante deve verificarne il risultato.

Un esempio famoso di suggerimenti sono gli URL inseriti nelle pagine web. La selezione di un collegamento non garantisce il fatto che la pagina web indicata esista effettivamente. In realtà, la pagina indicata potrebbe essere stata rimossa 10 anni fa, quindi l'informazione sulla pagina indicata è effettivamente solo un suggerimento.

I suggerimenti sono utilizzati anche nella connessione con file remoti. L'informazione nel suggerimento indica qualcosa a proposito del file remoto, come la posizione dove si

trovava l'ultima volta che è stato referenziato. Tuttavia, il file può essere stato spostato o cancellato da quando il suggerimento è stato registrato, quindi è sempre necessario un controllo per verificare l'esattezza dell'informazione.

12.4.6 Sfruttare la località

I processi e i programmi non agiscono a caso. Essi mostrano una discreta quantità di località nel tempo e nello spazio; questa informazione può essere impiegata in vari modi per aumentare le prestazioni. Un esempio famoso di località spaziale è rappresentato dal fatto che i processi non si muovono a caso all'interno dei loro spazi degli indirizzi. Essi tendono a utilizzare un numero relativamente piccolo di pagine durante un dato intervallo di tempo. Le pagine utilizzate in modo attivo da un processo possono essere annotate come il suo working set; il sistema operativo può assicurarsi che mentre il processo è autorizzato a essere in esecuzione, il suo working set si trovi in memoria, riducendo così il numero di errori di pagina.

Il principio della località ha senso anche per i file. Quando un processo ha selezionato una particolare directory di lavoro, è probabile che molti dei suoi futuri riferimenti ai file siano relativi ai file in quella directory. Mettendo tutti gli i-node e i file di ogni directory strettamente vicini sul disco, si possono ottenere miglioramenti delle prestazioni. Questo principio corrisponde a ciò che è sottolineato nel Berkeley Fast File System (McKusick et al., 1984).

Un'altra area in cui la località è importante è nello scheduling dei thread nei multiprocessori. Come abbiamo visto nel Capitolo 8, un modo per schedulare i thread in un multiprocessore è tentare di eseguire ogni thread sulla CPU che ha usato l'ultima volta, sperando che alcuni dei suoi blocchi di memoria siano ancora nella cache della memoria.

12.4.7 Ottimizzazione del caso comune

È spesso una buona idea distinguere il caso più comune dal caso peggiore possibile e trattarli in modo diverso. Spesso il codice dei due casi è piuttosto diverso. È importante rendere veloce il caso comune; per il caso peggiore, se si verifica raramente, è sufficiente che sia corretto.

Come primo esempio, considerate l'ingresso in una zona critica. La maggior parte delle volte l'ingresso andrà a buon fine, specialmente se i processi non rimangono molto tempo all'interno delle zone critiche. Windows 8 sfrutta questa aspettativa fornendo una chiamata API Win32, chiamata `EnterCriticalSection`, che testa atomicamente un flag in modalità utente (utilizzando `TSL` o un equivalente). Se il test ha successo, il processo entra semplicemente nella zona critica e non viene richiesta alcuna chiamata nel kernel. Se il test fallisce, la procedura di libreria esegue un `down` su un semaforo per bloccare il processo. Quindi nel caso normale non è necessaria alcuna chiamata nel kernel. Nel Capitolo 2 abbiamo visto che i mutex in Linux sono similmente ottimizzati per il caso comune di mancanza di contenzione.

Come secondo esempio, considerate di impostare un allarme (utilizzando i segnali in UNIX). Se non ci sono allarmi in sospeso in quel momento, è semplice creare una voce e metterla in coda al timer. Tuttavia, se un allarme è già in attesa, deve essere trovato e rimosso dalla coda del timer. Poiché la chiamata `alarm` non specifica se già vi sia un allarme impostato, il sistema deve considerare il caso peggiore, cioè che vi sia. Tuttavia, considerato che per la maggior parte dei casi non vi sono allarmi in sospeso e che togliere un allarme esistente è oneroso, è una buona idea distinguere fra i due casi.

Uno dei modi per farlo è tenere un bit nella tabella del processo che indica se vi è un allarme in attesa. Se il bit è *off*, si segue il percorso facile (aggiungere una nuova voce alla coda del timer senza controllare). Se il bit è *on*, bisogna verificare la coda del timer.

12.5 Gestione dei progetti

I programmatori sono inguaribili ottimisti. La maggior parte di loro pensa che il modo migliore di scrivere un programma sia quello di mettersi davanti a una tastiera e iniziare a digitare. In poco tempo il programma, perfettamente funzionante, è pronto. Per programmi molto grandi, non funziona esattamente in questo modo. Nei paragrafi seguenti parliamo un po' della gestione di grandi progetti software, specialmente dei progetti di grandi sistemi operativi.

12.5.1 The Mythical Man Month

Nel suo libro classico, Fred Brooks, uno dei progettisti dell'OS/360 che poi è passato al mondo accademico, affronta il problema del perché sia così difficile creare grandi sistemi operativi (Brooks, 1975, 1995). La maggior parte dei programmatori, quando legge l'affermazione che i programmatori, su grandi progetti, possono produrre *in un anno* solo 1000 righe di codice *corretto* si chiede se il Prof. Brooks viva nello spazio siderale, forse sul Pianeta Bug. Dopotutto, molti di loro sicuramente ricordano la volta in cui produssero un programma di 1000 righe in una sola notte. Come potrebbe trattarsi della produzione annuale di una qualsiasi persona con un QI > 50?

Quello che Brooks voleva sottolineare è che i grandi progetti, con centinaia di programmatori, sono completamente differenti rispetto ai piccoli progetti e che il risultato ottenuto per i piccoli progetti non cresce proporzionalmente per quelli grandi. In un grande progetto, un'enorme quantità di tempo si passa a pianificare come suddividere il lavoro in moduli, specificando attentamente i moduli, le loro interfacce e tentando di immaginare come interagiranno fra loro, ancora prima di iniziare la scrittura del codice. Poi i moduli devono essere tradotti in codice e corretti uno per uno. Alla fine, i moduli devono essere integrati e il sistema deve essere testato come entità unica. La norma è che i moduli funzionano perfettamente quando testati da soli, ma quando i pezzi sono messi insieme il sistema si blocca all'istante. Brooks ha stimato la seguente ripartizione del lavoro:

- 1/3 di pianificazione
- 1/6 di scrittura del codice
- 1/4 di test dei moduli
- 1/4 di test del sistema.

In altre parole, scrivere il codice è la parte facile. La parte difficile è immaginarsi quali dovranno essere i moduli e fare sì che il modulo *A* comunichi correttamente con il modulo *B*. In un programma piccolo scritto da un singolo programmatore, tutto ciò che rimane è la parte facile. Il titolo del libro di Brooks deriva dalla sua asserzione per cui le persone e il tempo non sono intercambiabili. Non esiste un'unità uomo-mese (o persona-mese). Se un

progetto richiede 15 persone e 2 anni per essere realizzato, è inconcepibile che 360 persone possano farlo in un mese e probabilmente è impossibile avere 60 persone che lo completino in 6 mesi.

Ci sono tre motivi alla base di questa considerazione. Primo, il lavoro non può esser gestito interamente in parallelo. Nessuna operazione di codifica può essere avviata prima che sia stata preparata la pianificazione, siano stati identificati i moduli necessari e le loro interfacce. In un progetto di soli due anni la pianificazione può comportare 8 mesi.

Secondo, per utilizzare appieno un elevato numero di programmatori, il lavoro deve essere suddiviso in un elevato numero di moduli in modo che tutti abbiano qualcosa da fare. Considerato che ogni modulo può potenzialmente interagire con ogni altro modulo, il numero di interazioni modulo-modulo che deve essere considerato cresce con il quadrato del numero dei moduli, cioè con il quadrato del numero dei programmatori. Questa complessità sfugge rapidamente di mano. Misurazioni precise su 63 progetti software hanno confermato che il compromesso fra le persone e i mesi è molto lontano dall'essere lineare nei casi di grandi progetti (Boehm, 1981).

Terzo, il debug è altamente sequenziale. Stabilire di avere 10 revisori su un problema non velocizza di 10 volte il ritrovamento del bug. In effetti, dieci controllori sono probabilmente più lenti di uno, perché sprecherebbero tanto tempo in chiacchiere.

Brooks riassume la sua esperienza sui bilanciamenti persone-tempo all'interno della "Legge di Brooks":

Aggiungere manodopera a un progetto software in ritardo lo fa ritardare di più.

Il problema relativo all'aggiunta di persone è rappresentato dal fatto che queste devono essere addestrate sul progetto, i moduli devono essere ridistribuiti per essere destinati al maggior numero di programmatori ora disponibili, saranno necessari molti incontri per coordinare gli sforzi e così via. Abdel-Hamid e Madnick (1991) hanno confermato questa legge in modo sperimentale. Un modo un po' irriverente di riformulare la legge di Brooks è il seguente:

Servono 9 mesi per mettere al mondo un bambino, indipendentemente dal numero delle donne a cui affidate questo compito.

12.5.2 Struttura del team

I sistemi operativi commerciali sono grandi progetti software e richiedono inevitabilmente team di tante persone. La qualità delle persone è immensamente importante. Si sa da decenni che i programmatori migliori sono 10 volte più produttivi dei programmatori mediocri (Sackman et al., 1968). Il problema nasce quando occorrono 200 programmatori, in quanto è difficile trovare 200 programmatori eccellenti; dovrete accontentarvi di uno spettro più ampio di qualità.

Un altro aspetto importante in ogni grande progetto, software o di altro genere, è la necessità di coerenza architetturale. Dovrebbe esserci una sola mente che controlla il progetto. Brooks cita la cattedrale di Reims in Francia come esempio di un grande progetto che ha richiesto decenni per la costruzione, nel quale gli architetti che arrivarono in seguito subordinarono il proprio desiderio di lasciare una loro impronta nell'opera al fine di portare avanti i piani dell'architetto originale. Il risultato è una coerenza architetturale non riscontrata in altre cattedrali europee.

Negli anni Settanta Harlan Mills ha combinato l'osservazione che molti programmatori sono decisamente migliori di altri con la necessità di coerenza architetturale per proporre il paradigma del **team del programmatore capo** (Baker, 1972). La sua idea era organizzare un team di programmazione come un'équipe chirurgica, invece che come uno staff di macellai. Invece di avere ogni persona coinvolta a sezionare pezzi senza criterio, è meglio avere una sola persona che tiene il bisturi in mano; tutti gli altri sono lì a dare una mano. Per un progetto che coinvolge 10 persone, Mills ha suggerito la struttura del team descritta nella Figura 12.10.

Sono passati tre decenni da quando questa struttura è stata proposta e messa in produzione. Alcune cose sono cambiate (come la necessità di un esperto del linguaggio – il C è più semplice del PL/I), ma la necessità di avere una sola mente che controlla il progetto è sempre valida. E quella mente dovrebbe essere in grado di dedicarsi al 100% alla progettazione e alla programmazione, da qui l'esigenza di avere uno staff di supporto che oggi, con l'aiuto del computer, può essere composto da un numero minore di persone. Ma l'idea nella sua essenza è ancora valida.

Ogni grande progetto deve essere organizzato in modo gerarchico. Alla base ci sono molti piccoli team, ognuno diretto da un capo programmatore. Al livello successivo, gruppi di team coordinati da un manager. L'esperienza mostra che ogni persona da gestire comporta un 10% del vostro tempo, quindi serve un manager a tempo pieno per ogni gruppo di 10 team. Questi manager devono essere gestiti a loro volta e così via.

Brooks ha osservato che le cattive notizie non risalgono bene lungo questa struttura ad albero. Jerry Saltzer del M.I.T. ha chiamato questo effetto **diodo delle cattive notizie**. Nessun capo programmatore o il suo manager vuole dire al proprio capo che il progetto è in ritardo di 4 mesi e non ha alcuna possibilità di rispettare la scadenza, in quanto esiste una tradizione vecchia di 2000 anni sulla decapitazione del messaggero che porta cattive notizie. Di conseguenza, il top management è generalmente all'oscuro circa lo stato del progetto.

Titolo	Compiti
Capo programmatore	Predispone il progetto architetturale e scrive il codice
Co-pilota	Aiuta il capo programmatore e funge da portavoce
Amministratore	Gestisce persone, budget, spazio, strumentazione, report, ecc.
Editor	Fa la revisione della documentazione che deve essere scritta dal capo programmatore
Segretarie	L'amministratore e il correttore necessitano ciascuno di una segretaria
Responsabile dei programmi	Cura gli archivi del codice e dei documenti
Responsabile dei tool	Fornisce tutti gli strumenti necessari al capo programmatore
Collaudatore	Collauda il codice del capo programmatore
Specialista del linguaggio	Una funzione part-time che può consigliare il capo programmatore su questioni relative al linguaggio

Figura 12.10 Proposta di Mills per la composizione di un team del programmatore capo costituito da 10 persone.

Quando diventa ovvio che la scadenza non può essere rispettata, il top management risponde aggiungendo persone, con il conseguente verificarsi della Legge di Brooks.

In pratica, le grandi aziende che hanno maturato una lunga esperienza nella produzione di software e sanno che cosa può accadere se è prodotto a casaccio, hanno la tendenza almeno a tentare di farlo nel modo corretto. Al contrario, le aziende nuove e più piccole, che hanno una grande fretta di mettersi sul mercato, non sempre prendono le dovute precauzioni nel produrre i loro software. Questa fretta spesso porta ben lontano dai risultati ottimali.

Né Brooks né Mills hanno previsto la crescita del movimento del software open source. Anche se molti ne dubitavano (soprattutto chi gestisce grandi aziende di software closed-source), il software open source ha avuto un successo clamoroso. Dai grandi server ai sistemi embedded, dai sistemi di controllo industriali agli smartphone, il software open source si trova ovunque. Grandi aziende com Google e IBM hanno iniziato a contribuire in maniera sostanziale a grandi progetti open source come Linux. Quello che è da rilevare è che i progetti software open source che hanno avuto maggior successo sono quelli che hanno utilizzato chiaramente il modello del team del programmatore capo, in base al quale una sola mente controlla la progettazione architetturale (per esempio Linus Torvalds per il kernel di Linux e Richard Stallman per il compilatore C di GNU).

12.5.3 Ruolo dell'esperienza

I progettisti con esperienza rappresentano un punto critico nel progetto di un sistema operativo. Brooks fa notare che la maggior parte degli errori non è nel codice ma nel progetto. I programmatori hanno eseguito correttamente quello che gli era stato richiesto; è questo che era sbagliato. Nessun test, per quanti se ne possano fare, è in grado di rilevare delle cattive specifiche iniziali.

La soluzione proposta da Brooks è abbandonare il modello di sviluppo classico della Figura 12.11(a) e utilizzare il modello della Figura 12.11(b). In questo caso l'idea è di scrivere dapprima un programma principale che chiami solamente le procedure di alto livello, che all'inizio sono fittizie. A partire dal primo giorno di vita del progetto, il sistema potrà essere compilato ed eseguito, sebbene non faccia nulla. Man mano che il tempo passa, i moduli verranno inseriti nel sistema completo. Il risultato di questo approccio è che il test sull'integrazione nel sistema è effettuato continuamente, quindi gli eventuali errori di progettazione emergono prima. In effetti, il processo di apprendimento causato dalle decisioni di progettazione sbagliate entra prima nel ciclo.

Una conoscenza scarsa è una cosa pericolosa. Brooks ha osservato quello che chiama **effetto del secondo sistema**. Spesso il primo prodotto creato da un team di progettazione è ridotto perché i progettisti temono che possa non funzionare del tutto. Di conseguenza, esitano ad aggiungere molte funzionalità. Se il progetto incontra il favore degli utenti, costruiscono un secondo sistema. Impressionati dal loro personale successo, la seconda volta i progettisti includono tutte le possibili funzionalità che erano state intenzionalmente estromesse la prima volta. Quindi il secondo sistema è ridondante e presenta prestazioni scarse. La terza volta hanno smaltito l'effetto del fallimento della seconda e sono di nuovo cauti.

La coppia CTSS-MULTICS è un chiaro esempio di questo fenomeno. CTSS è stato il primo sistema in timesharing generalista e ha avuto un enorme successo pur avendo funzionalità minime. Il suo successore, MULTICS, è stato troppo ambizioso e ne ha sofferto

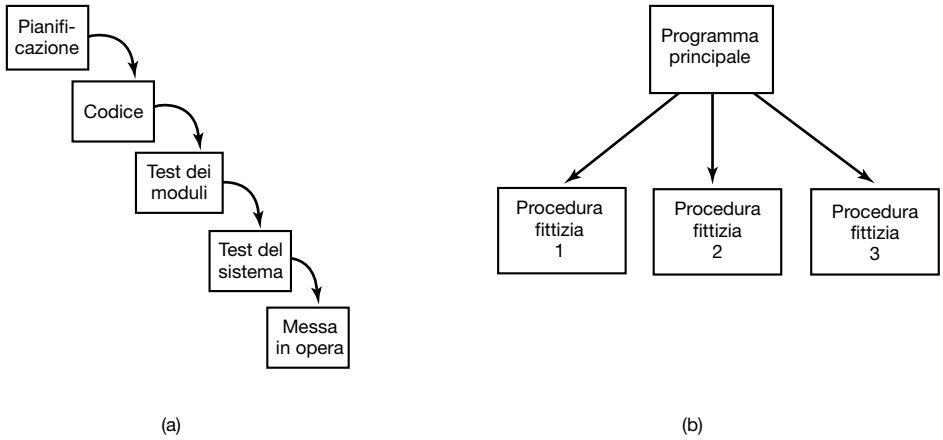

Figura 12.11 (a) Passi tradizionali di un progetto software. (b) Un progetto alternativo produce un sistema funzionante (che non fa niente) a partire dal primo giorno.

gravemente. Le idee alla base erano buone, ma c'erano troppe novità, quindi il sistema ha avuto per anni prestazioni scarse e non è mai stato un successo commerciale. Il terzo sistema in questa linea di sviluppo, UNIX, è stato molto più cauto e ha avuto molto più successo.

12.5.4 No Silver Bullet

Oltre a *The Mythical Man Month*, Brooks ha scritto un'altra influente pubblicazione intitolata *No Silver Bullet* (Brooks, 1987). In essa, sosteneva che nessuno dei tanti toccasana che venivano venduti in giro ai quei tempi sarebbe stato in grado di produrre un miglioramento significativo nella produttività del software entro un decennio. Il tempo ha dimostrato che aveva ragione.

Tra queste "pallottole d'argento" proposte, le migliori sono state i linguaggi ad alto livello, la programmazione orientata agli oggetti, l'intelligenza artificiale, i sistemi esperti, la programmazione automatica, la programmazione grafica, la verifica dei programmi e gli ambienti di programmazione. Forse nel prossimo decennio troveremo una "pallottola d'argento", ma è probabile che dovremo accontentarci di miglioramenti graduali e incrementali.

12.6 Tendenze nella progettazione dei sistemi operativi

È sempre difficile fare pronostici. Per esempio, nel 1899, il capo dell'ufficio brevetti, Charles H. Duell, chiese al Presidente McKinley di abolire l'ufficio brevetti (e il proprio lavoro!) perché, come disse: "Ogni cosa che può essere inventata è stata già inventata" (Cerf e Navasky, 1984). Tuttavia, Thomas Edison si presentò alla sua porta dopo pochi anni con un insieme di nuove invenzioni, tra cui la luce elettrica, il fonografo e il proiettore di filmati. Il punto è che il mondo è in costante cambiamento e i sistemi operativi si devono adattare di continuo alle nuove realtà. In questo paragrafo esamineremo alcune delle nuove tendenze importanti per progettisti di sistemi operativi di oggi.

Per evitare confusione, gli sviluppi hardware di cui si parla più avanti sono già qui. Ciò che ancora non esiste è il sistema operativo in grado di sfruttarli appieno.

In generale, quando nasce un nuovo hardware tutti cercano di adattarvi il vecchio software (Linux, Windows e così via). Alla lunga, si tratta di una pessima idea; è invece necessario un software innovativo in grado di gestire hardware innovativo. Se siete studenti di informatica o di ingegneria, o professionisti ICT, il vostro compito è pensare a questo software.

12.6.1 Virtualizzazione e cloud

La virtualizzazione è un'idea che è ritornata in auge. Emerse la prima volta nel 1967 con il sistema IBM CP/CMS, ma ora è tornata in pieno sulla piattaforma x86. Nel futuro immediato, molti computer eseguiranno gli hypervisor sull'hardware nudo, come illustrato nella Figura 12.12. L'hypervisor predispone una serie di macchine virtuali, ciascuna con il proprio sistema operativo. Questo fenomeno è stato affrontato nel Capitolo 7 e rappresenta sicuramente un segnale del futuro. Oggi molte aziende fanno un passo in più e virtualizzano anche le risorse. Per esempio, c'è molto interesse nella virtualizzazione del controllo dei dispositivi di rete, fino a portare anche il controllo delle reti nel cloud. Inoltre, produttori e ricercatori lavorano costantemente per migliorare gli hypervisor, rendendoli più piccoli, più veloci, con migliori capacità di isolamento.

12.6.2 Chip multicore

Ci fu un tempo in cui la memoria era così scarsa che un programmatore conosceva personalmente ciascun bit ed era invitato al suo compleanno. Oggi, i programmatori non si preoccupano quasi più di sprecare qualche megabyte qua e là. Per la maggior parte delle applicazioni, la memoria non è più una risorsa scarsa. Che cosa accadrà quando ci sarà una simile abbondanza di core? Detta in modo diverso: a mano a mano che i produttori inseriscono su un singolo chip sempre più core, che cosa accadrà se ce ne saranno talmente tanti che i programmatori smetteranno di preoccuparsi di sprecare qualche core qua e là?

I chip multicore sono già in uso, ma i sistemi operativi concepiti per essi non li usano bene. In effetti, i sistemi operativi oggi disponibili spesso non scalano nemmeno a poche decine di core e gli sviluppatori cercano continuamente di rimuovere tutti i colli di bottiglia che limitano la scalabilità.

Una domanda ovvia è: che cosa farsene di tutti questi core? Se si esegue un server che gestisce parecchie migliaia di richieste al secondo, la risposta può essere relativamente semplice. Per esempio, si può decidere di dedicare un core a ciascuna richiesta. Se non si presenta troppo spesso il problema del lock, il tutto può funzionare. Ma che cosa si può fare di tutti questi core su un tablet?

Un'altra domanda è che *tipo* di core può servire? Core superscalari con pipeline profonda, con un'esecuzione speculativa non sequenziale e velocità di clock elevatissime possono essere l'ideale per l'esecuzione di codice sequenziale, ma non per la bolletta dell'elettricità. Inoltre, se il lavoro ha molto parallelismo non sono ideali. Molte applicazioni funzionano meglio con core più piccoli e semplici, se si riesce ad averne parecchi. Alcuni esperti sostengono che i migliori sono i multicore eterogenei, ma la domanda resta la stessa: quali tipi di core, quanti e a quale velocità? E non abbiamo nemmeno iniziato a

parlare del problema dell'esecuzione di un sistema operativo e di tutte le sue applicazioni. Il sistema operativo verrà eseguito su tutti i core contemporaneamente o su uno alla volta? Ci sarà più di uno stack di rete o uno solo? Di quanta condivisione ci sarà bisogno? Alcuni core saranno dedicati a funzioni specifiche del sistema operativo (come lo stack di rete o di memorizzazione)? In questo caso, le funzioni dovranno essere replicate per migliorare la scalabilità?

Esplorando molte direzioni diverse, il mondo dei sistemi operativi sta oggi cercando di formulare risposte a queste domande. Se pure i ricercatori possono non essere d'accordo sulle risposte, la maggior parte di loro concorda su una cosa: sono tempi molto belli per la ricerca sui sistemi operativi!

12.6.3 Sistemi operativi con spazi degli indirizzi grandi

Quando le macchine si sono evolute dallo spazio degli indirizzi a 32 bit a quello a 64 bit è stato possibile introdurre importanti cambiamenti nella progettazione dei sistemi operativi. Uno spazio degli indirizzi a 32 bit non è poi così grande. Se voi aveste provato a dividere 2^{32} byte dandone una parte a ogni essere umano sulla terra, non ne avreste avuto abbastanza per tutti. Invece, 2^{64} corrisponde a 2×10^{19}. Adesso tutti avrebbero un bel pezzo da 3 GB.

Che cosa possiamo fare con uno spazio degli indirizzi di 2×10^{19} byte? Innanzitutto potremmo eliminare il concetto di file system. Al suo posto, tutti i file potrebbero essere concettualmente mantenuti tutto il tempo in memoria (virtuale). Dopotutto, c'è abbastanza spazio per più di 1 miliardo di lungometraggi, ognuno compresso in 4 GB.

Un altro possibile impiego è una memorizzazione persistente degli oggetti. Gli oggetti potrebbero essere creati nello spazio degli indirizzi ed esservi tenuti finché non sono stati eliminati tutti i riferimenti a essi, momento in cui potrebbero essere automaticamente cancellati. Questi oggetti sarebbero persistenti nello spazio degli indirizzi, anche allo spegnimento e al riavvio del computer. Con uno spazio degli indirizzi a 64 bit, gli oggetti potrebbero essere creati a una velocità di 100 MB/s per 5000 anni prima di esaurire tutto lo spazio. Naturalmente, per memorizzare effettivamente questa massa di dati, servirebbe molta memoria su disco per il traffico delle pagine, ma, per la prima volta nella storia, il fattore limitante sarebbe la memoria su disco e non lo spazio degli indirizzi.

Con grandi quantità di oggetti nello spazio degli indirizzi, diventa interessante permettere a più processi di funzionare nello stesso spazio degli indirizzi contemporaneamente per condividere gli oggetti in modo generale. Questo tipo di progettazione porterebbe chiaramente a sistemi operativi veramente differenti rispetto a quelli odierni.

Un'altra questione dei sistemi operativi che dovrebbe essere ripensata con indirizzi a 64 bit riguarda la memoria virtuale. Con 2^{64} byte di spazio degli indirizzi virtuali e pagine di 8 KB si ottengono 2^{51} pagine. Le tabelle delle pagine convenzionali non sono adatte per questa dimensione, per cui serve qualcos'altro. Una possibilità è rappresentata dalle tabelle delle pagine invertite, ma sono state proposte anche altre idee (Talluri et al., 1995). In ogni caso c'è spazio in abbondanza per la nuova ricerca nel campo dei sistemi operativi a 64 bit.

12.6.4 Accesso ai dati senza soluzione di continuità

Fin dalla nascita dei computer c'è stata una netta distinzione fra *questa* macchina e *quella* macchina. Se i dati si trovavano su *questa* macchina, non era possibile accedervi da *quella* macchina, a meno che non venissero prima esplicitamente trasferiti. Allo stesso modo, anche

se si avevano i dati, non li si poteva utilizzare a meno che non si fosse installato il software adeguato. Questo è un modello che sta cambiando.

Oggi gli utenti si aspettano che la maggior parte dei propri dati siano accessibili da qualsiasi luogo, in qualsiasi istante. Solitamente questo risultato si ottiene memorizzando i dati nel cloud, utilizzando servizi di storage online come DropBox, Google Drive, iCloud e SkyDrive. Tutti i dati memorizzati in questo modo possono essere recuperati da qualsiasi dispositivo dotato di una connessione di rete. Inoltre, i programmi per l'accesso a questi dati si trovano spesso anch'essi nel cloud e pertanto non è nemmeno necessario installarli sul proprio dispositivo. Le persone possono quindi leggere e modificare file di elaboratori di testi, fogli di calcolo e presentazioni utilizzando uno smartphone mentre se ne stanno seduti in bagno. Per lo più questo viene inteso come un progresso.

Riuscire a fare in modo che tutto questo non abbia soluzione di continuità non è un compito facile e richiede molte soluzioni intelligenti nel sistema operativo. Per esempio, che cosa si deve fare quando non c'è una connessione di rete? Ovviamente, le persone non possono permettersi di smettere di lavorare; si può preparare un buffer con tutte le modifiche in locale e aggiornare il documento principale quando la connessione torna disponibile, ma che cosa bisogna fare se più dispositivi hanno apportato modifiche in conflitto una con l'altra?

Si tratta di un problema molto comune quando ci sono più utenti che condividono dati, ma può capitare anche all'utente singolo; inoltre, se il file è di grandi dimensioni, non bisognerebbe dover aspettare troppo tempo per potervi accedere. Cache, precaricamento e sincronizzazione sono tutti problemi fondamentali. Gli attuali sistemi operativi possono unire più macchine, ma non senza soluzione di continuità; si può fare sicuramente di meglio.

12.6.5 Computer a batterie

PC potenti, con spazi degli indirizzi a 64 bit, collegamento in rete a banda larga, più processori e audio e video di alta qualità, sono oggi lo standard dei sistemi desktop e si stanno rapidamente diffondendo anche su portatili, tablet e perfino smartphone. Al proseguire di questa tendenza, i loro sistemi operativi dovranno essere apprezzabilmente diversi da quelli attuali, per essere in grado di gestire tutte queste richieste. Inoltre, dovranno bilanciare il consumo energetico e non surriscaldarsi; la dissipazione del calore e il consumo di energia sono alcune delle sfide più difficili anche nei computer di fascia più alta.

Peraltro, un settore del mercato con una crescita ancora più veloce è quello dei computer a batterie, inclusi i notebook, i tablet, i portatili ultraeconomici e gli smartphone. La maggior parte di essi hanno connessioni wireless verso il mondo esterno e hanno bisogno di sistemi operativi diversi, che siano più piccoli, più veloci, più flessibili e più affidabili rispetto a quelli attuali. Molti di questi dispositivi sono oggi basati su sistemi operativi tradizionali come Linux, Windows e OS X, ma con modifiche significative. Inoltre, spesso utilizzano una soluzione basata su microkernel/hypervisor per gestire lo stack delle radiocomunicazioni.

Questi sistemi operativi devono gestire le operazioni con connessione completa (cioè wired), con connessione debole (cioè wireless) o senza connessione, inclusa la raccolta dei dati prima della disconnessione e la gestione della loro consistenza al ritorno in linea. In futuro dovranno inoltre gestire i problemi di mobilità (per esempio rintracciare una stampante laser, collegarsi a essa e inviarle un file via radio). È essenziale anche la gestione dell'e-

nergia, incluso il dialogo assiduo tra il sistema operativo e le applicazioni circa la quantità di energia della batteria restante e su come meglio impiegarla. Può diventare importante l'adattamento dinamico delle applicazioni per gestire le limitazioni legate a schermi di piccole dimensioni. Infine, le nuove modalità di input e output, quali la scrittura manuale e la voce, potrebbero richiedere nuove tecniche nei sistemi operativi, per migliorarne la qualità. È probabile che il sistema operativo di un computer a batterie, palmare e wireless a controllo vocale, sia sostanzialmente diverso da quello di un computer desktop a 64 bit multiprocessore a quattro CPU con una connessione di rete a fibra ottica di un gigabit. Naturalmente non mancheranno innumerevoli macchine ibride con i loro propri requisiti.

12.6.8 Sistemi embedded

Un'ultima area nella quale i sistemi operativi prolifereranno è quella dei sistemi embedded. I sistemi operativi all'interno di lavatrici, forni a microonde, giochi elettronici, bambole, radio, lettori MP3, videocamere, ascensori e pacemaker saranno differenti da quelli descritti in questo libro e molto probabilmente diversi fra loro. Ognuno sarà probabilmente preparato attentamente su misura, in base alle proprie applicazioni specifiche, in quanto è improbabile che nessuno inserirà mai una scheda PCI in un pacemaker per trasformarlo nella scheda di controllo di un ascensore.

Dato che tutti i sistemi embedded funzionano con un numero ristretto di programmi, ben conosciuti al momento della progettazione, potrebbero essere possibili ottimizzazioni altrimenti non fattibili in sistemi per un utilizzo generico.

Un'idea promettente per i sistemi integrati è il sistema operativo estensibile (per esempio Paramecium ed Exokernel). Questi potranno essere realizzati in versione leggera o pesante come richiesto dall'applicazione in questione, ma in maniera consistente tra le applicazioni. Dato che i sistemi embedded saranno prodotti in centinaia di milioni, sarà un mercato importante per i nuovi sistemi operativi.

12.7 Riepilogo

Per progettare un sistema operativo si parte determinando che cosa dovrà fare. L'interfaccia deve essere semplice, completa ed efficiente. Deve essere molto chiaro il paradigma dell'interfaccia utente, dell'esecuzione e dei dati.

Il sistema dev'essere ben strutturato, utilizzando una delle tecniche note, come la struttura a livelli o quella client-server. I componenti interni devono essere ortogonali uno all'altro e separare in modo chiaro la policy dai meccanismi. Bisogna pensare con attenzione a questioni come la struttura statica o dinamica, il naming, il binding time e l'ordine di implementazione dei moduli.

Le prestazioni sono importanti, ma bisogna scegliere accuratamente le ottimizzazioni per non rovinare la struttura del sistema. Spesso vale la pena eseguire bilanciamento fra spazio e tempo, caching, suggerimenti, sfruttamento delle località e ottimizzazioni dei casi più comuni.

Scrivere un sistema con un paio di persone è molto diverso rispetto a produrre un grande sistema con 300 persone. In quest'ultimo caso, la struttura del team e la gestione del progetto sono essenziali per il successo o il fallimento del progetto.

Infine, i sistemi operativi stanno cambiando per adattarsi alle nuove tendenze e affrontare le nuove sfide, fra cui i sistemi basati su hypervisor, i sistemi multicore, gli spazi di indirizzamento a 64 bit, i computer palmari wireless e i sistemi embedded. Non c'è dubbio che per i progettisti di sistemi operativi questi siano tempi meravigliosi.