

SISTEMI OPERATIVI STORICI

Nel Capitolo 1, abbiamo presentato un breve esame storico riguardo lo sviluppo dei sistemi operativi. Quell'esame mancava di dettaglio, poiché non avevamo ancora presentato i concetti fondamentali (la schedulazione della CPU, l'organizzazione della memoria, i processi, e così via). Fino ad ora, comunque, avete appreso i concetti base. Siamo dunque ora nella posizione di esaminare come tali concetti siano stati applicati in molti sistemi operativi più vecchi e molto influenti. Alcuni di loro (come il XDS-940 o il sistema THE) erano sistemi presenti in un esemplare unico; altri (come OS/360) sono tuttora molto usati. L'ordine di presentazione evidenzia le somiglianze e differenze dei sistemi e non è strettamente cronologico o legato all'importanza. Un studente di sistemi operativi serio dovrebbe avere una certa familiarità con tutti questi sistemi.

Mentre si descrivono i primi sistemi, verranno inseriti riferimenti a letture supplementari. Questi libri, scritti dai progettisti dei sistemi, sono importanti sia per il loro contenuto tecnico che per il loro stile caratteristico.

1 I Primi Sistemi

I primi computer erano macchine fisicamente enormi che funzionavano grazie ad una console. Il programmatore, che era anche l'operatore del sistema, era tenuto a scrivere un programma, e poi ad avviarlo esso stesso direttamente dalla console dell'operatore. Il programma doveva per prima cosa essere caricato manualmente in memoria, tramite gli interruttori del pannello frontale (un'istruzione alla volta), da un nastro o da una scheda perforata. Poi era necessario premere dei pulsanti per impostare l'indirizzo iniziale e per iniziare l'esecuzione del programma. Mentre il programma era in esecuzione, il programmatore/operatore poteva controllarne l'esecuzione attraverso le luci sul display della console. Se venivano riscontrati degli errori, il programmatore poteva fermare il programma, esaminare i contenuti della memoria e dei registri, e rimuovere gli errori dal programma, direttamente dalla console. I dati venivano stampati, o registrati su carta o su una scheda per essere successivamente stampati.

Col passare del tempo vennero sviluppati software e hardware aggiuntivi. Lettori di schede, stampanti e nastri magnetici divennero di uso comune. Assembler, loader e linker venivano progettati per alleggerire il compito del programmatore. Furono create librerie di funzioni comuni, che potevano essere copiate in un programma nuovo senza bisogno di dover essere riscritte, permettendo il riutilizzo del software.

Erano soprattutto importanti le routine che riguardavano operazioni di I/O; ogni nuovo dispositivo I/O aveva delle caratteristiche peculiari, che richiedevano una programmazione accurata. Una sottoroutine speciale - chiamata driver della periferica - fu scritta per ogni dispositivo di I/O. Un driver di una periferica conosce il modo in cui utilizzare buffer, flag e registri, bit di controllo e di stato per ogni particolare dispositivo; ogni dispositivo ha quindi il proprio driver. Un semplice task, come leggere un carattere da un lettore di cassette, potrebbe comprendere complesse sequenze di operazioni specifiche per il dispositivo. Piuttosto che riscrivere ogni volta lo stesso codice, il driver per la periferica veniva ottenuto semplicemente dalla libreria.

In seguito, apparvero compilatori per FORTRAN, COBOL ed altri linguaggi, rendendo il compito del programmatore molto più semplice, ma complicando le operazioni svolte dal computer. Per preparare per l'esecuzione un programma FORTRAN, per esempio, il programmatore doveva caricare il compilatore FORTRAN sul computer. Il compilatore era solitamente su una cassetta, e

naturalmente andava inserita la cassetta giusta in un lettore. Il programma veniva letto attraverso il lettore di schede e scritto su un'altra cassetta.

Il compilatore FORTRAN produceva un output in linguaggio assembler, che poi doveva essere assemblato. Questa procedura richiedeva il caricamento di un'altra cassetta con l'assembler. L'output dell'assembler aveva bisogno di essere collegato alle routine della libreria di supporto. Alla fine, la forma binaria del programma sarebbe stata pronta all'esecuzione. Poteva, come in precedenza, essere caricata in memoria e controllata tramite la console.

Nell'esecuzione di un job, poteva essere impiegato un tempo di configurazione non trascurabile. La preparazione di ogni job consisteva di molti passi separati:

1. Caricare il nastro del compilatore FORTRAIN
2. Avviare il compilatore
3. Estrarre la cassetta del compilatore
4. Caricare il nastro dell'assemblatore
5. Avviare l'assemblatore
6. Estrarre il nastro dell'assemblatore
7. Caricare il programma oggetto
8. Avviare il programma oggetto

Se avveniva un errore in uno dei passaggi, il programmatore/operatore avrebbe dovuto ripartire dall'inizio. Ogni passo del lavoro poteva prevedere il caricamento e l'estrazione di nastri magnetici, di carta o di schede perforate.

Il tempo di configurazione del job era un vero problema: mentre venivano caricate le cassette o il programmatore stava operando sulla console, la CPU rimaneva inattiva. Bisogna ricordare che, a quei tempi, erano disponibili pochi computer, ed erano molti costosi. Un computer poteva costare milioni di dollari, esclusi i costi delle operazioni di alimentazione, di raffreddamento, di programmazione, e così via. Quindi, il tempo del computer era estremamente prezioso, e i proprietari volevano che i loro computer fossero usati il più possibile. Necessitavano di un **impiego** elevato per ottenere il maggior ritorno possibile dai loro investimenti.

La soluzione fu divisa in due parti. Prima: veniva assunto un operatore professionista di computer. Il programmatore non operava più sulla macchina. Quando un job veniva terminato, l'operatore poteva subito eseguire il successivo. Dato che l'operatore aveva più esperienza nel caricare le cassette di un programmatore, il tempo di avvio fu ridotto. Il programmatore procurava le schede o le cassette necessarie, ed una breve descrizione di come andava avviato il job. Ovviamente, l'operatore non poteva controllare alla console un programma errato, dato che l'operatore non comprendeva il programma. Quindi, in caso di errore del programma, veniva tenuto un dump (copia del contenuto) della memoria e dei registri, e il programmatore doveva effettuare il debug tramite il dump. Eseguire il dump della memoria e dei registri permetteva all'operatore di continuare con il job successivo, ma lasciava al programmatore il compito più difficile della correzione.

Come seconda cosa, i lavori con simili necessità erano raggruppati e venivano avviati nel computer come un gruppo per ridurre il tempo d'avvio. Per esempio, supponete che l'operatore riceveva un job FORTRAN, uno COBOL, ed un altro FORTRAN. Se li avesse avviati in quel ordine, avrebbe dovuto configurare il FORTRAN (caricare le cassette del compilatore, e così via), poi configurare COBOL, e poi nuovamente il FORTRAN. Avviando i due programmi FORTRAN come un gruppo, poteva, quindi, lanciare solo una volta il FORTRAN, risparmiando sul tempo operatore.

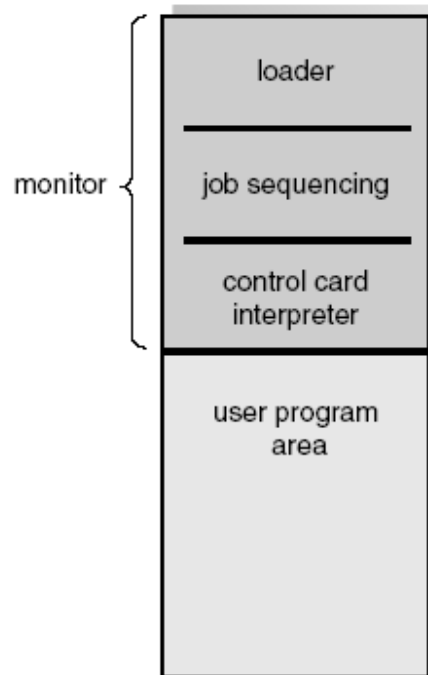


Figura 1. Disposizione della memoria per un monitor residente.

Monitor

Loader

Job sequencing = sequenza di job

Control card interpreter = interprete delle schede di controllo

User program area = area del programma utente

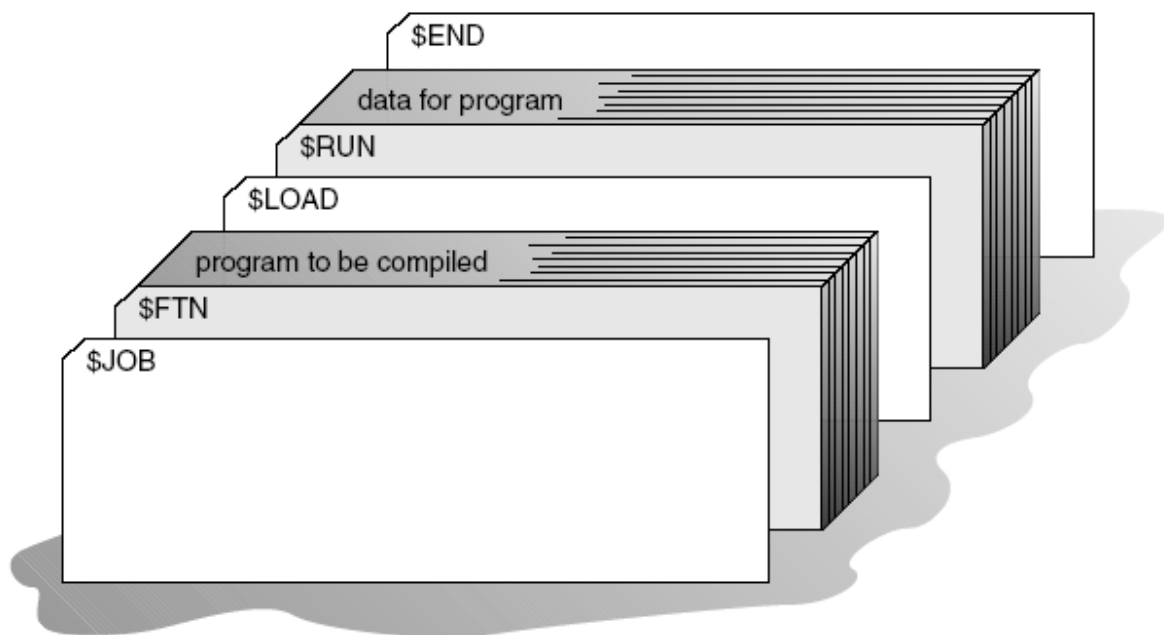


Figura 2. Sequenza di schede (card deck) per un semplice sistema a lotti.

Data for program = dati per il programma

Program to be compiled = programma da compilare

Ma c'erano ancora dei problemi. Per esempio, quando un job si bloccava, l'operatore doveva accorgersi che si fermava (osservando la console), determinare perchè si era fermato (in modo normale o anormale), eseguire il dump della memoria e, se necessario, dei registri, caricare il dispositivo appropriato per il lavoro successivo, e riavviare il computer. Durante la transizione tra un lavoro e l'altro, il computer rimaneva inattivo.

Per annullare questo tempo di inattività, è stato sviluppata la tecnica di **automatic job sequencing (ordinamento sequenziale automatico delle attività)**; con questa tecnica, furono creati i primi sistemi operativi rudimentali. Fu creato un piccolo programma, chiamato **resident monitor**, (monitor residente) per trasferire il controllo automaticamente da un job al successivo (Figura 1). Il resident monitor è sempre in memoria (o **residente**).

Quando un computer veniva acceso, veniva richiamato il resident monitor, e trasferiva il controllo ad un programma. Quando il programma terminava, faceva tornare il controllo al resident monitor, che poi avrebbe continuato con il programma successivo. Così, il resident monitor passava automaticamente da un programma all'altro, e da un job all'altro.

Ma come poteva il resident monitor sapere quale programma eseguire? In precedenza, all'operatore veniva data una breve descrizione di quali programmi dovevano essere avviati e con quali dati. Furono introdotte le **control card** per fornire questa informazione direttamente al monitor. L'idea è semplice: oltre al programma o ai dati per il job, il programmatore includeva le control card, che contenevano direttive per il resident monitor che indicavano il programma da

avviare. Per esempio, un normale programma dell'utente poteva avere come requisito uno di tre programmi: il compilatore FORTRAN (FTN), l'assemblatore (ASM), o il programma dell'utente (RUN). Potremmo usare una differente control card per ognuno di questi:

\$FTN - Esegue il compilatore FORTRAN.
\$ASM - Esegue l'assemblatore.
\$RUN - Esegue il programma dell'utente.

Queste card dicono al resident monitor quale programma avviare.
Possiamo usare due control card aggiuntive per definire i limiti di ogni job:

\$JOB - Prima card di un job.
\$END - Ultima card di un job.

Queste due card potevano essere utili per misurare le risorse-macchina usate dal programmatore. Possono essere usati dei parametri per definire il nome del job, il numero di account da caricare, e così via. Altre control card possono essere definite per altre funzioni, come, ad esempio, chiedere all'operatore di caricare o estrarre una cassetta.

Un problema con le control card è quello di distinguerle le card dei dati o del programma. La soluzione abituale è identificarle con un carattere speciale o una sequenza di caratteri (pattern) sulla card. Molti sistemi usavano il simbolo del dollaro (\$) nella prima colonna per identificare una control card. Altri usavano invece un codice diverso. Il Job Control Language dell'IBM (JCL), usava i caratteri slash (/) nelle prime due colonne. La Figura 2 mostra la disposizione di una sequenza di schede (card-deck) per un semplice sistema a lotti.

In questo modo, il resident monitor dispone di diverse parti identificabili:

- il **control card interpreter** (interprete di control card) che è responsabile di leggere ed eseguire le istruzioni nella card nel punto di esecuzione.
- il **loader** (caricatore), che è richiamato dal control card interpreter, per caricare programmi di sistema e programmi di applicazioni nella memoria ad intervalli specifici.
- i **driver delle periferiche**, che sono usati sia dal control card interpreter che dal loader per i dispositivi di I/O di sistema, al fine di eseguire l'I/O. Spesso, i programmi di sistema e di applicazioni sono collegati a questi stessi driver, assicurando la continuità nelle operazioni, come risparmiare spazio di memoria e tempo di programmazione.

Questi sistemi a lotti funzionano molto bene. Il resident monitor mette in atto la sequenzializzazione automatica dei job (automatic job sequencing) come indicato nelle control card: quando una control card indica che bisogna avviare un programma, il monitor carica il programma in memoria e gli trasferisce il controllo; quando il programma arriva al termine, trasferisce ancora il controllo al monitor, che legge la control card successiva, carica il programma appropriato, e così via. Questo ciclo viene ripetuto fino a quando tutte le control card relative al job sono state interpretate. Poi, il monitor continua automaticamente con il job successivo.

Il passaggio ai sistemi a lotti con job sequencing automatico fu finalizzato all'aumento del rendimento. Il problema, semplicemente, è che gli uomini sono considerevolmente più lenti del computer. Di conseguenza, si desidera rimpiazzare il lavoro umano con software del sistema operativo. Il job sequencing automatico elimina il bisogno di configurazione e di job sequencing da parte dell'uomo.

Come sottolineato nel Paragrafo 1.2.1, persino con questa soluzione, la CPU è comunque spesso inattiva. Il problema è la velocità delle periferiche di I/O meccaniche, che sono intrinsecamente più

lente dei dispositivi elettronici. Persino una CPU lenta lavora nel range del microsecondo, con migliaia di istruzioni eseguite al secondo. Un lettore di schede veloce, d'altra parte, può leggere 1.200 schede al minuto (o 20 schede al secondo). Così, la differenza di velocità tra la CPU e le sue periferiche I/O può essere di tre ordini di grandezza o più. Col passare del tempo, sicuramente, il progresso nella tecnologia ha dato come frutti periferiche di I/O più veloci. Sfortunatamente, la velocità della CPU è aumentata ancora più velocemente, quindi il problema non solo non fu risolto, ma fu persino amplificato.

Una soluzione comune fu sostituire i lettori di schede lenti (periferiche di input) e le stampanti in linea (periferiche di output) con unità a nastro magnetico. La maggior parte dei sistemi di computer negli ultimi anni 50 e nei primi anni 60 erano sistemi a lotti che leggevano da lettori di schede e scrivevano su stampanti in linea o su schede perforate. Invece di far leggere la CPU direttamente dalle schede, esse venivano prima copiate su nastro magnetico tramite una periferica separata. Quando il nastro era pieno, veniva smontato e portato al computer. Quando c'era bisogno di una scheda per passare l'input ad un programma, il record equivalente veniva letto dal nastro. Allo stesso modo, gli output venivano scritti su nastro e poi il contenuto del nastro veniva stampato successivamente. I lettori di schede e le stampanti venivano utilizzate fuori linea (*off-line*), invece che dal computer principale (Figura 3).

Il vantaggio principale delle operazioni off-line era che il computer principale non era vincolato dalla velocità dei lettori di schede e delle stampanti, ma solo da quella delle unità magnetiche, molto più veloci. Questa tecnica di utilizzo del nastro magnetico per tutte le operazioni I/O poteva essere applicato a qualsiasi dispositivo simile (come lettori di schede, perforatore di schede - card punch, plotter, nastro di carta, stampanti). Il vantaggio reale nelle operazioni off-line per una CPU deriva dall'opportunità di gestire sistemi multipli di flussi da input a nastro e da nastro a stampante. Se la CPU può elaborare input due volte più velocemente rispetto alla lettura delle schede del lettore, allora due lettori che lavorano simultaneamente possono produrre abbastanza nastro da tenere impegnata la CPU. D'altra parte, in questo modo c'è un ulteriore ritardo nell'avvio di un job. Esso deve prima essere scritto su nastro. Poi, deve aspettare fino a quando altri job vengono scritti su nastro fino a "riempirlo"; poi il nastro deve essere riavvolto, estratto, portato a mano fino alla CPU e montato su un lettore di cassette. Ovviamente, questo processo non è irragionevole per i sistemi a lotti. Molti job di questo tipo possono essere caricati su una cassetta prima che questa venga portata al computer.

Sebbene la preparazione dei lavori off-line sia continuata per qualche tempo, fu presto rimpiazzata nella maggior parte dei sistemi. I sistemi muniti di unità disco divennero disponibili in larga scala e migliorarono molto nel campo delle operazioni off-line. L'intero nastro doveva essere scritto prima di essere riavvolto e letto, dal momento che le cassette sono per loro natura **periferiche ad accesso sequenziale**. I sistemi a dischi eliminarono questo problema essendo **periferiche ad accesso casuale**. Poiché la testina viene mossa da un'area all'altra del dischetto, un lettore di dischi può commutare rapidamente dall'area del disco in uso al lettore di schede per memorizzare nuove schede, nella posizione richiesta dalla CPU per leggere la scheda "successiva".

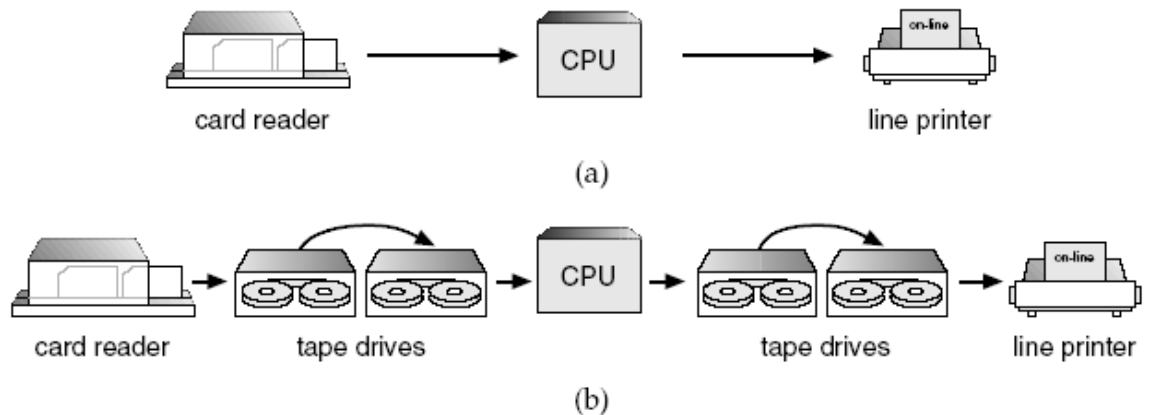


Figura 3. Funzionamento di dispositivi di I/O. (a) On-line. (b) Off-line.

Card reader = lettore di schede
 Line printer = stampante in linea
 Tape drivers = lettori di nastri

In un sistema a dischi, le schede sono lette dal lettore di schede direttamente nel dischetto. La posizione delle immagini della scheda è registrata in una tabella mantenuta dal sistema operativo. Quando viene eseguita un'operazione, il sistema operativo, per soddisfare la richiesta di input dal lettore di schede, legge i dati dal disco. In modo simile, quando il job richiede alla stampante di scrivere una linea, quella linea viene copiata in un buffer di sistema e salvata su disco. L'output viene realmente stampato quando il job è terminato. Questo tipo di elaborazione è chiamata **spooling** (Figura 4); il nome è un acronimo per: "simultaneous peripheral operation on-line (operazioni periferiche simultanee on-line)". Lo spooling, in pratica, utilizza il disco come se fosse un grande buffer, per leggere il più possibile dalle periferiche di input e per salvare l'output sui file finché le periferiche non sono pronte ad accettare i dati.

La tecnica di spooling viene anche utilizzata per elaborare dati in remoto. La CPU trasmette i dati tramite i canali di comunicazione ad una stampante remota(o accetta un job di input completo da un lettore di schede remoto). L'elaborazione remota avviene alla propria velocità, senza intervento della CPU a cui è necessario solo notificare la fine dell'operazione, così che sia in grado di processare il prossimo lotto di dati. Lo spooling sovrappone l'I/O di un job con l'elaborazione di un altro job. Anche in un sistema semplice, lo spooler potrebbe leggere l'input di un job mentre stampa l'output di un altro. Durante questo tempo, uno o più job potrebbero venir eseguiti, leggendo le proprie "schede" dal disco e "stampando" le linee di output sulla medesima periferica. Lo spooling porta un effetto benefico diretto alle prestazioni del sistema. Al costo di un po' di spazio su disco e di poche tabelle, l'elaborazione di un job si può sovrapporre all'I/O di un altro job. In questo modo, lo spooling permette sia alla CPU che all'I/O di lavorare con un tasso più elevato.

Lo tecnica di spooling porta naturalmente alla multiprogrammazione, che rappresenta le fondamenta di tutti i moderni sistemi operativi.

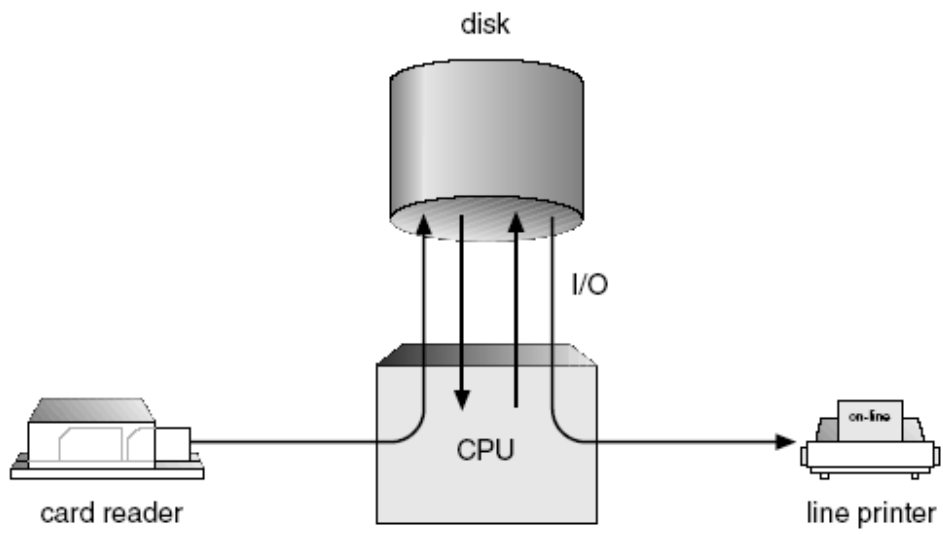


Figura 4. Spooling.

Disk =disco

Card reader = lettore di schede

Line printer = stampante in linea

2 Atlas

Il sistema operativo Atlas (Kilburn et al. [1961], Howarth et al. [1961]) fu disegnato alla University of Manchester in Gran Bretagna alla fine degli anni 50 ed ai primi degli anni 60. Molte delle sue caratteristiche di base, che all'epoca erano delle novità, divennero parti standard dei moderni sistemi operativi. I driver delle periferiche costituivano una delle parti principali del sistema. Inoltre, furono aggiunte delle chiamate di sistema tramite una serie di istruzioni speciali chiamate *extra code*.

Atlas era un sistema operativo a lotti con spooling. Lo spooling permetteva al sistema di registrare i job in base alla disponibilità delle periferiche, quali unità a nastro magnetico, unità per la scrittura o la lettura di nastri di carta, stampanti, unità di lettura o di scrittura su schede.

La funzionalità più ragguardevole di Atlas, comunque, era la sua organizzazione della memoria. La **core memory** (memoria centrale) era nuova e costosa allo stesso tempo. Molti computer, come l'IBM 650, usavano un tamburo per la memoria principale. La paginazione su richiesta veniva usata per trasferire automaticamente informazioni tra la core memory e il tamburo.

Il sistema Atlas usava un computer britannico con parole a 48-bit. Gli indirizzi erano a 24-bit, ma venivano codificati in decimale, la qual cosa permetteva l'indirizzamento di solo un milione di parole (parole). A quei tempi, questo era uno spazio di indirizzamento estremamente grande. La memoria fisica di Atlas era un tamburo di 98 KB parole e di 16 KB parole del core. La memoria era divisa in pagine da 512 parole, fornendo 32 frame di memoria fisica. Una memoria associativa con 32 registri realizzava la mappatura da un indirizzo virtuale ad uno fisico.

Se si verificava un errore di pagina, veniva richiamato un algoritmo di rimpiazzo. Un frame di memoria era sempre mantenuto vuoto, cosicché il trasferimento dal tamburo potesse iniziare immediatamente. L'algoritmo di sostituzione di pagina cercava di prevedere il comportamento degli accessi futuri alla memoria basandosi su quelli precedenti. Un bit di riferimento veniva impostato per ogni frame ogni volta che si accedeva a quest'ultimo. I bit di riferimento venivano letti dalla memoria ogni 1.024 istruzioni, e gli ultimi 32 valori di questi bit venivano conservati. Lo storico veniva utilizzato per definire il tempo intercorso dall'ultimo riferimento (t_1), e l'intervallo tra gli ultimi due riferimenti (t_2). Le pagine venivano scelte per il rimpiazzo nell'ordine seguente:

1. Qualsiasi pagina, con $t_1 > t_2 + 1$, veniva considerata inutilizzata.
2. Se $t_1 \leq t_2$ per tutte le pagine, allora la pagina si rimpiazza con quella con l'intervallo maggiore $t_2 - t_1$.

L'algoritmo di sostituzione dà per scontato che il programma acceda ciclicamente alla memoria. Se il tempo intercorso tra gli ultimi due riferimenti è t_2 , allora un altro riferimento è atteso in seguito per t_2 unità di tempo. Se un riferimento non avviene ($t_1 > t_2$), si deduce che la pagina non viene più utilizzata e quindi viene sostituita. Se tutte le pagine sono ancora in uso, allora quella di cui non ci sarà più bisogno per il tempo più lungo verrà sostituita. Ci si aspetta che il tempo per il riferimento successivo sia $t_2 - t_1$.

3 XDS-940

Il sistema operativo XDS-940 (Lichtenberger e Pirtle [1965]) fu progettato a Berkeley all'Università della California. Come il sistema Atlas, utilizzava la paginazione per la gestione della memoria; a differenza di Atlas, XDS-940 era un sistema a condivisione del tempo.

La paginazione veniva utilizzata solo per il riposizionamento, e non veniva usata la paginazione

su richiesta. La memoria virtuale di ogni processo dell'utente era di soli 16 KB parole, mentre la memoria fisica era di 64 KB parole. Ogni pagina era di 2 KB parole e la tabella delle pagine veniva mantenuta nei registri. Dato che la memoria fisica era più grande della memoria virtuale, molti processi dell'utente potevano risiedere in memoria contemporaneamente. Il numero di utenti poteva essere aumentato con la condivisione delle pagine, quando queste contenevano codice rientrante a sola lettura. I processi venivano mantenuti su un tamburo e venivano scambiati, in base alle necessità, dentro e fuori dalla memoria.

Il sistema XDS-940 fu costruito basandosi su di un XDS-930 modificato. Le modifiche furono rappresentative dei cambiamenti apportati al computer base per permettere di scrivere un sistema operativo in modo corretto. Fu aggiunta la modalità user-monitor; alcune istruzioni (come Halt e quelle di I/O), vennero definite come privilegiate: un tentativo di eseguire un'istruzione privilegiata in modalità utente sarebbe stato bloccato dal sistema operativo.

Fu aggiunta un'istruzione di chiamata di sistema al gruppo di istruzioni in modalità utente, la quale fu utilizzata per creare nuove risorse, come i file, permettendo al sistema operativo di organizzare le risorse fisiche. I file, per esempio, venivano allocati in blocchi da 256 parole sul tamburo. Veniva usata una mappa di bit per la gestione dei blocchi liberi sul tamburo. Ogni file aveva un blocco-indice con puntatori ai blocchi dei dati attuali. I blocchi-indice erano concatenati tra loro.

Il sistema XDS-940 forniva anche chiamate di sistema per permettere ai processi di creare, eseguire, sospendere, e distruggere i sottoprocessi. Un programmatore poteva costruire un sistema di processi. Processi separati potevano condividere la memoria per le comunicazioni e la sincronizzazione. La creazione di un processo definiva un struttura ad albero, dove un processo è la radice e i suoi sottoprocessi sono nodi nell'albero sotto di esso. Ogni sottoprocesso poteva, a sua volta, creare altri sottoprocessi.

4 THE

Il sistema operativo THE (Dijkstra [1968], McKeag e Wilson [1976]) fu progettato alla Technische Hogeschool a Eindhoven in Olanda. Era un sistema a lotti che funzionava su un computer olandese, EL X8, con 32 KB di parole a 27-bit. Il sistema fu notato principalmente per il suo progetto semplice e lineare, soprattutto per la sua struttura a livelli, e per l'uso di un gruppo di processi concorrenti che impiegavano semafori per la sincronizzazione.

A differenza del sistema XDS-940, tuttavia, il gruppo di processi nel sistema THE era statico. Il sistema operativo stesso fu creato come un insieme di processi tra loro cooperanti. Inoltre, furono creati cinque processi utente che servivano come agenti attivi per compilare, eseguire e stampare i programmi utente. Quando un job veniva concluso, il processo ritornava alla coda di input per selezionare un altro job.

Questo sistema utilizzava un algoritmo di schedulazione della CPU basato sulle priorità, che venivano ricalcolate ogni 2 secondi ed erano inversamente proporzionali al tempo in cui la CPU era stata usata recentemente (negli ultimi 8-10 secondi). Questo schema dava maggiore importanza ai processi I/O bound e a quelli nuovi.

La gestione della memoria era limitata dalla mancanza di supporto hardware. Tuttavia, dato che il sistema era limitato e i programmi potevano essere scritti solo in Algol, veniva usato un sistema di paginazione software. Il compilatore Algol generava automaticamente chiamate alle routine di sistema, che assicuravano la presenza in memoria delle informazioni richieste, usando lo swapping, se necessario. Il dispositivo di immagazzinamento dei dati era un tamburo di 512 KB parole. Veniva utilizzata una pagina di 512 parole, con una strategia di sostituzione di pagina LRU.

Un'altra delle caratteristiche principali del sistema THE era il controllo degli stalli. Per evitare che si verificassero stalli, veniva usato l'algoritmo del banchiere.

Fortemente legato al sistema THE è il sistema Venus (Liskov [1972]). Il sistema Venus era anche esso un progetto con struttura a strati, che usava i semafori per sincronizzare i processi. I livelli più bassi del progetto furono implementati in microcodici, fornendo, tuttavia, un sistema molto più veloce. La gestione della memoria fu modificata nel tipo paginata a segmenti. Il sistema fu inoltre progettato per essere a condivisione del tempo, piuttosto che un sistema a lotti.

5 RC 4000

Il sistema RC 4000, come il sistema THE, si distingueva principalmente per le caratteristiche progettuali. Fu ideato per il computer danese RC 4000 da Regnecentralen, in particolare da Brinch-Hansen (Brinch-Hansen [1970], Brinch-Hansen [1973]). L'obiettivo non era di progettare un sistema a lotti, o un sistema a condivisione del tempo, o altri sistemi specifici, ma di creare un nucleo di sistema operativo, o kernel, su cui poter costruire un sistema operativo completo. Così, la struttura del sistema era a strati, e venivano forniti solo quelli più bassi: il kernel.

Il kernel supportava diversi processi concorrenti. Uno schedatore della CPU di tipo round robin supportava i processi. Anche se i processi stessi potevano condividere la memoria, il meccanismo principale di comunicazione e sincronizzazione era il **sistema di messaggi** fornito dal kernel. I processi potevano comunicare tra di loro scambiandosi messaggi di lunghezza fissa di otto parole. Tutti i messaggi venivano immagazzinati in buffer facenti parte di un gruppo comune di buffer. Quando un buffer del messaggio non era più necessario, veniva restituito al gruppo comune.

Una **message queue** (coda di messaggi) era associata ad ogni processo e conteneva tutti i messaggi che erano stati inviati a quel processo, ma che non erano ancora stati ricevuti. I messaggi venivano rimossi dalla coda seguendo l'ordine FIFO. Il sistema supportava quattro operazioni primarie, che venivano eseguite automaticamente:

- **send message** (*in receiver, in message, out buffer*)
- **wait-message** (*out sender, out message, out buffer*)
- **send-answer** (*out result, in message, in buffer*)
- **wait-answer** (*out result, out message, in buffer*)

Le ultime due operazioni permettevano ai processi di scambiare diversi messaggi contemporaneamente.

Queste primitive richiedevano che un processo elaborasse i propri messaggi secondo l'ordine FIFO, e che si bloccasse mentre gli altri processi gestivano i suoi messaggi. Per rimuovere queste restrizioni, gli sviluppatori fornirono due ulteriori primitive di comunicazione che permettevano ad un processo di aspettare l'arrivo del messaggio successivo o di rispondere ed elaborare la propria coda in qualsiasi ordine:

- **wait-event** (*in previous-buffer, out next-buffer, out result*)
- **get-event** (*out buffer*)

Anche le periferiche di I/O venivano trattate come processi. I driver delle periferiche erano codici che convertivano gli interrupt e i registri della periferiche in messaggi. In tal modo, un processo avrebbe scritto ad un terminale, inviandogli un messaggio. Il driver di periferica avrebbe ricevuto il messaggio e fatto apparire il carattere sul terminale. Un carattere in input poteva interrompere il sistema e venire trasferito a un driver di periferica. Il driver di periferica poteva creare un messaggio a partire dal carattere immesso, e inviarlo ad un processo in attesa.

6 CTSS

Il Compatible Time-Sharing System (CTSS) (Corbato et al. [1962]) fu progettato presso MIT come sistema sperimentale a condivisione del tempo. Fu implementato su un IBM 7090 ed all'occorrenza supportava fino a 32 utenti interattivi. Agli utenti veniva fornito un gruppo di comandi interattivi che permetteva loro di manipolare i file, di compilare ed avviare i programmi tramite un terminale.

Il 7090 aveva una memoria di 32 KB, formata da parole a 36-bit. Il monitor utilizzava 5 KB parole, e lasciava 27 KB agli utenti. Le immagini della memoria utente venivano scambiate tra la memoria e un tamburo veloce. La schedulazione della CPU impiegava un algoritmo a coda multilivello con retroazione (multilevel-feedback-queue). Il quanto di tempo per il livello i era $2 * i$ unità di tempo. Se un programma non finiva di utilizzare la CPU in un quantum di tempo, veniva spostato al livello successivo della coda, assegnandogli il doppio del tempo. Il programma al livello più alto (con il quantum più breve) veniva avviato per primo. Il livello iniziale di un programma veniva determinato dalla sua dimensione, in modo che il quantum di tempo fosse lungo almeno quanto il tempo di swap.

CTSS era estremamente ben riuscito e rimase in uso fino al 1972. Anche se era limitato, riuscì a dimostrare che il time-sharing era una tecnologia di elaborazione pratica e conveniente. Un risultato di CTSS fu di aumentare lo sviluppo dei sistemi a condivisione del tempo; un altro risultato fu lo sviluppo di MULTICS.

7 MULTICS

Il sistema operativo MULTICS (Corbato and Vyssotsky [1965], Organick [1972]) fu ideato presso l'MIT come naturale estensione di CTSS. CTSS e gli altri sistemi a condivisione del tempo del primo periodo risultarono così di successo che crearono un desiderio immediato di procedere velocemente verso sistemi di maggiori dimensioni e migliori. Quando divennero disponibili computer più grandi, gli ideatori di CTSS decisero di creare una funzione di sistema per la condivisione del tempo. I servizi di elaborazione sarebbero stati forniti come avviene per l'energia elettrica: i grandi sistemi di computer sarebbero stati connessi da cavi telefonici a terminali negli uffici e nelle case in tutta la città. Il sistema operativo sarebbe stato un sistema a tempo condiviso funzionante in modo continuativo e dotato di un vasto sistema di programmi e dati condivisi.

MULTICS fu progettato da un gruppo dell'MIT: GE (che in seguito vendette il suo dipartimento di computer alla Honeywell), ed ai Bell Laboratories (che abbandonarono il progetto nel 1969). Il GE 635 di base fu modificato per evolversi in un nuovo sistema chiamato GE 645, principalmente con l'aggiunta di hardware con memoria a segmentazione paginata.

Un indirizzo virtuale era composto da un numero di segmento a 18-bit e da uno spiazamento di parole a 16-bit. I segmenti venivano poi divisi in pagine da 1 KB parole. Veniva usato l'algoritmo di rimpiazzo delle pagine detto second-chance.

Lo spazio per l'indirizzo virtuale segmentato era fuso nel file system; ogni segmento era un file, e il suo indirizzo il nome del file stesso. Il file system stesso era una struttura multilivello ad albero che permetteva agli utenti di creare le loro strutture di sottodirettori.

Come CTSS, MULTICS utilizzava una coda multilivello con retroazione per la schedulazione della CPU. La protezione era realizzata da una lista di accesso associata ad ogni file e da un gruppo di anelli di protezione per i processi in esecuzione. Il sistema, che era scritto quasi interamente in

PL/1, comprendeva circa 300.000 righe di codice. Fu esteso ad un sistema multiprocessore, permettendo ad una CPU di essere messa fuori servizio per la manutenzione mentre il sistema continuava a funzionare.

8 OS/360

La linea di sviluppo più duratura di un sistema operativo è, senza dubbio, quella dei computer IBM. I primi computer IBM, come l'IBM 7090 e l'IBM 7094, sono i primi esempi di sviluppo di procedure comuni di I/O, seguite da un monitor residente, da istruzioni privilegiate, dalla protezione della memoria, e da semplice elaborazione a lotti. Questi sistemi furono sviluppati separatamente, spesso in luoghi diversi e indipendenti. Come risultato, IBM si trovò di fronte a molti computer differenti, con linguaggi e software di sistema diversi.

L'IBM/360 fu progettato per cambiare questa situazione: fu realizzato come una famiglia di computer che copriva l'intera gamma di macchine per piccole imprese a grosse macchine per uso scientifico. Serviva solo un solo software per questi sistemi, che utilizzavano tutti lo stesso sistema operativo: OS/360 (Mealy et al. [1966]). Si pensava che questa disposizione potesse ridurre i problemi di manutenzione per l'IBM e che permettesse agli utenti di spostare liberamente programmi e applicazioni da un sistema all'altro.

Sfortunatamente, OS/360 cercò di fare troppe cose, e, come risultato, non svolse i propri compiti particolarmente bene. Il file system includeva un campo di informazioni che definiva il genere di ogni file; erano definiti diversi tipi di file, con record di lunghezza fissa e variabile, e i file bloccati o non bloccati. Veniva utilizzata l'allocazione contigua, in modo che l'utente potesse indovinare la dimensione di ogni file in output. Il Job Control Language (linguaggio di controllo del job: JCL) aggiunse dei parametri per ogni possibile opzione, rendendolo incomprensibile all'utente medio.

Le routine di gestione della memoria fu ostacolata dall'architettura. Sebbene fosse utilizzata una modalità di indirizzamento tramite registro base, il programma poteva accedere e modificare tale registro, in modo da venir generati indirizzi assoluti dalla CPU. Questa modifica impediva la rilocalizzazione dinamica; il programma era legato alla memoria fisica nella fase di caricamento. Furono prodotte due versioni diverse del sistema operativo: OS/MFT che utilizzava regioni fisse e OS/MVT che utilizzava regioni variabili.

Il sistema fu scritto in linguaggio assembler da migliaia di programmatori, dando come risultato milioni di righe di codice richiedendo molta memoria per i propri codici e tabelle, e spesso l'overhead della CPU consumava metà dei cicli totali.

Col passare degli anni, furono rilasciate nuove versioni per aggiungere caratteristiche nuove e per correggere gli errori. Tuttavia, la correzione di un errore spesso ne provocava un altro in qualche parte remota del sistema, in modo che il numero di errori conosciuti nel sistema restava costante.

Ad OS/360 venne aggiunto il supporto per la memoria virtuale con l'introduzione dell'architettura IBM 370. L'hardware sottostante permetteva la gestione di memoria virtuale a pagine segmentate. Versioni più recenti del sistema operativo utilizzavano tale hardware in modalità diverse. L'OS/VS1 creava un unico grande spazio di memoria virtuale, ed eseguiva in esso OS/MFT. In questo modo, il sistema operativo stesso era paginato, così come i programmi degli utenti. L'OS/VS2 Release 1 eseguiva OS/MVT in memoria virtuale; infine, OS/VS2 Release 2, conosciuto come MVS, forniva ad ogni utente un proprio spazio di memoria virtuale.

MVS è principalmente un sistema operativo a lotti. Il sistema CTSS veniva eseguito su un IBM 7094, ma l'MIT decise che lo spazio di indirizzamento del 360, successore dell'IBM 7094, era troppo piccolo per il MULTICS, e così cambiarono fornitore. Come conseguenza, IBM decise di

creare un proprio sistema a tempo condiviso: TSS/360 (Lett e Konigsford [1968]). Come MULTICS, TSS/360 si proponeva di essere una funzione di sistema per la gestione della condivisione del tempo. L'architettura base del modello 360 fu modificata nel modello 67 per gestire la memoria virtuale. Molte aziende acquistarono il sistema 360/67 prima dell'uscita di TSS/360.

TSS/360, tuttavia, fu ritardato nell'uscita, e così vennero sviluppati altri sistemi a condivisione del tempo come sistemi temporanei, fino alla piena disponibilità. Una opzione di condivisione del tempo (time-sharing option: TSO) venne aggiunta ad OS/360. Il centro scientifico di IBM a Cambridge sviluppò CMS, un sistema a singolo utente, e CP/67: una macchina virtuale su cui eseguirlo. (Meyer and Seawright [1970], Parmelee et al. [1972]).

Quando il TSS/360 venne infine distribuito, fu un fallimento; era troppo grande e troppo lento. Come conseguenza, nessuno passò dal proprio sistema temporaneo a TSS/360. Attualmente, la condivisione del tempo nei sistemi IBM viene largamente gestita da TSO sotto MVS o da CMS sotto CP/67 (rinominato in VM).

Cosa andò storto con TSS/360 e MULTICS? Una parte del problema consisteva nel fatto che questi sistemi avanzati erano troppo grandi e troppo complessi per essere compresi. Un altro problema risiedeva nell'assunto che la potenza di elaborazione sarebbe stata disponibile tramite un grande computer remoto a condivisione del tempo. Oggi, invece, sembra che la maggior parte delle elaborazioni verrà svolta da piccole macchine individuali: i personal computer, e non da grandi computer remoti che si propongono di gestire ogni cosa per ogni singolo utente.

9 Mach

Il sistema operativo Mach ha come antenato Accent, sviluppato alla Carnegie Mellon University: CMU (Rashid e Robertson[1981]). Il sistema di comunicazione e la filosofia di Mach deriva da quella di Accent, ma molte altre porzioni significative del sistema (ad esempio, la gestione della memoria virtuale, la gestione dei task e dei thread) vennero sviluppate dall'inizio (Rashid [1986], Tevanian et al. [1989], Accetta et al. [1986]). Lo schedatore del sistema Mach viene descritto in dettaglio da Tevanian et al. [1987] e Black [1990]. Una versione precedente del sistema a memoria condivisa e di mappatura della memoria di Mach fu presentata da Tevanian et al. [1987b].

Il sistema operativo Mach fu progettato avendo in mente tre obiettivi principali:

1. Emulare 4.3BSD UNIX, in modo che i file eseguibili, provenienti da un sistema UNIX, potessero essere correttamente eseguiti da Mach.
2. Essere un sistema operativo moderno, che supporta molti modelli di memoria, e l'elaborazione parallela e distribuita.
3. Avere un kernel più semplice e più facilmente modificabile rispetto a quello 4.3BSD.

Lo sviluppo di Mach ha seguito un percorso evolutivo che ha preso l'avvio nel sistema BSD UNIX. Il codice di Mach fu inizialmente sviluppato all'interno del kernel di 4.2BSD, con le componenti del kernel di BSD rimpiazzate da quelle di Mach nel momento in cui si venivano completate. Le componenti di BSD furono aggiornate a quelle di 4.3BSD, quando furono disponibili. A partire dal 1986, i sottosistemi di comunicazione e di gestione della memoria virtuale furono fatti funzionare su computer della famiglia DEC VAX, tra cui le versioni a multiprocessore di VAX. Le versioni per IBM RT/PC e per le workstation SUN 3 seguirono a breve distanza. Nel 1987 vennero completate le versioni a multiprocessore Encore Multimax e Sequent Balance, che includevano il supporto per i task e i thread; come pure le prime versioni ufficiali del sistema: la versione 0 e la versione 1.

Con la Release 2, Mach assicurava la compatibilità con i sistemi BSD corrispondenti, includendo grandi parti del suo codice del kernel. Le nuove funzionalità e capacità di Mach resero i kernel di queste release più grandi di quelli corrispondenti di BSD. Mach 3 spostò il codice di BSD al di fuori del kernel, lasciando così un microkernel di dimensioni molto più ridotte. Questo sistema implementa nel kernel solo le funzionalità base di Mach; tutto il codice specifico di UNIX è stato spostato per funzionare su server in modalità utente. L'esclusione del codice specifico di UNIX dal kernel consente la sostituzione di BSD con un altro sistema operativo, o l'esecuzione simultanea di diverse interfacce del sistema operativo sulla base del microkernel. In aggiunta a BSD, le implementazioni in modalità utente sono state sviluppate per i sistemi operativi DOS, Macintosh e OSF/1. Questo metodo ha dei punti in comune con il concetto di macchina virtuale, definita però tramite software (l'interfaccia del kernel di Mach), piuttosto che tramite hardware. Con la versione 3.0, Mach divenne disponibile in un'ampia gamma di sistemi, tra i quali le macchine a singolo processore SUN, Intel, IBM e DEC, e Sequent, Encore e DEC per quelle a multiprocessore.

Mach fu messo in primo piano, e portato all'attenzione del mercato, quando la Open Software Foundation (OSF) annunciò che avrebbe usato Mach 2.5 come base per il proprio sistema operativo: OSF/1. La versione iniziale di OSF/1 fu rilasciata con un anno di ritardo, e ora compete con UNIX System V, Release 4, il sistema operativo scelto dai membri di UNIX International (UI). Tra i membri di OSF vi erano compagnie tecnologicamente importanti come IBM, DEC e HP. OSF ha da allora cambiato strategia, basando la sola versione DEC di Unix è basata sul kernel Mach.

Mach 2.5 è pure la base del sistema operativo su workstation NeXT, nato da un'idea di Steve Jobs, famoso per la Apple Computer.

Al contrario di UNIX, che fu sviluppato senza tenere in conto la possibilità di multiprocessing, Mach ne incorporava il supporto completo. Tale supporto è enormemente flessibile, passando da sistemi con memoria condivisa ad altri senza alcuna memoria condivisa tra i processori. Mach usa processi leggeri, nella forma di thread di esecuzione multipli nell'ambito di un task (o spazio di indirizzamento), al fine di supportare l'elaborazione multiprocessore e parallela. L'uso estensivo di messaggi, quale unico metodo di comunicazione, assicura che il meccanismo di protezione sia completo ed efficiente. Integrando i messaggi con il sistema di memoria virtuale, Mach assicura inoltre che i messaggi possano essere gestiti in modo efficiente; infine, l'utilizzo di messaggi, da parte della memoria virtuale per comunicare con i demoni che gestiscono l'immagazzinamento in memoria, consente una grande flessibilità nella progettazione e implementazione di questi tipi di task di gestione della memoria. Tramite chiamate di sistema a basso livello, o primitive, da cui possono essere costruite funzioni più complesse, Mach riduce la dimensione del kernel permettendo l'emulazione del sistema operativo al livello utente, in modo simile a quanto fanno i sistemi macchina virtuale dell'IBM.

Le edizioni precedenti di Operating Systems Concepts comprendevano un intero capitolo dedicato a Mach. Tale capitolo, così come appare nella quarta edizione, è disponibile sul web all'indirizzo (<http://www.bell-labs.com/topic/books/os-book/Mach.ps>).

10 Altri Sistemi

Vi sono, ovviamente, altri sistemi operativi, e molti di essi hanno caratteristiche interessanti. Il sistema operativo MPC per la famiglia di computer Burroughs (McKeag e Wilson[1976]) fu il primo ad essere scritto in un linguaggio di programmazione per i sistemi. Esso supportava la segmentazione e CPU multiple. Il sistema operativo SCOPE per CDC 6600 (McKeag e Wilson [1976]) supportava anch'esso CPU multiple. Il coordinamento e la sincronizzazione dei processi

multipli era sorprendentemente ben progettata. Tenex (Bobrow et al.[1972]) fu uno dei primi sistemi con richiesta di paginazione per il PDP-10, ed ebbe una grande influenza sui successivi sistemi a condivisione del tempo, come TOPS-20 per DEC-20. Il sistema operativo VMS per VAX è basato sul sistema RSX per PDP-11. CP/M fu il sistema operativo più comune per i microcomputer a 8-bit, di cui oggi ne esistono pochi, mentre MS-DOS è il sistema più comune per i microcomputer a 16-bit. Stanno diventando sempre più popolari le **interfacce grafiche utente** (Graphical User Interfaces: GUI), per rendere più semplice l'utilizzo dei computer. Il sistema operativo Macintosh e Microsoft Windows sono i due leader in quest'area.