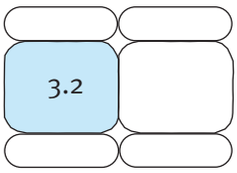
EVOLUZIONE DEI SISTEMI OPERATIVI

Iª Parte: dai sistemi batch a Linux

Paolo Ancilotti
Maurelio Boari

Nel ripercorrere le motivazioni che hanno portato all'evoluzione delle funzioni di un sistema operativo viene evidenziato come tale evoluzione sia stata fortemente influenzata dai progressi di altre branche dell'intera disciplina informatica e al tempo stesso ne abbia a sua volta influenzato il progresso. L'articolo è suddiviso in due parti: la prima dedicata all'evoluzione del concetto stesso di sistema operativo con riferimento, soprattutto, ai sistemi *general purpose*, dai primi sistemi *batch* agli attuali sistemi *multithreading*; la seconda verrà dedicata all'evoluzione di specifiche categorie di sistemi.



1. INTRODUZIONE

La conoscenza del modo in cui una disciplina si è sviluppata, delle motivazioni che stanno alla base delle innovazioni che ha prodotto e delle tecnologie da questa derivate, hanno spesso un valore che trascende il puro e semplice interesse culturale in quanto aiutano a comprendere meglio il perché di certe scelte e ad orientare gli stessi sviluppi futuri sia scientifici che tecnologici. Se ciò è vero per discipline che hanno alle spalle un percorso evolutivo molto lungo, spesso di secoli, le precedenti motivazioni possono sembrare prive di fondamento per una disciplina come l'informatica che ha soltanto pochi decenni di vita. In realtà ciò è vero, ma è altrettanto vero che la rapidità con cui è avvenuta l'evoluzione di questa scienza, con cui sono state introdotte nuove tecnologie e con cui tali tecnologie hanno pervaso tutte le altre discipline, non ha eguali in nessun altro ambito scientifico e tecnologico. È proprio da questa frenetica evoluzione, dalla rapidità con cui nuove tecnologie vengono introdotte ed altre abbandonate, che deriva l'opportunità anche, in questo campo, di ripercorrere il cammino evolutivo, cercando di capire e giustificare perché certe scelte siano state privilegiate nei confronti di altre.

Nel caso poi dello specifico tema dei Sistemi Operativi, l'importanza di conoscere il loro percorso evolutivo è ulteriormente avvalorata dal fatto che le funzioni che vengono svolte da questo importante componente di un sistema di elaborazione sono molto varie, spesso anche molto diverse da un sistema all'altro, al punto che negli stessi libri dedicati a questo argomento [3, 20, 21, 22] il sistema operativo viene definito in vari modi che, come ad esempio quelli riportati di seguito, non sempre sono coincidenti:

- il sistema operativo è un programma che agisce come intermediario fra l'utente di un calcolatore e le componenti hardware del calcolatore stesso.
- Il sistema operativo è quel componente di un sistema di elaborazione destinato a fornire l'ambiente nel quale vengono eseguiti i programmi degli utenti del sistema.

❑ Il sistema operativo è il gestore delle risorse hardware e software di un sistema di elaborazione; alloca tali risorse ai programmi e ai relativi utenti, secondo le loro necessità.
❑ Il sistema operativo è quel componente del sistema di elaborazione che ha il compito di creare una macchina astratta più semplice da usare rispetto alla macchina fisica.

Ciascuna delle precedenti definizioni, e altre ancora presenti in molti libri dedicati a questo tema, è sicuramente corretta ma, in generale, da sola non è sufficiente a identificare l'intero insieme delle funzioni che, di volta in volta, possono trovarsi in un sistema operativo, limitandosi a mettere in evidenza soltanto alcuni aspetti che, in certi casi possono essere quelli più importanti, ma senza che ciò sia necessariamente vero per tutti i sistemi.

L'obiettivo di questo articolo è proprio quello di ripercorrere le motivazioni che hanno portato, da un lato alla necessità di dotare ogni sistema di elaborazione di un proprio sistema operativo e dall'altro all'evoluzione delle diverse funzioni che, di volta in volta, gli sono state aggiunte a seconda dei diversi ambiti applicativi per i quali il sistema di elaborazione viene prioritariamente progettato.

Come avremo modo di evidenziare nel corso dell'articolo, non è facile isolare l'evoluzione dei sistemi operativi da quella delle altre branche dell'intera disciplina informatica essendo l'una parte integrante dell'altra. Le motivazioni e le scelte secondo le quali sono stati sviluppati i sistemi operativi hanno fortemente risentito dei progressi di altre branche dell'intera disciplina e al tempo stesso ne hanno a loro volta influenzato l'evoluzione. Il caso più evidente è quello relativo alla reciproca influenza fra l'evoluzione dei sistemi operativi e quella delle architetture dei sistemi di elaborazione. Per esempio, l'introduzione del meccanismo di interruzione automatica può essere considerata la pietra miliare nello sviluppo della multiprogrammazione che è alla base di tutti i moderni sistemi operativi. Ma da quel punto in poi molte sono state le connessioni tra architettura e sistema operativo. Tanto per citarne alcune delle più significative, ricordiamo l'introduzione dei meccanismi per la gestione della memoria (MMU - *Memory Management Unit*) quali quelli della paginazione e della segmentazione; i meccanismi architetturali per la protezione, come il doppio stato di esecuzione del processore; i meccanismi hardware per la mutua esclusione e la realizzazione degli *spin semaphore* ed altri ancora. Anche i settori relativi ai linguaggi di programmazione e all'ingegneria del software hanno risentito dell'evoluzione dei sistemi operativi. Per esempio, l'intera branca dei linguaggi concorrenti è una diretta conseguenza dell'introduzione del meccanismo della multiprogrammazione e, per quanto riguarda le metodologie di progetto del software è indubbia l'influenza che il progetto del sistema operativo THE (Figura 1), [9] ha avuto nello sviluppo delle metodologie di strutturazione di un sistema software a livelli gerarchici. Infine, con riferimento alle basi di dati, è notevole l'influenza del lavoro di Gray [11] nello sviluppo delle tecniche per garantire l'atomicità delle transazioni.

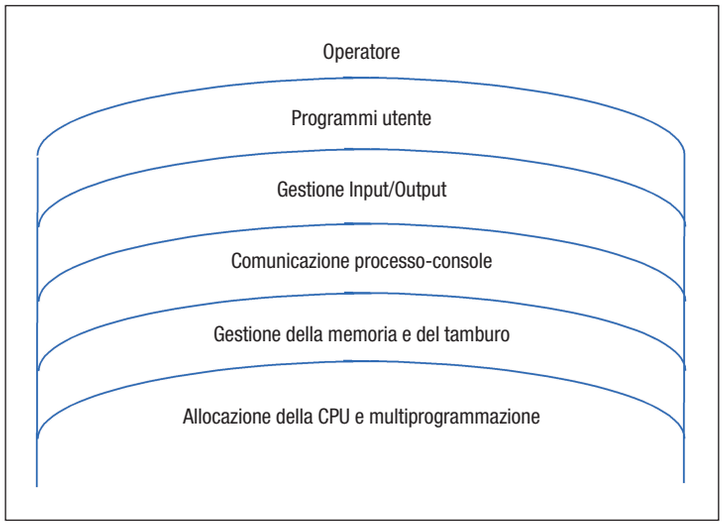


FIGURA 1
Organizzazione gerarchica di un sistema operativo: i sei livelli del sistema operativo THE

2. L'AVVENTO DEI PRIMI CALCOLATORI (1945-1955: ASSENZA DI SISTEMI OPERATIVI)

Il periodo che intercorre tra la fine della II Guerra Mondiale e la metà degli anni '50 è caratterizzato dalla nascita dei primi calcolatori elettronici, inizialmente programmabili dall'esterno tramite connessioni su un pannello di controllo e successivamente mediante programma memorizzato. I primi calcola-

tori a programma memorizzato sono del 1948 (il Mark I presso l'*università di Manchester*) e del 1949 (l'EDSAC progettato da M. Wilkes presso l'*università di Cambridge*).
I calcolatori di questa prima generazione, completamente realizzati mediante valvole termoioniche, erano estremamente costosi, poco affidabili e di difficile utilizzabilità essendo privi di qualunque supporto software. La stessa programmazione era effettuata inizialmente in linguaggio macchina e i programmi caricati manualmente in memoria tramite interruttori presenti sulla consolle del sistema. Successivamente, fu introdotto il linguaggio assemblativo e solo nel 1954 J. Backus annunciò il primo compilatore FORTRAN per il calcolatore IBM 704. Le applicazioni cui erano dedicati erano quelle relative alla risoluzione di problemi di tipo scientifico e tecnologico.
A causa della difficoltà di utilizzazione di questi sistemi, il loro uso era praticamente limitato ai soli specialisti per cui una stessa persona era contemporaneamente progettista, operatore di macchina, programmatore e utente. Era l'unico responsabile della scrittura di un programma applicativo, ivi incluse le particolari istruzioni destinate a controllare il funzionamento dei dispositivi d'ingresso/uscita (I/O), oltre che di caricare in memoria il programma e di controllarne l'esecuzione tramite la consolle. I tipici dispositivi di I/O erano costituiti da macchine elettromeccaniche estremamente lente rispetto alla velocità di esecuzione delle istruzioni da parte della CPU (lettori di schede perforate, perforatori di schede, stampanti) e da nastri magnetici come supporto di memoria massa.
Chiaramente veniva eseguito un solo lavoro (*job*) alla volta. L'utente-programmatore-operatore, aveva il sistema completo a sua esclusiva disposizione per tutto il tempo in cui rimaneva seduto alla consolle.
Con la realizzazione dei primi traduttori, sia di linguaggi assemblativi che di linguaggi ad alto livello (FORTRAN), la tipica esecuzione di un *job* consisteva nel caricare in memoria lo specifico programma traduttore (residente ad esempio su nastro magnetico) ed eseguirlo per tradurre il programma applicativo (tipicamente disponibile su un pacco di schede precedentemente preparato perforando su ogni scheda una diversa istruzione del programma). La traduzione forniva il programma tradotto in linguaggio macchina direttamente su nastro magnetico, oppure perforandolo su un diverso pacco di schede. A quel punto il programma tradotto poteva essere caricato ed eseguito.
Questa modalità operativa era caratterizzata dal grosso inconveniente di limitare fortemente sia l'efficienza di uso delle costose risorse di macchina sia la produttività del sistema (*throughput*). Infatti, a causa delle numerose operazioni manuali, l'efficienza di uso della CPU aveva valori estremamente bassi. Contemporaneamente il *throughput* non superava il valore di uno o due *job* l'ora. D'altro canto, il fatto che l'utente fosse anche programmatore ed operatore e cioè avesse l'intera macchina a sua disposizione e potesse quindi monitorare l'esecuzione dei programmi direttamente dalla consolle, consentiva allo stesso di eseguire facilmente la messa a punto di un programma potendo effettuarne il *debugging* direttamente da consolle, per esempio, potendo bloccare l'esecuzione del programma, modificare manualmente i contenuti di alcune locazioni di memoria o di alcuni registri e rilanciare subito l'esecuzione.

3. NASCITA DEI PRIMI SISTEMI BATCH (1955-1959)

Nella seconda metà degli anni '50 cominciarono ad uscire i primi sistemi realizzati mediante transistor, molto più affidabili dei precedenti a valvole. Erano ancora sistemi fondamentalmente dedicati a elaborazioni di tipo tecnico/scientifico, notevolmente costosi sia in termini di acquisto che di gestione e manutenzione. Tali calcolatori, noti come *mainframes*, erano disponibili quasi esclusivamente presso i centri di calcolo universitari o di banche e grandi imprese ed erano programmati prevalentemente in FORTRAN o in linguaggio assembler.
La disponibilità di linguaggi ad alto livello rese possibile l'accesso al calcolo elettronico anche a ricercatori e tecnici non strettamente informatici ai quali però non poteva essere più lasciato libero accesso al sistema di elaborazione che era riservato, vice-

versa, a particolari tecnici specializzati, gli operatori di macchina. L'utente-programmatore, non più anche operatore, portava al centro calcolo il proprio pacco di schede contenente il programma applicativo da tradurre, caricare ed eseguire, lasciandolo a disposizione degli operatori. Alla fine dell'esecuzione di un singolo *job*, l'operatore prelevava dalla stampante il tabulato con i risultati della sua esecuzione, lasciava il tabulato a disposizione dell'utente del *job* appena terminato e prelevava un pacco di schede relativo al *job* di un altro utente al fine di sottoporlo al sistema.
Pur essendo le operazioni manuali svolte da un tecnico specializzato, queste continuavano a limitare drasticamente l'efficienza di uso del sistema e la sua produttività. È per limitare l'intervento dell'operatore che nacque l'esigenza di dotare la macchina di un programma di sistema, il primo vero e proprio sistema operativo, spesso identificato col termine di *monitor*, a cui demandare il compito di ridurre al minimo l'intervento manuale dell'operatore. Per ottenere questo risultato, per prima cosa i programmi componenti il sistema di programmazione (traduttori dei linguaggi ad alto livello, assemblatore ecc.) furono resi disponibili su nastri magnetici. Quindi, per automatizzare le singole fasi relative all'esecuzione di un *job*, fu necessario definire un semplice linguaggio di controllo, detto anche *Job Control Language* (JCL) (l'antenato dei moderni *shell*). Utilizzando tale linguaggio l'utente preparava il pacco di schede relativo al proprio *job*, in modo tale che, oltre alle schede su cui erano perforati sia il programma sorgente che i dati da elaborare, fossero presenti anche le cosiddette schede di controllo (nelle quali erano perforati i comandi del JCL) facilmente riconoscibili contenendo, nelle prime colonne della scheda, dei caratteri particolari come, ad esempio $. Mediante queste schede l'utente identificava i programmi da eseguire come singoli passi del proprio *job*. Il compito di caricare in memoria i programmi da eseguire non era più demandato all'operatore, ma a un programma (*monitor*) il quale era residente in una porzione della memoria ad esso riservata. Questo era un semplice programma ciclico (*cyclic executer*) che si limitava a leggere una scheda e, se questa era riconosciuta come scheda di controllo, ne veniva interpretato il contenuto, mediante un semplice interprete del JCL, prelevando dal nastro magnetico il programma eseguibile richiesto mediante quella scheda e caricandolo in memoria. Terminato il trasferimento in memoria del programma richiesto, mediante un'istruzione di salto veniva trasferito il controllo della CPU alla prima istruzione del programma caricato. Al termine della sua esecuzione il programma cosi attivato, invece di eseguire l'istruzione di *halt* che avrebbe fermato la macchina, restituiva il controllo al monitor, ancora tramite un salto, e questo continuava con l'esame della successiva scheda di controllo e così via fino alla scheda di controllo che rappresentava la fine dell'esecuzione dell'intero *job*.
Per ridurre ulteriormente l'intervento dell'operatore, fu anche automatizzata la transizione da un *job* al successivo caricando sul lettore di schede invece di un solo pacco, corrispondente al *job* di un singolo utente, un insieme di pacchi di schede, corrispondenti ad altrettanti *job* di utenti diversi che l'operatore aveva raccolto precedentemente (da ciò deriva il nome di *sistemi batch* con cui questi primi sistemi operativi furono conosciuti). In questo modo il monitor poteva passare dall'esecuzione di un *job* a quella del *job* successivo senza soluzione di continuità. L'intervento manuale dell'operatore avveniva solo alla fine dell'esecuzione di un *batch di job*. A quel punto veniva inserito sul lettore un nuovo *batch*, collezionato durante l'esecuzione del *batch* precedente, e la macchina veniva di nuovi riattivata manualmente.
Il primo sistema operativo (GM OS) funzionante con questo criterio fu realizzato dalla *General Motors Research Laboratories* per il loro calcolatore IBM 701 nel 1956 [12]. Successivamente IBM realizzò il sistema operativo FMS (*Fortran Monitor System* [17]) per il proprio calcolatore IBM 709 da cui ebbe origine nel 1959 il *mainframe* IBM 7090 completamente transistorizzato e su cui continuò ad operare lo stesso sistema operativo FMS.
L'introduzione di questi primi sistemi operativi funzionanti con la tecnica *batch* garantirono un drastico aumento dell'efficienza di

uso delle risorse dei sistemi di elaborazione. Alcuni problemi restavano comunque ancora aperti e altri furono introdotti a causa dalle modalità operative proprie dei sistemi *batch*:

a. il primo inconveniente, di natura tecnologica, riguardava l'efficienza. Pur essendo migliorata, questa era comunque ancora molto bassa (dell'ordine di poche unità percentuali). Il principale motivo di questa inefficienza era dovuto alla notevole differenza fra la velocità della CPU e quella, molto più lenta, dei dispositivi periferici (soprattutto lettore di schede e stampante) e al fatto che, non essendo ancora disponibile il meccanismo hardware delle interruzioni, tutti i trasferimenti di I/O avvenivano ancora con la tecnica del "*controllo di programma*". Per cui, quando la CPU attivava un trasferimento di dati (per leggere una scheda o stampare i risultati di un programma) doveva restare in attesa della fine del trasferimento, e questi tempi di attesa erano di alcuni ordini di grandezza superiori a quelli con cui la CPU era in grado di eseguire un intero programma.

b. Il secondo inconveniente, legato alla tecnica *batch*, riguardava la scarsa flessibilità nella scelta dell'ordine con cui i *job* venivano eseguiti che corrispondeva allo stesso ordine con cui comparivano i relativi pacchi di schede sul lettore. Poteva quindi accadere che un utente che aveva presentato un programma molto semplice, e quindi eseguibile molto rapidamente, dovesse aspettare un tempo molto lungo prima di ricevere i propri risultati (*turnaround time*) e ciò solo per il fatto che il pacco di schede del proprio programma era stato posizionato sul lettore di seguito a quello di un utente che aveva richiesto un tempo di esecuzione molto lungo. In altri termini l'ordine con cui i vari *job* venivano eseguiti (*scheduling*) era preordinato dall'operatore senza che il sistema operativo potesse intervenire per modificarlo.

c. Il terzo inconveniente, anch'esso legato alla tecnica *batch*, riguardava il tempo estremamente lungo che era richiesto per completare la "messa a punto" di un programma applicativo. Come già indicato precedentemente, quando il programmatore era anche operatore, poteva quindi: beneficiare del fatto che tutto sistema era a sua competa disposizione e quindi poteva interagire direttamente con la consolle della macchina. Aver allontanato il programmatore dalla macchina influì in maniera estremamente negativa sul processo di scrittura, compilazione, *testing* e *debugging* di un programma. L'utente iniziava tale processo sottoponendo il suo pacco di schede all'operatore. Dopo un tempo che poteva essere dell'ordine di alcune ore, se non di giorni, riceveva il tabulato da cui poteva risultare che il programma non era stato eseguito per alcuni errori sintattici, al limite anche molto banali. Iniziava quindi tutta una serie di tentativi per ottenere un programma sintatticamente corretto e quindi eseguibile. A quel punto iniziava la fase di *testing* per verificare la presenza di errori semantici. Per ogni errore individuato era necessario correggerlo e ricominciare con la compilazione.

4. NASCITA DELLA MULTIPROGRAMMAZIONE: I SISTEMI BATCH MULTIPROGRAMMATI (1960-1970)

Agli inizi degli anni '60 la IBM, per garantire una maggiore efficienza nell'uso delle costose risorse di calcolo dei propri *mainframe* (7090 e 7094) decise di eliminare l'inconveniente a) visto nel precedente paragrafo, obbligando ad utilizzare per tali macchine solo dispositivi di I/O che consentissero elevate velocità di trasferimento dati come i nastri magnetici. Per trasferire un *batch* di *job* da lettore di schede a nastro magnetico veniva usata, fuori linea, un macchina satellite poco costosa come la IBM 1401. Analogamente per trasferire i risultati delle elaborazioni di tutti i *job* di un *batch* dal nastro su cui erano stati scritti alla stampante, veniva di nuovo utilizzata fuori linea la 1401. In questo modo mentre il *mainframe* eseguiva un *batch*, operando sia in ingresso che in uscita soltanto con nastri magnetici, contemporaneamente venivano trasferiti da schede a nastro i *job* del *batch* successivo e da nastro alla stampante i risultati del *batch* precedente.

Il sistema operativo IBSYS della IBM funzionava con tale criterio riuscendo ad ottenere efficienze di uso della CPU del *mainframe* dell'ordine del 90%.

Con l'introduzione delle memorie di massa ad accesso casuale come i dischi magnetici e

tenendo conto che i calcolatori della classe 7090/7094 erano dotati di canali di I/O, veri e propri processori *special purpose* che operavano in maniera asincrona con la CPU, fu possibile ottenere gli stessi risultati dello schema precedente senza utilizzare il calcolatore satellite. Infatti, utilizzando il disco come un *buffer*, mentre il calcolatore, mediante il canale di input, leggeva i *job* di un *batch* trasferendoli su disco, lo stesso poteva eseguire i *job* del *batch* precedentemente trasferito su disco, prelevando i dati e inserendo i risultati sempre su disco. E, contemporaneamente mediante il canale di output, poteva trasferire alla stampante i risultati, presenti anch'essi su disco e relativi al *batch* ancora precedente. Essendo inoltre i *job* da eseguire residenti su disco, era possibile scegliere l'ordine con cui eseguirli non essendo più vincolati dalla sequenzialità con cui gli stessi comparivano sul nastro magnetico, eliminando quindi anche il precedente inconveniente b). Questo tipo di sistemi *batch* fu noto col nome di "*Spooling*" (*Simultaneous Peripheral Operation On Line* [2]).

Nei primi anni '60, con l'introduzione dei meccanismi hardware delle interruzioni automatiche e del DMA (*Direct Memory Access*) fu possibile estendere ulteriormente le funzionalità di un sistema operativo abilitandolo a caricare in memoria principale più programmi contemporaneamente in modo tale che, quando l'esecuzione di uno di essi non poteva proseguire, per esempio in seguito all'attivazione di un trasferimento di I/O, fosse possibile salvare i suoi risultati parziali e passare in esecuzione un altro programma presente in memoria. Soltanto all'arrivo dell'interruzione relativa alla fine del precedente trasferimento, l'esecuzione del primo programma poteva riprendere dal punto in cui era stata precedentemente interrotta. Questa tecnica, nota col nome di *miltiprogrammazione*, consentiva di ridurre al minimo il tempo di inattività della CPU e, al tempo stesso, garantire l'utilizzazione contemporanea di tutte le risorse del sistema (CPU, dispositivi periferici, memoria), aumentando ancora l'efficienza globale di uso delle risorse e diminuendo il *tournaround time*. Come contropartita, la complessità del sistema operativo aumentò drasticamente. Dovendo infatti gestire l'esecuzione contemporanea di più programmi, fu necessario delegare al sistema operativo il compito di implementare le strategie di allocazione di tutte le risorse di calcolo che, se rese disponibili per l'esecuzione di un programma, non potevano essere rese disponibili anche per altri. Analogamente, al fine di evitare che l'esecuzione di un programma erroneo producesse interferenze nelle esecuzioni di altri programmi contemporaneamente presenti in memoria, fu necessario implementare ben precise strategie di protezione anche con l'ausilio di meccanismi hardware all'uopo concepiti. La conseguenza di ciò fu un aumento delle dimensioni e quindi dell'occupazione di memoria del sistema operativo e un maggior tempo di CPU richiesto per l'esecuzione dalle sue funzioni, tempo sottratto all'esecuzione dei programmi applicativi (*overhead* di sistema). I vantaggi della multiprogrammazione furono comunque tali da rendere del tutto trascurabili questi problemi, tenendo anche conto dei continui progressi sul versante delle architetture.

Nei primi sistemi multiprogrammati la risorsa che limitò maggiormente le prestazioni del sistema fu la memoria. Infatti, la necessità della contemporanea presenza in memoria dei programmi fra i quali poteva essere commutato l'uso della CPU, limitava il numero dei programmi stessi in base alle loro dimensioni e alle dimensioni della memoria. Poiché in un sistema multiprogramnmato la CPU resta inutilizzata quando tutti i programmi presenti in memoria sono in attesa della terminazione di un trasferimento di I/O da essi attivato, è ovvio che questo rischio è tanto minore quanto maggiore è il numero di tali programmi.

Uno dei primi sistemi operativi *batch* multiprogrammati fu il sistema OS/360 realizzato dalla IBM per i propri calcolatori della serie System/360 [13].

Fino alla metà degli anni '60 le case costruttrici di computer producevano due diverse tipologie di macchine del tutto incompatibili tra loro: i *mainframe*, macchine di grosse dimensioni, fondamentalmente orientate al calcolo tecnico/scientifico, e macchine di più piccole dimensioni orientare al calcolo di tipo commerciale. Nel 1964 l'IBM annun-

ciò la produzione di una serie di macchine di terza generazione (utilizzando la tecnica dei circuiti integrati), la serie System/360. Ad essa appartenevano macchine di dimensioni diverse, con prezzi e prestazioni diverse, ma tutte software-compatibili tra loro e concepite per soddisfare le esigenze di qualunque tipo di utenza sia scientifica che commerciale.
Lo sviluppo del sistema operativo OS/360 fu concepito in modo tale che tutte le macchine della serie dovessero impiegare varianti, tra loro compatibili, dello stesso sistema operativo. Naturalmente le esigenze delle varie macchine della serie erano diverse tra loro, a seconda che dovessero lavorare in ambiente scientifico o commerciale, che fossero dotate di molte o poche periferiche. Il risultato fu un sistema operativo molto complesso, di enormi dimensioni (milioni di righe di codice, forse il più grande e complesso programma mai sviluppato) scritto completamente in assembler da moltissimi programmatori. Oltre ad impegnare molta memoria e tempo di CPU, il sistema operativo si caratterizzò fin dall'inizio per il rilevante numero di errori in esso contenuti. 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. Un interessante rapporto sull'esperienza con OS/360 è descritto da *Fred Brooks*, uno dei progettisti del sistema, che scrisse in seguito un libro famoso nel settore dell'ingegneria del software [5].
L'esperienza vissuta servì a rendere sempre più evidente la necessità di adottare tecniche per una maggiore strutturazione nella realizzazione di sistemi software di grandi dimensioni e di significativo grado di complessità come nel caso dei sistemi operativi. È dell'ottobre 1968 la prima conferenza di *software engineering*, tenuta a *Garmisch* in Germania, nell'ambito della quale emerse chiara la necessità di fare riferimento, nello sviluppo di software, a metodi formali e metodologie di programmazione strutturata per far fronte alla inerente complessità di sistemi software di grandi dimensioni [18]. Sempre del 1968 è un famoso articolo di *Dijkstra* sulla programmazione strutturata [10].
Un approccio che fu ritenuto valido per affrontare la complessità dello sviluppo di un sistema operativo fu quello di fare riferimento a tecniche di modularizzazione in modo tale da suddividere il sistema in componenti (moduli) ciascuno destinato a fornire una delle sue funzionalità. Secondo questo criterio, ogni modulo deve essere caratterizzato da una ben precisa interfaccia che specifichi la funzionalità offerta dal modulo ed il modo di utilizzarla, e da un corpo contenente la sua implementazione, non visibile all'esterno del modulo.
Nonostante gli indubbi vantaggi indotti dall'uso di tecniche di modularità la complessità dei sistemi operativi andava progressivamente crescendo, richiedendo quindi ulteriori paradigmi di progetto. Fra questi ha svolto un ruolo particolarmente importante la tecnica di organizzazione del software a *livelli gerarchici di astrazione*. Senza entrare nel merito di queste tecniche, peraltro ben note nel campo della programmazione (*top-down*, *buttom-up*), uno degli aspetti principali della tecnica di organizzazione gerarchica consiste nel ridurre il numero di possibili interconnessioni fra i moduli di un sistema, semplificando quindi sia la fase di progetto che quella di verifica. I moduli che costituiscono il sistema operativo vengono organizzati gerarchicamente in diversi livelli di astrazione in modo tale che i moduli realizzati ad un certo livello utilizzano esclusivamente le funzionalità offerte dai moduli di livello più basso e forniscono le loro funzionalità ai moduli di livello più alto. Questo tipo di organizzazione è oggi presente, in misura più o meno completa, in tutti i moderni sistemi operativi. Va ricordato che il primo sistema operativo realizzato secondo questo modello strutturale fu il sistema THE, un sistema batch realizzato alla fine degli anni '60 presso la *Technische Hogeschool di Eindhoven da Edsger Dijkstra* e dai suoi studenti [9]. L'importanza di tale sistema, che rimase un caso di studio, fu proprio nella possibilità offerta di sperimentare in che modo poteva trovare la sua realizzazione il modello a livelli gerarchici calato nella realtà di un semplice sistema operativo.

5. I PRIMI SISTEMI TIME-SHARING (1960-1970)

Lo sviluppo dei sistemi batch multiprogrammati consentì di raggiungere alti livelli di efficienza e di produttività (*throughput rate*), e ciò decretò il loro successo in tutti i casi nei quali non era necessario prevedere interazioni con l'utente; quindi si imposero come sistemi ideali per far girare sia programmi che richiedevano lunghe elaborazioni, come nel caso di applicazioni tecnico/scientifiche, sia programmi che richiedevano l'elaborazione di grosse moli di dati come nel caso di applicazioni commerciali. Restava comunque aperto il problema legato allo sviluppo di tali programmi. Come abbiamo già messo in evidenza nel terzo paragrafo (vedi inconveniente c), l'aver negato al programmatore l'accesso alla macchina implicò una crescita molto marcata del *turnaround time* di un *job* e quindi la corrispondente crescita del tempo necessario per la messa a punto di un programma [23].

È per risolvere questo problema che agli inizi degli anni '60 nacque una nuova linea di ricerca tendente a sviluppare una nuova categoria di sistemi operativi che consentissero ad un utente di tornare a diretto contatto con la macchina, abilitandolo ad operare in maniera conversazionale col sistema stesso. Per questo motivo fu necessario interfacciare la macchina con un diverso tipo di unità periferiche, in particolare di terminali provvisti di una tastiera, con la quale inviare al sistema i comandi che lo stesso avrebbe dovuto eseguire, e di un monitor su cui avrebbero dovuto comparire i risultati dell'esecuzione dei comandi stessi. È del 1960 la nascita del primo computer commerciale provvisto di monitor e tastiera, il PDP-1 della *Digital Equipment Corporation*.

Utilizzando il terminale l'utente poteva quindi inviare al sistema i comandi battendoli direttamente sulla tastiera invece che perforarli su schede. Il sistema operativo, interpretando il comando digitato dall'utente, caricava in memoria ed eseguiva il programma destinato a svolgere il compito da lui richiesto. Alla fine dell'esecuzione del programma, i risultati venivano visualizzati sul video sul quale poi compariva un carattere di *prompt* per indicare all'utente la disponibilità del sistema a ricevere il prossimo comando. Questo tipo di comportamento implicava che l'utente operasse col sistema all'interno di *sessioni di lavoro* che iniziavano quando l'utente si sedeva al terminale. Specificando il proprio codice identificativo (*user-ID*) e la propria parola d'ordine (*password*), si collegava al sistema operativo (*login*) iniziando una sessione di lavoro (invio di un comando al sistema – esecuzione da parte del sistema del compito richiesto – invio del comando successivo – ecc.) fino a quando, eseguendo la procedura di chiusura (*logout*) non decideva di terminare la sessione di lavoro.

Inoltre, per evitare che il sistema fosse monopolizzato da un solo utente alla volta, avendo disponibili più terminali connessi alla macchina e sfruttando il meccanismo della multiprogrammazione, era possibile far accedere più utenti contemporaneamente. In questo modo il sistema poteva eseguire, in multiprogrammazione, i programmi richiesti dai singoli utenti tramite i comandi da loro digitati sulle rispettive tastiere.

Essendo il comportamento del sistema di tipo conversazionale, durante una sessione di lavoro la frazione di tempo che la CPU dedicava all'esecuzione dei programmi richiesti da un utente era sicuramente trascurabile nei confronti del tempo che l'utente stesso dedicava a battere i propri comandi sul terminale. Per cui il sistema operativo poteva sfruttare questi tempi "morti" relativi a un utente per far avanzare i programmi richiesti da altri utenti. Il risultato era quello di dare ad ogni utente l'illusione che la macchina fosse a sua completa disposizione (*macchina virtuale*).

Ciò richiese però anche una diversa filosofia di funzionamento del sistema. L'obiettivo prioritario di questo tipo di sistemi non era più quello di raggiungere la massima efficienza di uso delle risorse di macchina ma bensì quello di ridurre il tempo di risposta cioè il tempo che l'utente avrebbe dovuto attendere per avere sul video i risultati dell'esecuzione del programma richiesto mediante un comando. Inoltre, non era solo importante minimizzare il tempo di risposta di un programma ma, ancora più importante era rendere questo tempo proporzionale alla complessità del programma. Infatti, un uten-

Multics

Il sistema operativo *Multics* fu sviluppato nella seconda metà degli anni 60 presso il *Massachusetts Institute of Technology* (MIT) in collaborazione con i *Bell Telefone Laboratories* e il *Computer Department* della General Electric che fornì il calcolatore sul quale sviluppare Multics (versione modificata del GE 635, denominata GE 645). Il sistema fu pensato come una *computer utility* in grado di venire incontro ad una vasta comunità di utenti. In modo suggestivo il sistema fu presentato come un servizio, alla stessa stregua dell'acqua o dell'elettricità a disposizione di una vasta comunità di utenti (nella sua concezione originale il sistema avrebbe dovuto poter servire le necessità di calcolo dell'intera città di Boston). Il servizio quindi doveva avere caratteristiche di alta affidabilità, in pratica funzionare 7 giorni su 7, 24 h al giorno. Gli utenti avrebbero potuto accedere sia da terminali remoti che centralizzati.
Multics si rivelò un progetto troppo ambizioso per il suo tempo: la sua trasformazione in un prodotto industriale risultò molto complicata anche perché i *Bell Laboratories* abbandonarono il progetto e la GE cessò l'attività nel settore computer. Il MIT continuò tuttavia a lavorare sul progetto e lo condusse a termine. Il sistema fu venduto dalla Honeywell che aveva acquisito il ramo calcolatori della GE ed installato in 80 grandi aziende e università nel mondo.

te è tendenzialmente disponibile ad attendere al terminale il completamento di un programma molto complesso per un tempo maggiore rispetto a quanto sia disponibile ad attendere il completamento di un programma banale. I principi di funzionamento tipici dei sistemi *batch* non erano adatti a fornire questo tipo di comportamento. In tali sistemi, infatti, quando entrava in esecuzione un programma che richiedeva lunghe elaborazioni (*CPU-bound*) senza trasferimenti di I/O, questo poteva mantenere il controllo della CPU per molto tempo senza consentire l'esecuzione di altri programmi, neppure di quelli molto semplici che, viceversa, avrebbero dovuto terminare il più rapidamente possibile. Per ovviare a questo problema fu deciso che la CPU, di volta in volta, fosse dedicata all'esecuzione di un programma per un intervallo di tempo dell'ordine di qualche decina di millisecondi (*time slice*), trascorso il quale il programma, se non fosse terminato o non si fosse sospeso in attesa di un trasferimento di I/O, doveva comunque essere interrotto e la CPU essere dedicata all'esecuzione di un altro programma e così via in maniera circolare prima che la CPU fosse di nuovo dedicata al programma interrotto. È da questo tipo di comportamento che nacque il nome di *time-sharing*, o a divisione di tempo, con cui furono identificati i sistemi operativi di questa nuova categoria.
I primi studi su questa tipologia di sistemi operativi furono svolti da *John McCarthy, Robert Fano e Fernando Corbato al Department of Electrical Engineering* del MIT. Tali studi condussero alla realizzazione, nel 1961, del primo sistema *time-sharing*, il sistema *CTSS – Compatible Time-Sharing* [7]. Questo sistema utilizzava un calcolatore della classe IBM 7090 ritenuto, per quei tempi, molto potente (IBM 7094, con 32 K di memoria, di cui 5 k occupate dal monitor, parole a 36 bit, tamburo veloce come memoria secondaria). Gestiva fino a 32 utenti che avevano la possibilità, utilizzando un insieme di comandi del sistema operativo, di operare sul sistema in modo interattivo tramite terminali compilando e mettendo in esecuzione i propri programmi e operando sui propri file.
Come prosecuzione dell'esperienza, sotto molti aspetti positiva, del CTSS, nel 1964 presso il MIT fu finanziato il Progetto MAC (*Machine Aided Cognition*), che avrebbe dovuto condurre al successivo passo in avanti nello sviluppo dei sistemi *Time-Sharing* con la progettazione del sistema *Multics* [6] in collaborazione con i *Bell Telefone Laboratories* ed il *Computer Department* della General Electric (riquadro). Il *Multics* introdusse una serie di innovazioni di grande importanza che troveremo anche nei recenti sistemi.
Con l'introduzione dei sistemi *time-sharing* aumentò molto la complessità del sistema operativo rispetto ai precedenti sistemi di tipo *batch*. Infatti, a causa della *multiutenza* fu necessario affrontare nuovi problemi di sicurezza e di protezione delle informazioni, controllando gli accessi al sistema (gestione degli *account*) e implementando meccanismi di protezione sia del sistema che dei dati e dei programmi dei singoli utenti. Inoltre si complicò notevolmente il problema relativo alla gestione della memoria. Infatti, per non limitare il numero degli utenti contemporaneamente attivi fu necessario garantire elevati livelli di multiprogrammazione e cioè di programmi contemporaneamente in esecuzione. Fu quindi necessario svincolare il numero di questi programmi dalle dimensioni della memoria fisica centrale che li avrebbe dovuti contenere. Nacque quindi la necessità di implementare strategie di allocazione dinamica della memoria e di realizzazione del concetto di memoria virtuale [8]. Il *Multics* fu il primo

sistema nel quale molte di queste nuove tecniche furono sperimentate.
I progettisti di *Multics* furono i primi ad introdurre il concetto di processo. Per la prima volta il comportamento dei programmi in esecuzione veniva regolamentato facendo loro corrispondere processi dotati di uno stato (pronto, in esecuzione, bloccato) cui venivano assegnate da parte del sistema operativo le risorse sulla base di determinate politiche. Il concetto di processo è stato poi adottato da tutti i sistemi operativi ed è alla base delle metodologie e tecniche di programmazione concorrente [3].
Le tecniche di comunicazione tra processi utilizzate in *Unix* ed in altri sistemi operativi attuali derivano da quelle introdotte in *Multics*.
L'aspetto forse più interessante introdotto da *Multics* è quello che riguarda la protezione con l'adozione di un'architettura basata su più livelli (*ring architecture*) che è una generalizzazione di quella classica basata sui due livelli (*user e supervisor*). I livelli di protezione sono organizzati in modo gerarchico in una struttura ad anelli numerati da 0 a 7. In generale indicando con *Ri* e *Rj* due generici anelli, se $i < j$ significa che *Ri* ha più diritti di *Rj*. I primi livelli sono quelli utilizzati dal sistema operativo (*R*0 è il livello del nucleo), gli altri dai programmi di utente. Il tipo di soluzione adottato è stato ripreso ed è presente in molte architetture di moderni microprocessori.
È importante notare che, praticamente, nello stesso periodo e nello stesso luogo, la IBM sviluppò a sua volta il primo sistema operativo *time-sharing* per le proprie macchine. In particolare, nel 1964, presso il centro scientifico IBM di Cambridge, fu sviluppato il sistema CP/CMS per un particolare modello del sistema IBM/360 [16]. Questo modello, il 67, fu l'unico di tutta la serie 360 a disporre di meccanismi hardware utili per la realizzazione di sistemi *time-sharing*, primo fra tutti il meccanismo della paginazione a due livelli necessario per implementare memorie virtuali di grandi dimensioni.
Il CP/CMS è stato il primo sistema operativo ad implementare l'architettura a macchine virtuali che, successivamente, verrà utilizzata in altri settori applicativi. CP/CMS possedeva infatti due componenti principali: il CP (*Control Program*) era un sistema a macchine virtuali il quale forniva ad ogni singolo utente un ambiente assimilabile ad un *mainframe* personale; il CMS (*Conversational Monitor System*) era un semplice sistema operativo monoutente progettato per essere eseguito, principalmente, in ambiente CP. Ad ogni utente, quindi, di CP/CMS è fornita la propria macchina virtuale su cui eseguire CMS. Sulla macchina virtuale, però, l'utente poteva caricare un altro sistema operativo al posto del CMS. Ciò risultò estremamente utile come ambiente nel quale sviluppare nuovi sistemi operativi.
Il CP/CMS è stato il predecessore di tutta la famiglia di sistemi operativi VM (*Virtual Machine*) successivamente utilizzati i tutte le macchine della serie IBM/370 [23].

6. I SISTEMI GENERAL PURPOSE (1970-1980)

Dalla fine degli anni '60 e per tutto il decennio 70-80, l'evoluzione dei sistemi operativi continua sull'onda del successo riscontato dai sistemi *time-sharing* e favorita anche dalla realizzazione di architetture dei calcolatori sempre più efficienti e sofisticate, dalla nascita dei primi minicomputer realizzati utilizzando circuiti integrati e dallo sviluppo dei protocolli di comunicazione tra calcolatori (TCP/IP). In particolare, nascono i primi sistemi operativi dedicati ad applicazioni *real-time*. Vengono inoltre sviluppati sistemi operativi *general purpose* dove gli utenti possono far girare programmi con la filosofia tipica di sistemi *batch* e, al tempo stesso, eseguire altri programmi in *time-sharing* e specificare al sistema anche la necessità di eseguire programmi con caratteristiche *real-time*.
Nel 1979 esce il primo minicomputer a 16 bit della serie PDP-11 della DEC, serie che avrà un enorme successo sia per la semplicità con cui poteva essere programmato sia per il costo notevolmente inferiore ai grossi *mainframe*.
Nel 1973 la Digital fornisce per la serie PDP-11 un sistema operativo molto semplice dedicato sia allo sviluppo di programmi che a semplici applicazioni *real-time* (*data acquisition*): il sistema RT-11 (Real-Time-11). Si trattava di un sistema operativo monoutente che non forniva il *multitasking* sebbene potesse es-

sere configurato per fornire supporto ad un singolo processo in "*foreground*" (ad alta priorità) contemporaneamente ad un singolo processo in "background" (bassa priorità).
Nel 1974, sempre per la famiglia di calcolatori PDP-11, esce il sistema RSX-11, un sistema multiprogrammato ancora adatto ad applicazioni *real-time*, un sistema che avrà notevoli influenze sul progetto di successivi sistemi, ivi incluso Windows NT.
Nel 1977 la DEC fornisce una nuova serie di sistemi destinati a sostituire le macchine della serie PDP-11: i sistemi della serie VAX-11 per i quali viene realizzato il sistema operativo VMS (*Virtual Memory System* [24]) che recepisce molte caratteristiche dal sistema RSX-11 e che avrà un notevole successo, in particolare in ambienti scientifici, dove è rimasto in uso anche in anni recenti operando anche sulle macchine della serie Alpha e apprezzato in particolare per la sua flessibilità e sicurezza. VMS è un sistema operativo multiutente, multiprogrammato concepito per essere utilizzato come sistema *time-sharing*, particolarmente adatto per applicazioni basate su transazioni, ma abilitato ad eseguire anche programmi in *batch* e, specificando opportunamente le relative priorità, anche in tempo reale. È stato utilizzato per l'architettura di rete DECnet, sviluppata da Digital, oltre che per l'implementazione dei protocolli TCP/IP.
Dopo la vicenda OS/360 anche IBM decise di sviluppare un sistema operativo di tipo *general purpose* e nel 1974 rilasciò la prima versione del sistema MVS (*Multiple Virtual Storage*) destinato ad operare sulle macchine della serie 370 e, successivamente, 390 (in quel caso noto anche col nome di OS/390). MVS, in quanto successore di OS/360 era adatto ad eseguire *job* di tipo *batch* ma, mentre uno di questi girava in background, era possibile operare anche in maniera interattiva mediante l'opzione *time-sharing*. Era un sistema in grado di fornire il supporto anche a configurazioni multielaboratore e garantiva l'interconnessione di più macchine mediate i protocolli SNA (*System Network Architecture*), protocolli proprietari, e successivamente anche mediante i protocolli TCP/IP.
Nel 1969 Brinch Hansen sviluppò il sistema RC 4000 secondo un modello strutturale che diventerà noto come "modello a scambio di messaggi" o "modello a memoria locale" duale rispetto a quello utilizzato per lo sviluppo del THE basato sul "modello a memoria comune". RC 4000 è il primo sistema che introduce il concetto di *microkernel*. Anche in questo sistema un utente può scegliere di aprire una sessione *multitasking* con *preemption*, un altro può scegliere la modalità *single-user* mode per operare in *batch*. Anche lo *scheduling real-time* è previsto utilizzando la possibilità di inviare messaggi al processo *timer*.

7. LO SVILUPPO DI UNIX

La nascita di *Unix* si deve in gran parte agli ostacoli incontrati nel trasformare il progetto *Multics* in un prodotto industriale (riquadro). L'obiettivo degli sviluppatori fu quello di realizzare un sistema operativo che pur mantenendo molte delle caratteristiche di *Multics*, ne semplificasse la struttura, riducendone sensibilmente le dimensioni e la complessità.
Mentre la prima versione era stata scritta in assembler, le versioni successive furono scritte in un linguaggio ad alto livello ideato da Ken Thompson che fu chiamato B (basato

Nascita di UNIX
La nascita di *Unix* si deve in gran parte agli ostacoli incontrati nel trasformare il progetto *Multics* in un prodotto industriale. I *Bell Labs* che avevano partecipato fin dall'inizio al progetto, ritenendo troppo complessa l'operazione, la abbandonarono nel 1969. Alcuni ricercatori dei *Labs* decisero nonostante tutto di continuare lo sviluppo del progetto. Furono in particolare Ken Thompson e Dennis Ritchie a non arrendersi: fu grazie ai loro sforzi che, su una macchina non più utilizzata all'interno dei *Labs*, un PDP-7, nacque la versione finale di *Unics* (in seguito *Unix*). Nome che stava a sottolineare la voluta semplicità del progetto rispetto alla mal gestita complessità di *Multics*. Visto l'interesse destato tra i ricercatori dei *Labs*, *Unix* fu trasportato su un calcolatore più potente della Digital, il PD11/20 e più tardi sul PDP11/45 ed infine sul PDP11/70.
Nel 1974 Ritchie e Thompson pubblicarono un articolo su *Unix* [19], per il quale i due autori ricevettero più avanti il prestigioso premio Turing dell'ACM, che spinse molte Università a chiedere ai *Bell Labs* una copia del sistema operativo. Il che avvenne, a condizioni particolarmente vantaggiose essendo fatto divieto dalle leggi federali americane alla AT&T (proprietaria dei *Bell Labs*) di vendere prodotti per computer.

sul linguaggio BCPL) con l'obiettivo di aumentarne la portabilità. La versione definitiva fu scritta in un nuovo linguaggio progettato da Ritchie, chiamato C con il risultato, anche, di dimostrare quanto fosse vantaggioso utilizzare un linguaggio di alto livello per scrivere, se non tutto, almeno la gran parte del codice di un sistema operativo.
La versione 6 divenne la prima disponibile fuori dai Bell Labs nel 1976. La versione 7, successiva, uscita nel 1978 cominciò ad essere usata, oltre che nelle Università, anche nelle aziende industriali.
Una realizzazione importante di *Unix* fu quella sviluppata presso l'Università di Berkeley in California, conosciuta come *Berkeley Software Distribution Unix* (BSD *Unix*), di cui furono prodotte diverse versioni fino alla più nota, la 4.3BSD sviluppata per il calcolatore VAX [15].
Questa versione apportò significativi arricchimenti alle versioni precedenti, tra cui l'utilizzo della memoria virtuale e dell'impaginazione ed in particolare l'utilizzo delle socket come strumento per la comunicazione tra processi in rete tramite TCP/IP.
Negli anni '80 la grande popolarità di *Unix* ed il fatto che fosse disponibile il codice sorgente, inevitabilmente determinò il proliferare di svariate realizzazioni, spesso con notevoli differenze tra loro che hanno ostacolato una piena portabilità delle applicazioni. In particolare, le due famiglie di sistemi *Unix* maggiormente affermatesi sono state *Unix System* V [4], prodotto dai laboratori dell'A&T[1] e BSD *Unix*.
Nel 1988, grazie all'intervento dell'IEEE Standard Board, fu definito lo standard POSIX (*Portable Operating System Unix*) [14] che definisce un'interfaccia (API) tra sistema operativo e applicazioni. Le applicazioni che utilizzano i servizi del sistema operativo attraverso l'interfaccia POSIX hanno ottime caratteristiche di portabilità verso altri sistemi operativi conformi allo standard. Lo standard POSIX fu riconosciuto dall'*International Standard Organization* (ISO) e le due versioni *Unix System* V e BSD (versione 4.3), pur mantenendo alcune differenze tra loro, sono state sviluppate entrambe in maniera conforme allo standard POSIX[2].
Alcune aziende si crearono la loro versione di *Unix*, come *Solaris* della SUN (basata su System V)[3] e AIX di IBM (basato sia su System V sia su BSD). Anche Microsoft si interessò per un certo numero di anni ad Unix creando il sistema operativo XENIX.
La derivazione di *Unix* da *Multics* fece sì che fin dalle sue prime versioni fossero presenti nel sistema operativo molte delle caratteristiche di *Multics*. Innanzi tutto il concetto di processo, la multiprogrammazione e la gestione della multiutenza secondo le tecniche del *time-sharing*, l'uso del file per la rappresentazione di ogni risorsa di sistema, l'organizzazione gerarchica del file system. Il sistema di protezione fu volutamente semplificato con l'introduzione delle liste di controllo degli accessi per regolare l'accesso degli utenti ai file.
Le versioni successive portarono ad una crescita delle funzionalità di *Unix*, in particolare la memoria virtuale, le modalità di interazione tra i processi con l'utilizzo dei segnali per la loro sincronizzazione e delle *pipe* e *socket* per la comunicazione rispettivamente in ambiente a memoria comune e in sistemi distribuiti. Per facilitare l'interazione dell'utente, all'originario linguaggio di comandi fu affiancato un sistema di interfacce grafiche che fanno uso di finestre, icone etc. La più nota tra queste è CDE (*Common Dekstop Environment*) frutto di un lavoro di standardizzazione portato avanti dalle più importanti aziende produttrici di *Unix*, che presenta caratteristiche analoghe a quelle dell'interfaccia grafica del sistema operativo Windows NT.
Al di là di questi aspetti, che oggi sono propri di tutti i moderni sistemi operativi, *Unix*

[1] Nel 1984 il governo degli Stati Uniti permise la suddivisione della AT&T; nacque così una consociata che trattava computer che mise sul mercato UNIX System III, che fu infine rimpiazzato da UNIX System V.

[2] Un altro gruppo di sviluppatori di UNIX (IBM, DEC, HP e molti altri) preoccupati dal fatto che AT&T avesse il controllo di UNIX formarono la OSF (*Open Software Foundation*) con lo scopo di produrre una propria versione di UNIX senza vincoli di proprietà. Nel tempo la OSF lentamente scomparve a favore delle versioni AT&T e BSD.

[3] Solaris, il cui primo nome era SunOS, fu creato da Bill Joy che fondò nel 1982 la Sun Microsystems.

ricopre nella storia dei sistemi operativi una posizione di grande importanza. Innanzitutto, per i motivi detti, è il primo sistema operativo non proprietario di cui fu reso disponibile liberamente il codice. Il fatto che fin dalle sue prime versioni, *Unix* venne scritto utilizzando, per la maggior parte del progetto, il linguaggio C (circa il 60%), ne facilitò la diffusione presso Università ed aziende. Infatti, oltre ai vantaggi relativi ad una sua più agevole comprensione, il sistema risultava dotato di una buona portabilità, intesa come capacità di fornire lo stesso ambiente di esecuzione e di sviluppo su architetture diverse. Questa caratteristica, decisamente inusuale per l'epoca in cui *Unix* è nato, è stata la chiave di volta della capillare diffusione che ha raggiunto ai tempi nostri.
Di pari passo anche il linguaggio C, nato come formalismo per lo sviluppo di *Unix*, si è diffuso in maniera tale da divenire uno standard di fatto tra i linguaggi di programmazione imperativi rimpiazzando, per esempio, gradualmente il linguaggio *Pascal* come linguaggio prescelto nei corsi di programmazione nelle università (anche se, a parere degli autori, si possono nutrire fondati dubbi sulla sua validità a fini didattici).

8. LINUX

Visto il successo di *Unix*, furono sviluppati molti altri sistemi operativi basati su di esso. Nel 1985, presso l'*Università di Carnagie Mellon*, fu sviluppato il progetto Mach destinato a realizzare un sistema operativo (il sistema Mach [1]) con lo scopo di fornire un ambiente di supporto alla ricerca sui sistemi operativi. Tale sistema forniva un'interfaccia (API) simile a quella di *Unix* ed era in grado di girare su un'ampia varietà di calcolatori commerciali, fornendo anche il supporto ad architetture multielaboratore. Ma la caratteristica più importante di questo sistema è stata quella di essere strutturato secondo il modello a *microkernel* di cui è stato sicuramente uno dei primi esempi e forse il più famoso. L'idea che sta alla base del *microkernel* è quella di fornire minime funzionalità nel *kernel* per renderlo efficiente ed affidabile (in generale le funzioni del *microkernel* coincidono con la gestione dei processi, delle loro interazioni e con la gestione delle interruzioni), mentre funzioni come la gestione della memoria ed il file system operano come processi utente.
Pur essendo ormai terminata l'esperienza di ricerca di Mach, questa ha influenzato molti sistemi commerciali, fra questi Mac-OS, il sistema operativo dei calcolatori Macintosh.
Nel 1987, Andrew Tanenbaum dell'*Università di Vrije di Amsterdam* sviluppò MINIX [22], una versione ridotta di *Unix* realizzata per insegnare le basi dei sistemi operativi, funzionalmente molto simile a *Unix* e conforme a POSIX. Va ricordato che MINIX fu anch'esso uno dei primi sistemi operativi tipo-*Unix* che utilizzò una struttura a *microkernel*.
Il desiderio di scrivere una versione professionale gratuita (contrapposta a quella didattica) di MINIX fece sì che uno studente dell'Università di Helsinki, Linus Torvalds, nel 1991 sviluppasse una nuova versione del *kernel* di MINIX per il processore 80386, la prima vera CPU a 32 bit dell'Intel (riquadro p. 61).
I sorgenti della prima versione di *Linux* (il codice totale era di 9300 righe di C e 950 righe di assembler) furono resi accessibili gratuitamente attraverso Internet [25].
Questa possibilità invogliò molti programmatori ad analizzare, modificare ed estendere le varie componenti del sistema operativo. Questo fenomeno provocò un velocissimo progresso nel completamento e nel miglioramento del sistema, che in breve tempo, da un prototipo "giocattolo" di sistema operativo divenne un sistema operativo completo, moderno, robusto ed efficace che aveva come suo punto di riferimento *Unix*, la sua struttura, le sue chiamate di sistema, le librerie, gli algoritmi (più dell'80% delle chiamate di sistema *Linux* sono una copia esatta delle corrispondenti chiamate di sistema in POSIX, BSD o System V), ma che nel tempo è stato modificato ed esteso acquisendo alcune caratteristiche che oggi lo differenziano rispetto al sistema da cui ha tratto ispirazione.
Una delle differenze più significative riguarda la gestione dei processi; in *Linux* sono presenti e gestiti a livello di *kernel* i *threads* (processi leggeri) che rappresentano quindi l'unità di *scheduling* e che consentano rispetto ad una gestione basata solo sui *processi pesanti* di ottenere migliori prestazioni in termini di efficienza e sfruttamento del parallelismo.

Nascita e sviluppo di Linux

Linux nasce nel 1991 in seguito al lavoro di uno studente finlandese, Linus Torvalds, che, con lo scopo di studiare, sperimentare e, al tempo stesso, divertirsi, riprogettò il *kernel* del sistema operativo MINIX, realizzato da Tanenbaum per scopi didattici, cercando di sfruttare al meglio le caratteristiche delle architetture basate sui microprocessori Intel i386, all'epoca le più diffuse e, secondo Linus, non efficientemente utilizzate da MINIX che era strutturato secondo il modello a *microkernel*.
Con questo obiettivo il nuovo *kernel* fu organizzato secondo il modello monolitico, tipico di *Unix*, sicuramente meno flessibile rispetto a un'organizzazione a *microkernel*, ma certamente più efficiente.
La prima versione di *Linux*, oltre al nuovo *kernel* completamente riprogrammato, continuava ad utilizzare, per semplicità, la restante parte del sistema operativo MINIX da cui il nuovo progetto era originato. Successivamente però, anche per svincolarsi dai soli scopi didattici a cui MINIX era orientato, Torvalds sostituì quella parte di MINIX con il software relativo al progetto GNU (progetto lanciato nel 1984 dalla *Free Software Foundation* per sviluppare un sistema operativo *Unix-compatibile* completo e che fosse reso disponibile come software libero).
Per questo motivo, Torvalds decise inoltre di rendere disponibile tutto il software prodotto con la licenza GPL (*General Public License*) della GNU dando quindi vita a un progetto *open source* aperto a contributi esterni, cosa che, come è noto, ha fortemente contribuito al successo di *Linux*.
La nascita di *Linux*, originariamente come alternativo a MINIX, dette vita a una lunga diatriba su internet fra Torvalds e Tanenbaum. Quest'ultimo, nel 1992, probabilmente infastidito dal crescente interesse verso *Linux*, scrisse un messaggio dal titolo estremamente chiaro: "*Linux is obsolete*", criticando aspramente l'architettura basata su *kernel* monolitico sicuramente meno innovativa di quella di MINIX. A tale messaggio rispose Torvalds cercando di mettere in evidenza i vantaggi del nuovo sistema in termini di efficienza. Il dibattito è poi continuato nel tempo come si può verificare su vari siti internet (vedi per esempio, http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html).

Linux supporta inoltre molte funzionalità avanzate quali l'elaborazione multipla parallela (SMP, *Simmetric Multiprocessing*), l'accesso a memoria non uniforme (NUMA, *Non Uniform Memory Access*) ed il supporto ad un ampio spettro di architetture hardware.
La complessità raggiunta dall'attuale prodotto rende difficile, ed in alcune situazioni infattibile, la pratica adottata negli anni passati di scaricare personalmente il software da Internet e provvedere direttamente alla sua installazione. Sono nate, per ovviare a questo problema, delle distribuzioni che includono il *kernel*, le applicazioni ed interfacce utente, insieme ad altri strumenti ed accessori. Attualmente si contano più di 300 tipi di distribuzione. Tra questi possiamo citare *Red Hat, SuSE* (*Novell*), *Debian, Mandria Linux, Slackware*.

Bibliografia

[1] Accetta M., Baron R., Bolosky W., Golub D.B., Rashid R., Tevanian AS., Young M.: *Mach: a New Kernel Foundation for Unix Development*. Proceedings of the Summer 1986 USENIX Conference, June 1986.

[2] Ancilotti P., Boari M., Ciampolini A., Lipari G.: *Sistemi Operativi*. McGraw-Hill,2008, 2° edizione.

[3] Ancilotti P., Boari M.: *Programmazione concorrente e distribuita*. McGraw-Hill, 2007.

[4] Bach M.J.: *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, N.Y., 1987.

[5] Brooks Jr. F.P.: *The Mythical Man-Month: Essays in Software Engineering*. Reading, Mass: Addison Wesley, 1975.

[6] Corbato F.J., Vyssotsky,: *Introduction and overview of the Multics system*. Proceedings of the AFIPS Fall Join Computer Conference, 1962.

[7] Crisman P.A. et al.: T*he Compatible Timesharing System*. Cambridge, Mass: MIT Press, 1964.

[8] Denning P.J.: *Virtual Memory*. ACM Computing Surveys, Settembre 1970.

[9] Dijkstra E.W.: The structure of the THE Multiprogramming System. *Communications of the ACM*, Vol. 11, n. 5, May 1968.

[10] Dijkstra E.W.: Go To Statement Considered Harmful. *Communications of the ACM*, Vol. 11, 1968, p. 147-148.

[11] Gray J.N: Notes on Database Operating Systems. *Lectures Notes in Computer Science*, Vol. 60, Operating Systems – An Advanced Course, Springer Verlag, 1978.

[12] Grosch H.R.J.: The Way It Was in 1957. *Datamation*, September 1977, p. 121-132.

[13] IBM: *IBM System/360 Operating System – Introduction*. IBM System Reference Library, file Number S360-20, Order number GC28-6534-3.

[14] Inst of Elect & Electronic Eng: *IEEE Standard Portable Operating System Interface for Computer Environments 1003.1*. Revised edition (September 1988).

[15] Leffler S.J., McKusick M.K., Karels M.J., Quarterman J.S.: *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, November, 1989.

[17] Meyer R. A., Seawright L. H.: A virtual machine time-sharing system. *IBM Systems Journal*, Vol. 9, n. 3, September 1970, p. 199–218.

[17] Mooney J.: *Early Operating Systems*. West Virginia University, www.csee.wvu.edu/~jdm/classes/cs258/OScat/early.html, January 27, 2004.

[18] Naur P., Randell B. (eds.): *Software Engineering*. NATO Scientific Affairs Division, Brussels. Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October, 1998, 1969.

[19] Ritchie D.M., Thompson K.: The Unix Time-Sharing System. *Communications of the ACM*, Vol. 17, n. 7, July 1974.

[20] Silberschatz A., Galvin P.B., Gagne G.: *Operating System Concepts*. Seventh Edition, John Wiley & Sons, December 2004.

[21] Stallings W.: *Operating Systems: Internals and Design Principles*. Fifth edition, Prentice Hall, 2005.

[22] Tanenbaum A.S, Woodhull A.S.: *Operating System: design and implementation*. 3 ed. Upper Saddle River, NJ: Prentice Hall, 2006.

[23] Weizer N.: A History of Operating Systems. *Datamation*, January 1981, p. 119-126.

[24] http://h18000.www1.hp.com/info/SP2501/SP2501PF.PDF

[25] http://www.linux.org/info/index.html

Paolo Ancilotti è professore ordinario di Sistemi Operativi presso la Scuola Superiore Sant'Anna di Pisa di cui è stato Direttore. È autore di numerosi articoli scientifici ed alcuni libri. Ha interessi di ricerca nel settore dei sistemi operativi, dei sistemi real-time e della programmazione concorrente.
E-mail: p.ancilotti@sssup.it

Maurelio Boari è professore ordinario di calcolatori elettronici presso la Facoltà di Ingegneria dell'Università di Bologna. È autore di numerosi articoli scientifici ed alcuni libri. Ha interessi di ricerca nel settore dei sistemi distribuiti, linguaggi di programmazione e sistemi operativi. È attualmente delegato del Rettore per l'informatica e le reti di Ateneo.
E-mail: maurelio.boari@unibo.it