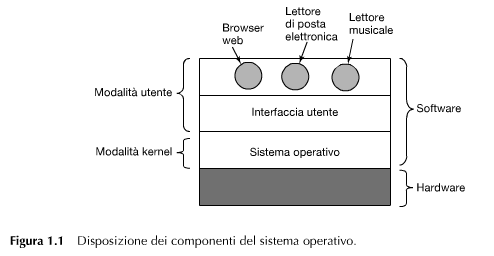
**Introduzione**

Il Sistema Operativo (SO) è quello strato software che avvolge la macchina e che permette l’utilizzo ottimizzato ed astratto delle risorse che la compongono.

Un errore comune è considerare la GUI (Graphical User Interface) come parte del SO, in realtà è solo un’applicazione che lo utilizza.

La maggior parte dei computer ha due modalità operative: kernel (o supervisor) e user.

L’interprete dei comandi (shell), i sistemi a finestre, i compilatori, gli editor e gli altri programmi indipendenti dalle applicazioni vengono eseguite in modalità user. Il sistema operativo è quella porzione di software che viene eseguito in modalità kernel, ed è protetto dalle manomissioni degli utenti dell’hardware.



**Che cos’è un sistema operativo?**

Risulta difficile definire esattamente cosa sia un sistema operativo. Il sistema operativo realizza due funzionalità distinte:

- fornisce ai programmatori ed alle applicazioni software un insieme di risorse astratte invece del complicato hardware;

- gestisce le risorse hardware.

**Il sistema operativo come macchina estesa**

L’astrazione è un concetto chiave per gestire la complessità: in un sistema complesso si identificano delle caratteristiche salienti rispetto ad altre considerate trascurabili e lo si rappresenta attraverso un modello che contiene solo gli aspetti principali.

Una buona astrazione trasforma un problema impossibile in uno risolubile attraverso due semplici passaggi:

- definizione e realizzazione delle astrazioni;

- utilizzo delle astrazioni per risolvere il problema.

Per esempio il file è un concetto astratto che ogni utente conosce e utilizza: mentre l’utente “vede” il nome, la data, il tipo, il percorso astratto ove è memorizzato, il SO conosce tutti i dettagli che servono per la sua memorizzazione sul dispositivo I/O, come ad esempio la sua posizione fisica.

**Il sistema operativo come gestore delle risorse**

Il SO ha come compito principale quello di mettere a disposizione dell’utente una comoda interfaccia.

I SO moderni permettono di eseguire più programmi contemporaneamente.

Tra i compiti del SO troviamo:

- gestire i conflitti derivanti dall’accesso alle risorse condivise (es. stampanti, memoria,...);

- garantire che ogni processo abbia accesso alle risorse in coerenza con le autorizzazioni di cui dispone;

- tenere traccia di quali processi stanno utilizzando quale risorsa.

La gestione delle risorse include il multiplexing (in termini di condivisione) delle risorse e può avvenire sotto due aspetti:

- rispetto al tempo: la risorsa è utilizzata a turno dai processi;

- rispetto allo spazio: la risorsa è partizionata e più processi accedono concorrentemente alla risorsa (es. memoria).

**Storia dei sistemi operativi**

**La prima generazione (1945-1955): valvole e schede a spinotti**

Tutta la programmazione veniva effettuata in linguaggio macchina, predisponendo una serie di spinotti su schede particolari che servivano per controllare le funzioni elementari della macchina.

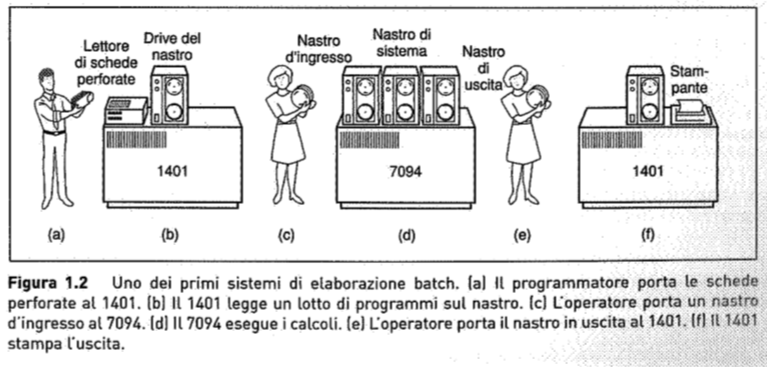
Negli anni ’50 vennero introdotte le schede perforate, che rendevano possibile scrivere programmi sulle schede e leggerli tramite il calcolatore, invece di usare le schede a spinotti.

**La seconda generazione (1955-1965): transistor e sistemi batch**

L’introduzione dei transistor durante la metà degli anni ’50 cambiò radicalmente la situazione. Questi calcolatori, ora chiamati mainframe, erano molto costosi e solo grosse compagnie, agenzie governative o università potevano spendere milioni di dollari.

Per far girare un job (cioè un programma o un insieme di programmi), un programmatore doveva prima scrivere il programma su carta (in FORTRAN o in assembler), poi doveva copiarlo su schede perforate (successivamente sui nastri), infine portare il pacchetto da un operatore e aspettare i dati in uscita.

Successivamente, per velocizzare le operazione, vennero inventati i sistemi batch (sistemi di elaborazione a lotti). L’idea era quella di trasferire i job su nastro magnetico usando un calcolatore poco costoso, ma non adatto ad eseguire i calcoli che venivano eseguiti su calcolatori molto più costosi.



**La terza generazione (1965-1980): circuiti integrati e multiprogrammazione**

Nasce l’IBM System/360, che utilizza i circuiti integrati, ed il sistema operativo OS/360 che permette la multiprogrammazione (la zSeries è un suo discendente).

Questi sistemi possono leggere i programmi dal disco attraverso una tecnica denominata di spooling (poi utilizzata per l’output).

Nasce il primo sistema timesharing, CTSS (Compatible Time Sharing System) sviluppato al M.I.T. e, successivamente, il MULTICS. Da una versione derivata e ridotta, utilizzata su un minicomputer PDP-7, si gettano le basi per la progettazione del sistema UNIX.

**La quarta generazione (dal 1980 a oggi): i Personal Computer**

Digital Research riscrive il CP/M (Control Program for Microcomputers) per utilizzare il processore Zilog Z80.

IBM progetta il primo PC con il DOS (Disk Operating System) e il linguaggio Basic.

Apple progetta Lisa (troppo costoso) e Macintosh, il primo ambiente user-friendly.

Microsoft, influenzata dal successo di Macintosh, realizza Windows e Apple MacOS.

Iniziano ad essere utilizzate alcune versioni di UNIX sui PC: Linux, FreeBSD, ….

**Concetti di base sui sistemi operativi**

**I processi**

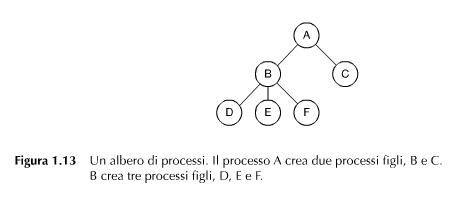
Il processo è un concetto chiave di tutti i SO: un processo è un programma in esecuzione. Ad ogni processo vengono associati:

- un suo spazio di indirizzamento dove sono memorizzati il programma eseguibile, i dati e lo stack;

- un insieme di risorse, tra cui registri (inclusi PC e SP), file aperti, elenco di processi correlati, allarmi in sospeso, …;

- una riga all’interno della tabella dei processi.

I processi possono creare uno o più altri processi, detti processi figli e questi possono ripetere nuovamente l’operazione.



I processi collegati possono cooperare per raggiungere un obiettivo comune ed hanno bisogno di sincronizzarsi attraverso la comunicazione. Questa comunicazione è detta interprocess communication (IPC).

Ad ogni persona autorizzata all’uso di un sistema (utente) l’amministratore del sistema assegna un UID (User Identification). Ogni processo che viene eseguito ha l’UID della persona che l’ha lanciato (un processo figlio ha lo stesso UID del processo che l’ha generato).

Gli utenti possono essere membri di gruppi, identificato da un GID (Group Identification).

**La gestione della memoria**

Ogni computer ha una memoria principale per gestire l’esecuzione dei programmi.

Nei sistemi operativi più semplici c’è sempre solo un programma alla volta in memoria. In quelli più sofisticati possono risiedere in memoria allo stesso tempo più programmi, ma sono necessari meccanismi di protezione per evitare interferenze.

Nei primi sistemi operativi c’era il problema del processo a cui non bastava il suo spazio di indirizzamento. Ora, la virtualizzazione della memoria consente al processo di vedere uno spazio indirizzo unico anche se parte dei dati sono davvero in memoria principali ed altri sul disco (richiamandoli solo all’occorrenza).

**File**

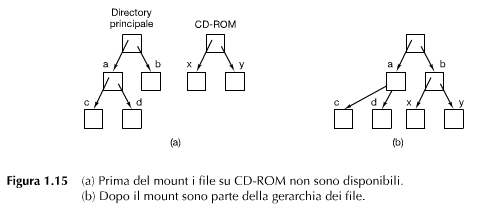
I file sono il modello astratto per raggruppare insiemi di dati nascondendo le peculiarità dell’hardware che li memorizza o, in generale, utilizza (nastri, dischi, dispositivi di I/O,...).

Il file system è l’ambiente che gestisce i file ed è organizzato in modo gerarchico: ciascun nodo intermedio è un contenitore di altri file (directory) o altri contenitori, in modo ricorsivo.

L’elenco delle directory che occorre attraversare dalla radice (root) per raggiungere il file è detto percorso o path del file (il simbolo “/” o “\” è utilizzato come separatore dei nomi).

Prima che un file possa essere letto/scritto è necessario aprirlo e in questo momento sono controllati i permessi, se l’accesso è consentito è restituito un numero intero detto descrittore del file; se l’accesso è proibito, viene restituito un codice di errore.

In UNIX è importante notare il concetto di file system montato: cioè il file system del dispositivo di I/O viene attaccato all’albero principale.



Un altro concetto importante in UNIX sono i file speciali, che permettono di considerare i dispositivi di I/O come file e sono di due tipi:

- file speciali a blocchi: sono utilizzati per modellare dispositivi in grado di indirizzare casualmente blocchi di dati (es. dischi);

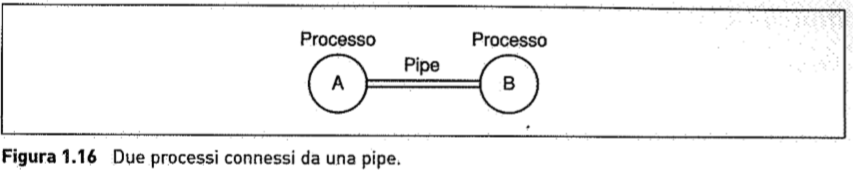
- file speciali a caratteri: sono utilizzati per modellare dispositivi che utilizzano le sequenze (o stream) di caratteri (es. stampanti, tastiere, …).

Si può utilizzare uno pseudofile, chiamato pipe, per far comunicare due processi. Se i processi A e B vogliono comunicare utilizzando una pipe devono:

- inizializzarla in anticipo;

- il processo A, che vuole spedire dati al processo B, scrive sulla pipe come se fosse un file in uscita;

- il processo B può leggere i dati leggendo dalla pipe, come se fosse un file d’ingresso.



**Input/output**

Tutti i computer hanno dispositivi fisici per acquisire input e produrre output.

Esistono molti tipi di input e di output, incluse tastiere, monitor, stampanti e così via. Sta al sistema operativo gestire questi dispositivi. Di conseguenza, ogni sistema operativo ha un sottosistema di I/O per gestire i suoi dispositivi di I/O. Parte del software di I/O è indipendente dai dispositivi, ossia si applica a molti o a tutti i dispositivi allo stesso modo. Altre parti, come i driver dei dispositivi, sono specifici per particolari dispositivi di I/O.

**Sicurezza**

La protezione dei dati è una caratteristica importante dei SO.

Considerando come esempio UNIX, i file sono protetti mediante l’assegnazione di un codice di protezione a 9 bit, costituito da tre campi di 3 bit, uno per il proprietario del file, uno per gli altri membri del gruppo a cui appartiene il proprietario e uno per tutti gli altri; ogni campo ha un bit per l’accesso in lettura, un bit per l’accesso in scrittura, un bit per l’accesso in esecuzione (bit rwx).

Esempio:

*owner other everyone*

r w x r - x - - -

**La shell**

Il SO è il codice che esegue le chiamate di sistema; editor, i compilatori, gli assemblatori, i linker e l’interprete dei comandi (la shell in UNIX) non fanno parte del SO.

La shell (interprete dei comandi) è l’interfaccia principale tra utente e SO.

La GUI (Graphic User Interface) è un’applicazione che sta sopra al SO proprio come l’interprete dei comandi, ma non fa parte del SO.

**Le chiamate di sistema**

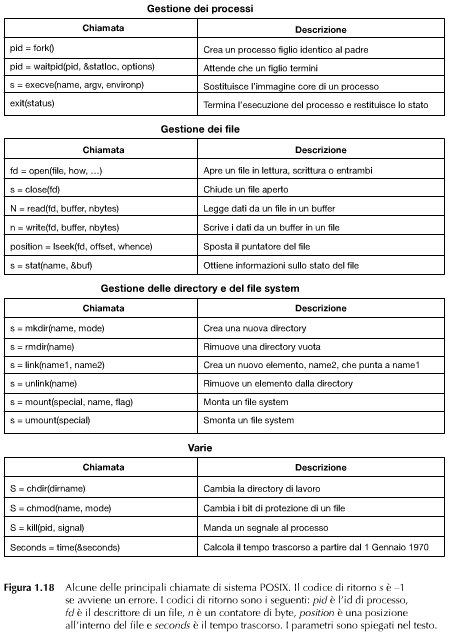
Se un processo utente ha bisogno di leggere un file deve chiamare un servizio di sistema, che restituirà poi il controllo al chiamante una volta eseguito il servizio.

Una chiamata di sistema o system call è una procedura speciale che viene eseguita in modalità kernel, ad esempio:

*count = read(file, buffer, nbyte);*

La chiamata di sistema *read()* restituisce il numero di byte realmente letti nel buffer: non è detto che si riesca a leggere *nbyte*.

Se la chiamata non può essere completata, a causa di un parametro non corretto o di un errore sul disco, *count* viene messo uguale a *-1*.



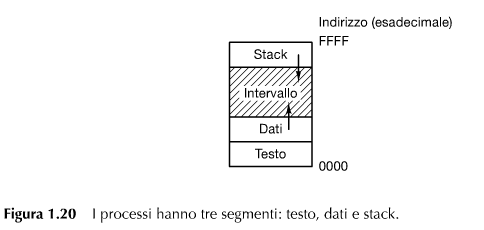
**Chiamate di sistema per la gestione di processi**

Il primo gruppo della Figura 1.18 riguarda la gestione del processo.

*fork* è l’unico modo per creare un nuovo processo in POSIX (lo standard per UNIX). Crea un duplicato esatto del processo originale, includendo tutti i descrittori dei file, registri, ecc…

Dopo la chiamata *fork*, il processo originale e la copia (il padre e il figlio) prendono vie separate.

La memoria dei processi in UNIX è suddivisa in tre segmenti: il segmento testo (cioè il codice programma), il segmento dati (cioè le variabili) e il segmento stack (utilizzato per gestire le attivazioni delle procedure). Il segmento dati cresce verso l’alto e lo stack cresce verso il basso, come mostrato nella Figura 1.20. In mezzo a loro c’è un intervallo di spazio inutilizzato,



**Chiamate di sistema per la gestione dei file**

Molte chiamate di sistema fanno riferimento al file system.

Prima di poter leggere o scrivere un file, è necessario aprirlo con la chiamata di sistema *open*,

*read* e *write* servono per leggere e scrivere il file.

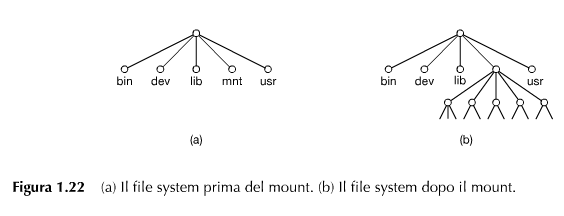
Con *close* è possibile chiudere il file.

**Chiamate di sistema per la gestione delle directory**

Le chiamate *mkdir* e *rmdir* rispettivamente creano directory o rimuovono directory vuote.

La chiamata *link* permette di creare un riferimento ad un file o una directory.

La chiamata *mount* permette di fondere insieme due file system. Quest’ultima è utilizzata quando si vuole montare il file system di un dispositivo (es. CD-ROM) all’interno del file system del sistema: *mount(“/dev/fd0”, “/mnt”, 0);* il primo parametro è il nome di un file speciale a blocchi, il secondo specifica dove montarlo e il terzo la modalità (lettura o scrittura ecc…).

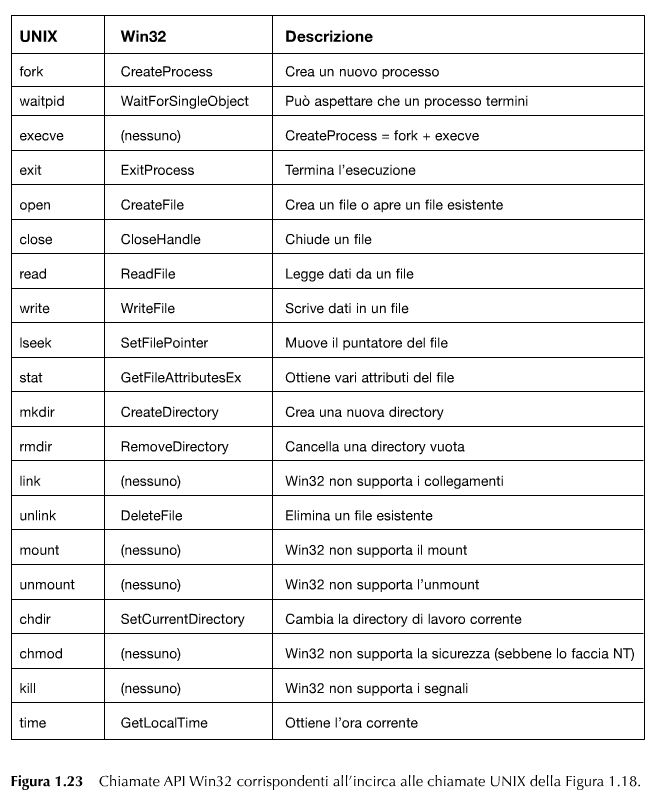


**Le API Win32 di Windows**

Windows e UNIX differiscono per il modello di programmazione: un programma UNIX richiede dei servizi attraverso le chiamate di sistema mentre un programma Windows è guidato dagli eventi. Detto questo, anche Windows ha le chiamate di sistema.

Microsoft ha definito un insieme di procedure chiamate le API Win32 (Application Program Interface) che i programmatori possono utilizzare per ottenere i servizi del SO.

**Confronto tra system call**

****

**Struttura di un sistema operativo**

**Sistemi monolitici**

I sistemi monolitici sono l’organizzazione più comune; in questo approccio l’intero SO viene eseguito come un programma singolo e in modalità kernel.

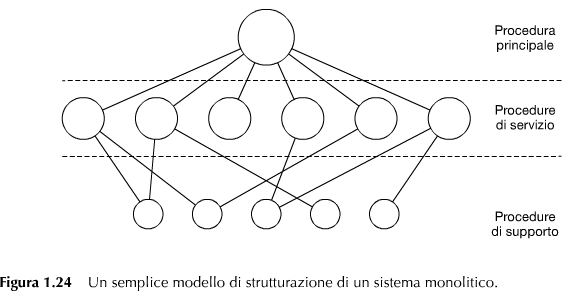
Il SO è un unico grande programma binario eseguibile che contiene all’interno un insieme di procedure collegate tra loro. Ogni procedura nel sistema è libera di richiamarne un’altra, se quest’ultima fornisce alcune elaborazioni necessarie. La presenza di migliaia di procedure che si possono chiamare reciprocamente senza limiti rende il sistema di difficile comprensione.

Questa organizzazione suggerisce la seguente struttura base per il SO:

- un programma principale che richiama la procedura di servizio richiesta;

- un insieme di procedure di servizio che realizzano le chiamate di sistema;

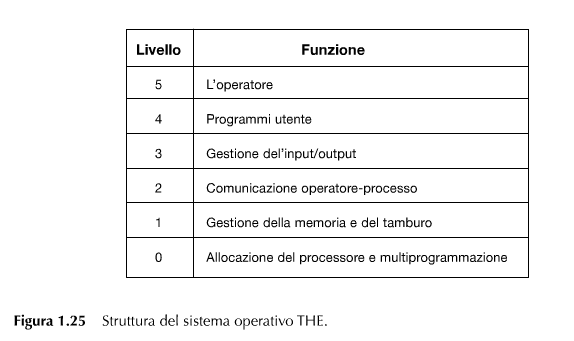
- un insieme di procedure di supporto che aiutano le procedure di servizio.



**Sistemi a livelli**

Una generalizzazione del modello strutturale dei sistemi monolitici è l’organizzazione del SO come una gerarchia di livelli, ognuno costruito sopra quello che sta sotto.

Il primo sistema di questo tipo fu il sistema THE (sviluppato da Dijkstra), ma era ancora un unico eseguibile.



Un altro sistema era il MULTICS ed era descritto come una serie di anelli concentrici, dove quelli più interni erano più privilegiati di quelli esterni.

**Microkernel**

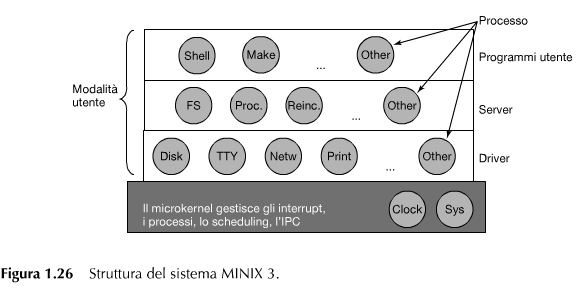
Con l’approccio a strati, i progettisti potevano scegliere dove definire il confine tra kernel e utente.

Inizialmente tutti i livelli erano nel kernel. Ma, in realtà, bisogna collocare il meno possibile in modalità kernel, dato che errori del kernel possono far cadere immediatamente il sistema. Al contrario, si possono predisporre processi utente con minor influenza, in modo che un errore non risulti fatale.

La difettosità di un codice dipende dalla dimensione del modulo, dalla sua età (più è vecchio più è stato testato e quindi più sicuro) e altri fattori. Statisticamente sono presenti 10 bug in 1000 righe di codice.

L’idea di base della struttura microkernel è di incrementare l’affidabilità suddividendo il sistema operativo in piccoli moduli collaudati e di avere un solo modulo eseguito in modalità kernel (il microkernel).

I SO a microkernel sono utilizzati in quelle applicazioni che richiedono alta affidabilità (come gli ambienti real-time, industriali, militari).

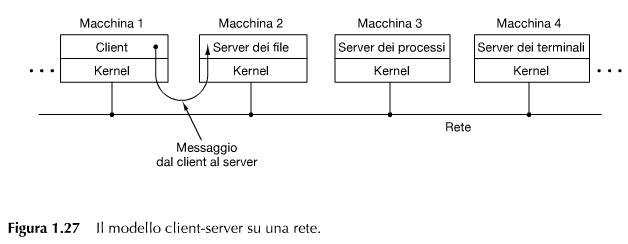


**Modello client-server**

Una leggera variazione rispetto all’idea del microkernel è distinguere due classi di processi, i server, ognuno dei quali mette a disposizione alcuni servizi, e i client, che li utilizzano.

Spesso il livello più basso è un microkernel, ma non è necessario.

La comunicazione tra client e server avviene spesso attraverso lo scambio di messaggi. Per ottenere un servizio, un processo client costruisce un messaggio indicando la propria richiesta e lo invia al servizio appropriato. Il servizio esegue il lavoro e rimanda indietro la risposta.



**Macchine virtuali**

Il sistema di timesharing TSS/360 di IBM (1967) è stato il primo tentativo di virtualizzazione: più utenti potevano lavorare sui terminali collegati alla stessa macchina.

Il sistema z/VM, utilizzati sugli attuali mainframe IBM zSeries, è il discendente diretto del TSS.

**Riscoperta delle macchina virtuali**

Mentre IBM disponeva già da quarant’anni di un prodotto di macchina virtuale, l’idea della virtualizzazione è stata ignorata nel mondo dei PC/server fino a pochi anni fa.

La virtualizzazione si è diffusa anche nel mondo del web hosting.

Oggi anche gli utenti finali possono eseguire due o più sistemi operativi contemporaneamente sulla stessa macchina (VirtualBox, VMware, …).

**La macchina virtuale java**

Quando Sun Microsystem (oggi Oracle) inventò il linguaggio di programmazione Java, creò anche una macchina virtuale (cioè un’architettura di computer) chiamata JVM (Java Virtual Machine).

**Exokernel**

Piuttosto che clonare la macchina reale, come avviene con le macchine virtuali, un’altra strategia è partizionare, assegnando quindi a ogni utente un sottoinsieme delle risorse. Una macchina virtuale potrebbe così prendersi i blocchi del disco da 0 a 1023, la successiva i blocchi da 1024 a 2047 e così via.

Il livello base, che gira in modalità kernel, è un programma chiamato exokernel. Il suo compito è quello di allocare le risorse alle macchine virtuali e di controllare i tentativi di impiego, in modo che ciascuna VM utilizzi esclusivamente le proprie risorse.

Ogni macchina virtuale a livello utente esegue il suo SO personale ed è limitata ad utilizzare le uniche risorse che ha chiesto e che le sono state assegnate.

Il vantaggio dello schema dell’exokernel è che risparmia uno strato di corrispondenza.

Negli altri progetti ogni macchina virtuale pensa di avere il proprio disco, con blocchi che vanno da 0 a un massimo, così il monitor della macchina virtuale deve mantenere delle tabelle per rimappare gli indirizzi del disco (e di tutte le altre risorse). Con l’exokernel, questo rimappaggio non è più necessario, poiché esso necessita solo di tenere traccia di quale sia la macchina virtuale a cui è stata assegnata una certa risorsa.

Inoltre questo metodo tiene separata la multiprogrammazione (che è nell’exokernel) dal codice del sistema operativo utente (che è nello spazio utente).

**Il mondo secondo il C**

I sistemi operativi sono normalmente grandi programmi C.

**Il linguaggio C**

Java è basato su C quindi presenta molte somiglianze ad esso. Però, una caratteristica che C possiede e che Java non ha sono i puntatori espliciti.

Un puntatore è una variabile che punta (cioè contiene l’indirizzo di) a una variabile o a una struttura dati. Si considerino le seguenti istruzioni

*char c1, c2, \*p;*

*c1 = ’x’;*

*p = &c1;*

*c2 = \*p;*

in cui si dichiarano *c1* e *c2* come variabili carattere e *p* come una variabile che punta un carattere. La prima assegnazione salva il codice ASCII per il carattere *’x’* nella variabile *c1*. La seconda assegna l’indirizzo della *c1* alla variabile puntatore *p*. La terza assegna il contenuto della variabile puntata da *p* alla variabile *c2*. Così facendo, alla fine anche *c2* contiene il codice ASCII di *’x’*.

I puntatori sono costrutti estremamente potenti ma allo stesso tempo molto delicati: non avendo controlli è possibile indirizzare la memoria in modo improprio e commettere errori (motivo per il quale un programmatore Java non può utilizzarli, se non implicitamente).

La gestione della memoria del C è statica e allocata in modo esplicito (con la funzione *malloc()*) e rilasciata dal programmatore al termine del lavoro svolto (con la funzione *free()*).

I sistemi operativi sono dei veri e propri sistemi real-time utilizzati per scopi generali, quindi l’esistenza di un collector che entra in azione in modo arbitrario non è ammissibile.

Inoltre il C è un linguaggio che ha un livello concettuale non troppo distante dalla macchina fisica. Tutte queste considerazioni, unite al controllo totale della memoria da parte dei programmatori, rendono il C il linguaggio ideale per scrivere i sistemi operativi.

**File di intestazione**

I file di intestazione (o header) hanno estensione *.h* e contengono dichiarazioni e definizioni usate da uno o più file di codice. I file di intestazione possono anche contenere semplici macro.

Le macro sono dichiarate attraverso la parola chiave *#define*.

Un file sorgente C ha estensione .c e può includere uno o più file intestazione che utilizzano la direttiva *#include*

I file di intestazione possono anche contenere compilazioni condizionate, per esempio:

*#ifdef X86*

*intel\_int\_ack();*

*#endif*

che viene compilata in una chiamata verso la funzione *intel\_int\_ack* se la macro *X86* è definita, altrimenti nulla.

**Il modello di run-time**

1) Il preprocessore assembla insieme i file sorgenti (inclusi quelli di intestazione).

2) Il compilatore produce i file oggetto (estensione *.0*);

3) Il linker collega tutti i file oggetto e le altre librerie creando un file eseguibile binario dipendente dalla macchina e ottimizzato per quella architettura.

