Appunti di Sistemi Operativi

Enzo Mumolo

e-mail address :mumolo@units.it
web address :www.units.it/mumolo

Indice

1 Introduzione **1**
1.1 Struttura di un computer . . . 1
1.1.1 Sistema operativo come macchina astratta . . . 2
1.1.2 Sistema operativo come gestore di risorse . . . 2
1.1.3 Sistema operativo come architettura . . . 2
1.1.4 Classificazione e servizi dei Sistemi Operativi . . . 2
1.2 Strutture dei sistemi operativi . . . 4
1.2.1 Sistemi a strati . . . 4
1.3 Le risorse . . . 9
1.3.1 La protezione delle risorse . . . 10
1.4 Alcune strutture fondamentali . . . 11
1.4.1 Il descrittore dei processi: Process Control Block (PCB) . . . 11
1.5 Lo Spooling . . . 12
1.5.1 I file . . . 13
1.5.2 La memoria di un processo . . . 13

Capitolo 1

Introduzione

1.1 Struttura di un computer

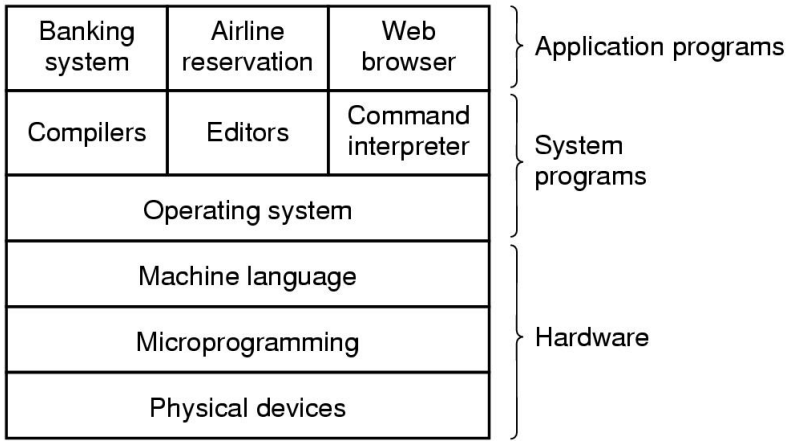

Figura 1.1 Struttura semplificata di un calcolatore

Esistono almeno due modi per poter definire e analizzare un moderno calcolatore: il primo consiste nel guardarlo *dall'alto verso il basso*, il secondo *dal basso verso l'alto*. Più precisamente:

1. *dall'alto verso il basso*, ovvero vederlo come **macchina astratta** (all'utente interessa utilizzare la macchina senza occuparsi dell'implementazione che c'è sotto); da questo punto di vista viene nascosta la complessità dei livelli sottostanti.

2. *dal basso verso l'alto*, ovvero come **gestore di risorse**: ci si pone dal punto di vista dell'hardware, ovvero si guarda a come possono essere gestiti i vari dispositivi.

Fino ad alcuni anni fa dare una definizione di sistema operativo era tutto sommato facile: era quel particolare strato di software che gestiva l'hardware. Al giorno d'oggi invece è difficile tracciare un solco netto tra HW e SW; si pensi ad esempio al **firmware** (o microprogramma): esso è un microcodice solitamente allocato in una ROM che serve ad interpretare un insieme di istruzioni ed eseguirle attraverso una sequenza di passi elementari (l'insieme delle istruzioni interpretate definisce il **linguaggio macchina**): ora questo strato può considerarsi parte integrante dell'hardware. Pertanto oggi si è più propensi a definire un sistema operativo come *un sistema che offre servizi* di macchina astratta o di gestione delle risorse.

Esistono moltri altri punti di vista dai quali poter analizzare un computer: nel proseguio del corso verrà considerato anche una terza prospettiva: l'**architettura**.

Vediamo ora in dettaglio cosa vuol dire studiare un sistema da questi tre diversi punti di vista.

1.1.1 Sistema operativo come macchina astratta

Significa sostanzialmente studiare il modo in cui l'utente si approccia alla macchina: studiare la *shell* e i servizi che questa offre, ad esempio:

- permette di spostare, copiare e rinominare file di varia natura;

√ permette (e questo è un aspetto molto importante) di *creare applicazioni*: ovvero il sistema operativo crea, attraverso la macchina astratta, un ambiente per creare, eseguire e terminare programmi (o applicazioni). Qui dentro ci sono ad esempio i linguaggi compilati e quelli interpretati (questi ultimi sono i più numerosi: si pensi ad esempio ai linguaggi di shell, l'HTML, il perl...)

1.1.2 Sistema operativo come gestore di risorse

Studiare il sistema come gestore di risorse può significare migliorare le prestazioni della macchina:

- Facendo lo *scheduling* della CPU: quest'ultima infatti deve essere continuamente "suddivisa" tra i vari lavori...
- La memoria dev'essere sempre utilizzata al meglio: bisogna evitare il più possibile la frammentazione, bisogna garantire una buona allocazione dei dati...
- Bisogna gestire bene l'I\O in termini di tempi d'accesso e di affidabilità...
- Bisogna gestire bene le risorse (e si badi che queste non sono solo HW, ma anche SW: si pensi ad esempio ai processi)...

1.1.3 Sistema operativo come architettura

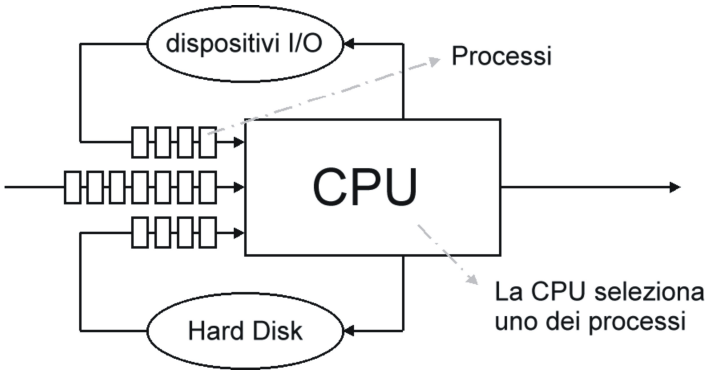

Figura 1.2 Possibile rete di code d'attesa.

Dal punto di vista del progettista (ovvero colui che deve *dimensionare* la macchina) il sistema operativo può essere visto come una rete di **code d'attesa**: la figura a destra mostra una possibile rappresentazione (estremamente semplificata) delle code che un processore deve essere in grado di gestire in maniera efficiente.

1.1.4 Classificazione e servizi dei Sistemi Operativi

Un Sistema operativo puó essere classificato n base al suo utilizzo da parte dell'utente:

- Utilizzo di tipo Generale (General Purpose). Lo sono i sistemi Operativi noti per l'utilizzo da parte dell'utente generico, come Windows, Linux o Mac OS. In questo caso non é possibile prevedere quale tipo di applicazione verrá eseguita.
- Utilizzo di tipo Batch. In questo caso il sistema operativo non interagisce direttamente con l'utente, ma esegue un certo insieme di programmi utente assegnati leggendo i dati d'ingresso e scrivendo i risultati sulla memoria di massa.
- Utilizzo di tipo Interattivo. Il sistema attende i comandi dell'utente e visualizza i risultati della esecuzione.
- Utilizzo dedicato. Il sistema é progettato per eseguire solo determinate applicazioni. Puó essere il controllo di un'auto (computer di bordo), una memoria di massa di tipo disco fisso, un sistema multimediale di lettura di DVD, un cellulare.
- Sistemi Transazionali. Sistemi progettati per gestire e comunicare grandi quantitá di dati.

Ogni sistema operativo nasce con delle particolari modalitá di utilizzo che hanno delle influenze molto pesanti sul tipo di struttura di cui sará dotato. **É fondamentale osservare che non esiste un sistema operativo che possa soddisfare tutte le diverse modalitá di utilizzo perché sono tipicamente in contrasto.**

I servizi fondamentali di un Sistema Operativo sono i seguenti:

- Creazione, esecuzione e terminazione dei programmi utente. Questi servizi legati ai programmi utente sono i piú importanti; non sono validi ovviamente per i sistemi dedicati.
- Operazioni di Ingresso Uscita
- Manipolazione di file
- Gestione degli errori di tipo hardware, di I/O, generati dall'utente
- Allocazione delle risorse: sono elementi hardware come CPU, memoria file, canali di I/O o elementi software come strutture dati o codice rientrante.
- Contabilitá e statistiche sull'uso delle risorse. Queste misure sono importanti sia per sapere il costo dell'uso delle risorse, ma anche per effettuare ottimizzazioni sulle prestazioni del sistema.
- Protezione e sicurezza. Le strutture del sistema operativo devono essere robuste rispetto all'esecuzione dei processi. Stiamo parlando ovviamente del caso in cui non ci sia malizia nell'esecuzione dei processi: ad esempio se viene fatto girare un programma bisogna assolutamente evitare che questo (per errori di programmazione) possa in alcun modo modificare alcune strutture del sistema operativo. Diverso è il caso della **sicurezza**, caso in cui bisogna difendere il sistema operativo da intrusioni esterne con scopi maliziosi.

1.2 Strutture dei sistemi operativi

Un modo per classificare i sistemi operativi è quello di farlo in base alla loro struttura interna.

1.2.1 Sistemi a strati

Il primo modo in cui si è iniziato a concepire un sistema operativo è stato mediante delle strutture a livello.

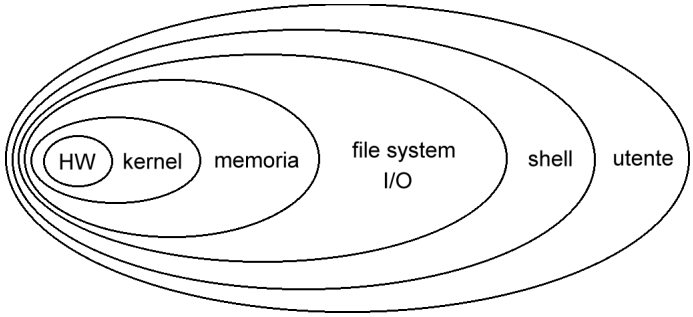

Figura 1.3 Struttura onion skin.

Questo tipo di struttura è chiamata *onion skin* (a buccia di cipolla): in questi modelli il sistema operativo è organizzato come una gerarchia a strati, ognuno costruito sopra il precedente; compito di ogni strato è quello di fornire servizi allo strato superiore (per esempio la memoria utilizza servizi che chiede al kernel).

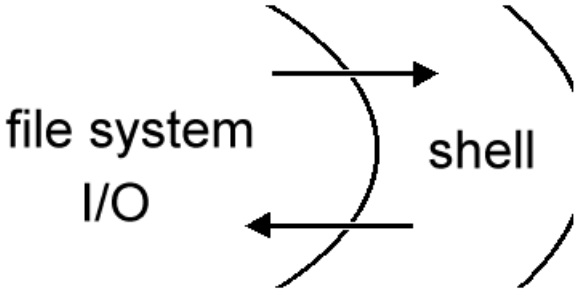

Figura 1.4 Interazione ai confini del nucleo

Ogni strato interagisce col sottostante attraverso delle opportune interfacce (figura 1.4); questo avviene ad ogni livello: se per esempio la shell ha bisogno del nucleo deve passare diversi "confini". Questo comporta che la struttura risulti molto pesante al fine delle prestazioni (infatti l'attraversamento dei vari confini si traduce in elevati tempi di risposta).

In precedenza era stato sottolineato che il primo servizio di un sistema operativo è quello di creare, eseguire e terminare processi: questo significa che il sistema operativo deve offrire un ambiente per l'esecuzione delle applicazioni che dev'essere il più efficiente possibile: pertanto risulta immediato capire che le massime attenzioni saranno rivolte affinché l'esecuzione sia più veloce possibile: tutto quello che si contrappone a questa esigenza è chiamato **sovraccarico**. Ogni sistema operativo possiede un sovraccarico che deve tenere il più basso possibile[1]. In una struttura del tipo *onion skin* il sovraccarico è massimizzato. Ovviamente un sistema operativo con alto sovraccarico ha una bassa **efficienza**.

[1] Ed è anche il motivo per cui si cercano sempre algoritmi il più semplici possibile.

Chiariti questi aspetti è naturale osservare come una struttura del tipo onion skin, benché sia molto carente dal punto di vista delle presazioni, risulti la più robusta in assoluto.

Esempi di Sistemi a strati

'61 → ATLAS Sistema operativo scritto in assembler che (assieme al Multics) ha portato importantissime innovazioni come lo *spooling*, l'*elaborazione batch* e le prime idee di *memoria virtuale*.

'64 → MULTICS Sistema operativo anch'esso scritto in assembler progettato da Bell Labs, General Electric e MIT con lo scopo di creare una macchina in grado di supportare contemporaneamente centinaia di utenti in *timesharing*. La struttura era quella dell'onion skin: quando una procedura del livello esterno dewsiderava chiamare una procedura del livello interno si serviva dell'istruzione `TRAP` i cui parametri venivano accuratamente controllati prima che la chiamata potesse procedere.

'65 → THE Sistema operativo progettato in Olanda da Dijkstra e dai suoi studenti. Fu il primo a risolvere problemi legati alla concorrenza (ovvero legati all'uso di risorse condivise) e a introdurre le *strutture semaforiche*. Anche il THE possedeva uno schema a strati che però era solo una convenzione progettuale in quanto tutte le parti del sistema venivano alla fine collegate in un unico programma oggetto.

'66 → OS/360 Con questo progetto l'IBM voleva introdurre un sistema operativo che fosse in qualche modo "IL" sistema operativo in assoluto: ovvero doveva essere in grado di fornire tutti i servizi possibili. Il progetto del OS/360 fallí miseramente. Il risultato infatti fu un nucleo enorme e, come tutti gli *oggetti* di dimensioni eccessive, era un oggetto difficilmente sviluppabile, controllabile e gestibile e dalla manutenzione ardua. Infatti modificare in un punto un programma di decine di pagine tipicamente vuol dire dover introdurre molte altre modifiche[2].

Ritornando al nucleo dell'OS/360, questo aveva un codice assembler di milioni di righe sul quale lavoravano migliaia di programmatori: introdurre anche solo una modifica significava comunicarlo agli altri programmatori, i quali a loro volta dovevano mettere mano al codice per adeguarlo alle modifiche e via dicendo. Ogni nuova release sistemava alcuni bug introducendone di nuovi, così che il numero totale di bag rimase pressoché inalterato. Il progetto OS360 costó alla IBM diverse decine di milioni di dollari.

Possiamo dire che la prima funzione di un nucleo è quella di gestire gli interrupt; pertanto quello del nucleo dev'essere il codice il più affidabile dell'intero sistema operativo. Un'altra importantissima funzione è la gestione dei processi (ovvero creazione-terminazione-esecuzione, schedulazione e IPC). Il nucleo di un sistema batch può essere quindi molto diverso dal nucleo di un sistema interattivo

'78 → VMS Sistema operativo onion skin costruito dalla Digital che per moltissimi anni ha rappresentato una tappa obbligata per tutti coloro che avevano bisogno di grande affidabilità, caratteristica dovuta ad un nucleo estremamente protetto. Col tempo i sistemi VMS sono stati soppiantati sai sistemi Unix per motivi di efficienza.

Alla luce di tutto ciò si può facilmente capire come il fatto di voler costruire un sistema unico per poter essere utilizzato in tutti i casi è (ed è stato dimostrato) un problema eccessivamente complesso.

[2]Questo è uno dei problemi che hanno fatto nascere la programmazione ad oggetti.

Macchine virtuali

'66 → VM/370 In parallelo al progetto OS/360 all'IBM un gruppo di programmatori, sempre con l'obiettivo di far nascere un sistema operativo che fosse il più generico possibile (quella era l'idea guida dell'IBM in quegli anni), sviluppava il principio delle **macchine virtuali**. I programmatori presero coscienza del fatto che esistevano diversi sistemi operativi sviluppati in diversi ambiti, e anziché progettare un sistema mostruosamente complesso che fosse in grado di soddisfare tutte le esigenze si concentrarono sullo sviluppo di un software che permettesse l'utilizzo di diversi sistemi operativi *contemporaneamente* sulla stessa macchina, dando così vita al sistema VM/370.

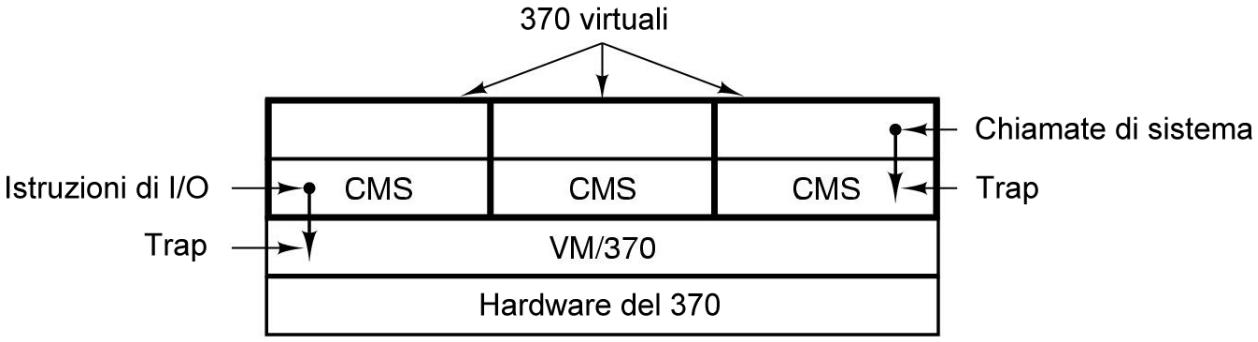

Figura 1.5 La struttura del VM/370.

Questo strato di software gira direttamente sull'hardware e realizza la multiprogrammazione fornendo al livello superiore non una, bensì svariate macchine virtuali; diversamente da tutti gli altri sistemi operativi queste macchine sono delle repliche virtuali esatte dell'hardware sottostante, e come tali ciascuna di esse può eseguire un qualsiasi sistema operativo in grado di girare direttamente sull'hardware.

Modelli client-server

Nei sistemi operativi moderni c'è sempre più la tendenza a spostare ancora più il codice verso i livelli superiori, e rimuovere quanto possibile dal sistema operativo, lasciando un nucleo minimale. L'approccio consueto consiste nell'implementare la maggior parte delle funzioni del sistema operativo all'interno di processi utente. Per richiedere un servizio, come la lettura di un blocco di un file, un processo utente (ora chiamato **processo client**) invia la richiesta a un **processo server** che effettua il lavoro e invia indietro la risposta.

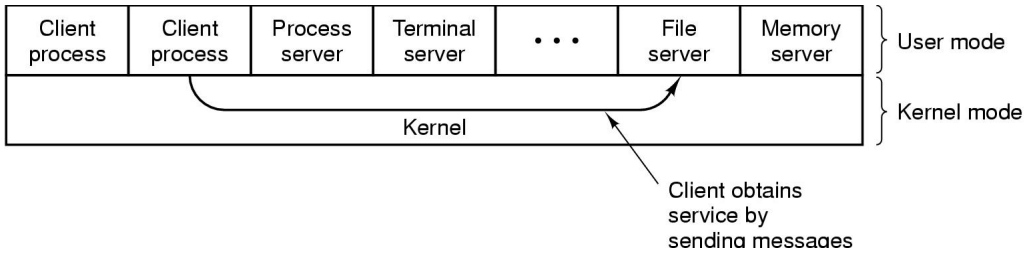

Figura 1.6 Modello client-server.

In questo modello (mostrato in figura **??**) il nucleo deve solo gestire la comunicazione tra processi client e server. Dividendo inoltre il sistema operativo in parti, ognuna delle quali si occupa di un solo aspetto del sistema, ogni parte diviene piccola e maneggevole.

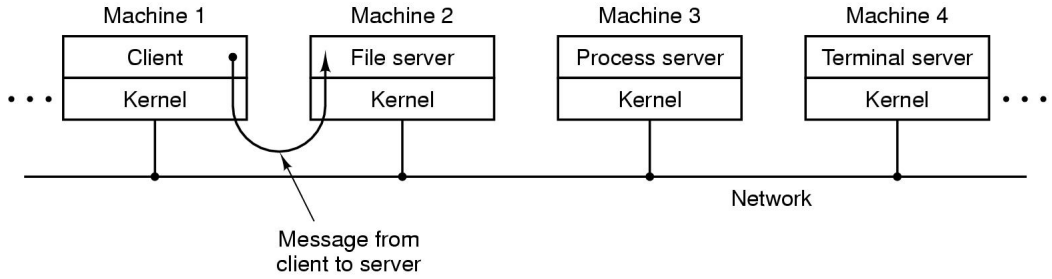

Figura 1.7 Modello client-server in un sistema distribuito.

Un altro vantaggio del modello client-server è la sua adattabilità all'utilizzo in sistemi distribuiti (figura **??**). Se un client comunica con un server inviandogli messaggi, il client non ha bisogno di sapere se il suo messaggio viene gestito localmente, sulla sua macchina, o se esso sia inviato attraverso la rete a un processo server su una macchina remota. Questi processi inoltre non necessitano di macchine dedicate: possono girare tranquillamente (e di fatto lo fanno) su altri sistemi operativi in modo concorrente rispetto ad altri processi.

Unica differenza nei siostemi distribuiti è che in questo caso il nucleo non può occuparsi solo il traffico dei messaggi ma deve saper gestire la comunicazione con altri nodi della rete.

Sistemi monolitici

La struttura dei sistemi monolitici consiste (sembra un gioco di parole) nell'assenza di ogni strutturazione. Il sistema operativo è scritto come un insieme di procedure, ognuna delle quali all'interno del sistema ha una ben definita interfaccia (in termini di parametri e risultati) e può indifferentemente chiamare tutte le altre in qualsiasi momento ne abbia bisogno.

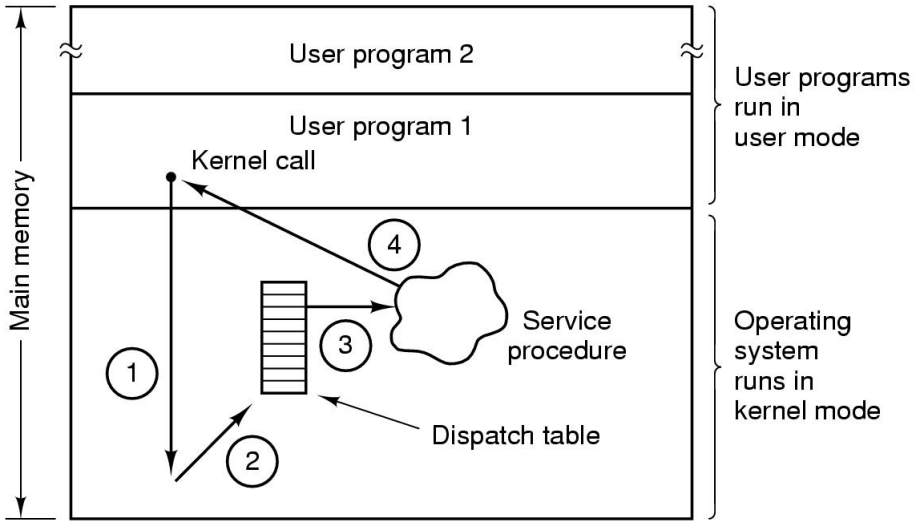

Figura 1.8 Struttura monolitica di un Sistema Operativo

Usando questo approccio, per costruire il programma oggetto del sistema operativo, si devono anzitutto compilare tutte le singole procedure, poi collegarle assieme in un unico file oggetto utilizzando il linker di sistema; di fatto non c'è nessuna astrazione: ogni procedura è sonovisibile a tutte le altre.

Anche nei sistemi monolitici, comunque, è possibile è possibile avere un certo grado di strutturazione. I servizi (chiamate di sistema) forniti dal sistema operativo vengono invocati immettendo

i parametri in locazioni ben definite, come i registri o lo stack, ed eseguendo quindi una speciale istruzione di trap denominata **kernel call**.

Come mostrato in figura 1.8, questa istruzione cambia la modalità della macchina da user mode a kernel mode trasferendo il controllo al sistema operativo (1). Il sistema esamina quindi i parametri della chiamata per determinare quale chiamata di sistema debba essere eseguita (2). In seguito il sistema operativo accede a una tabella (tabella di *dispatch*) che contiene nel k-esimo elemento un puntatore alla procedura che effettua quanto richiesto dalla k-esima chiamata di sistema: questa operazione identifica la procedura di servizio che quindi viene chiamata (3). Quando l'operazione termina, e la chiamata di sistema ha fine, il controllo viene restituito al programma utente (4) in modo da consentirgli di proseguire con l'istruzione immediatamente successiva alla chiamata di sistema.

Un caso particolare di un sistema monolitico: il kernel di Unix

Facciamo ora una rapida panoramica del kernel di Unix. In genere tutte le versioni commerciali di Unix hanno un kernel monolitico[3]. Nella figura sotto si può vedere una rappresentazione (semplificata) a blocchi del kernel.

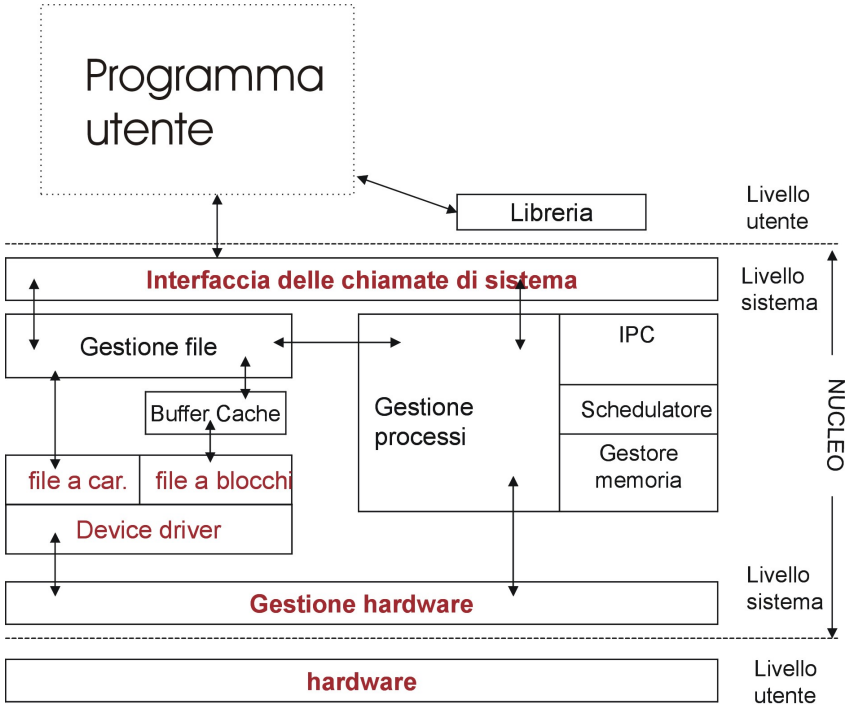

Figura 1.9 Schema a blocchi del kernel di Unix

Quello che un utente vede del kernel è la **system call interface**: quest'interfaccia traduce le richieste esterne (ovvero le invocazioni delle trap) in opportune procedure di sistema (interne). La figura suddivide l'insieme delle chiamate di sistema in quelle che interagiscono con il sottosistema di gestione dei file e quelle che interagiscono con il sottosistema di gestione dei processi.

Gestione dei file: il sottosistema di gestione dei file gestisce i file, alloca lo spazio per i file, amministra lo spazio libero, controlla l'accesso ai file e ricerca dati per l'utente. I processi interagiscono con questo sottosistema per mezzo di un insieme di chiamate di sistema, come `open`, `close`, `read`, `write`, `chmod` ecc...

[3] **Linux** - ovviamente free - ha un kernel **modulare**: ogni funzione è realizzata con moduli a collegamento dinamico che possono essere aggiunti o tolti anche durante il funzionamento.

Gestione dei processi: il sottosistema di gestione dei processi è il responsabile della sincronizzazione dei processi, della comunicazione tra i processi, della gestione della memoria e dello scheduling dei processi. Alcune chiamate di sistema per il controllo dei processi sono: `fork`, `exec`, `exit`, `wait`, `socket` ecc...
In particolare il modulo per la **gestione della memoria** si occupa di giostrare l'allocazione dei processi tra la memoria centrale e le memorie secondarie in modo da permettere a tutti le migliori condizioni di esecuzione.
Infine il modulo di **scheduling** si occupa di assegnare la CPU ai vari processi, scegliendo tra di loro quello di priorità più alta.

È importante osservare come in realtà gran parte dei comandi di Unix sono realizzati a livello utente per mezzo di applicazioni che usano le chiamate di sistema: di conseguenza *non* fanno parte del sistema operativo. Il loro punto di forza è che sono dei programmi applicativi che realizzano degli algoritmi *molto* efficienti[4]. L'aspetto interessante è che in molti casi le applicazioni utente che diventano un'estensione dei comandi del sistema operativo sono realizzate con degli script, quindi con linguaggi *interpretati* come il perl o gli script di shell. È un fatto noto che in genere i sistemisti non programmino in C, bensì in shell o in perl: infatti, nonostante il C sia un linguaggio molto potente che viene poi *compilato* dando luogo ad eseguibili molto efficienti, ci sono tre grossi vantaggi a favore dei linguaggi interpretati in Unix:

1. la programmazione in shell è molto potente (quello che si fa con 10 righe in C può essere tranquillamente fatto con una sola riga in shell);

2. un programma scritto in shell utilizza tutti gli altri programmi di Unix che, come già detto, sono estremamente efficienti (e questo compensa l'interpretazione);

3. poiché gli script di shell utilizzano di solito sia gli applicativi di Unix sia altri script, tutto questo consiste in cercare e caricare in memoria dei file: Unix è etremamente efficiente nell'utilizzo dei file (infatti è un sistema operativo che utilizza molto il concetto di *caching*, per cui lavorare con i file vuol dire lavorare in memoria).

1.3 Le risorse

Ci sono diversi tipi di risorse (queste infatti non sono solo hardware, bensì anche software), ed in particolare ci sono diversi modi di poter classificarle. Si possono suddividere in:

√ **Condivisibili**

√ **Non condivisibili**

Le risorse condivisibili sono frammenti di codice rientrante, ovvero puro codice: non contengono dati ma solo istruzioni; infatti non appena nel codice ci sono dei dati il codice non può essere più condiviso in quanto i dati sono peculiari di ogni singola utilizzazione. Le risorse non condivisibili perciò sono tipicamente una struttura dati. Quello che può essere invece condiviso ad esempio è un file (infatti tipicamente più processi possono accedere allo stesso file per svolgere determinate operazioni).

Il fatto di avere del codice non condivisibile crea alcuni problemi nella contesa dell'utilizzo di tali risorse richiedendo meccanismi di sincronizzazione.

Un'altra classificazione può essere fatta discernendo tra risorse

√ **Riutilizzabili**

[4] Basti pensare che su alcuni comandi sono state dedicate intere pubblicazioni...

√ **Non riutilizzabili**

Le risorse riutilizzabili sono quelle che non si consumano durante l'utilizzo, come ad esempio l'unità centrale; esempi invece di risorse non riutilizzabili possono essere i file o le strutture dati che possono subire modifiche quando vengono impiegate.

Un'ulteriore classificazione può essere fatta distinguendo le risorse in

√ **Sottraibili**

√ **Non sottraibili**

La sottraibilità è in un certo senso sinonimo di *preemption*: la risorsa può essere interrotta in qualunque istante: caso tipico è la CPU che può venir sottratta ad un processo e assegnata ad un altro processo. Da notare che i moderni sistemi operativi sono tutti preemptive (si pensi ad esempio al multitasking).

Alcune caratteristiche delle risorse stanno però anche alla base di alcuni problemi che devono essere assolutamente risolti. Le risorse infatti hanno anche delle anomalie, ovvero dei comportamenti che fanno sì che l'utilizzo del'unità centrale venga a essere diminuita. Queste anomalie sono di due tipi:

Strozzature: situazione in cui una risorsa viene utilizzata molto più delle altre (per cui non c'è più equilibrio); il sistema operativo può essere bloccato sull'utilizzo di una determinata risorsa.

Anelli di reazione: situazione in cui l'utilizzo errato di una risorsa provoca per reazione un errato utilizzo di altre risorse.

Esempio

Per spiegare questi concetti prendiamo l'esempio più comprensibile. Sappiamo che i processi devono essere allocati nella memoria centrale per poter essere eseguiti: quindi c'è una determinata quantità di memoria disponibile per questa operazione. In multiprogrammazione la memoria è condivisa: questo vuol dire che più processi sono simultaneamente in esecuzione, meno memoria si ha a disposizione per ogni singolo processo (la memoria è limitata...). Il principio che sta alla base del funzionamento della multiprogrammazione è il **pricipio di località**: in ogni istante ogni processo ha bisogno in realtà solo di una piccola quantità di memoria. Ora però succede che se in processo richiede altre informazioni bisogna accedere al disco per prelevare il codice (o i dati) che servono in quel momento e caricarli in memoria sovrascrivendo il codice (o i dati) non più necessari. L'anello di reazione può innescarsi nel momento in cui si aumenta il livello di multiprogrammazione: aumenta il numero di processi, quindi la memoria allocata per ogni processo diminuisce, il che implica la necessità di reperire informazioni dal disco con una frequenza maggiore rischiando una strozzatura del disco... a questo punto il sistema operativo si trova nelle condizioni di dover gestire pesantemente il disco con la conseguenza di una diminuzione delle prestazioni.

1.3.1 La protezione delle risorse

Un argomento importante è la protezione delle risorse: il sistema operativo deve avere sempre il controllo delle sue risorse; in questa sezione faremo riferimento ad una risorsa in particolare.

Protezione dell'unità centrale

Il caso più noto di perdita di controllo della CPU si ha quando questa rimante vincolata in un loop infinito.

Il controllo dell'unità centrale avviene utilizzando un meccanismo di interrupt: il **clock**. Questo è un *interrupt hardware* gestito da alcuni circuiti integrati che, a intervalli programmati, interrompe la CPU che per un determinato lasso di tempo (tempo di **context switch**) viene dedicata *solo* al sistema operativo. Facciamo un esempio: supponiamo che la CPU rimanga bloccata a tempo infinito da parte di un processo utente; il meccanismo del clock permette al sistema operativo di rilevare un comando di sistema (ad esempio un `kill -9`) da parte dell'utente: senza questo meccanismo il sistema diventerebbe inutilizzabile.

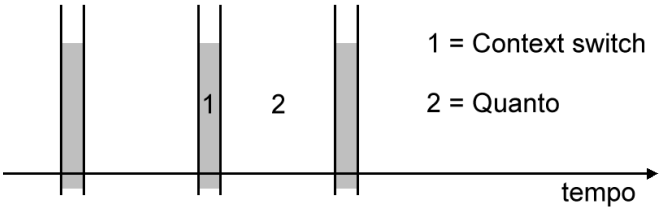

Figura 1.10 Rappresentazione del clock.

Questo meccanismo è fondamentale anche per poter gestire la programmazione concorrente, ambiente nel quale si ha la possibilità di *switchare* tra i processi utente presenti nel sistema.

Naturalmente è importante che il context switch sia il più piccolo possibile per mantenere il sovraccarico del sistema il più piccolo possibile: il principale compito di un sistema operativo è infatti quello di eseguire applicazioni... Per diminuire questo tempo esistono diverse tecniche, alcune delle quali sono legate alla programmazione a thread.

1.4 Alcune strutture fondamentali

1.4.1 Il descrittore dei processi: Process Control Block (PCB)

Per descrivere un processo in un dato istante si usano delle strutture chiamate **PCB Process Control Block**: la sua implementazione dipende dal particolare sistema operativo e può essere realizzato in molti modi; in generale comunque contiene dei puntatori che puntano alle informazioni necessarie.

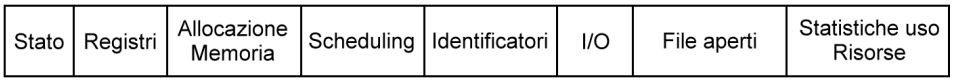

Figura 1.11 La struttura del PCB.

In particolare questo descrittore di processi sotto Unix si chiama **Process Structure** e viene realizzato tramite un array; l'insieme di tutti gli array che descrivono l'insieme dei processi presenti nel sistema è organizzato in un vettore chiamato **Process Table**.

Questa implementazione permette di identificare il processo con il numero dell'array che descrive il processo (PCB) dentro la Process Table: nasce così il PID.

I sistemi operativi che supportano il concetto di processo devono fornire un modo per creare tutti i processi di cui si ha bisogno: in Unix i processi sono creati con la chiamata di sistema `fork()` (presente dentro al kernel) che crea una copia identica del processo chiamante. In questo contesto il programma chiamante si chiama **padre** e quello creato si chiama **figlio**. Un processo figlio può a sua volta eseguire la `fork()` in modo da consentire la creazione di un intero albero di processi.

1.5 Lo Spooling

Lo Spooling é un meccanismo di gestione simultanea dei dispositivi di I/O. I dati d I/O vengono quindi letti o scritti dal/sul disco, evitando cosí di allocare una periferica di I/O. Il termine SPOOL nasce come acronimo per Simultaneous Peripheral Operation On Line. Il meccanismo é in principio é semplice. Prendendo come esempio lo spooling della stampante. In un ambiente multiprogrammato il processo che vuole stampare dovrebbe allocarsi la stampante e stampare il suo output. Tuttavia se il controllo viene passato ad un altro processo che vuole stampare anch'esso, la stampa del secondo processo deve essere bloccata per non avere delle stampe sovrapposte, risultando in un ritardo inaccetabile. Mediante lo Spooling questa allocazione della stampante viene evitata perché i processi non stampano direttamente sulla stampante ma su un dispositivo di memoria di massa veloce e capace, come il disco. La stampa viene differita e gestita da un processo che monitorizza lo stato della stampante e dei processi che hanno richiesto una stampa. Questo viene fatto mediante una tabella chiamata **Spooling Directory** dove il processo che vuole stampare scrive il nome del file da stampare.

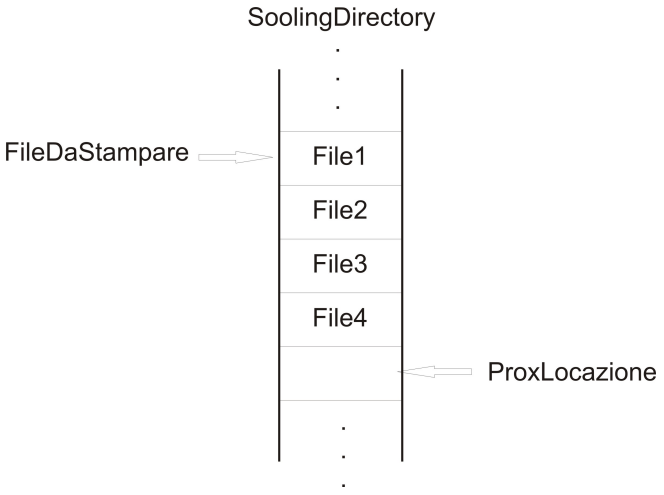

Figura 1.12 Spooling Directory

Il file da stampare contiene i dati generati dal processo; quindi il processo non stampa sulla stampante ma su un file. Quando il processo di stampa é terminato, il nome del file viene scritto nella Spooling directory, alla locazione corrispondente ad un puntatore, chiamato ad esempio **ProxLocazione**. Subito dopo la scrittura del nome del file da stampare il puntatore viene incrementato in modo tale che punta sempre alla prossima locazione libera della tabella. Un processo in background, chiamato per esempio **FileDaStampare** esamina continuamente la tabella, inizia la stampa del file puntato dal puntatore FileDaStampare se la stampa precedente é terminata e incrementa il valore del puntatore quando una stampa é terminata. I due puntatori devono essere condivisi tra tutti i processi.

Da notare che la stampa non púo iniziare prima che il file non sia stato completamente scritto, perché altrimenti si avrebbe lo stesso problema della sovrapposizione delle stampe.

Naturalmente lo spooling púo essere usato anche in molte altre situazioni, nelle quali invece di scrivere direttamente sulla periferica si scrive sul disco e si demanda la scrittura ad un processo **demone**. Questo é il caso di alcuni trasferimenti di file sulla rete, come quelli che erano usati nella trasmissione di mail tramite USENET. In questo caso si parlava di **Network Spooling Demon**.

I sistemi di Spooling possono peró introdurre molti problemi legati sia al **context switch** che alla saturazione dello spazio di disco. Supponiamo infatti che un processo che vuole stampare un file abbia scritto il nome del file nella SpoolingDirectory e mentre sta incrementando il puntatore

ProxLocazione ma prima di averlo incrementato avviene un cambio di contexto. Il prossimo processo che esegue vuole stampare anch'esso un file, prende il puntatore **ProxLocazione** per scrivere il nome del file da stampare ma essendo il puntatore non ancora incrementato, viene sovrascritto il nome precedente. La stampa del primo processo viene cosí persa.

Altri problemi possono accadere se si raggiunge la saturazione dello spazio disco mentre un processo sta scrivendo i dati che vuole stampare su un file. Essendo finito lo spazio disco, il processo blocca lascrittura e aspetta che si liberi lo spazio. Lo spazio si potrebbe liberare se si iniziasse la stampa del file, ma la stampa non puó cominciare perché il file non é completamente scritto. Il processo e il demone di stampa sono bloccati in stallo.

1.5.1 I file

Tutte le applicazioni informatiche hanno bisogno di memorizzare e recuperare informazioni. Ci sono tre requisiti essenziali per la memorizzazione delle informazioni a lungo termine:

1. Si deve poter memorizzare una grande quantità di informazioni; un processo in esecuzione infatti può salvarne una quantità limitata all'interno del suo spazio di indirizzamento: per alcune applicazioni questo spazio è assolutamente insufficiente (si pensi ad applicazioni data base).

2. Le informazioni devono sopravvivere alla terminazione del processo che le usa.

3. Più processi devono poter accedere alle informazioni in modo concorrente; infatti se le informazioni sono memorizzate nello spazio di indirizzamento di un singolo processo solo quest'ultimo può accedervi.

La soluzione più comune a tutti questi problemi consiste nel memorizzare le informazioni in unità chiamate **file**, contenute nei dischi o in altri mezzi fisici. I processi possono leggere i file o scriverne di nuovi se ne hanno bisogno. Le informazioni memorizzate nei file devono essere **persistenti**, ovvero non devono essere influenzate dalla creazione e terminazione dei procassi; un file deve scomparire solo quando il suo proprietario lo rimuove esplicitamente.

I file sono gestiti dal sistema operativo. Il modo in cui sono strutturati e denominati, il modo in cui vengono definiti gli accessi, usati, protetti e realizzati è un argomento di grande importanza nella progettazione di un sistema operativo. La parte del sistema operativo che ha a che fare con i file è complessivamente nota come **file system**.

1.5.2 La memoria di un processo

La memoria assegnata a un processo viene divisa in tre parti: dati, stack e codice. Queste tre parti sono descritte all'interno del PCB nel campo assegnato all'allocazione della memoria.

Un aspetto importante è che gli indirizzi utilizzati sono tutti **indirizzi virtuali**: ogni processo alloca la memoria necessaria tra 0 e max; tutto ciò serve a svincolare il più possibile la programmazione dai dettagli implementativi dell'hardware.

Per fare in modo che l'unità centrale possa accedere alla memoria fisica si utilizzano alcune tabelle chiamate **Page Map Table** (dette anche tabelle delle regioni attive): queste sono delle tabelle gestite direttamente dal sistema operativo che servono a mappare gli indirizzi virtuali in indirizzi fisici direttamente accessibili.

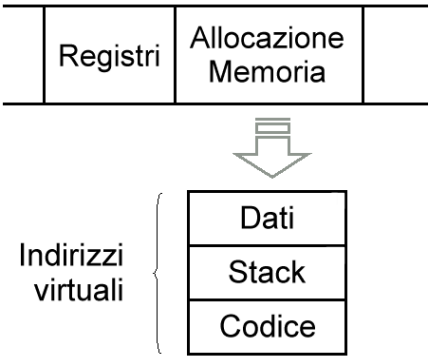

Figura 1.13 Indirizzamento della memoria di un processo.