Sistemi Operativi Teoria

Parte 1 - Panoramica

Appunti a cura di Liccardo Giuseppe
Università degli Studi di Napoli "Parthenope"

Indice – Parte 1

CAPITOLO 1. Introduzione

Un Sistema Operativo controlla l'uso delle risorse di un sistema di un computer come CPU, memoria e dispositivi di I/O per soddisfare le richieste di elaborazione dei suoi utenti.

Gli obiettivi principali di un SO sono:

- la **convenienza per l'utente**; infatti gli utenti si aspettano qualità dai servizi offerti
- l'**uso efficiente**; infatti gli amministratori si aspettano che il SO sfrutti al meglio le risorse hardware a disposizione in modo da avere buone prestazioni nell'esecuzione dei programmi
- l'**assenza di interferenze**; infatti gli utenti si aspettano la garanzia che altri utenti non siano in grado di interferire con le proprie attività

Soprattutto i primi due punti dipendono *dall'ambiente di elaborazione*, cioè dall'hardware di sistema del computer, dalle interfacce di rete e dal tipo di elaborazioni richieste dai suoi utenti.

Come detto, i progettisti di sistemi operativi devono gestire tre problemi: l'uso efficiente delle risorse, la convenienza per gli utenti e la sicurezza. L'uso efficiente delle risorse è più importante quando il sistema è dedicato ad applicazioni specifiche, la convenienza è più importante nei personal computer, mentre entrambe sono importanti nei sistemi condivisi da molti utenti; per queste ragioni, il progettista punta a una giusta combinazione tra uso efficiente e convenienza per l'utente. Per quanto riguarda la sicurezza, essa è importante in tutti gli ambiti di utilizzo.

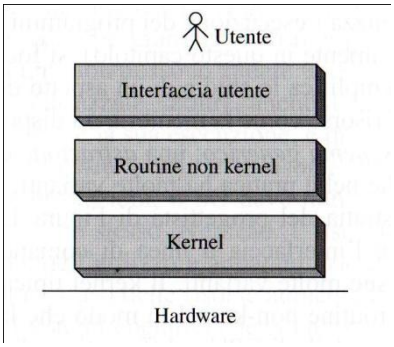
Un moderno SO deve poter essere utilizzato su computer con diverse architetture; inoltre deve restare al passo con l'evoluzione degli ambienti di elaborazione. Infatti il SO funge da *intermediario* tra gli utenti e il sistema: fornisce agli utenti dei servizi utilizzando le risorse HW del sistema. Oltre a ciò permette anche l'aggiornamento nel tempo dei servizi per adeguare i cambiamenti alle esigenze degli utenti.

1.1 Viste di un sistema operativo

La definizione di Sistema Operativo dipende dal tipo di utilizzo che ne deve fare l'utente che lo utilizza. Ad esempio, per uno studente il SO è semplicemente il software che permette di accedere ad internet, mentre per un programmatore il SO è il software che consente di sviluppare applicazioni, mentre per un agente di polizia è il software che permette di catalogare i vari criminali.

Tutte queste sono delle *visioni astratte* di un sistema operativo.

Un progettista di SO ha una sua visione astratta del sistema operativo stesso:



Le funzionalità tipiche di queste parti sono le seguenti:

- *Interfaccia utente*: questa accetta comandi per eseguire programmi e usa le risorse e i servizi forniti dal sistema operativo. Può essere un'interfaccia a linea di comando che mostra all'utente un prompt dei comandi oppure può essere un'interfaccia grafica (*GUI*).

- *Routine non di sistema*: queste implementano i comandi utente relativi all'esecuzione dei programmi e all'uso delle risorse del computer; sono richiamate dall'interfaccia utente.
- *Kernel*: questo è il cuore del SO. Controlla il computer e fornisce un insieme di funzioni e servizi per utilizzare la CPU, la memoria e le altre risorse del computer. Le funzioni e i servizi del kernel sono richiamati dalle routine non-kernel e dai programmi utente.

Dal punto di vista del progettista di un SO emergono due caratteristiche, mostrate nella figura precedente. Il sistema operativo è una collezione di routine che facilitano l'esecuzione dei programmi utente e l'uso delle risorse del computer. Il SO consiste di un'organizzazione gerarchica a livelli in cui ogni routine di livello superiore utilizza le operazioni fornite dalle routine di livello immediatamente inferiori.
Dal punto di vista dell'utente, l'interfaccia appare come una macchina che interpreta i comandi nel linguaggio del sistema operativo.

1.2 Obiettivi di un SO

Ribadiamo ancora una volta che gli obiettivi di un SO sono l'*uso efficiente* delle risorse, la *convenienza per l'utente* e la *sicurezza e prevenzione* intese come l'assenza di interferenze nelle attività degli utenti.

I primi due obiettivi possono, talvolta, generare conflitti. Ad esempio, per avere dei servizi veloci si dovrebbero tenere allocata la memoria per un programma anche quando questo non è in esecuzione, ma ciò porterebbe ad un uso inefficiente delle risorse. In questi casi è il progettista che deve giungere ad un compromesso per ottenere la combinazione ottimale tra uso efficiente e convenienza per l'utente.

1.2.1 Uso efficiente delle risorse

Un SO deve assicurare un uso efficiente delle risorse di sistema quali la memoria, la CPU e le periferiche di I/O come dischi e stampanti.
Si può verificare una scarsa efficienza se un programma non usa le risorse assegnatigli. Una situazione del genere può avere un effetto a cascata: poiché la risorsa è allocata per questo programma, essa non può essere assegnata ad altri programmi. Questi programmi non possono andare in esecuzione e dunque le risorse ad essi allocate non vengono usate.
Inoltre, anche il sistema operativo utilizza risorse di CPU e memoria durante la sua esecuzione, e questo utilizzo costituisce l'*overhead* che contribuisce a ridurre le risorse disponibili per i programmi utente.

Quindi, per ottenere buone prestazioni, occorre sia minimizzare lo spreco di risorse da parte dei programmi che l'overhead.

1.2.2 Convenienza per l'utente

In passato, convenienza per l'utente, indicava la possibilità di eseguire un programma scritto in un linguaggio di alto livello; mentre ai giorni nostri, convenienza per l'utente, significa semplicemente maggiore velocità nel rispondere alle richieste dell'utente.

Inoltre, un altro fattore che col passare del tempo ha inciso sulla convenienza per l'utente è stato il passaggio dal prompt dei comandi alle interfacce grafiche (GUI). E' solo grazie a quest'ultime che l'uso dei personal computer si è diffuso su scala mondiale. Va inoltre indicato che lo sviluppo delle GUI è in costante evoluzione per facilitare sempre più i compiti dell'utente ed introdurre nuove caratteristiche e tecnologie.

1.2.3 Assenza di interferenze

L'utente di un sistema operativo può dover fronteggiare diversi tipi di interferenza nell'utilizzo del computer. L'esecuzione dei suoi programmi può essere disturbata dalle azioni di altri utenti. Le interferenze riguardano non solo i programmi utente ma anche i servizi offerta dal SO stesso.
E' compito del sistema operativo prevenire queste interferenze allocando risorse ad uso esclusivo dei programmi utente e dei servizi del sistema stesso, e prevenendo l'acceso illegale alle risorse.

1.3 Funzionamento di SO

Gli obiettivi principali di un SO durante il funzionamento sono l'esecuzione dei programmi, l'utilizzo delle risorse e la prevenzione delle interferenze tra programmi e tra risorse. Un SO soddisfa questi obiettivi eseguendo tre funzioni primarie durante la sua esecuzione:

- *Gestione dei programmi*: il SO inizializza i programmi, organizza l'uso della CPU, e li termina quando hanno completato la loro esecuzione. Visto che vengono eseguiti più programmi in contemporanea, il SO esegue una funzione chiamata *scheduling* che serve per gestire l'esecuzione dei programmi.
- *Gestione delle risorse*: il SO alloca le risorse come la memoria e i dispositivi di I/O quando un programma li richiede. Al termine dell'esecuzione del programma, il SO dealloca queste risorse per assegnarle ad altri programmi.
- *Sicurezza e protezione*: il SO non dovrebbe permettere a nessun utente di usare in modo illegale programmi o risorse, o di interferire con il loro funzionamento.

BOOT: quando un computer viene avviato, automaticamente carica un programma memorizzato in una parte riservata di un periferica di I/O, generalmente un hard disk, e avvia l'esecuzione del programma. Questo programma esegue una tecnica software conosciuta come *bootstrapping* che carica in memoria il software necessario per la fase di boot (*procedura di boot*): in questa fase il programma caricato inizialmente carica altri programmi, che a loro volta caricano altri programmi, e così via fino a completare la fase di boot.
La procedura di boot crea una lista di tutte le risorse HW disponibili e passa il controllo del sistema al SO.

LOGIN: un amministratore di sistema specifica quali persone sono registrate come utenti del sistema. Il SO consente solo a queste persone di autenticarsi (nella *fase di login*) per utilizzare le sue risorse e i suoi servizi.

Un utente autorizza i suoi collaboratori ad accedere ai programmi e ai dati segnalandoli al SO che annota queste informazioni per implementare la protezione. In pratica un utente può modificare i privilegi di accesso ad un file in modo tale da permettere ai suoi collaboratori di accedere a quel file.

Il SO esegue anche un insieme di funzioni utili per la gestione dei processi e delle risorse (ad esempio lo scheduling dei programmi). Quindi un utente può modificare i privilegi di accesso ad un file in modo tale da permettere anche ad altri utenti di accedere a quel file.

Nella tabella seguente vengono descritte le operazioni comunemente effettuate da un SO.

Task	Eseguito
Costruire una lista di risorse	Durante la fase di boot
Memorizzare le informazioni per la sicurezza	Mentre vengono registrati nuovi utenti
Verificare l'identità di un utente	Al momento del login
Inizializzare l'esecuzione dei programmi	Quando richiesto dall'utente
Memorizzare le informazioni per l'autorizzazione	Quando un utente specifica che i suoi collaboratori possono avere accesso a i suoi programmi o ai suoi dati
Eseguire l'allocazione delle risorse	Quando richiesto dagli utenti o dai programmi
Preservare lo stato corrente delle risorse	Durante l'allocazione e la deallocazione delle risorse
Preservare lo stato corrente dei programmi ed effettuare lo scheduling	In maniera continuativa durante l'esecuzione del SO

I paragrafi successivi presentano una breve panoramica sulle responsabilità del SO nella gestione dei programmi e delle risorse e nell'implementazione della sicurezza e della protezione.

1.3.1 Gestione dei programmi

Le CPU moderne hanno velocità talmente elevate che possono alternare l'esecuzione di più programmi continuando a fornire un buon servizio all'utente. La funzione chiave per ottenere l'esecuzione alternata dei programmi è lo **scheduling**, che decide di volta in volta a quale programma deve essere concesso l'utilizzo della CPU.
Lo **scheduler**, implementato come routine del SO, mantiene una lista dei programmi in attesa di essere eseguiti dalla CPU e ne seleziona uno per l'esecuzione. Inoltre specifica anche per quanto tempo il programma può utilizzare la CPU. Al termine di questo tempo, il SO sottrae al programma la CPU e la concede ad un altro programma. Questa azione prende il nome di **prelazione**. Un programma che perde la CPU a causa della prelazione ritorna nella lista dei programmi in attesa di essere eseguiti dalla CPU.

1.3.2 Gestione delle risorse

L'allocazione e la deallocazione delle risorse possono essere effettuate usando una **tabella delle risorse**.

Questa tabella viene creata dalla procedura di boot rilevando la presenza di dispositivi di I/O e viene aggiornata dal SO con le allocazioni e le deallocazioni eseguite.

Ogni elemento della tabella contiene il nome e l'indirizzo di una risorsa e il suo stato attuale, che indica se è disponibile o allocata a qualche programma. Di seguito è riportato un esempio di tabella delle risorse.

Nome della risorsa	Classe	Indirizzo	Stato dell'allocazione
stampante1	Stampante	101	Allocata a P_1
stampante2	Stampante	102	Libera
stampante3	Stampante	103	Libera
disco1	Disco	201	Allocata a P_1
disco2	Disco	202	Allocata a P_2
cdw1	CD writer	301	Libera

Sono due le strategie di allocazione delle risorse più utilizzate: approccio basato sul **partizionamento delle risorse** e **approccio pool-based**.

Nell'approccio basato sul *partizionamento delle risorse* il SO decide a priori quali risorse dovrebbero essere allocate a ogni programma utente; ad esempio può decidere che a un programma sia allocato 1MB di memoria, 1000blocchi del disco e un monitor. Il SO divide le risorse in molte *partizioni*, ciascuna delle quali include 1M di memoria, 1000 blocchi del disco e un monitor. Una partizione è allocata a un programma utente nel momento in cui deve iniziare l'esecuzione.
Questo approccio è semplice da implementare, dunque è soggetto a meno overhead, ma pecca in flessibilità. In pratica ci potrebbe essere uno spreco di risorse se una partizione contiene più risorse di quelle che occorrono. Inoltre non si può mandare in esecuzione un programma se le sue richieste eccedono le risorse disponibili nella singola partizione. Questo vale anche se vi sono risorse libere in un'altra partizione.

Nell'approccio *per la gestione delle risorse (pool-based)*, il SO alloca le risorse prelevandole da un insieme unico di risorse. Quando un programma richiede una risorsa, il SO consulta la tabella delle risorse e, se disponibile, la alloca. Questo metodo è soggetto all'overhead dovuto alla allocazione e alla deallocazione ma evita entrambi i problemi dell'approccio del partizionamento delle risorse.

RISORSE VIRTUALI
Una *risorsa virtuale* è una risorsa fittizia, creata dal SO attraverso l'uso di risorse reali. Un SO può usare la stessa risorsa reale per supportare diverse risorse virtuali. In questo modo, può dare l'impressione di avere un numero di risorse maggiore di quelle effettivamente disponibili.
Ad esempio, avendo a disposizione un solo hard disk, lo si potrebbe partizionare in due parti, avendo così l'impressione di avere due hard disk.

Queste risorse vengono usate anche nei sistemi operativi moderni. Infatti molti SO forniscono una risorsa virtuale chiamata *memoria virtuale*, attraverso la quale si ha l'impressione di avere un quantitativo di memoria superiore di quella fisicamente disponibile.

1.3.3 Sicurezza e protezione

Come accennato in precedenza, un SO deve garantire che nessun utente possa utilizzare in maniera illegale programmi e risorse del sistema, o interferire con il loro funzionamento.

In un classico ambiente stand-alone (senza connessione a internet), un computer opera in completo isolamento, dunque le minacce alla sicurezza e alla protezione possono essere gestite facilmente. Si può utilizzare un computer solo se, in fase di autenticazione (login), si immette la password corretta.

In un sistema collegato ad internet, la sicurezza e la protezione vengono gestite solitamente con software appositi. Sono molteplici le minacce alla sicurezza: *cavalli di troia*, *virus* e *worm*.
Un *trojan horse* è un malware le cui funzionalità sono nascoste all'interno di un programma apparentemente utile. E' l'utente che, eseguendo questo programma, esegue inconsapevolmente il codice trojan.
Un *virus* è un malware che, una volta eseguito, è in grado di infettare i dati presenti sul computer e di riprodursi facendo copie di sé stesso. Solitamente riesce a fare tutto questo senza farsi rilevare.
Un *worm* è un malware in grado di autoreplicarsi. E' simile ad un virus, ma a differenza di questo non necessita di legarsi ad altri eseguibili per diffondersi.

CAPITOLO 2. Il SO, il computer e i programmi utente

Come visto nel capitolo precedente, il SO esegue molti compiti in modo ripetitivo, come l'inizializzazione dei programmi e l'allocazione delle risorse. Ognuno di questi compiti prende il nome di **funzione di controllo**.
Poiché il SO è un insieme di routine, esegue le funzioni di controllo utilizzando le istruzioni della CPU. In questo modo, la CPU esegue sia i programmi utente che il sistema operativo.

In questo capitolo verranno descritte alcune caratteristiche rilevanti di un computer e di come vengono utilizzate dal SO e dai programmi utente. Inoltre verrà discusso di come un SO interagisce con il computer e con i programmi utente, cioè qual è il procedimento mediante il quale prende il controllo della CPU per eseguire una funzione di controllo e come passa il controllo a un programma utente.

Si utilizza il termine **context switch** per identificare un'azione che forza la CPU a sospendere l'esecuzione di un programma e a iniziare l'esecuzione di un altro programma. Quando il kernel ha la necessità di eseguire una funzione di controllo, la CPU passa all'esecuzione del kernel. Dopo aver completato la funzione di controllo, la CPU passa nuovamente all'esecuzione dei programmi utente.

2.1 Principi fondamentali del funzionamento di un SO

E' importante comprendere il funzionamento di un SO prima di affrontare le caratteristiche dei SO e la loro progettazione. In particolare è importante capire quali sono le caratteristiche importanti di un computer dal punto di vista del SO, come il SO sfrutta tali caratteristiche durante il funzionamento e come i programmi ottengono i servizi dal SO.

Come detto in precedenza, il **kernel** è l'insieme di routine che costituiscono il cuore del SO. Esso controlla il funzionamento del computer implementando le operazioni note come *funzioni di controllo*. Inoltre fornisce servizi all'utente. Il kernel risiede in memoria durante il funzionamento del SO ed esegue le istruzioni utilizzando la CPU per implementare le funzioni di controllo ed i servizi. In questo modo la CPU è usata sia dai programmi utente che dal kernel.

Per questa serie di motivi è, dunque, necessario capire i dettagli del funzionamento del SO in termini di:

- come il kernel controlla il funzionamento del computer;
- come la CPU passa ad eseguire il codice del kernel quando si verifica un evento;
- come un programma utente utilizza i servizi messi a disposizione dal kernel;
- come il kernel assicura la mancanza di reciproche interferenze tra programmi utente e tra un programma utente e il sistema operativo

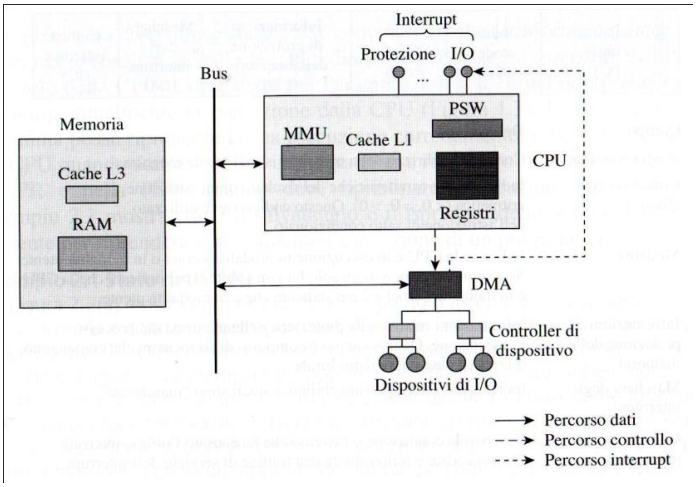
In riferimento a quest'ultimo punto si può dire che esistono due modalità di funzionamento della CPU, **modalità kernel** e **modalità utente**, che assicurano la mancanza di interferenze tra diversi programmi utente e tra programmi utente e sistema operativo.

Quando la CPU è in *modalità kernel* può eseguire tutte le istruzioni del computer.
La CPU viene impostata in *modalità utente* per eseguire i programmi utente.
Un aspetto molto importante consiste nel conoscere come la CPU viene impostata in modalità kernel per eseguire il codice kernel e come viene impostata in modalità utente per eseguire i programmi utente.

Per un utilizzo efficiente del computer, la CPU dovrebbe eseguire i programmi utente per la maggior parte del tempo. Tuttavia dovrebbe anche eseguire il codice del kernel quando si verifica un *evento*, cioè quando si verifica una situazione che richieda l'attenzione del kernel.

Ora verranno trattati gli elementi dell'architettura di un computer e verrà descritto come il kernel usa le caratteristiche dell'architettura del computer per controllarne il funzionamento.

2.2 Il computer



La figura mostra lo schema di un computer indicando le unità rilevanti dal punto di vista di un sistema operativo.
La CPU e la memoria sono direttamente connesse al bus di sistema, mentre i dispositivi di I/O sono connessi al bus attraverso un controller e il DMA.

2.2.1 La CPU

Sono due le caratteristiche della CPU visibili ai programmi utente o al sistema operativo:

- i **registri GPR** (noti anche come *general-purpose registers* o come *registri visibili dagli utenti*) che sono usati per memorizzare i dati, gli indirizzi, gli indici o lo stack pointer durante l'esecuzione di un programma;
- i **registri di controllo** che contengono l'informazione necessaria a controllare il funzionamento della CPU. Per semplicità l'insieme dei registri di controllo verrà chiamato *program status word* (PSW) e i singoli registri verranno individuati come *campi* della PSW.

Program counter (PC)	Condition code (CC)	Modalità (M)	Informazioni di protezione della memoria (MPI)	Maschera degli interrupt (IM)	Codice interrupt (IC)

Campo	Descrizione
Program counter	Contiene l'indirizzo della prossima istruzione da eseguire.
Condition code (flag)	Indica alcune caratteristiche del risultato di un'istruzione aritmetica (< 0, = 0, > 0). Questo codice viene utilizzato nell'istruzione di salto condizionato.
Modalità	Indica se la CPU è in esecuzione in modalità kernel o in modalità utente. Si assume un campo di un solo bit con valore 0 per indicare che la CPU è in modalità kernel e 1 per indicare che è in modalità utente.
Informazioni di protezione della memoria	Informazioni relative alla protezione della memoria del processo in esecuzione. Questo campo è composto di sottocampi che contengono il registro base e il registro limite.
Maschera degli interrupt	Indica quali interrupt sono abilitati e quali sono "mascherati".
Codice interrupt	Descrive la condizione o l'evento che ha causato l'ultimo interrupt. Questo codice è utilizzato da una routine di servizio dell'interrupt.

Nella figura precedente sono riportati i campi importanti della PSW.

PROGRAM COUNTER E CONDITION CODE

I primi due campi sono comunemente noti ai programmatori.

Il *program counter* (PC) contiene l'indirizzo della prossima istruzione che deve essere eseguita dalla CPU.

Il *codice di condizione* (CC) contiene un codice che descrive alcune caratteristiche relative al risultato dell'ultima operazione aritmetica o logica eseguita dalla CPU (< 0 , = 0 , > 0).

MODALITA' DI FUNZIONAMENTO KERNEL E UTENTE DELLA CPU

Abbiamo detto che la CPU può operare in due modalità chiamate *modalità kernel* e *modalità utente* e che la CPU può eseguire determinate istruzioni solo quando è in modalità kernel.

Queste istruzioni prendono il nome di **istruzioni privilegiate**; se queste fossero eseguite da parte dei programmi utente si creerebbero delle interferenze con il funzionamento del SO.

Il campo *modalità* (M) è di un solo bit: contiene 0 quando la CPU è in modalità privilegiata, mentre contiene 1 quando la CPU è in modalità utente.

STATO DELLA CPU
Sia i registri GPR che la PSW contengono tutte le informazioni necessarie per conoscere cosa sta facendo la CPU. Queste informazioni costituiscono lo *stato* della CPU.

Abbiamo detto in precedenza che il kernel può prelazionare il programma attualmente in esecuzione dalla CPU. Quando viene eseguita questa operazione, il kernel salva lo stato della CPU nei GPR e nel PSW in modo tale che quando il programma verrà ripristinato, saranno caricate queste informazioni.

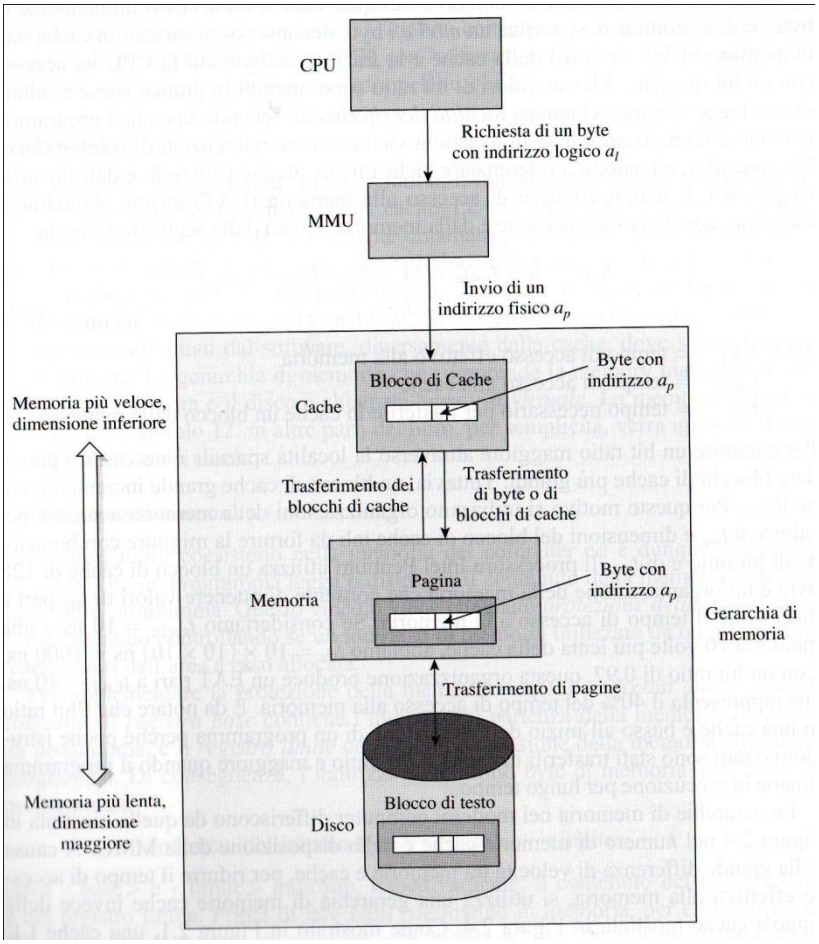
2.2.2 *Memory Management Unit (MMU)*

Questa unità si occupa di effettuare la ***traduzione degli indirizzi***:
- un ***indirizzo logico*** è l'indirizzo usato dalla CPU per fare riferimento ad un dato o ad un'istruzione
- un ***indirizzo fisico*** è l'indirizzo in memoria dove risiede il dato o l'istruzione richiesta dalla CPU

La MMU si occupa proprio di "tradurre" un indirizzo logico in un indirizzo fisico.

Inoltre, il SO implementa la memoria virtuale usando l'allocazione della memoria non contigua e la MMU. La tecnica della *memoria virtuale* consiste nel creare l'illusione di una memoria più grande della reale memoria installata.

2.2.3 *Gerarchia della memoria*



Un computer dovrebbe idealmente contenere una memoria abbastanza capiente e abbastanza veloce, così che gli accessi alla memoria non rallentino la CPU. Tuttavia la memoria veloce è costosa. La soluzione consiste in una gerarchia della memoria che contenga un numero di unità di memoria con differenti velocità.
La memoria più veloce è quella di dimensioni più piccole, mentre la memoria più lenta è quella di dimensioni maggiori.

La CPU accede solo alla memoria più veloce: se il dato (o l'istruzione) di cui necessita è presente nella memoria più veloce allora viene usato direttamente, altrimenti il dato richiesto viene copiato dalla memoria più lenta in quella più veloce e successivamente utilizzato. Il dato rimane nella memoria più veloce finchè non viene rimosso per fare spazio ad altri dati.
Questa organizzazione ha, quindi, lo scopo di velocizzare gli accessi ai dati

utilizzati più di frequente.

Un esempio di memoria gerarchica è quello costituito da:

- *memoria cache*, piccola e veloce
- *memoria RAM*, più capiente ma più lenta della cache
- *hard disk*, è la memoria più lenta ma anche quella più capiente

MEMORIA CACHE

La memoria cache contiene alcune istruzioni e valori di dati cui la CPU ha avuto accesso più di recente.

Per aumentare le prestazioni della cache, l'hardware della memoria non trasferisce un singolo byte dalla memoria alla cache, ma carica sempre un blocco di memoria di dimensioni standard in un'area della cache chiamata *cache block* o *cache line*. In questo modo, l'accesso a un byte vicino a un byte caricato di recente, può essere effettuato senza accedere nuovamente alla memoria.

Quando la CPU scrive un nuovo valore in un byte, il byte modificato viene scritto nella cache. Prima o dopo dovrà anche essere scritto in memoria. Sono diversi gli schemi usati per scrivere un byte in memoria: ad esempio, il *write-through* prevede la scrittura del byte nella cache e nella memoria allo stesso tempo.

Per ogni dato o istruzione richiesti durante l'esecuzione di un programma, la CPU effettua una ricerca nella cache. Si verifica un *hit* se i byte richiesti sono presenti nella cache, e in questo caso si ha accesso diretto ai byte; si verifica un *miss* se i byte richiesti non sono presenti, e in questo caso devono essere caricati nella cache dalla memoria.

Le gerarchie di memoria nei moderni computer differiscono da quella mostrata nella figura precedente nel numero di memorie cache e nella disposizione della MMU. A causa della grande differenza di velocità tra cache e memoria, per ridurre il tempo di accesso effettivo alla memoria, si utilizza una gerarchia di memorie cache invece della singola cache mostrata in figura. Esistono, infatti, cache di vari livelli che consentono di migliorare il tempo effettivo di accesso alla memoria: una cache L1 (cache di livello 1) è montata sul chip della CPU; a questa è solitamente affiancata una cache L2 (cache di livello 2), più lenta ma più capiente della cache L1; normalmente è presente anche una cache L3 ancora più capiente e lenta. Quest'ultima è tipicamente esterna alla CPU.

Un'altra differenza è che la MMU viene sostituita da una configurazione parallela della MMU e della cache L1. in questo modo alla cache L1 viene inviato un indirizzo logico piuttosto che un indirizzo fisico eliminando di fatto il processo di traduzione degli indirizzi (da indirizzo logico a indirizzo fisico).

MEMORIA RAM

Il funzionamento della memoria centrale è analogo a quello della memoria cache. Le similitudini riguardano il trasferimento di un blocco di byte, solitamente chiamato *pagina*, dal disco rigido alla memoria RAM.
La differenza sta nel fatto che la gestione della memoria e il trasferimento dei blocchi tra memoria e disco sono effettuati dal software, mentre dalla cache sono effettuati dall'hardware.

PROTEZIONE DELLA MEMORIA

Molti programmi sono eseguiti contemporaneamente e quindi usano contemporaneamente la memoria RAM. Per questo motivo è necessario prevenire che un programma legga o cancelli il contenuto della memoria utilizzata da un altro programma (*protezione della memoria*).
Per implementare la protezione della memoria vengono utilizzati due registri di controllo. Il *registro base* contiene l'indirizzo di partenza della memoria allocata a un programma, il *registro limite* contiene la dimensione della memoria allocata al programma. Prima di fare ogni riferimento in memoria viene controllato se un indirizzo di memoria utilizzato da un programma risiede fuori dall'area ad esso allocata, cioè se si trova al di fuori dell'intervallo degli indirizzi definiti dal contenuto dei registri base e limite. In questo caso l'hardware genera un interrupt per segnalare una violazione della protezione e interrompe l'accesso in memoria.

E' il campo del PSW relativo all'*informazione di protezione della memoria* (MPI) che contiene i registri base e limite. In questo modo l'informazione di protezione della memoria diventa una parte dello stato della CPU e viene salvata e caricata quando il programma è prelazionato o riportato in esecuzione.

Un programma potrebbe compromettere la protezione della memoria se avesse la possibilità di caricare informazioni a sua scelta nel registro base e nel registro limite. Per evitare questo inconveniente, le istruzioni per caricare i valori nei registri base e limite sono implementate come istruzioni privilegiate, cioè eseguibili solo quando la CPU è in modalità kernel.

La protezione della cache è più complessa. Ci sono vari metodi, quello più conveniente è quello dove viene memorizzato l'identificativo del programma le cui istruzioni o dati sono caricati in un blocco della cache e solo a quel programma è consentito l'accesso al contenuto del blocco.
Quando un programma genera un indirizzo logico contenuto in un blocco della cache, si verifica un hit solo se l'identificativo del programma coincide con l'identificativo del programma le cui istruzioni o dati sono caricati nel blocco della cache. Tale tecnica non influisce più di tanto sulle prestazioni dei programmi.

2.2.4 Input/Output

Una delle operazioni più lente che deve eseguire un sistema è il trasferimento di I/O, cioè il trasferire i dati da o verso una periferica di I/O. Quest'operazione risulta, solitamente, un'operazione molto lenta, che richiede l'intervento della CPU, della memoria e di una periferica di I/O.

Il tasso di trasferimento è determinato dal modo in cui è implementato il trasferimento tra memoria e periferica. L'organizzazione dell'I/O utilizzata nei moderni computer si è evoluta attraverso una sequenza di passi mirati a ridurre il coinvolgimento della CPU e a consentire tassi di trasferimento più elevati.
Infatti, visto che il sistema di I/O è il più lento di un computer, la CPU può eseguire milioni di istruzioni nella quantità di tempo richiesta per effettuare un'operazione di I/O; per tale motivo si preferisce utilizzare tra i vari modi esistenti per effettuare le operazioni di I/O quello che non prevede l'intervento della CPU.

I modi per effettuare le operazioni di I/O sono i seguenti:

Modalità di I/O	Descrizione
I/O programmato	Il trasferimento dati tra la periferica di I/O e la memoria avviene attravero la CPU. La CPU non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O.
Interrupt di I/O	La CPU è libera di eseguire altre istruzioni dopo aver eseguito l'istruzione di I/O. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte. Questa sequenza di operazioni viene ripetuta finché tutti i byte sono trasferiti.
I/O basato sul direct memory access (DMA)	Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus. La CPU non è coinvolta nel trasferimento dei dati. Il controller DMA genera un interrupt quando il trasferimento di tutti i byte è stato completato.

Nella *modalità I/O programmato* il trasferimento dei dati avviene attraverso la CPU, dunque il trasferimento è lento e la CPU è interamente impegnata nella sua gestione. Di conseguenza può essere effettuata solo un'operazione di I/O alla volta.

La *modalità interrupt* è ugualmente lenta poiché effettua il trasferimento dati un byte alla volta con l'aiuto della CPU. Tuttavia la CPU è libera tra i diversi trasferimenti.

La *modalità DMA* può trasferire un blocco dati tra la memoria e una periferica di I/O senza coinvolgere la CPU, ottenendo tassi di trasferimento elevati e supportando operazioni concorrenti di CPU e dispositivi di I/O. Inoltre sia la modalità interrupt che la modalità DMA consentono di eseguire operazioni di I/O simultaneamente su diversi dispositivi.

Le operazioni del DMA sono effettuate dal **controller DMA**, ovvero un processore dedicato all'esecuzione delle operazioni di I/O. Come si può vedere nell'immagine del paragrafo 2.2 diversi dispositivi di I/O della stessa classe sono collegati ad un controller. I vari controller sono poi collegati al DMA. Quando viene eseguita un'operazione di I/O, la CPU "delega" il DMA in modo tale da non essere coinvolta nell'operazione ed essere dunque libera di svolgere altre funzioni. E' il DMA, mediante il controller, ad effettuare l'operazione. Al termine dell'operazione, il DMA genera un *interrupt di I/O*. La CPU passa all'esecuzione del kernel quando rileva un interrupt; il kernel analizza la causa dell'interrupt e deduce che l'operazione di I/O è stata completata.

2.2.5 Interrupt

Un **evento** è una situazione che richiede l'attenzione del sistema operativo. Ad ogni evento è associato un **interrupt** il cui scopo è quello di segnalare al SO il verificarsi dell'evento a cui è associato in modo da consentirgli di effettuare le azioni appropriate per gestirlo.

Implementazione di un interrupt:
Nel ciclo di esecuzione di un'istruzione della CPU, è recuperata, decodificata ed eseguita (*fetch – decode – execute*) l'istruzione il cui indirizzo è contenuto nel Program Counter. Dopo l'esecuzione (*execute*) dell'istruzione si controlla se si è verificato un interrupt durante l'esecuzione dell'istruzione. In caso affermativo esegue un'*azione di interrupt* che salva lo stato della CPU (cioè il contenuto del PSW e dei GPR) e carica i nuovi dati nel PSW e nei GPR in modo che la CPU cominci l'esecuzione, in modalità kernel, delle istruzioni di una *routine di servizio dell'interrupt* (chiamata anche ISR). A un certo punto, il kernel può ripristinare l'esecuzione del programma interrotto ricaricando lo stato salvato della CPU.

Ad ogni interrupt è associata anche una **priorità**. Se diversi interrupt si verificano nello stesso tempo, la CPU seleziona l'interrupt con priorità più alta, mentre gli altri interrupt restano pendenti finchè non verranno selezionati.

CLASSI DI INTERRUPT

Classe	Descrizione
I/O interrupt	Causato da condizioni come il completamento dell'I/O e il malfunzionamento delle periferiche di I/O.
Timer interrupt	Generato a intervalli di tempo o quando è trascorso uno specifico intervallo di tempo.
Program interrupt	(1) Causato da condizioni di eccezioni che si verificano durante l'esecuzione di una istruzione, per esempio eccezioni aritmetiche come overflow, eccezioni di indirizzamento e violazioni di protezione della memoria. (2) Causato dall'esecuzione di una speciale istruzione chiamata *istruzione di interrupt software*, il cui unico scopo è di generare un interrupt.

In questa tabella vengono descritte le tre classi di interrupt che risultano molto importanti durante il normale funzionamento del SO.

Un **I/O interrupt** indica la fine di un'operazione di I/O o il verificarsi di condizioni come il malfunzionamento della periferica di I/O.

Un **timer interrupt** è utilizzato per implementare un'organizzazione scandita dal tempo. Un interrupt può essere generato sia periodicamente (dopo un predefinito numero di tick) oppure dopo un intervallo di tempo programmato.

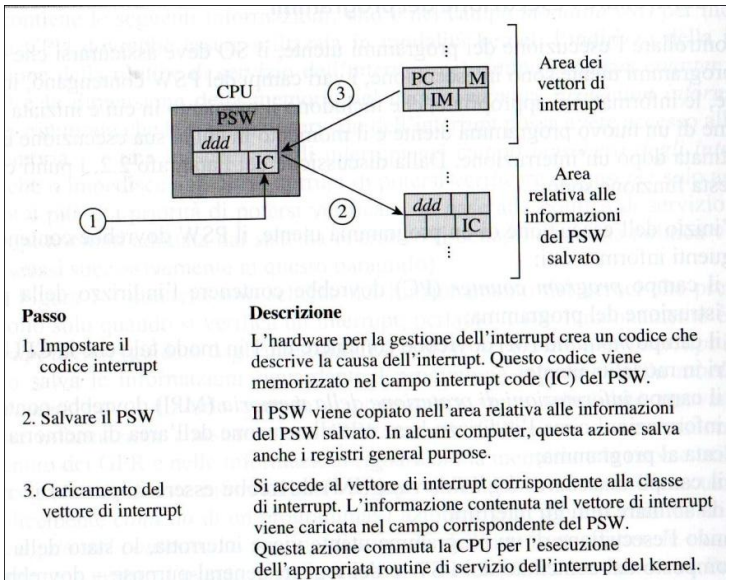
Un **program interrupt** è usato per due scopi. Nel primo, l'hardware del computer utilizza l'interrupt per indicare il verificarsi di una condizione durante l'esecuzione di un'istruzione, come l'overflow in un'operazione o come una violazione di protezione. Nel secondo, i programmi utente utilizzano l'interrupt per richiedere servizi o risorse al kernel a cui non possono accedere direttamente. In quest'ultimo caso, l'istruzione speciale che serve per passare il controllo al kernel prende il nome di *istruzione di interrupt software*, il cui unico scopo è quello di generare un interrupt.

Per la gestione degli interrupt vengono usati due campi nel PSW: **IC** (*Interrupt Code*) e **IM** (*Interrupt Mask*).

CODICE INTERRUPT (IC)
Quando si verifica un interrupt di una certa classe, l'hardware imposta un **codice interrupt** nel campo IC del PSW per indicare quale specifico interrupt, all'interno di quella classe, si è verificato. Questa informazione è utile per sapere la causa che ha generato l'interrupt.
I codici interrupt sono specifici per ogni architettura. Solitamente, per un I/O interrupt, il codice interrupt è l'indirizzo della periferica di I/O.

MASCHERAMENTO DEGLI INTERRUPT (IM)
Il campo *maschera degli interrupt* (IM) del PSW indica quali interrupt sono consentiti in un dato momento. Gli interrupt rilevabili sono detti *abilitati* mentre gli altri sono detti *mascherati* o *disabilitati*. Se si verifica un evento corrispondente ad un interrupt mascherati, l'interrupt generato non viene perso, ma rimane pendente finchè non viene abilitato e può dunque essere rilevato.

L'AZIONE DI INTERRUZIONE
Dopo l'esecuzione di ogni istruzione, la CPU controlla se sono stati generati interrupt. In caso affermativo, la CPU esegue il gestore dell'interrupt (*interrupt handler*), che salva lo stato della CPU e passa ad eseguire la routine di servizio dell'interrupt nel kernel.



Passo	Descrizione
1. Impostare il codice interrupt	L'hardware per la gestione dell'interrupt crea un codice che descrive la causa dell'interrupt. Questo codice viene memorizzato nel campo interrupt code (IC) del PSW.
2. Salvare il PSW	Il PSW viene copiato nell'area relativa alle informazioni del PSW salvato. In alcuni computer, questa azione salva anche i registri general purpose.
3. Caricamento del vettore di interrupt	Si accede al vettore di interrupt corrispondente alla classe di interrupt. L'informazione contenuta nel vettore di interrupt viene caricata nel campo corrispondente del PSW. Questa azione commuta la CPU per l'esecuzione dell'appropriata routine di servizio dell'interrupt del kernel.

Come si può vedere nella figura il gestore di interrupt si compone di tre passi.

1. **Impostare il codice interrupt**
   Il primo passo imposta il codice dell'interrupt nel campo IC del PSW in base alla causa dell'interrupt.
2. **Salvare il PSW**
   Il secondo passo salva il contenuto del PSW in memoria in modo tale che il kernel possa ricreare lo stato della CPU del programma interrotto quando questo verrà ripristinato.
3. **Caricamento del vettore di interrupt**
   Il terzo passo esegue l'appropriata routine di servizio dell'interrupt nel kernel. In pratica viene servito l'interrupt.
   Nell'are del *vettore degli interrupt* vi sono diversi vettori degli interrupt; in base alla classe di interrupt si accede al vettore di interrupt corrispondente. L'informazione contenuta nel vettore di interrupt viene caricata nel campo corrispondente del PSW (si assume, per semplicità, che il vettore di interrupt abbia lo stesso formato del PSW). Quindi nel terzo passo vengono caricate le informazioni dal vettore degli interrupt nei campi *program counter*, *maschera degli interrupt* e *modalità* del PSW, consentendo alla CPU di eseguire in modalità kernel la routine di servizio dell'interrupt.

2.3 Interazione del SO con il computer e i programmi utente

In questo paragrafo verrà discusso di come il SO interagisce con il computer per garantire che lo stato del programma interrotto sia salvato, in modo da poterne riprendere l'esecuzione successivamente e di come una routine di servizio dell'interrupt ottenga le informazioni relative all'evento che ha causa un interrupt in modo da poter effettuare le azioni appropriate.

I programmi hanno bisogno di usare i servizi messi a disposizione dal SO. Per questo motivo hanno bisogno di generare un interrupt per passare le loro richieste al SO. Una **chiamata di sistema** (in inglese *system call*) è il meccanismo usato da un programma a livello utente per richiedere un servizio a livello kernel del SO.

2.3.1 Controllare l'esecuzione dei programmi

Per controllare l'esecuzione dei programmi utente, il SO deve assicurarsi che quando questi sono in esecuzione, i vari campi del PSW contengano, in ogni istante, le informazioni appropriate.
In pratica:

1. All'inizio dell'esecuzione di un programma utente, il PSW dovrebbe contenere le seguenti informazioni:
    a. il campo *program counter* (PC) dovrebbe contenere l'indirizzo della prima istruzione del programma;
    b. il campo *modalità* (M) dovrebbe contenere un 1, cioè che la CPU opera in modalità utente;
    c. il campo *informazioni di protezione della memoria* (MPI) dovrebbe contenere informazioni circa l'indirizzo base e la dimensione dell'area di memoria dedicata al programma;
    d. il campo *maschera degli interrupt* (IM) dovrebbe essere impostato in modo da abilitare tutti gli interrupt.
2. Quando l'esecuzione di un programma utente viene interrotta, lo stato della CPU (contenuto del PSW e dei GPR) dovrebbe essere salvato.
3. Quando deve essere ripristinata l'esecuzione del programma interrotto, lo stato salvato della CPU dovrebbe essere caricato nel PSW e nei GPR.

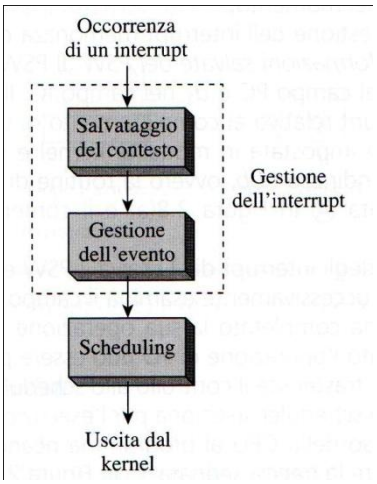
Il SO conserva una tabella delle informazioni, che al momento possiamo chiamare *tabella dei programmi*.
Ogni elemento della tabella contiene informazioni relative a un programma utente ed è creato quando viene avviato il programma.
Un campo di questa tabella viene usato per conservare informazioni circa lo stato della CPU. Il kernel:

- scrive le informazioni menzionate nel punto 1 in questo campo quando si deve iniziare l'esecuzione del programma
- salva lo stato della CPU in questo campo quando l'esecuzione del programma viene interrotta
- quando un programma viene ripristinato, le informazioni contenute in questo campo vengono caricate

2.3.2 Servire gli interrupt

Come già detto, per semplicità, si assume che un vettore degli interrupt abbia lo stesso formato del PSW. Il kernel, all'avvio del sistema operativo, costruisce il vettore degli interrupt per diverse classi di interrupt. Ciascun vettore contiene queste informazioni:

- uno 0 nel campo *modalità* (M) per indicare che la CPU dovrebbe essere usata in modalità kernel;
- l'indirizzo della prima istruzione della routine di servizio dell'interrupt nel campo *program counter* (PC);
- uno 0 e la dimensione della memoria nel campo MPI in modo tale che possa avere accesso all'intera memoria;
- una maschera nel campo IM che, o impedisce agli altri interrupt di potersi verificare o consente solo agli interrupt ad alta priorità di potersi verificare.



Il kernel prende il controllo solo quando si verifica un interrupt, pertanto il suo funzionamento è detto *interrupt-driven* (guidato dagli interrupt). La routine di servizio dell'interrupt salva il **contesto**, cioè le informazioni riguardanti il programma interrotto, per poterlo poi caricare quando il programma verrà ripristinato. Dopo aver completato il *salvataggio del contesto*, la routine di servizio dell'interrupt gestisce l'evento che ha causato l'interrupt in base al codice ad esso associato. Infine, una volta servito l'interrupt, la *routine di scheduling* seleziona un programma da mandare in esecuzione. Questo programma può o non può essere quello precedentemente interrotto: se viene eseguito il programma interrotto precedentemente viene anche caricato il contesto salvato in precedenza.

GESTIONE DEGLI INTERRUPT ANNIDATI
Col termine *interrupt annidati* si intende il verificarsi di interrupt contemporaneamente.

I sistemi operativi usano due approcci per la gestione degli interrupt annidati.
Alcuni SO usano il campo IM del vettore di gestione dell'interrupt per mascherare tutti gli interrupt mentre è in esecuzione una routine di gestione dell'interrupt. Questo approccio rende il ***kernel non prelazionabile***, cioè in grado di gestire un solo interrupt per volta. Tuttavia, si potrebbero verificare dei ritardi nella gestione degli interrupt a più alta priorità.
Nel secondo approccio, il kernel imposta la maschera degli interrupt in ogni vettore degli interrupt per mascherare solo gli interrupt con priorità più bassa in modo tale da poter servire gli interrupt più critici in maniera annidata. Tali kernel sono chiamati ***kernel prelazionabili***. In questo approccio potrebbero esserci problemi di inconsistenza dei dati se due o più routine di gestione degli interrupt annidate aggiornassero gli stessi dati nel kernel; per tale motivo occorre utilizzare uno schema di sincronizzazione per assicurare che solo una routine di gestione degli interrupt possa accedere ai dati in ogni istante.

PRELAZIONE DEI PROGRAMMI UTENTE
Va ricordato che il contesto dei programmi interrotti è conservato nella *tabella dei programmi* e dunque non c'è nessuna difficoltà nel ripristinare l'esecuzione di un programma prelazionato quando viene ripristinato.

2.3.3 System call

Un programma, durante la sua esecuzione, potrebbe aver bisogno di utilizzare le risorse del computer, come le periferiche di I/O. Tuttavia, queste risorse sono condivise tra i programmi utente e dunque è necessario prevenire possibili interferenze nel loro utilizzo.
Per facilitare la gestione delle risorse, le istruzioni che allocano o hanno accesso ad esse sono *istruzioni privilegiate*, ciò vuol dire che la CPU può accedervi solo in modalità kernel; i programmi utente non hanno accesso diretto alle risorse ma è il kernel che, per loro, accede ad esse.

Una **chiamata di sistema** (in inglese *system call*) è il meccanismo usato da un programma a livello utente per richiedere un servizio a livello kernel del sistema operativo. Una system call è implementata attraverso il metodo degli interrupt, dunque la si può definire anche come una richiesta che un programma fa al kernel attraverso un interrupt software.

Ovviamente una system call può prendere parametri che forniscono informazioni importanti al servizio del kernel invocato. Ad esempio, la *open*, che apre un file, prende come parametro il nome del file da aprire.

CAPITOLO 3. Panoramica dei sistemi operativi

Un *ambiente di elaborazione* si compone di un computer, delle sue interfacce con altri sistemi e dei servizi forniti dal suo sistema operativo agli utenti e ai loro programmi.

Nel corso della storia, sia gli ambienti di elaborazione che i sistemi operativi si sono evoluti con l'evolversi della tecnologia, delle applicazioni e delle necessità degli utenti per fornire una qualità sempre maggiore agli utenti stessi. Infatti col passare del tempo sono state sviluppate nuove classi di sistemi operativi che, in parte, sfruttavano concetti e tecniche presenti nei sistemi operativi già usciti in commercio.

In questo capitolo verranno affrontati i concetti e le tecniche utilizzate nei fondamentali sistemi operativi.

3.1 Ambienti di elaborazione e natura delle elaborazioni

Abbiamo detto che col passare del tempo gli ambienti di elaborazione si sono evoluti; tuttavia il sistema operativo deve svolgere compiti sempre più complessi all'aumentare della potenza computazionale del computer, della complessità delle interfacce con le periferiche di I/O e di nuovi servizi agli utenti.

Ad esempio, in un tipico ambiente di elaborazione moderno, l'utente avvia diverse attività simultaneamente (client di posta elettronica, browser per la navigazione web, player audio/video, ecc), quindi il SO deve essere in grado, in base alle risorse di cui è dotato, di permettere l'esecuzione ottimale di queste attività.

Iniziamo col dare uno sguardo veloce a come gli ambienti di elaborazione si sono evoluti fino alla loro configurazione attuale.

AMBIENTI DI ELABORAZIONE NON INTERATTIVI
Rappresentano le forme più vecchie di ambienti di elaborazione. In questi ambienti un utente fornisce al SO sia il programma che i dati. L'elaborazione viene effettuata dal SO e i risultati vengono restituiti all'utente, che non può interagire con l'elaborazione.
Esempi di elaborazioni non interattive sono le elaborazioni scientifiche.
In questi ambienti di elaborazione, l'obiettivo del SO è l'uso efficiente delle risorse.

Le elaborazioni utilizzate negli ambienti non interattivi sono programmi o *job* dove:

- un programma è un insieme di funzioni o moduli
- un job è una sequenza di programmi che insieme raggiungono l'obiettivo desiderato; viene eseguito solo se i programmi precedenti del job sono stati eseguiti con successo

AMBIENTI DI ELABORAZIONE INTERATTIVI
In questo tipo di ambiente, l'utente può interagire con l'elaborazione mentre è in esecuzione.
L'obiettivo del SO è di ridurre il tempo medio richiesto per implementare l'interazione tra un utente e la sua elaborazione. Inoltre l'utente interagisce col SO per avviare un'elaborazione, dunque non ha senso parlare di job in quanto è l'utente a scegliere cosa avviare, ovviamente tenendo conto della dipendenza dei programmi prima di eseguire il comando successivo.

AMBIENTI REAL-TIME, DISTRIBUITI ED EMBEDDED
Alcune elaborazioni hanno delle necessità speciali, per le quali sono stati sviluppati speciali ambienti di elaborazione.

In un *ambiente real-time* il SO deve utilizzare tecniche speciali per assicurare che le elaborazioni siano completate rispettando dei vincoli temporali.

In un *ambiente distribuito* le risorse presenti possono essere utilizzate da diversi sistemi attraverso una rete.

In un *ambiente embedded* il computer è parte di uno specifico sistema hardware, come un elettrodomestico ed esegue elaborazioni volte a controllare il sistema stesso.

I MODERNI AMBIENTI DI ELABORAZIONE
Come detto, nei moderni ambienti di elaborazione c'è la necessità di supportare diverse applicazioni, dunque il SO deve adottare complesse strategie per gestire i programmi e le risorse.

Ora affrontiamo lo studio delle strategie utilizzate nei moderni SO, in particolare, in questo capitolo, ci soffermeremo sulle strategie utilizzate dai SO in ognuno degli ambienti di elaborazione appena menzionati.

3.2 Classi di sistemi operativi

Classe del SO	Periodo	Obiettivo principale	Concetti chiave
Elaborazione batch	anni '60	Tempo idle della CPU	Automatizzare la transizione tra i job
Multiprogrammazione	anni '60	Utilizzo delle risorse	Proprietà del programma, prelazione
Time-sharing	anni '70	Buon tempo di risposta	Scheduling, time slice, round-robin
Real time	anni '80	Rispettare i vincoli temporali	Scheduling real-time
Distribuiti	anni '90	Condivisione delle risorse	Controllo distribuito, trasparenza

La tabella elenca cinque classi fondamentali di sistemi operativi il cui nome rispecchia le loro caratteristiche peculiari. Inoltre mostra anche il periodo in cui i vari SO hanno avuto diffusione, l'obiettivo principale per il quale sono stati sviluppati e i concetti chiave sui quali si basa per poter raggiungere l'obiettivo.

Negli anni '60 l'hardware era molto costoso per cui i SO si focalizzavano sull'uso efficiente della CPU e delle altre risorse del sistema.
Negli anni '70 l'hardware divenne un po' più economico, per questo motivo l'obiettivo principale si spostò verso la produttività degli utenti.
Negli anni '80 ci fu l'avvento delle applicazioni real-time, di conseguenza i SO si focalizzarono sull'ottimizzazione di queste applicazioni.
Negli anni '90 si decise di puntare forte sullo sviluppo dei SO distribuiti grazie ai quali diversi computer condividevano le proprie risorse attraverso la rete.

SISTEMI DI ELABORAZIONE BATCH
L'obiettivo principale è l'uso efficiente della CPU. Questo sistema opera elaborando un job alla volta, eseguendo i programmi uno dopo l'altro. In questo modo un solo programma è in esecuzione in un dato momento.

SISTEMI MULTIPROGRAMMATI
L'obiettivo è sia l'uso efficiente della CPU che dei dispositivi di I/O. Questo sistema gestisce diversi programmi in uno stato di parziale completamento per ogni istante di tempo ed attraverso le *priorità del programma* concede o meno l'utilizzo della CPU ai vari programmi.

SISTEMI TIME-SHARING
L'obiettivo principale è l'ottimizzazione della velocità di risposta alle richieste fatte dai processi. Questo obiettivo è raggiunto dando un'equa opportunità di esecuzione a ogni processo attraverso due metodi: il SO serve tutti i processi a turno (*scheduling round-robin*) ed evita che un processo utilizzi per troppo tempo la CPU (*time-slicing*).

SISTEMI REAL-TIME
L'obiettivo principale è il supporto di applicazioni real-time, cioè quelle applicazioni dove è necessario ottenere una risposta dal SO in un tempo prefissato. Non è importante l'intervallo di tempo in cui il SO deve reagire; l'importante è che risponda entro un tempo massimo pre-determinato. In altre parole il sistema deve essere *prevedibile*.

SISTEMI DISTRIBUITI
L'obiettivo principale è quello di consentire ad un utente di avere accesso alle risorse presenti in altri computer in modo conveniente ed efficace. Per migliorare la convenienza, il SO non richiede all'utente di conoscere dove si trovano le risorse (*trasparenza*); per migliorare l'efficienza, il SO può eseguire parti di un'elaborazione su computer differenti allo stesso tempo.

3.3 Efficienza, prestazioni del sistema e servizio per l'utente

Tre parametri molto importanti sono l'efficienza, le prestazioni del sistema e il servizio per l'utente.

EFFICIENZA DI UNA RISORSA
Valutare l'efficienza nell'uso di una risorsa equivale a vedere quanto la risorsa non è utilizzata e, relativamente all'utilizzo della risorsa, quanto è stata produttiva.

Consideriamo, ad esempio, come risorsa la CPU. Innanzitutto sappiamo che una parte del tempo di CPU è usata dal SO per la gestione degli interrupt e per lo scheduling, costituendo l'**overhead**. La parte del tempo rimanente della CPU è usato per eseguire i processi degli utenti. Per valutare l'efficienza della CPU basta considerare la percentuale di utilizzo della CPU per eseguire i processi degli utenti.

In modo analogo è possibile determinare l'efficienza della memoria e dei dispositivi di I/O: valutare la percentuale di utilizzo in rapporto al totale.

PRESTAZIONI DEL SISTEMA
Le prestazioni del sistema consistono nella quantità di lavoro svolto nell'unità di tempo.
Una volta stabilita la giusta combinazione di efficienza della CPU e servizio dell'utente è importante poter misurare le prestazioni del SO. Le prestazioni del sistema sono caratterizzate dalla quantità di lavoro svolto per unità di tempo e sono generalmente misurate dal *throughput*.

Il **troughput** è il numero di job, programmi e processi completati in una unità di tempo.

L'unità di lavoro usata per misurare il throughput dipende dall'ambiente di elaborazione. Ad esempio, il throughput di un hard disk può essere misurato come il numero di bytes trasferiti in un'unità di tempo. In un sistema bancario potrebbe essere il numero di transazioni per unità di tempo.

SERVIZIO PER L'UTENTE
Il servizio per l'utente indica quanto velocemente un'elaborazione dell'utente è stata completata dal SO.

Definiamo due misure del servizio dell'utente:

- **tempo di turnaround**, è il tempo di completamento di un job o di un processo
- **tempo di risposta**, è il tempo di risposta ad una sottorichiesta dell'utente

3.4 Sistemi di elaborazione batch

I computer negli anni '60 non erano interattivi. Le schede perforate erano il mezzo di input principale e dunque un job e i suoi dati consistevano in un gruppo di schede. Un operatore caricava le schede in un

lettore per impostare l'esecuzione di un job. Questa azione causava una perdita di tempo prezioso di CPU; l'elaborazione batch fu introdotta per prevenire questo spreco.

Un **batch** è una sequenza di job utente assemblati per essere elaborati dal sistema operativo. Un operatore assemblava il batch organizzando alcuni job utente in sequenza, delimitando l'inizio e la fine del batch con schede speciali. Il SO eseguiva uno dopo l'altro i vari job del batch. In questo modo l'operatore doveva intervenire solo all'inizio e alla fine del batch.

I lettori di schede usati negli anni '60 rappresentavano un collo di bottiglia, per questo motivo i sistemi batch (anni '70) usarono il concetto di lettori di schede virtuali attraverso i nastri magnetici, per migliorare il throughput del sistema.

In definitiva possiamo dire che i sistemi operativi per l'elaborazione batch si focalizzano sul processo di automatizzazione di una collezione di programmi, in modo da ridurre i tempi idle della CPU.

3.5 Sistemi multiprogrammati

Un sistema multiprogrammato mantiene molti programmi utente in memoria.
Esso utilizza il DMA per le operazioni di I/O. In pratica mentre il DMA esegue le operazioni di I/O di alcuni programmi, la CPU esegue istruzioni di altri programmi.
Questa organizzazione fa un uso efficiente sia della CPU che dei dispositivi di I/O: i SO forniscono servizio a diversi programmi simultaneamente sovrapponendo un'operazione di I/O in un programma con l'esecuzione di istruzioni in un altro programma.

Poiché diversi programmi sono in memoria contemporaneamente, le istruzioni, i dati e le operazioni di I/O di un programma dovrebbero essere protette dall'interferenza di altri programmi.
In un sistema multiprogrammato:

- il **DMA** consente la multiprogrammazione
- la **protezione della memoria** previene eventuali accessi non autorizzati al di fuori dello spazio di indirizzamento definito dal contenuto del *registro base* e del *registro size* (*limite*)
- le **modalità kernel** e **utente** della CPU prevengono le interferenze tra i programmi permettendo l'esecuzione delle istruzioni privilegiate solo al kernel (se un programma utente cercasse di effettuare un'operazione del genere verrebbe generato un interrupt)

3.5.1 Priorità dei programmi

Il SO mantiene sempre, in memoria, un numero sufficiente di programmi, in modo che la CPU e i dispositivi di I/O abbiano lavoro sufficiente da effettuare. Questo numero è detto *grado di multiprogrammazione*.

Tuttavia, un elevato grado di multiprogrammazione non può garantire un buon utilizzo sia della CPU che dei dispositivi di I/O, perché la CPU resterebbe idle se ognuno dei programmi eseguisse operazioni di I/O per la maggior parte del tempo, oppure i dispositivi di I/O rimarrebbero idle se ognuno dei programmi eseguisse elaborazioni per la maggior parte del tempo. Per tale motivo il sistema multiprogrammato adotta due tecniche per assicurare una sovrapposizione della CPU e dei dispositivi di I/O:

- la prima tecnica è quella di utilizzare un *mix appropriato* di programmi:
  - **programmi CPU-bound**, che necessitano di molta elaborazione ma poche operazioni di I/O
  - **programmi I/O-bound**, che richiedono poca elaborazione ma eseguono molte operazioni di I/O
- la seconda tecnica è quella di utilizzare lo *scheduling a priorità con prelazione* per condividere la CPU tra i programmi assicurando una buona sovrapposizione

La **priorità** è quel criterio mediante il quale lo scheduler decide quale richiesta debba essere schedulata quando molte richieste sono in attesa di essere servite.

Il kernel utilizza la priorità con prelazione, cioè prelaziona un programma a bassa priorità in esecuzione dalla CPU se un programma ad alta priorità richiede la CPU. In questo modo, la CPU esegue sempre il programma a più alta priorità che la richiede.
E' il kernel che assegna le priorità numeriche ai programmi. L'assegnazione delle priorità ai programmi è una decisione cruciale che influenza il throughput del sistema. Nei sistemi multi programmati vengono assegnate alte priorità ai programmi I/O-bound.

3.6 Sistemi time-sharing

Questi sistemi sono progettati per fornire tempi di risposta veloci ai programmi degli utenti. L'obiettivo è ottenuto condividendo il tempo di CPU tra i processi in modo tale che ogni processo che ha fatto richiesta ottenga l'uso della CPU senza attendere troppo.

Questo obiettivo è raggiunto utilizzando lo scheduling *round-robin con time-slicing*.

Lo scheduling round-robin funziona in questo modo: il kernel mantiene una *coda di scheduling* dei processi che richiedono l'uso della CPU e schedula il processo in testa alla coda. Quando il processo schedulato termina l'esecuzione o avvia un'operazione di I/O il kernel lo rimuove dalla coda e schedula un altro processo. Il processo precedente viene inserito in fondo alla coda se effettua una nuova richiesta o se termina l'operazione di I/O. Questa organizzazione fa sì che tutti i processi debbano aspettare più o meno lo stesso tempo prima di poter usare la CPU.

In questi sistemi viene anche utilizzata la nozione di *time-slicing*: se ogni processo utilizzasse la CPU per troppo tempo, il tempo di attesa degli altri processi sarebbe troppo alto, dunque, per evitare questa situazione, il kernel utilizza il **time-slice** che è la più grande porzione di tempo di CPU che ogni processo può utilizzare quando viene schedulato per essere eseguito dalla CPU. Il time-slice viene implementato dal kernel mediante l'uso di un interrupt timer.
In pratica, se la porzione di tempo termina prima che il processo completi l'esecuzione, il kernel prelaziona il processo, lo sposta in fondo alla coda di scheduling e schedula un altro processo. In questo modo, un processo può anche dover essere eseguito più volte prima che completi l'operazione.

3.6.1 Swapping dei programmi

Il throughput delle sottorichieste è la misura appropriata delle prestazioni di un SO time-sharing. La tecnica dello *swapping* viene usata per servire un maggior numero di processi rispetto a quelli che possono essere effettivamente presenti in memoria. Essa ha dunque la potenzialità di migliorare sia le prestazioni del sistema sia i tempi di risposta dei processi.

Lo **swapping** è la tecnica di rimuovere temporaneamente un processo dalla memoria di un computer.

Il kernel effettua un'operazione di *swap-out* su un processo che non sarà schedulato nel prossimo futuro, copiando le sue istruzioni e i dati sul disco rigido. In questo modo si libera l'area di memoria allocata per quel processo. Il kernel carica un altro processo in questa area di memoria con un'operazione di *swap-in*.

In pratica, quando la memoria RAM libera non è più sufficiente per contenere tutte le informazioni che servono ai programmi, il SO si fa carico di spostare una certa quantità di dati (quelli meno recentemente utilizzati) dalla memoria al disco rigido, liberando quindi una parte della RAM per permettere il corretto funzionamento dei programmi. È chiaro che nel momento in cui si rende necessaria tale operazione, le prestazioni del sistema crollano bruscamente, essendo la scrittura su disco molto più lenta di quella in RAM.

3.7 Sistemi operativi real-time

I sistemi operativi real-time sono quei sistemi che devono eseguire applicazioni real-time, cioè quei programmi che devono rispettare dei vincoli temporali imposti da un sistema esterno.

Una **applicazione real-time** è un programma che risponde alle attività in un sistema esterno entro un intervallo di tempo massimo determinato dal sistema esterno.
In pratica deve soddisfare vincoli temporali imposti da un sistema esterno. Se tali vincoli non vengono rispettati si verifica un malfunzionamento del sistema esterno.

Ad esempio, si consideri un sistema che salva in un file i dati ricevuti da un satellite. Il satellite invia i dati ogni 2ms, per cui il sistema deve essere in grado di elaborare e salvare i dati ricevuti dal satellite in massimo 1.99ms (che è il requisito di risposta), altrimenti si verifica un malfunzionamento come la perdita di dati. Supponendo che un campione è ricevuto al tempo t, il tempo per elaborare e memorizzare i dati (la deadline) è dato da t+1.99ms.

3.7.1 Sistemi hard e soft real-time

Sono stati sviluppati due tipi di sistemi real-time.

Un **sistema hard real-time** è solitamente *dedicato* all'elaborazione di applicazioni real-time e richiede che i suoi vincoli temporali siano rispettati in maniera garantita.

Un **sistema soft real-time** può tollerare che in modo occasionale non vengano soddisfatti i suoi vincoli temporali. In pratica fa del suo meglio per soddisfare il requisito di risposta di un'applicazione real-time ma non garantisce che lo farà sempre. Solitamente lo fa in maniera probabilistica, ad esempio, il 97% delle volte.

3.7.2 Caratteristiche di un sistema operativo real-time

In questi sistemi sono molto importanti il requisito di risposta e la deadline.
Usiamo il termine *requisito di risposta* di un sistema il tempo massimo di risposta entro il quale il sistema può funzionare correttamente.
Indichiamo con *deadline* il tempo entro il quale l'azione dovrebbe essere effettuata.

Caratteristica	Spiegazione
Concorrenza all'interno di un'applicazione	Un programmatore può indicare che alcune parti di un'applicazione debbano essere eseguite in maniera concorrente l'una con l'altra. Il SO considera l'esecuzione di ognuna di queste parti come un processo.
Priorità del processo	Un programmatore può assegnare delle priorità al processo.
Scheduling	Il SO utilizza politiche di scheduling basate su priorità o su deadline.
Eventi specifici per il dominio, interrupt	Un programmatore può definire speciali situazioni nel sistema esterno come eventi, associare a essi degli interrupt e specificare azioni per la gestione di questi eventi.
Predicibilità	Le politiche e l'overhead del SO dovrebbero essere predicibili.
Affidabilità	Il SO garantisce che un'applicazione possa continuare a funzionare anche quando si verificano malfunzionamenti nel computer.

In questa tabella sono riassunte le caratteristiche di un SO real-time.

Le prime tre caratteristiche consentono ad un'applicazione real-time di soddisfare il vincolo di risposta di un sistema, infatti:

- la *concorrenza* permette di eseguire le parti di un'applicazione real-time in maniera concorrente, ovvero come processi separati
- la *priorità* assegnata ai vari processi permette, insieme alla politica di *scheduling basata su priorità*, di sovrapporre l'attività della CPU e dell'I/O

- la politica di *scheduling basata su deadline* è usata dal kernel e permette di schedulare i processi in maniera tale che possano rispettare le rispettive deadline

La possibilità di specificare *eventi specifici* (gestiti mediante *interrupt*) consente ad un'applicazione real-time di rispondere prontamente a speciali situazioni del sistema esterno.

La *predicibilità* delle politiche e dell'overhead del SO consente a uno sviluppatore di calcolare il tempo di esecuzione di un'applicazione nel caso peggiore in modo da determinare se il vincolo di risposta possa essere rispettato in ogni caso. Questa caratteristica vincola i sistemi hard real-time a non usare la memoria virtuale in quanto le prestazioni di quest'ultima non possono essere predette in modo preciso.

L'ultima caratteristica è l'*affidabilità*. Per garantirla, un SO real-time usa due tecniche: la *fault tolerance* e la *graceful degradation*.
Un sistema fault tolerance utilizza risorse ridondanti (in più) per garantire il funzionamento anche in caso di malfunzionamenti, ad esempio può usare due hard disk anche se l'applicazione ne richieda solo uno.
La graceful degradation è l'abilità di un sistema di passare a un livello di servizio ridotto in caso di malfunzionamenti per poi ritornare a pieno servizio una volta risolto il problema. Il programmatore può assegnare priorità alte alle funzioni più importanti in modo che queste vengano eseguite quando il sistema è a servizio ridotto.

3.8 Sistemi operativi distribuiti

Un SO distribuito si compone di diversi computer singoli connessi attraverso una rete. Ogni computer può essere un PC, un sistema multiprocessore o anche un *cluster* (insieme, gruppo) di computer. Quindi in un sistema distribuito esistono molte risorse e un SO distribuito cerca di sfruttarle al meglio.
Il SO esegue le sue funzioni di controllo in diversi computer tra quelli collegati alla rete. Ciò consente l'uso efficiente delle risorse di tutti i computer consentendo ai programmi di condividerle attraverso la rete, l'aumento della velocità di esecuzione di un programma eseguendo le sue parti su differenti computer allo stesso tempo e di fornire affidabilità attraverso la ridondanza delle risorse e dei servizi.

Un **sistema distribuito** è un sistema composto di due o più nodi, in cui ogni nodo è un computer con un proprio clock e una propria memoria, dell'hardware di rete con la capacità di effettuare alcune funzioni di controllo di un SO.

3.9 Moderni sistemi operativi

Gli utenti di un SO moderno spesso eseguono più attività, anche di natura diversa, contemporaneamente. Per questo motivo un moderno SO non può usare una strategia uniforme per tutti i processi ma usare una strategia appropriata per ogni singolo processo.

In pratica un moderno SO controlla un diverso ambiente di elaborazione che ha elementi di tutti i classici ambienti di elaborazione visti fin qui (batch, time-sharing, real-time, distribuiti), e deve pertanto utilizzare tecniche differenti per differenti applicazioni. Utilizza una strategia adattiva che seleziona le tecniche più appropriate per ogni applicazione in base alla sua natura.

CAPITOLO 4. Struttura dei sistemi operativi

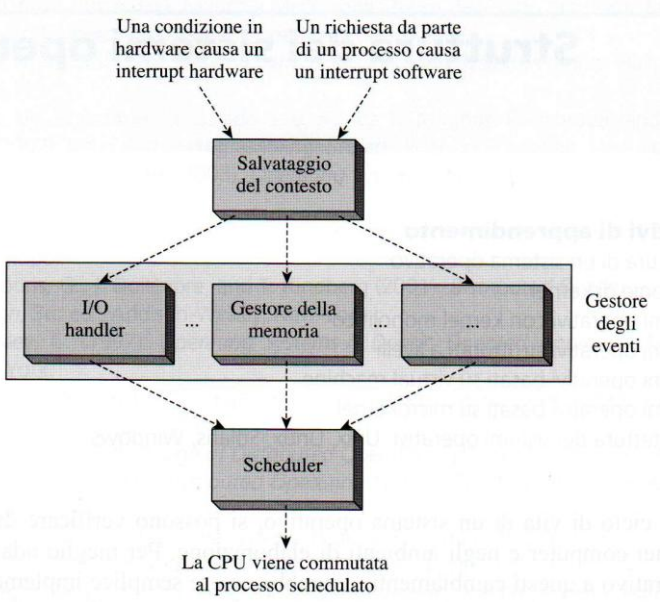
Durante il ciclo di vita di un sistema operativo, si possono verificare diversi cambiamenti nei computer e negli ambienti di elaborazione. Per questi motivi, dovrebbe essere semplice implementare il SO su un nuovo computer e successivamente aggiungervi nuove funzionalità. Si parla pertanto rispettivamente di **portabilità** ed **espandibilità**. Queste due caratteristiche sono diventate requisiti fondamentali a causa del lungo ciclo di vita dei moderni sistemi operativi.

La *portabilità* di un SO si riferisce alla facilità con cui il SO può essere implementato su un computer che ha una differente architettura. L'*espandibilità* di un SO si riferisce alla facilità con cui le sue funzionalità possono essere migliorate per adattarle a un nuovo ambiente di elaborazione.

I moderni SO sono implementati nella forma di nucleo, detto *kernel* o *microkernel*, e costruiscono il resto del sistema operativo usando i servizi offerti dal nucleo. Questa struttura fa si che la portabilità di un SO sia determinata dalle proprietà del suo kernel (o microkernel), mentre l'espandibilità di un SO sia determinata dalla natura dei servizi offerti dal kernel (o microkernel).

In questo capitolo verrà discusso dei modi differenti di strutturare i SO per soddisfare questi due requisiti.

4.1 Funzionamento di un SO



Come già detto nei capitoli precedenti, all'accensione di un computer, la *procedura di boot* analizza la sua configurazione: tipo di CPU, quantità di RAM, dispositivi di I/O e altri dettagli dell'hardware. Successivamente carica una parte del SO in memoria, inizializza le sue strutture dati con le informazioni ottenute e gli passa il controllo del sistema.

Le funzioni di un SO sono implementate da gestori di eventi e sono attivate dalle procedure di servizio degli interrupt. Queste funzioni riguardano principalmente la gestione dei processi, la gestione della memoria, la gestione dell'I/O, la gestione dei file e l'implementazione della sicurezza e protezione.

4.2 Struttura di un SO

4.2.1 Politiche e meccanismi

Nel determinare come un SO debba svolgere una sua funzione si devono considerare due livelli distinti:

- *politica*: è un principio guida in base al quale il SO svolgerà la funzione
- *meccanismo*: è un'azione specifica necessaria per implementare una politica

In pratica, la politica decide cosa dovrebbe essere fatto, mentre un meccanismo determina come dovrebbe essere fatto (e in effetti lo fa). Dunque, la politica decide quale modulo va richiamato e in quali circostanze. Il meccanismo è implementato nel modulo ed esegue un'azione specifica.

Una funzionalità di un SO tipicamente contiene una *politica*, che specifica il principio che deve essere usato per svolgere la funzionalità e alcuni *meccanismi* che effettuano le azioni per implementare la funzionalità.

4.2.2 *Portabilità ed espandibilità dei sistemi operativi*

La progettazione e l'implementazione dei SO coinvolgono grandi investimenti finanziari. Per proteggere tali investimenti, la progettazione del sistema operativo dovrebbe avere un ciclo di vita pari a più di un decennio. Comunque, in tale periodo di tempo, dovrebbe essere possibile adattare un SO ai cambiamenti.

Come già detto, sono due i requisiti da implementare in questo contesto: *portabilità* e *espandibilità*.
Questi due requisiti dipendono da come è strutturato il codice e dalle sue politiche e meccanismi.

Il problema della portabilità del SO è affrontato separando le parti dipendenti dall'architettura da quelle indipendenti. Infatti ci sarebbe un'elevata portabilità se il codice di SO dipendente dall'architettura del sistema fosse di dimensione ridotta.
L'espandibilità di un SO è necessaria per due motivi: per incorporare nuovo hardware in un computer e per fornire nuove funzionalità in risposta a nuove aspettative degli utenti. I nuovi hardware vengono gestiti nella fase di boot oppure installando sulla macchina il driver del dispositivo. I moderni SO utilizzano anche la funzionalità *plug-and-play* mediante la quale è possibile aggiungere nuovo hardware anche quando il SO è in esecuzione.

4.3 Sistemi operativi con struttura monolitica

I primi sistemi operativi avevano una struttura *monolitica*, secondo cui il SO formava un singolo strato software tra l'utente e la macchina (hardware). L'interfaccia utente consisteva in un interprete dei comandi. Sia l'interprete dei comandi che i processi degli utenti richiamavano le funzioni e i servizi del SO attraverso le chiamate di sistema.

Questo tipo di sistemi aveva un portabilità molto limitata poiché il codice dipendente dall'architettura era presente in gran parte nel SO. Inoltre, nella struttura monolitica tutte le componenti del SO erano in grado di interagire con l'hardware, e questo rendeva complicate e dispendiose (anche in termini economici) le fasi di test e debug a causa del **gap semantico**, ossia l'assenza di corrispondenza tra la natura delle operazioni necessarie all'applicazione e la natura delle operazioni fornite dall'hardware.
Questi problemi portarono alla ricerca di modi alternativi di strutturare un SO.

4.4 Sistemi operativi strutturati a livelli

La progettazione a livelli dei sistemi operativi utilizzava il principio dell'astrazione per controllare la complessità della progettazione del SO. Questa progettazione vede il SO come una gerarchia di livelli, in cui ogni livello forniva un insieme di servizi al livello superiore ed esso stesso usava i servizi messi a disposizione dal livello inferiore. Ciò voleva dire che nessun livello poteva essere "saltato".

Un'organizzazione del genere semplifica notevolmente le fasi di test, di debug e di modifica di un modulo.

Nella versione base di questi sistemi, quella composta da due soli livelli, il livello più basso costituisce la ***macchina estesa***, capace di offrire molte funzioni al SO. Il livello superiore, il ***livello top***, sfrutta queste funzioni. Col passare del tempo le strutture a livelli si sono evolute, usando diverse *astrazioni* e un numero differente di livelli.

La progettazione a livelli causa tre problemi.
Il primo è legato al numero di livelli e al fatto che ognuno di essi può "comunicare" solo coi livelli adiacenti, quindi una azione richiesta da un processo utente deve "scalare" i vari livelli partendo da quello in cima fino ad arrivare a quello in fondo. Questo modo di operare genera un elevato overhead.
Il secondo problema è legato all'ordine dei livelli. in alcune circostanze si preferisce suddividere un livello in più livelli in modo tale che i "frammenti" possano essere spostati.

L'ultimo problema riguarda la stratificazione delle funzionalità del SO. Questa si verifica poiché ogni funzionalità deve essere divisa in parti che appartengano a differenti livelli del SO. Inoltre la stratificazione crea problemi anche per l'inserimento di nuove funzioni nel SO che può portare alla modifica di più livelli.

4.5 Macchina virtuale e sistemi operativi

Questi sistemi furono adottati perché diverse classi di utenti hanno la necessità di differenti tipologie di servizi, dunque, utilizzare un unico SO su di un computer può provocare una scarsa soddisfazione da parte di diversi utenti.

I *sistemi operativi basati su macchina virtuale* (SO VM) supportavano il funzionamento di diversi sistemi operativi su un computer simultaneamente, creando una *virtual machine* per ogni utente e permettendo all'utente di eseguire i suoi programmi sul SO di sua scelta nella virtual machine.
Chiameremo ognuno di questi sistemi operativi *SO ospite*, e chiameremo il SO della macchina virtuale *host*.

Il SO VM realizza il funzionamento concorrente dei SO ospite attraverso un'azione simili alla commutazione dei processi, quindi con una procedura analoga allo scheduling. Quando una virtual machine veniva schedulata, il suo SO organizzava l'esecuzione delle applicazioni degli utenti in esso attive.

La distinzione tra modalità kernel e modalità utente della CPU comporta alcune difficoltà nell'uso di un SO VM. Quest'ultimo deve infatti proteggersi dai SO ospite, per cui li deve eseguire con la CPU in modalità utente. In questo modo sia il SO ospite che i programmi utente al suo interno vengono eseguiti in modalità utente, cosa che rende vulnerabile il SO ospite a operazioni non legittime da parte di un processo utente.

4.6 Sistemi operativi basati su kernel

Il *kernel* è il cuore di un SO e fornisce un insieme di istruzioni e servizi per supportare differenti funzioni. Il resto del SO è organizzato come un insieme di *routine non kernel*, che implementano operazioni sui processi e risorse di interesse per l'utente, e un'*interfaccia grafica*.

Il funzionamento del kernel è guidato dagli interrupt, di fatti prende il controllo quando un interrupt gli notifica l'occorrenza di un evento o quando un interrupt viene generato per servire una *system call*. Quando un interrupt viene generato, viene eseguita la funzione di *salvataggio del contesto* e viene invocato un *gestore dell'evento*, che è una routine non kernel del SO.

Le motivazioni storiche di una struttura del SO basato su kernel risiedono nella portabilità del SO e nella semplicità di progettazione e codifica delle routine non kernel. La portabilità si ottiene inserendo nel kernel le parti del codice del SO dipendenti dall'architettura, mantenendo al di fuori del kernel le parti di codice indipendenti dall'architettura.
I SO basati su kernel presentano una ridotta espandibilità poiché l'aggiunta di nuove funzionalità può richiedere cambiamenti nelle funzioni e nei servizi offerti dal kernel.

4.6.1 Evoluzione della struttura basata su kernel

La struttura dei SO basati su kernel si è evoluta per compensare alcuni dei suoi svantaggi. Gli elementi fondamentali di tale evoluzione sono i moduli del kernel caricabili dinamicamente e i driver dei dispositivi a livello utente.

In pratica, un *kernel base* viene caricato in memoria durante la fase di boot, mentre gli altri moduli sono caricati quando le loro funzionalità sono richieste e sono rimossi dalla memoria quando non sono più necessari. In questo modo viene preservata la memoria perché vengono caricati solo i moduli che servono

effettivamente. Inoltre anche l'espandibilità viene migliorata in quanto è possibile aggiungere nuove funzioni al SO modificando i moduli già presenti o, meglio ancora, aggiungendone di nuovi.

Un *driver di dispositivo* gestisce una specifica classe di dispositivi di I/O. Inoltre se operanti in modalità utente consentirebbero facilità di sviluppo, debug e robustezza, poiché sia il codice del kernel che il suo funzionamento non risentirebbero della presenza di tali driver.

4.7 Sistemi operativi basati su microkernel

Abbiamo detto nel paragrafo precedente che mettere tutto il codice del SO dipendente dall'architettura nel kernel fornisce una buona portabilità. Tuttavia, in pratica, anche i kernel contengono del codice indipendente dall'architettura. Questo fa si che la dimensione del kernel sia elevata allontanando l'obiettivo della portabilità.
Inoltre, per incorporare nuove funzionalità spesso è necessario effettuare modifiche al kernel, e ciò porta a poca espandibilità.

Il microkernel fu sviluppato negli anni '90 per superare i problemi relativi alla portabilità, all'espandibilità e all'affidabilità del kernel. Un *microkernel* è il nucleo essenziale del codice di un SO. E' di dimensione ridotta, contiene pochi meccanismi, supporta un piccolo numero di system call e non contiene nessuna politica.
I moduli contenenti le politiche sono implementati come *processi server*, ovvero semplici processi che non terminano mai; possono essere cambiati o sostituiti senza coinvolgere il microkernel, fornendo in tal modo elevata espandibilità al SO. I processi server e i programmi utente operano al di sopra del microkernel.