Sistemi Operativi

Luca Tagliavini

25 febbraio 2022

Indice

1 Definizione di sistema operativo

Il sistema operativo realizza un'atrazione che definisce il concetto di processo come unità che può svolgere istruzioni sulla parte della memoria ad esso associata o invocare l'aiuto del sistema operativo stesso tramite system call. Il sistema operativo dunque è un livello di astrazione sul quale i programmi user mode giaciono e che interagisce a bassissimo livello con l'hardware e offre una interfaccia per consentire una comunicazione tra processi e kernel. Il sistema operativo decide come allocare le risorse quali tempo di esecuzione e memoria ai processi, gestendo in particolare alcuni primitivi di concorrenza.

I principali obbiettivi del sistema operativo sono l'*efficienza* e la *semplicità*. Il sistema operativo nel gestire i processi deve lasciare a sua volta il controllo ai processi stessi in attesa di una system call o di un interrupt. Il sistema operativo lascia dunque controllo ai programmi in una modalità limitata qualè la *user mode* in attesa che l'**hardware** lo interpelli quando se ne ha bisogno.

Rendendo le *system call* l'unica interfaccia di comunicazione tra i processi e il sistema sottostante si astraggono i programmi dall'hardware e si rende il software più portabile.

1.1 Syscall minime

Ecco alcune tra le più base system call che sono necessarie in un sistema moderno:

1. esecuzione di programmi: poichè il sistema deve gestire processi le chiamate più essenziali sono quelle che consento di creare altri programmi (i.e. `fork`) e eseguire altri tool (i.e. `exec`)
2. accesso *semplice* a dispositivi di IO e filesystem
3. accesso a dispositivi di networking
4. accesso al sistema e gestione di account e delle siorse

Inoltre è vitale che tutte queste system call abbiano modo di segnalare all'utente l'occorrenza di errori nell'esecuzione.

Poichè i processi in *user mode* hanno una grande libertà nelle istruzioni del processore che possono eseguire il sistema deve anche gestire istruzioni che generano errori a livello di processore, come divisione per zero o accesso a una locazione di memoria non acconsentita. Ad esempio in unix gli errori vengono segnalati al kernel e gestiti tramite l'invio di segnali al processo desiderato.

1.2 Multiprogrammazione

Il processore non viene lasciato inattivo durante le lunghe operazioni di Input/Output ma bensì questo tempo viene sfruttato per svolgere altri job in attesa.

In questo modo non vengono sprecate risorse preziose quali cpu time e memoria, in quanto se ne può massimizzare l'utilizzo svolgendo tutti i job possibili nei momenti di attesa.

1.3 Time sharing

È l'estensione naturale della multiprogrammazione: indipendentemente dalla presenza di chieste I/O ad ogni processo in esecuzione viene data una quantità di tempo e viene messo in pausa se ancora non ha eseguito una chiamata I/O in questo timeframe. Così il tempo è distribuito in modo più equo tra tutti i processi in esecuzione e viene garantito l'avvicendamento tra i processi in tempi prestabiliti. Per questa migliora è necessario un dispositivo I/O detto *interval timer* che invia interrupt ad un rateo programmabile.

1.4 Memoria virtuale

Con la possibilità di eseguire molti processi contemporaneamente si nota che grande parte della memoria occupata non viene letta/scritta, e dunque si decide di fingere di avere più memoria e spostare quella meno usata su dispositivi secondari non volatili.

1.5 Parallelismo

I sistemi operativi possono (o no) supportare il parallelismo, ovvero l'abilità di svolgere più azioni contemporanemanete sfruttando una architettura a più core o la disponibilità di più CPU. Quando questa funzionalità è supportata le varie implementazioni si distinguono per la loro tipologia:

- SIMD: Single instruction, Multiple Data. La stessa istruzione viene svolta su una grande mole di dati, ad esempio nelle GPU.
- MIMD: Multiple instruction, Multiple Data. La CPU può eseguire programmi totalmente diversi all'unisono.

Una diversa nomenclatura deriva dalla qunatità di parallelismo: esistono processori con pochi core e dunque denominati *a basso parallelismo* o sistemi con una grandissima quantità di core a discapito del clock che vengono chiamati *massicciamente paralleli*.

Solitamente si hanno due standard di parallelismo:

1. sistemi *tightly* coupled. Si hanno pochi processori e la memoria/bus sono condivisi (ciò ne limita la scalabilità e costituisce un collo di bottiglia su configurazioni estreme).
2. sistemi *loosely* coupled. Tanti processori con memoria privata che comunicano tramite canali.

Essendoci un unico kernel e più processori, bisogna gestire il caso in cui più processori ricevono gli interrupt nello stesso momento, e bisogna dunque coordinarsi affinchè il codice kernel mode venga eseguito in mutua esclusione.

In altre occasioni si può pensare di assegnare il sistema opeartivo (kernel) ad un solo processore che denomineremo *master* e lasciare il lavoro a tutti gli altri processori denominati *slave*. Questo approccio è più comune in calcolatori estremamente grandi.

1.6 Sistemi distribuiti

Si prendono vari sistemi completi, con proprio hardware dedicato e sistema operativo, che venogno collegati tramite una rete e che lavorano all'unisono su un dato problema dividendo il lavoro. Questo approccio richiede ovviamente la scrittura di software appositamente pensato per questo utilizzo.

Un esempio di sistema distribuito è quello di ateneo, che condivide e sincronizza le cartelle home, i pacchetti di sistema attraverso tutto il sistema distribuito.

1.7 Sistemi in realtime

Sono sistemi in cui il risultato computazionale dipende dal tempo di esecuzione. Spesso si desidera un risultato entro un determinato tempo. Nei sistemi *hard real-time* il mancato rispetto dei vincoli temporali causa fallimenti *hard*, catastrofici. Se i vincoli non sono così stringenti si adatta una strategia *soft real-time*.

2 Architettura Hardware

Seconodo l'architettura di Von Neumann il processore ha accesso alla memoria nel quale sia i programmi e i dati sono contenuti e comunica con dispositivi esterni tramite un bus e gli interrupt.

2.1 Interrupt

Un interrupt è un comando speciale che consente di interrompere l'esecuzione dell'attuale istruzione da parte della CPU. Nella pratica vengono usati per restituire il controllo al sistema operativo che deve svolgere compiti importanti o gestire dispositivi terzi. Sono introdotti per evitare il polling e il busy waiting e migliorare le performance.

Gli interrupt possono essere mascherati se la CPU sta svolgendo compiti importanti, come quando ad esempio si è in kernel mode e non si vuole bloccare il codice del kernel (si esegue essenzialmente il codice del kernel in mutua esclusione dagli interrupt).

Gli interrupt hardware generati dai device verranno chiamati *interrupt*, mentre quelli generati dal software in esecuzione venogno detti *trap*.

2.1.1 Interrupt software (trap)

Sono interrupt generati dal processo in esecuzione in modalità utente, che vengono generati quando il codice esegue una istrzione errata o quando si vuole invocare una system call.

2.2 Cosa succede quando avviene un interrupt

Quando il processore riceve un interrupt sospende il processo corrente, salva tutti i dati che verranno sovrascritti dal passaggio in kernel mode, e ritorna il controllo al kernel. Una volta gestito l'interrupt il controllo viene restituito al processo bloccato o ad un altro (nel caso di scheduling).

I sistemi operativi moderni si dicono *interrupt-driven* e si affidano dunque agli interrupt per la maggior parte delle proprie funzionalità. Esistono due approcci possibili:

- disabilitazione degli interrupt: gli interrupt venogno disabilitati completamente mentre si è nella parte di *interrupt handling* in kernel mode. Di conseguneza gli interrupt vengono gestiti in modo sequenziale e l'implementazione appare più semplice, tuttavia si perde la relazione temporale tra i vari interrupt che potrebbe essere importante.

- interrupt innestati: si deve tener traccia di molti più dettagli per gestire la possibilità che gli interrupt possano interrompere il codice stesso dell'interrupt handler. Tuttavia in questo modo si possono gestire interrupt annidati (con priorità maggiore di quello intercettato) che vengono processati appena arrivano.

2.3 IO con interrupt o diretto

Implementare le operazioni di Input/Output tramite interrupt in modo che la CPU parli con il controller attrverso il bus e gli iterrupt. Tuttavia, per quanto questa tecnica sia più efficiente del busy waiting, rimane comunque troppo lenta. Si utilizza allora l'*accesso a memoria diretto*.

Si decide allora di specificare ad un controller DMA (Direct Memory Access) l'indirizzo di memoria a cui leggere o scrivere i dati ed esso può tranquillamente scrivere/leggere dalla memoria centrale i dati di cui si ha bisogno. Questo approccio è sicuramente il più efficiente ma può creare conflitto sul bus, semplificando tuttavia il controller e sottranedo lavoro inutile alla CPU.

2.4 Memoria e MMU

Il processore può accedere alla memoria tramite funzioni LOAD/STORE e specificando l'indirizzo logico. L'indirizzo logico viene poi tradotto dalla MMU in un indirizzo fisico. Questo step viene aggiunto per sicurezza e per facilitare la comunicazione tra i device attraverso il bus.