1 Introduzione

Un Sistema operativo è un insieme di programmi con solo scopo di coordinare e rendere efficente e allocare risorse il livello hardware. Il sistema operativo inoltre offre efficenza e astrazione, indirettamente proporzionali

1.1 Principali evoluzioni dei sistemi operativi

1.1.1 1a generazione

Primi calcolatori a valvole termoioniche con esecuzione da console, implementano la gestione delle prime periferiche e lo sviluppo dei software

1.1.2 2a generazione

Con i calcolatori a transistor, nasce il batching (batch = lotto), ossia il raggruppamento dei job simili. Grazie all'automatic job sequency, è il SO e non un operatore a passare da un job al successivo. Questa sequenzializzazione è gestita tramite il Job Control Language.

Per la prima volta abbiamo la sovrapposizione delle operazioni I/O e di elaborazione, per gestirle vengono implementati due meccanismi asincroni: Interrupts e DMA

Concettualmente, quando la CPU riceve un segnale di interrupt, interrompe il job che stava eseguendo e salva lo stato, effettua l'azione richiesta dal'I/O e poi riprende il suo lavoro. Questo risulta poco utile dato che sarebbe la CPU a venire continuamente interrotta, influendo sulla capacità di svolgere un job rapidamente

In alternativa all'interrupt, viene implementato un controller HW che si occupa del trasferimento I/O invece di appesantire la CPU. Questo metodo è chiamato DMA o Direct Memory Access

Per la gestione di sovrapposizioni nello stesso job, vengono usufruite le tecnologie di buffering e spooling. In particolare, il Simultaneous Peripheral Operation Online (o SPOOL), usufruisce di un disco magnetico usato come grande buffer per tutti i job. Richiede un qualche tipo di job scheduling

1.1.3 3a generazione

Introduzione della multiprogrammazione (sfruttamento dell'attesa I/O per l'esecuzione di un'altro job) e dei circuiti integrati. Nascono anche i primi sistemi Time sharing (o multitask/multi utente), questo però ha introdotto nuovi problemi di protezione della memoria, CPU e I/O. I meccanismi sviluppati sono l'esecuzione Dual mode (USER e SUPERVISOR/KERNEL) alle risorse I/O, dove tutte le operazioni I/O sono privilegiate e gestite dal SO, l'associazione di un registro per ogni processo (protezione della memoria) e l'accesso di ogni job alla CPU limitato da un timer

1.1.4 4a generazione

Nascita di nuovi tipi di SO e specializzazioni: per PC, di rete, distribuiti, mobile, embedded...

2 Componenti di un SO

2.0.1 Gestione dei Processi

Il SO è responsabile dei processi (programmi in esecuzione) in quanto creazione, distruzione, sospensione, riesumazione e sincronizzazione per la communicazione tra processi

2.0.2 Gestione della memoria primaria

La memoria primaria contiene i dati condivisi dalla CPU e dispositivi I/O. Il sistema operativo gestisce lo spazio, allocazione e rilascio della memoria, e scelta del processo da caricare in memoria

2.0.3 Gestione della memoria secondaria

Il SO gestisce oltre che gestione e allocazione, anche lo scheduling degli accessi al disco

2.0.4 Gestione dell'I/O

Il SO gestisce un sistema per l'accumulo degli accessi ai dispositivi e gestisce driver specifici o generici per i dispositivi

2.0.5 Gestione dei file

Il SO gestisce creazione e cancellazione di file e dyrectories, operazioni primitive, gestione dei backup e corrispondenza tra file e spazio fisico

2.0.6 Protezione

gestione degli accessi autorizzati e non, degli strumenti per la verifica delle policies e definizione dei controlli da imporre

2.0.7 Sistemi distribuiti

Corrispondono a una serie di calcolcatori con clock e memorie diverse, generalemente usano una connessione gestita dal SO sulla rete

2.1 Interfacce

2.1.1 Interprete dei comandi

Chiamato anche Shell, offre all'utente i comandi con la quale interagire con il SO. Questi programmi rientrano nella gestione dei file, status del sistema, strumenti per la programmazione...

Esisteno vari modi per interfacciarsi con i programmi del SO: GUI, CLI e Batch. Queste sono definite interfaccie utente

2.1.2 System calls e API

Le funzioni di un SO vengono definite system calls sono utilizzate dai processi tramite le API. Quest'ultime permettono di interporsi e mascherare l'implementazione dei servizi del SO.
Per ogni system call viene associato un numero, che viene usato dall'interfaccia per comunicare col SO. A volte queste call contengono dei parametri. quest'ultimi vengono passati al SO in 3 modi:

- Tramite registri
- Tramite stack del programma
- Tramite tabella in memoria, passando poi l'indirizzo della tabella tramite registro o stack

In dettaglio, la comunicazione tramite stack:

1. vengono salvati i parametri sullo stack e chiamata la funzione dalla libreria
2. viene caricato il numero della system call in registro ed eseguita la Trap
3. selezionata la system call indicata nel registro, viene gestita appropriatamente
4. infine ritorno della funzione di libreria e al codice utente

Le API più comuni sono Win32 per windows e POSIX (Portable Operating System Interface for Unix) API per sistemi unix, linux e Mac OS X

3 Architettura

Nella creazione di un SO è inportante distinguire policies e meccanismi, ossia cosa e come deve essere fatto qualcosa. Altri principi importanti sono il KISS (Keep It Small & Simple) e il POLA (Principle Of the Least Privileges), ossia l'assegnazione dei privilegi minimi

3.1 Strutture di un SO

3.1.1 Sistemi monolitici

Non presentano nessun tipo di gerarchia, il software comunica direttamente con l'HW. Questo però comportava una forte dipendenza dalla tipologia di HW

3.1.2 Sistemi a struttura semplice

Sviluppati per ridurre i costi di sviluppo, implementano gerarchie molto flessibili. Come esempio abbiamo MS-DOS e UNIX originale

MS-DOS offre nel minimo spazio, più funzionalità possibili. Usa una struttura minima, ma con interfaccie e livelli poco ben definiti. UNIX invece aveva una base limitata a causa delle funzionalità HW

3.1.3 Sistema a livelli

Sono organizzati gerarchicamente, con la user interface al livello più alto e l'HW al livello più basso, e implementano un modello a layers. Questo permette una maggiore modularità, ma con minor efficenza e portabilità.

Un esempio è THE, SO accademico sviluppato da Dijkstra

3.1.4 Sistemi basati su kernel

Evoluzione del sistema a livelli, divide le funzionalità in servizi kernel e non-kernel. Questo allieva gli svantaggi e i vantaggi del sistema a livelli

3.1.5 Micro-kernel

Evoluzione del kernel, ma che contiene le funzioni strette necessarie, come Inter Process Communication (IPC), gestione memoria e scheduling CPU

3.1.6 Virtual Machine

È un approccio a livelli dove HW e VM vengono trattati come HW, in modo da offrire timesharing multiplo. Il concetto chiave è la separazione tra multiprogrammazione (VM) e presentazione (SO).

Questo metodo permette la protezione completa del sistema, ottimizzazione delle risorse, portabilità e la possibilità di installare più SO sulla stessa macchina. D'altro canto però risulta più pesante a livello di prestazioni e richiede la gestione della dual mode virtuale, ossia la VM e VMM funzionano una in modalità kernel e l'altra in modalità user

Esistono due tipi di hypervisor, il type-1 che si posiziona tra hardware e VM in modalità kernel, e il type-2, che si appoggia al SO dell'host e permette in contemporanea l'esecuzione di altre applicazioni. Il type-1 permette anche di implementare hypervisors micro kernel o monolitici.

3.1.7 Sistemi client-server

Molto efficente nei SO distribuiti, si basa sulla gestione delle comunicazioni da parte del kernel e il resto delle funzioni nei processi utente

4 Processi e Threads

4.1 Concetto di processo

Un processo è un istanza di un programma. In un sistema multiprogrammato, i processi, devono avanzare in modo concorrente.

A livello di memoria il processo è composto in un immagine che condtiene codice, attributi, dati (variabili globali) e nello stack salva variabili locali e chiamate, mentre nell'heap la memoria allocata dinamicamente. Il processo è rappresentato dal Process Controll Block (PCB)

4.2 Stato di un processo

Lo schema di evoluzione di un processo è formato dagli stati "pronto", "in attesa" e "in esecuzione"

4.2.1 Scheduling

Per garantire multiprogrammazione e time-sharing, c'è bisogno di selezionare quale processo eseguire e in che ordine. Questo è il compito dei schedulers che si occupano di selezionare quale processi trasferire in coda (Long-term) e quali far eseguire dalla CPU (Short-term). I long-term schedulers sono responsabili del grado di multiprogrammazione

Esistono due code di scheduling: la ready queue che determina i processi pronti all'esecuzione e la coda a un dispositivo.
Quando un processo viene eseguito nella ready queue viene definito dispatched. Da qui può proseguire in diversi modi:

- viene inserito nella coda di un dispositivo in attesa di I/O
- impiega troppo tempo di esecuzione e viene reinserito nella ready queue
- crea un processo figlio e ne attende l'esecuzione
- il processo attende un evento

L'operazione di dispatch determina un cambio di contesto (o caricamento di un nuovo processo), il passaggio alla user mode o un salto a un'istruzione da eseguire. Il cambio di processo implica la registrazione dello stato del processo (salvataggio del PCB) ed è un semplice peso al calcolatore

4.3 Operazioni sui processi

Un processo può generare un processo figlio in modo sincrono, dove si attende la terminazione dei figli, e asincrono, o con concorrenza.
In UNIX abbiamo le funzioni fork (duplicazione), exec (programma diverso) e wait (esecuzione sincrona).

Un processo invece termina alla fine naturale della sua esecuzione, oppure forzatamente da parte del processo padre o del SO. Questo può essere causato per eccessi nell'uso delle risorse, errori o chiusura del programma/processo padre.

4.4 Threads

Un thread è l'unità minima di utilizzo della CPU. Rispetto a un processo, viene associato lo stato di esecuzione, l'insieme dei registri, il program counter...
Nei SO moderni è possibile supportare il multithreading per un singolo processo, ne consegue la separazione tra flusso di esecuzione (thread) e spazio di indirizzamento.

I vantaggi del multithreading sono la riduzione del tempo di risposta, la condivisione delle risorse e la scalabilità (parallelismo). Inoltre, la creazione, terminazione e context switch tra processi risulta più lento rispetto che tra threads

NB: lo stato del thread può divergere da quello del processo

Un thread può essere livello utente (user-level), dove la gestione è affidata alle applicazioni, o a livello kernel (kernel-level), gestito dal kernel e invocato tramite system call.
I vantaggi dello user-level thread sono l'efficienza, data dall'assenza del mode switch tra user e kernel, portabilità e la possibilità di usare uno scheduling implementato nell'applicazione. Tutto questo al costo dell'impossibilità di accesso a multiprocessori e il blocco del processo in caso di blocco del thread.