1 Lezione del 23-09-25

1.1 Introduzione

Il corso di sistemi operativi riguarda l'ultima parte dello studio delle "architetture", che è partita con l'implementazione hardware in reti logiche, è continuata con lo studio del kernel in calcolatori elettronici, e termina appunto con lo studio dei sistemi operativi. Nello specifico, si considereranno sistemi operativi derivanti dalla famiglia **UNIX**.

Argomento del corso è la conoscenza di tecniche di programmazione usate nello sviluppo del sistema operativo **multiprogrammato** (più *processi* o più *thread*), con riferimento particolare alla programmazione **concorrente**, lo **scheduling** e la **memoria virtuale**.

Il corso mira a dare informazioni generiche utili allo studio di qualsiasi sistema operativo (anche non direttamente derivante da UNIX), in primis rivolte alla compresione di *come mai* una certa soluzione ad un problema è migliore di altre, e quali sono le tecniche che ci permettono di sviluppare soluzioni miglori.

1.1.1 Sistemi embedded e in tempo reale

Ci interesseremo anche ai sistemi **embedded** e sopratutto sistemi in **tempo reale**. Questi rappresentano sistemi *special-purpose* (per distinguere dai sistemi a scopo generale, *general-purpose*), dove dobbiamo rispettare coi nostri algoritmi di scheduling date **scadenze** temporali.

1.1.2 Programmazione concorrente

Con programmazione concorrente ci riferiamo alle tecniche che ci permettono di gestire più processi che si contendono le solite risorse, adottando politiche più o meno *eque* per i processi, o magari privilegiandole alcune. Obiettivo fondamentale sarà comunque quello di evitare *stalli* o **deadlock** dati da risorse occupate.

1.1.3 Programma del corso

Il corso è strutturato negli argomenti:

- **Concetti introduttivi** su sistemi operativi, architetture hardware e relativi cenni storici, in particolare ci interessano dettagli come la gestione della *pila* e le *interruzioni*;
- **Processi** e la loro gestione, inclusi gli algoritmi di *scheduling preemptive* e *non preemptive*, *prioritari* e *non prioritari* (FCFS, SJF, SRTF, RR). Inoltre si tratta la schedulazione nei sistemi **hard real-time** (RM, RDF);
- **Sincronizzazione** dei processi, quindi *programmazione concorrente*, *competizone* su risorse, e *scambio di informazioni fra processi* (IPC);
- Gestione della **memoria**, quindi *memoria virtuale*, *segmentazione* e *paginazione*;
- Gestione dei **dispositivi** di I/O, cioè i *driver*;
- **File system** su disco, cioè i componenti software che permettono la gestione di strutture di *file*, nella loro struttura sia *logica* che *fisica* di implementazione nei driver e nel sistema operativo;

- **Sicurezza**, quindi meccanismi di *protezione* fra processi, controlli sugli *accessi* sia in memoria che al filesystem, con riferimento al modello della *matrice degli accessi*.

1.1.4 Meccanismi e politiche

Una prima distinzione che possiamo fare è quella fra **meccanismo** e **politica**.

- Si dice **meccanismo** ciò che effettivamente implementato, in maniera sufficientemente veloce e compatta, nel kernel, per fornire il cosidetto *supporto architetturale* a risorse, dispositivi, ecc...
- Nei sistemi operativi ci interessano invece principalmente le **politiche**, cioè decisioni (che vanno poi implementate) su come gestire *a priori* date risorse, dispositivi, ecc...

1.1.5 Sistemi operativi

Un **sistema operativo** è in primo luogo un *programma software* che ha il compito di fare da intermediario fra l'*utente* e l'*hardware* di un calcolatore.

Far fronte ai bisogni dell'utente significa gestire e consentire l'accesso delle risorse ai *processi* di cui l'utente necessita. In questo individuiamo come obiettivi del sistema operativo:

- Eseguire i *programmi utente*;
- Rendere il sistema facile da usare;
- Utilizzare l'hardware in maniera efficiente.

1.1.6 Programmi

I programmi con cui abbiamo a che fare sono per noi *liste di istruzioni* (tralasciando il fatto che queste siano codificate in linguaggio macchina o in un suo linguaggio mnemonico), ordinate ma che possono presentare salti condizionali che cambiano il normale *flusso di esecuzione*.

Il **comportamento** e quindi i **risultati** di un programma dipendono sì dal codice, ma anche dai **dati** in ingresso allo stesso. In questo possiamo dire che il programma non esiste mai da solo ed è solo la parte **statica** di un processo.

1.1.7 Risorse

Iniziamo quindi a vedere quelle che sono le risorse che dobbiamo fornire ai programmi. Il modello che adottiamo è il più diffuso oggigiorno, cioè quello di *Von Neumann*. Questo modello comprende, a grandi linee:

- La **CPU** o *processore*, che ha il solo scopo di prelevare ed eseguire le istruzioni in maniera sequenziale, alterando il suo flusso come già detto solo nel caso di istruzioni condizionali, o come vedremo nel caso di interruzioni o altre situazioni simili;
- La **memoria principale**, che nell'architettura di Von Neumann contiene sia i dati che il programma in esecuzione, e che deve essere capace di fornire su richiesta alla CPU.

Ricordiamo che questa è spesso *volatile*, cioè i suoi contenuti vengono sostanzialmente invalidati allo spengimento della macchina. Potremmo interrogarci sul motivo di tale decisione: principalmente diciamo che le ragioni sono economiche, in quanto mantenere l'informazione per lunghi periodi di tempi è solitamente più costosto e delegato a dispositivi (come i dischi) che offrono risparmi in cambio di grandi tempo di accesso (inusuali sulla memoria principale);

- Altre **risorse** che si aggiungono a CPU e memoria, comunque indispensabili per eseguire qualsiasi istruzione. Queste sono:
  - I **dispositivi**, che comprendono ad esempio la memoria di *archiviazione* (il **disco**) e le *periferiche* di interfaccia con l'utente;
  - Le risorse **logiche**, cioè determinate strutture dati in memoria che devono essere fornite in maniera più o meno esclusiva ai processi. Anche gli stessi *file* del *file system* sono risorse logiche.

Risulta chiaro come la gestione delle risorse hardware e logiche è fondamentale anche alla **portabilità** dei programmi, che magari vogliono avere accesso a funzionalità simili su più sistemi operativi (accesso alla tastiera, ai file, ecc...), senza dover necessariamente conoscere l'implementazione interna di tali sistemi operativi.

Abbiamo quindi l'obiettivo di implementare tutte quelle **interfacce** di cui il programma bisogna per presentare all'utente le sue funzioni. Questo include le interfacce grafiche, audio, ecc... per la realizzazione di ambienti visuali e interattivi nei sistemi moderni.

Dal nostro lato, quello del *sistema*, vorremo che le soluzioni tecniche che adottiamo non impattino in maniera negativa le prestazioni o comunque il funzionamento dei programmi che mandiamo in esecuzione.

1.1.8 Struttura stratificata del S/O

La struttura di un sistema operativa può dividersi in più livelli, fra cui:

- Il livello **hardware**, fornito come già detto da risorse come:
  - La **CPU**;
  - La **memoria principale**;
  - Le **periferiche**, fra cui *video*, *disco*, *interfacce di rete*, ecc...

  Il livello hardware offre la cosiddettà *intefaccia hardware*, data dalle specifiche secono cui interagiamo con i dispositivi hardware stessi;
- Il livello **sistema operativo** (o *S/O*), che implementa la gestione delle risorse che studieremo nel corso, e offre a sua volta altre risorse logiche. In particolare notiamo:
  - Gestione della **CPU**;
  - Gestione della **memoria**;
  - Gestione del **file system** e quindi dei *file*;
  - Gestione dei **dispositivi** attraverso i *driver*.

Questo livello offre la sua interfaccia attraverso le **chiamate di sistema** o *primitive*, che implementano una certa **API** (*Application Programming Interface*) secondo le quali i programmi utente delegano all'S/O operazioni che non potrebbero normalmente portare avanti da soli (accesso a risorse, schedulazioni temporali, ecc...);

- Il livello delle **applicazioni**, che comprende i programmi utente.

Questa gerarchia implica chiaramente che ogni livello non conosce nulla riguardo al livello successivo, ma si preoccupa solo di fornire un'*interfaccia* conforme alle specifiche. A questo punto è compito del livello successivo stesso rispettare l'interfaccia e farne uso per i suoi scopi.

Il programmatore di **sistema** interagisce con i livelli *hardware* e *S/O*, mentre il programmatore di **applicazioni** interagisce con i livelli *S/O* e *applicazioni*.

Compito dell'*API* è quello di generare per i programmatori di applicazioni una macchina *astratta* più semplice da usare, più efficiente e più sicura (cioè realizzare gli obiettivi che ci eravamo posti in 1.1.5). Ricordiamo che per noi sicurezza significa *modelli* che controllano l'accesso da parte dei processi (altresì **soggetti**) alla memoria, e più in generale a tutte le risorse sistema (altresì **oggetti** dei programmi).

1.1.9 Definizione di S/O

Iniziamo a definire più nei dettagli cos'è un S/O.

- Un S/O è un **allocatore di risorse**, cioè gestisce *tutte* le risorse, e decide tra richieste conflittuali di accesso a tali risorse (inviate dai vari processi) al fine di garantirne un uso equo ed efficiente.
- Un S/O è però anche un **programma di controllo**, che controlla l'esecuzione dei programmi e lo stato dele risorse per prevenire usi impropri e stati inconsistenti.

Ricordiamo che in ogni caso l'unico programma effettivamente in esecuzione in ogni momento sulla macchina reale è il **kernel** cioè nucleo.