INTRODUZIONE SISTEMI OPERATIVI

Trascrizione lezione *Malchiodi Riccardo*

I programmi sono eseguiti dall'hardware, meglio dire sono eseguiti dalla **CPU.**
La CPU fa fetch decode execute

- **Fetch:** pesco dalla ram la prossima istruzione da eseguire.
- **Decode:** leggo qualcosa dalla ram e cerco di capire di cosa si tratta, più precisamente come si è visto ad ADC la decode decodifica la prossima istruzione da eseguire.
- **Execute:** eseguo.

Da che indirizzo faccio il fetch? In generale tutte le CPU hanno un registro, alcuni la chiamano **PC** (program counter) altri **IP** (instruction Pointer), che indica l'indirizzo della prossima istruzione da eseguire.
La CPU fa il fetch all'indirizzo indicato dal **PC** o da **IP** (stessa cosa). Nel caso **x86** (processori più comuni) viene chiamato **IP.**
Come si fa a sapere se è successo qualcosa nel mondo esterno? attraverso **Interact**, ci sono uno o più segnali sulla CPU di interruzione in questo modo le periferiche comunicano alla CPU cosa è successo. Quando viene attivata una linea di interruzione la CPU invece di continuare FETCH, **DECODE** ed **EXECUTE** darà una risposta all'interact, questa risposta tipicamente consiste di passare in una modalità privilegiata ed eseguire interact handler (gestore interruzione) che non è altro che un pezzo di codice tipicamente all'interno del S.O. che decide cosa fare in risposta a questa interruzione.
Quanti programmi per volta? In base ai **core**, se abbiamo una CPU da 1 core potremmo eseguire solo un'istruzione. Se avessimo 8 core allora possiamo eseguire 8 istruzioni contemporaneamente.
Un **core** è l'unità di elaborazione centrale di un processore (CPU).
Il sistema operativo gestisce l'hardware, inclusi la CPU e le periferiche, e si occupa di assegnare la CPU ai vari programmi che desideriamo eseguire. Questo avviene suddividendo il tempo della CPU in fette, che vengono allocate a ciascun programma. Una volta terminata una fetta di tempo, la CPU viene riassegnata a un altro programma. Questo processo ci dà l'illusione che tutti i programmi vengano eseguiti contemporaneamente, anche se in realtà l'elaborazione avviene in modo alternato.
Un programma è un concetto statico, un insieme di istruzioni (esempio stupido: libro da cucina). Quando lo mando in esecuzione diventa un processo, processo è un programma in esecuzione.
I programmi per essere eseguiti devono stare in **RAM,** finchè il programma sta su disco non possiamo eseguirlo.
Quando clicchiamo sull'icona di un app, questa applicazione deve essere portata in **RAM** perchè la **CPU** deve essere in grado di eseguirla, ovviamente tutto ciò avviene grazie al **S.O.**
Un'altra cosa che fa il **S.O.** è quella di rendere isolati I processi.
Ogni processo ha l'illusione che tutta la memoria sia per lui , questa astrazione si chiama spazio di indirizzamento ed è un'astrazione della **RAM.** Questa illusione è creata dal **S.O**.
Sistema operativo si intende un nucelo (kernel) e tutte le utilty che ci stanno intorno.
Un software particolare, il **kernel del SO** si occupa di gestire e virtualizzare le risorse come **CPU,** memoria ecc.
Bisogna dare questa illusione dei processi in modo efficienti, per farlo c'è bisogno di supporto hardware. Senza un supporto hardware per esempio alla paginazione sarebbe "impossibile" isolare I processi. Si potrebbe fare ma costerebbe troppo.

I nostri programmi non si devono preoccupare di parlare direttamente con l'hardware perchè ci pensa il S.O.
I driver sono pezzi di software che si inseriscono nel sistema per imparare ad "usare" una periferica. Sono necessari, ma sono anche un problema di sicurezza in quanto **I driver** girano insieme al S.O. e se c'è un bug all'interno dei driver potrebbe causare problemi al S.O.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int num;

int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: %s <value>\n", *argv);
                return EXIT_FAILURE;
        }
        num = atoi(argv[1]);
        for(;;++num) {
                printf("(pid:%jd) &num=%p num=%d\n", (intmax_t)getpid(), (void*)&num, num);
                sleep(1);
        }
}
```

Cosa fa questo programma ? Prima di tutto guarda se argc è diverso da 2, argc è il numero di argomenti che arriva dalla riga di comando + 1, nel senso che l'argomento 0 per convenzione è il nome del programma. Questa condizione quindi controlla se questo programma è stato chiamato con un argomento oppure no.
Se non è stato chiamato con un argomento stampo un errore ed esco dal programma. Printf mi permette di stampare una stringa, **stderr** si occupa di gestire errori.
***argv** sarebbe **argv[0]**, stiamo stampando quindi "usage + nome programma" + un numero che è rappresentato appunto da argv.
atoi è ask to int, se non sappiamo cosa voglia dire usiamo il manuale tramite il **comando "man"** in questo modo:

man atoi

atoi converte da stringa ad intero, quindi se avessi atoi("3") ottengo 3 come int.
Ogni processo viene identificato all'interno del sistema da un pid (process identifier) ed è un intero non negativo che identifica un particolare programma in esecuzione.
In ogni istante tutti i pid sono diversi.

Esempio di output che si ottengono con questo programma:

```
gio ~ didattica so 01_ … c-examples (e) py3 master 1 ./mem_example 10
(pid:9808) &num=0x80e4f1c num=10
(pid:9808) &num=0x80e4f1c num=11
(pid:9808) &num=0x80e4f1c num=12
(pid:9808) &num=0x80e4f1c num=13
(pid:9808) &num=0x80e4f1c num=14
```

Il pid cambia sempre ogni volta che lanciamo il programma.

Ogni processo ha la sua tabella delle pagine, quando entro in gioco la MMU a fare paginazione ogni processo ha la sua.
Gli indirizzi &num sono indirizzi logici che verranno tradotti in indirizzi fisici. Le traduzioni di

un certo processo saranno in un certo modo mentre la traduzione di un altro processo sarà diverso e così via.
Quando un processo è in esecuzione sta usando i registri della CPU. Quando mandiamo in esecuzione un altro processo i valori che erano nei registri del primo processo dovranno essere salvati da una parte e dentro ai registri della CPU ci finiranno i valori del nuovo processo.
Se facciamo questa operazione abbastanza velocemente l'illusione di noi umani è quella di mandare in esecuzione allo stesso tempo tutti i processi.
Perché abbiamo bisogno di una modalità priveligiata(kernel) e utente ? perché senza di loro non avremmo il modo di poter isolare i processi se ogni processo potesse eseguire qualsiasi istruzione potrebbe per esempio cambiare le sue tabelle delle pagine e quindi andare a scrivere ovunque nella memoria fisica.
Le chiamate di sistema sono un modo per cui i processi utente possono chiedere al SO di fare qualcosa. Per esempio leggere un file, chiedo al SO di poter aprire questo file, il SO controlla, per esempio, se l'utente ha l'autorizzazione per aprirlo.
Il codice utente gira in modalità NON priveligiata.
System call macro istruzioni che poi corrispondono a un sacco di istruzioni macchina che vengono eseguite dentro al Kernel del SO.
Tutti i SO hanno centinaia di system call anche se poi le più comuni sono una 20ina.
Il kernel è una parte del software estremamente delicata, da una parte deve essere efficace ma nell'altra deve però essere anche molto sicuro in quanto un bug all'interno del SO compromette la sicurezza dell'intero sistema.
Quando sono arrivati i minicomputer (non in modo letterale, un tempo per mini si intendeva come una lavatrice) avevano un sacco di memoria e c'era la possibilità di caricare più programmi in memoria ed è nata l'idea di sfruttare al meglio le risorse disponibili soprattutto per una questione di costi.
Multiprogrammazione è l'idea di avere molti più programmi in esecuzione ma ovviamente abbiamo il problema di proteggere un programma dall'altro e il SO da tutti questi programmi.
Il SO che useremo nel corso è UNIX, esso è un sistema operativo nato negli anni '60.
Scritto in C con qualche parte in assembler, semplice ma potente.