Struttura di un sistema operativo

0. I gestori del sistema operativo

Un sistema operativo multiprogrammato (e multiprocesso) è composto principalmente da:

- Processor/process management (**kernel**)
- I/O system manager
- Memory manager
- Filesystem manager
- Protection system
- Networking manager
- Command interpreter system (gestione CLI e/o GUI)

Ognuno di questi gestori può essere studiato da diversi punti di vista. Si possono studiare, ad esempio, le proprietà del kernel dal punto di visto dal programmatore (punto di vista **esterno**), mentre altri concetti possono essere spiegati dal punto di vista del sistema operativo (punto di vista **interno**). Ad esempio, il programmatore non si deve far carico dell'esecuzione del processo, quello lo fa il **kernel**, pertanto la situazione viene analizzata internamente.

1. Il kernel

Un sistema operativo gestisce generalmente una **collezione di processi** in modo **concorrente**.
Questi processi possono essere categorizzati come:

- processi di sistema: programmi **insiti al sistema operativo**;
- processi utenti: programmi esterni, eseguiti dall'utente;

Il gestore dei processi è proprio **il kernel**.
Il kernel deve **assegnare** il processore (nel caso single-core) o i processori (nel caso multi-core) ai vari processi. Deve pertanto implementare lo **scheduling** dei processi, ovvero fare in modo che questi ultimi possano avvicendarsi il processore per poter eseguire.
Il kernel si occupa anche di **creazione, distruzione, sospensione, riattivazione** e **intercomunicazione** di processi.
Da un punto di vista interno, è possibile scoprire che un processo passa, durante la sua vita, ***almeno*** (nel caso di Linux/Unix ce ne sono altri) per tre stati principali:

- Ready: il processo, a livello logico è **pronto per l'esecuzione**, ma non ha risorse CPU attribuite;
- Running: il processo ha risorse attribuite ed è in **esecuzione**;
- Sleeping: il processo è in attesa di un particolare evento, ovvero che a livello logico il processo non può eseguire codice (ad esempio eventi di I/O, eventi asincroni…).

All'interno di un S.O viene allocata una struttura dati chiamata **descrittore di processo** (**process control block**). All'interno di questa struttura è presente un campo che indica lo stato del processo. Tramite questa *struct* (perché è effettivamente una struct) è possibile identificare il processo e il suo stato. Il descrittore ha un ruolo importantissimo come **depositario** di tutte le informazioni che è necessario salvare quando un determinato processo **non è in esecuzione**.

Supponendo che il processo sia in esecuzione e supponendo che esso stia usando vari registri della macchina per varie operazioni. Nel momento in cui questo processo subisce una transizione di stato (ad esempio *exec => sleeping*), tutto il contesto di esecuzione (ovvero tutti i dati presenti nei registri) vengono salvati, appunto, nel **descrittore**.
Quando il processo viene rimesso in esecuzione i dati salvati nel descrittore vengono di nuovo rimessi nei registri che stava modificando. Ad esempio, nel descrittore viene salvato il **valore del program counter**, ed è uno dei valori che non può mancare all'interno di esso.

In sintesi il descrittore dei processi è una struttura dati (si può vedere come una **lista**) che contiene stato e informazioni salvate dallo stato **exec** del processo. Chiaramente quando il processo viene eseguito per la prima volta il suo descrittore sarà vuoto. Lo scambio di contesto (di processi, quindi) viene chiamato **context switching**.
Ogni descrittore viene linkato con altri (poiché è una lista), formando code di processi con stati affini.

1.1 Algoritmi di scheduling

Nell'ipotesi semplificativa di avere una singola CPU **solo un processo per volta potrà trovarsi in esecuzione**, mentre gli altri saranno *ready* o *sleeping*.
Lo scheduling viene dunque fatto sulla coda di processi *ready*.
Gli algoritmi di scheduling possono essere:

- preemptive: con prelazione. Se un algoritmo è preemptive, è lo scheduler stesso che "stacca" la CPU dal processo. Lo scheduler esegue un'operazione di *preemption* nei confronti di un processo quando a esso viene sottratta la CPU, proprio dallo scheduler.
- non-preemptive: senza prelazione. Negli algoritmi non-preemptive l'avvicendamento dei processi ai processori viene fatto quando il processo rilascia la CPU "spontaneamente", ovvero quando esso va in sleeping o in attesa di dati. È un algoritmo abbastanza pericoloso, poiché se un processo arriva ad uno stato di *loop infinito*, esso non rilascerebbe mai la CPU, causando un comportamento indefinito.

a) Round Robin

La coda dei processi ready viene gestita come una coda *FIFO* (first in, first out), pertanto il primo processo inserito nella coda è il primo che viene estratto per essere messo in esecuzione.
Questo algoritmo si basa sulla suddivisione di tempo: nel momento in cui un processo viene messo in esecuzione, il sistema operativo assegna al processo un *quanto* di tempo di CPU (time slice) durante la quale, se il processo non termina, subisce preemption e viene messo **al termine della coda**. Un processo sarà in esecuzione **al massimo** per un quanto di tempo.
Può succedere che durante la durata del quanto di tempo il processo si sospenda (I/O) o termini. Se ciò accade il processo rilascia la CPU (in entrambi i casi) e lo scheduler deve andare a prelevare dalla coda dei processi pronti il prossimo processo (nel caso in cui venisse sospeso, il processo andrebbe nella coda dei processi sospesi, mentre nel caso termini verrebbe rimosso logicamente dalle code di processi attivi).
Nel RR è molto importante la scelta della durata del time slice, che deve essere adeguata (tipicamente è 100/200ms), scelta in modo tale che sia sufficientemente grande rispetto al

tempo che il kernel ci mette per fare **context switching**, poiché la preemption prevede questo passaggio nel quale il kernel “switcha” processo, operazione che ha anch’essa un tempo.

b) Scheduling a priorità

1. Statica

Alla creazione dei processi, ad essi, viene assegnata una **priorità (staticamente)**. La coda di processi pronti viene suddivisa in base ai livelli di priorità: più in alto ci saranno quelli a priorità più elevata (0), fino ad arrivare alla n-esima priorità.
I processi nella coda a priorità elevata vengono ottenuti per primi, quando la coda di priorità maggiore si sarà svuotata si passerà a quella di livello inferiore e così via.
Può insorgere il problema di *starvation*: i processi di priorità più bassa rimangono fermi nella loro coda e non vengono eseguiti finché quelli di priorità più alta non terminano.
Si noti che anche in questo caso esistono algoritmi a priorità statica **con e senza preemption**. La preemption avviene quando un processo di priorità maggiore a quella del processo in esecuzione viene messo nella sua coda.

2. Dinamica

Nello scheduling a priorità dinamica la priorità viene abbassata/alzata dinamicamente.
Tutti i processi entrano nella coda a massima priorità, inizialmente, con un quanto di tempo molto corto. Se il processo è I/O bound, ad esempio, non consumerà mai totalmente il suo quanto di tempo, pertanto tornerà nella coda a massima priorità. Se un processo è CPU-bound ed esaurisce il suo quanto di tempo, esso subisce preemption e viene messo nella coda a livello inferiore, con time slice più lungo.
Esiste algoritmo di “promozione” dei processi, tramite il quale, se il processo non esaurisce il quanto di tempo assegnatogli, viene “promosso” alla coda con priorità più alta (con time slice più basso).
In questo modo si evitano fenomeni di starvation e si penalizzano processi che impegnano troppo la CPU. Esiste anche un processo chiamato di *aging priority*, secondo il quale processi che rimangono in una coda per troppo tempo vengono automaticamente promossi.
In sostanza un processo “virtuoso” (ovvero *“bravo”* a rispettare i tempi) viene promosso poiché, appunto, bravo.

1.2 Interazione fra processi

Quando i processi *comunicano* fra di loro, essi assumono un grado di interazione che il kernel deve gestire. Si parla dunque di **programmazione concorrente**.
L’interazione fra processi può essere di due tipi:

1) a competizione: i processi devono usare lo stesso pool di risorse. L’accesso alle risorse comuni deve essere *mutualmente esclusivo* in tutti i casi.
2) cooperazione: i processi devono scambiarsi messaggi o notificarsi eventi.

1.2.1 Il **deadlock**

Problema che si può manifestare quando processi interagenti si trovano ad essere bloccati in attesa del verificarsi di condizioni che non possono verificarsi
I processi possono trovarsi in una condizione di *sleeping* e lì rimanere per sempre nell'attesa di un evento che non si verificherà mai.
Supponiamo di avere due risorse (R1/2) e due processi (P1/2).
Questi due processi abbiano bisogno di usare le risorse R1, R2 in modo mutuamente esclusivo (caso competitivo).
P1 acquisisce R1 e poi R2, mentre P2 acquisisce R1 e poi R2.
Nell'ordine:

- P1 acquisisce R1, che da libera passa ad occupata;
- P2 acquisisce R2, che da libera passa ad occupata.

Una volta che le risorse vengono acquisite, queste risorse non vengono **mai rilasciate** (il rilascio delle risorse va fatto dal programmatore, pertanto va scritto il codice riguardante il rilascio all'interno del programma), poiché il programmatore, sbadatamente, si è scordato di rilasciare il *lock* delle risorse.
Quando P1 cerca di assumere il controllo di R2, si ha un blocco, poiché P1 non ha mai rilasciato il lock su R1 e viceversa. Ecco verificatosi il **deadlock**.

Nel caso di cooperazione, supponiamo che ci siano due processi P1 e P2 e che essi debbano comunicare.
P1 spedisce in modo sincrono l'informazione I1 a P2.
P2 a sua volta spedisce I2 a P1.
La spedizione di I1 a P2 deve trovare P2 *pronto a ricevere*, ma P2 sta tentando di spedire il messaggio.
In questo caso avremmo due processi **sospesi**, poiché P1 sta cercando di consegnare ad un processo non pronto per ricevere, così come P2.
In generale, infatti, per scambiare messaggi fra thread si utilizzano metodi di spedizioni **asincroni**.

2. Gestore I/O

La parte di kernel che si occupa della gestione di I/O comprende tutti i *buffer* utilizzati dall'I/O che vengono letti/scritti dal sistema operativo, dai quali i processi leggeranno i valori (e.g tastiera). Inoltre, in questa sezione del kernel, vengono gestite le routine di interrupt scatenate dall'I/O.

3. Gestore memoria centrale

Il compito principale di questa parte di sistema operativa è quello di evitare che i processi "sporchino" la memoria di altri *colleghi*.
Oltre a ciò, il gestore della memoria si occupa di allocare e deallocare memoria per caricare programmi e dati in memoria centrale.

3.1 Allocazione

Il metodo più utilizzato per gestire i programmi nella RAM è quello di **paginazione**.
A livello generale questo metodo è detto di *allocazione non contigua*: il programma, prima di essere caricato in RAM, viene **paginato**; lo spazio di indirizzamento del programma viene suddiviso in pagine logiche (da scrivere dentro la RAM, vedasi page table) con una dimensione d_p. Il numero di pagine deriva dalla lunghezza dell'eseguibile diviso la dimensione della pagina stessa (ovviamente) arrotondato all'intero superiore.

La paginazione è utile perché quando il S.O deve caricare un programma, basta verificare che esista un numero sufficiente di pagine logiche libere nella RAM per poter allocare "disgiuntamente" il nostro programma. Nella page table viene inserito l'indirizzo al quale ciascuna pagina viene caricata.
Nella tabella delle pagine abbiamo, per ogni processo attivo nel sistema, l'indirizzo fisico al quale è stato caricata ciascuna pagina logica.
Si associa dunque un indirizzo virtuale (logico) ad un indirizzo fisico.
Quando la CPU mette in esecuzione il processo genera un indirizzo logico, che viene dato in pasto alla MMU. La MMU suddivide, tramite algoritmi già visti nel corso di Calcolatori, l'indirizzo virtuale in offset e index. L'index indicherà l'elemento all'interno della page table e l'offset corrisponderà il byte dell'elemento in cui accedere alla page table.
Il problema introdotto da questo meccanismo si devono fare **2** accessi in memoria (come già visto in CE), pertanto si introduce un overhead del 100%, ovvero si utilizza il 50% di risorse in più (si fa una lettura che si potrebbe evitare, se il meccanismo fosse diverso).
Si utilizza pertanto **TLB** (lo sapete come funziona dai…).
Con il TLB l'overhead introdotto è molto minore.

3.1.2 Gestione della memoria virtuale

Nel corso del tempo, lo studio dei sistemi operativi ha portato allo sviluppo di una tesi secondo la quale il caricare un intero programma in RAM non sia completamente necessario. Può succedere che vengano sovradimensionati dati (dal programmatore) o che in determinati scenari certe funzioni non vengano mai chiamate, ecc…
L'idea è quella di gestire la memoria in modo **virtuale**: il gestore della memoria carica in RAM le pagine necessarie solo in determinati lassi di tempo. Le pagine non necessarie verranno caricate su una zona riservata (*swap* o su HDD).

Nei sistemi operativi UNIX il programma viene suddiviso a livello logico in segmenti di **codice** e **dati**, che vengono a loro volta paginati (in modo pseudo-harvardiano).
Il vantaggio di questo meccanismo è che il sistema operativo può caricare parzialmente più processi. Se un processo ha bisogno di una pagina che non è caricata in RAM il sistema di

traduzione degli indirizzi se ne accorge (tramite il bit di validità **nullo** si ha una page fault). Il processo viene sospeso e lo scheduling seleziona un altro processo da far eseguire.
Il sistema operativo si occuperà in modo concorrente di recuperare la pagina necessaria all'esecuzione del processo e di copiarla dal disco alla RAM.

Se non ci sono pagine libere il sistema operativo deve effettuare una sostituzione di pagina. Una pagina in RAM (scelta tramite LRU) viene sostituita con una su disco. Se la pagina in RAM aveva dirty bit a 1 (sovrascritta), allora viene salvata su disco, altrimenti viene sovrascritta semplicemente.