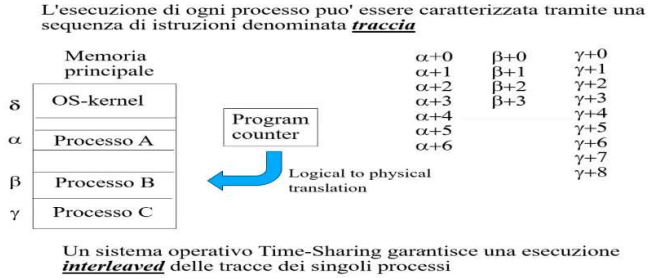
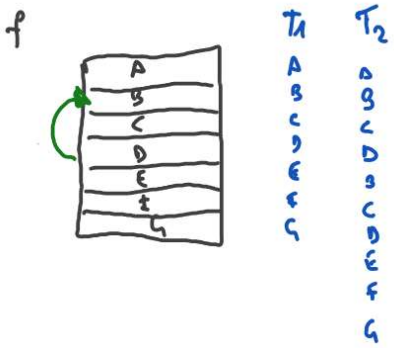
Processi e thread 1 Stati + multiprogramamzione + swap

lunedì 13 ottobre 2025 16:45

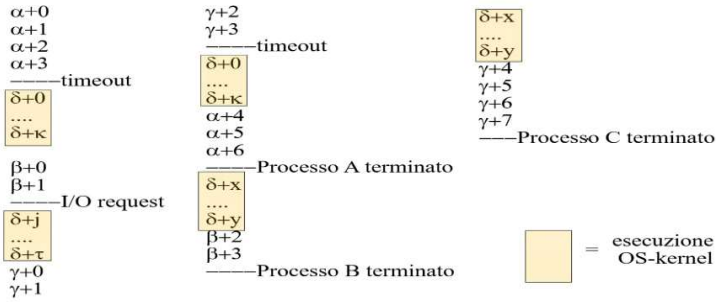
Ricordiamo che un processo , soprattutto nei sistemi time-sharing(attuali) è un'applicazione attiva. Vediamo in generale come un SO è organizzato dal punto di vista della gestione dei processi :



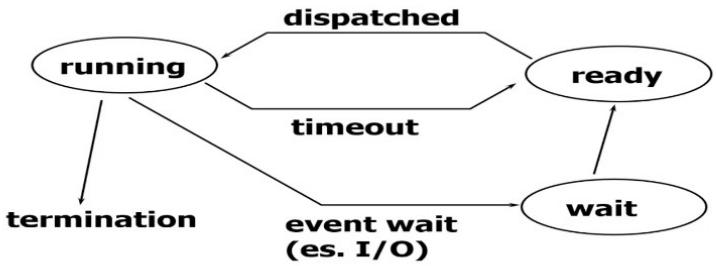
Nell'esempio se la CPU esegue il processo B , la CPU va a pescare tutte le istruzioni nel blocco B: analogamente se altra applicazione pesco altre istruzioni. Quindi in generale per ogni applicazione attiva in CPU, vi è una sequenza di istruzioni macchina , la quale prende il nome di traccia . In un sistema time-sharing l'esecuzione delle tracce avviene secondo uno schema **interleaved** . Vediamo in dettaglio cosa è una traccia : elenco istruzioni macchina in base a cosa c'è scritto all'interno di una funzione/programma in esecuzione e di ciò che sono i dati . In dettaglio :

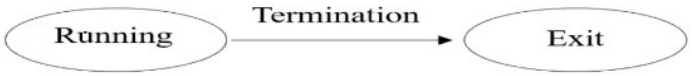


Dove T_1 e T_2 sono due trace differenti , magari in base ad un salto o una condizione. Vediamo ora come viene effettuata l'esecuzione interleaved dei processi (esecuzione mischiata) :



Quindi partendo dal processo A eseguo le istruzioni di A, poi se arriva un'interruzione (timeout) si va ad eseguire software di sistema, con eventualità di riorganizzazione della CPU (assegno altra applicazione) . Analogamente quando svolgo istruzioni processo B , le eseguo e se ricevo una richiesta di i/o si comporta come prima . *Quindi da questo esempio si evince che la traccia di un processo non è sequenziale (da inizio a fine) , ma può essere che inizia e finisce in istanti diversi.* Quindi a quale processo posso dare il controllo?? In base allo stato in cui si trova correntemente un processo

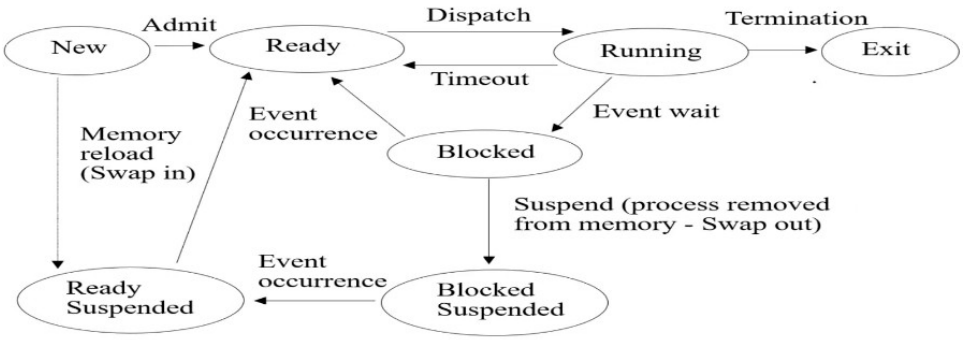


1. **Running**
   a. Il processore decide di dare il controllo alla traccia
2. **Ready**
   a. Si è il questo stato quando il processo, in esecuzione è soggetto ad interrupt o timeout
   b. Questa traccia messa in ready potrà riprendere il controllo della CPU più in avanti (dispatched) : ready->running
3. **Wait**
   a. Il processo è in esecuzione e chiamo una system call , la quale passa il controllo al software dl SO, quindi la traccia rimane in attesa che accada qualche evento .
   b. Se accade evento , si ha interrupt , quindi quella traccia può essere riportata nello stato ready, appena la CPU si libera
4. **Termination**
   a. Stato nel quale si ha che l'applicazione viene terminata o a causa di una system call (per terminarla) oppure per segmentation fault.
5. In generale oltre a questi 3 stati (ready, running e wait) ce se nono altri 2 addizionali : exit e ready. Vediamoli :
   a. **Exit**
   b. 

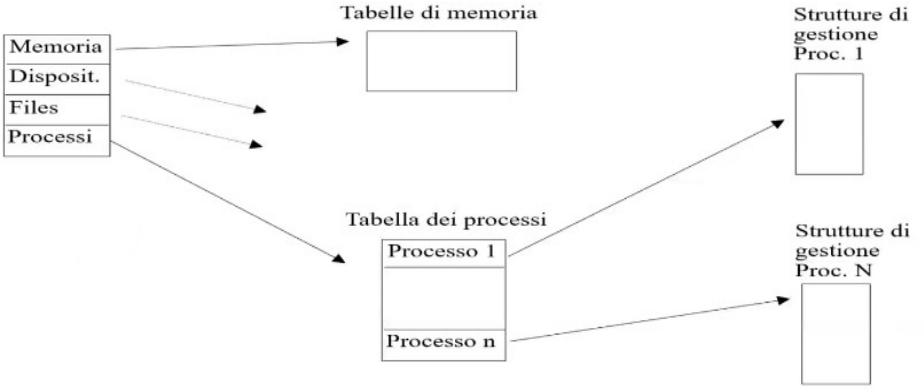
   c. Serve per gestire terminazione , la quale porta l'applicazione ad essere nello stato exit (è il software del sistema che la porta li), rilasciando le strutture dati (risorse) usate per quell'applicazione.
   d. Le applicazioni non riprendono più l'esecuzione, ma rimangono comunque "registrate" per un determinato tempo
   e. Ready
   f. 

   g. In questo caso il SO alloca le risorse necessarie (strutture dati e/o memoria). Se tutto corretto la registriamo ready.

Quindi vediamo ora quante applicazioni possiamo mantenere attive in un determinato momento? Si parla di **livello di multi programmazione .** In generale oltre al numero di applicazioni attive in memoria , vengono anche incluse quelle attive sul disco . Esempio : se abbiamo attive solo applicazioni attive su disco il grado di multiprogrammazione è vincolato alla memoria. Come buona pratica , utilizziamo sia applicazioni su disco che in memoria in quanto un processo potrebbe avere evoluzione del tipo : il processo sta in stato di wait per un determinato tempo (T_1) finché non torna nello stato di ready. Quindi durante tutto quel periodo si ha uno "spreco delle risorse" in quanto mantengo l'address space in memoria , il che fa presuppore che convenga caricare l'address space di altro processo , così si ha l'esecuzione di un nuovo processo. Dopo l'esaurimento del tempo di wait, il vecchio processo è pronto per riessere schedulato. Questo processo viene chiamato **swapping (sposto address space da memoria a disco e viceversa) .** In dettaglio si parla di *swap out* (processi blocked finché non accade qualcosa) se escludo applicazione da memoria di lavoro e la carico sul disco oppure di *swap in* se prendo applicazione e la ri/carico all'interno della memoria e lo eseguo. Quindi con questa tecnica riduco il sotto utilizzo del processore. Quanto detto finora (incluso lo swapping) porta il processo ad avere i seguenti stati :



Per ogni processo il sistema operativo mantiene informazioni sui processi stessi e risorse (strutture dati di controllo) :



Nelle tabelle di memoria vi sono informazioni su come può essere usata la memoria stessa (zone libere / occupate), mentre per i processi abbiamo una tabella (lista) nella quale per ogni processo attivo abbiamo una tabella ulteriore che contiene le strutture di gestione di ogni processo : quindi ogni entity di questa tabella è collegata con altra tabella che descrive le caratteristiche di quel processo. Per quanto riguarda invece quelle di memoria sono tabelle che contengono info sia riguardo la *memoria principale sia quella secondaria* : per la prima si ha una tabella chiamata **core map** (mappa cuore) la quale ci dice le zone libere della memoria stessa : serve quindi dove poter ricollocare gli addres space dei processi . Tutto ciò serve per gestire la memoria virtuale . Analogamente per la secondaria (dove sposto le applicazioni ,caricando gli address space delle applicazioni quando vengono eliminate dalla memoria principale) . Le informazioni (spazio libero/occupato) sono gestite dalla **swap map** .