di Pierluigi Panunzi

# Multi-tasking in Time-sharing con il Turbo Pascal

prima parte

*Quello di cui ci accingiamo a parlare, per una serie di puntate, è un argomento che riteniamo molto interessante, stimolante, al quale i lettori sono anche invitati a contribuire; si tratta senz'altro di un argomento alquanto complicato che richiede, per la sua corretta comprensione, tutta una serie di conoscenze a monte sui sistemi operativi «multi-tasking», conoscenze che, data la carenza cronica di sistemi operativi di tal genere, devono per forza di cose provenire da libri di testo universitari, se non dalla letteratura specifica sull'argomento. Evidentemente cercheremo di spiegare in termini il più possibile semplici i concetti che via via incontreremo*

Innanzitutto, prima di proseguire, sottolineiamo che, non tanto l'argomento ma bensì l'implementazione, è veramente del tutto originale: per la già citata carenza di spunti se non sui testi sacri (che poi sono sempre molto lontani dalla realtà applicativa...), il progettista di questo sistemino (si tratta proprio del redattore della rubrica!) ha praticamente lavorato «al buio» armato di quei potenti mezzi quali il Turbo Pascal, il Turbo Assembler ed il Turbo Debugger (che nel prosieguo citeremo con le abbreviazioni TP, TA e TD), ottimi per creare, testare, correggere il programma.

Detto questo, iniziamo dunque la nostra chiacchierata su come si possa, in ambiente TP, realizzare un sistema operativo multi-tasking, che consenta l'esecuzione «contemporanea» di più processi concorrenti, preveda l'uso di semafori per l'accesso a risorse condivise, e sia dotato inoltre delle «primitive» più comuni e note.

A beneficio di quei lettori che già temono di vedere descritto un tool che poi non potranno mai fare girare sul proprio sistema per problemi di microprocessore, diciamo subito che il sistema operativo proposto gira sull'8086 (e perciò anche sui suoi successori), in quanto non utilizza nessuna delle facility introdotte con il 286 e relative alla programmazione in Modo Protetto: il fatto di poter girare sull'8086, oltre che un requisito di compatibilità verso il basso (... amici che non possiedono un 386...) è nato anche per il fatto che il TP genera file eseguibili codificati in 8086, senza la possibilità di generare codice «superiore» ed allora era perfettamente inutile scrivere routine in Assembler 386, quando poi il nucleo principale era in 8086.

Forse per un approccio differente al problema e cioè prima di poter utilizzare appieno le risorse multi-tasking dei vari 286, 386 e successivi, bisognerà attendere le versioni 6.0 o 7.0 del TP, che prevederanno innanzitutto (è una richiesta implicita alla Borland oltreché una speranza...) una codifica migliore e magari già un nucleo di multi-programmazione.

Lasciamo dunque tutto questo ad un futuro (prossimo...) e ribadiamo che nel caso in esame abbiamo «sintetizzato» le funzionalità di un OS (d'ora in poi abbrevieremo così il termine «Operative System») per mezzo di semplici funzioni scritte per la maggior parte in TP, più alcune routine particolarmente delicate scritte in TA: il risultato è dunque il sistemino che andiamo a presentare e che è stato battezzato con la sigla TPMT (da «Turbo Pascal Multi-Tasking»).

Prima di addentrarci nell'analisi del TPMT, vediamo insieme alcune nozioni fondamentali riguardanti gli OS e che risulteranno necessarie nel seguito.

### *Caratteristiche fondamentali*

Un sistema operativo di tipo multitasking è concettualmente diverso da un sistema operativo «semplice», «normale», in quanto consente di far girare «contemporaneamente» più applicazioni, piuttosto che una unica: mentre infatti in un OS, quale l'MS-DOS, è solo un programma per volta che gira e che può girare, in un OS multi-tasking invece più programmi possono girare in un unico ambiente, avendo a disposizione un certo numero di risorse (condivise e non per mezzo di semafori) e potendosi scambiare in modo univoco informazioni per mezzo di meccanismi appositi (mailbox).

Ma alla base di tutto c'è un programma (che in gergo prende il nome Kernel, che in inglese significa «nocciolo di un frutto», «nucleo centrale»), che è il vero e proprio nucleo «intelligente» del sistema e che ha il compito di gestire programmi, risorse, tempi, ecc.

Visto che abbiamo parlato di tempi, vediamo come è possibile, in un computer dotato di un microprocessore solo e che perciò può eseguire un'istruzione alla volta, far eseguire più programmi «contemporaneamente», laddove questo termine è stato messo sempre tra virgolette: data l'enorme velocità di esecuzione della CPU, basta far sì che quest'ultima dedichi la sua attenzione (esecuzione di istruzioni, cioè) non ad un solo programma, ma ad un certo numero di essi dedicando ad ognuno un certo tempo prefissato (detto slice).

Ecco che perciò, supponendo per ipotesi di fissare lo slice ad 1 secondo, la CPU eseguirà per 1 secondo le istruzioni del programma A, per 1 secondo le istruzioni del programma B, e così di seguito fino all'ultimo programma della serie, per poi ricominciare il ciclo dal programma A al quale dedica il successivo secondo.

Però questo era solo un esempio: con questo slice di tempo ci si accorgerebbe subito del trucco, in quanto tempi dell'ordine del secondo sono particolarmente alti.

Se invece riduciamo lo slice (scendendo ad un valore «comodo» pari ad 1/18 di secondo, sul cui valore torneremo nel seguito) avremo che in un secondo vengono eseguiti 18 programmi, uno di seguito all'altro: ma se i programmi sono solo 2, ecco che ovviamente verranno eseguiti 9 volte al secondo ognuno e questo fatto già può

dare l'illusione di «contemporaneità».

Proprio quella della contemporaneità e continuità è una sensazione che abbiamo noi esseri umani di fronte ad eventi che si susseguono uno dopo l'altro, ma per piccoli intervalli: basti pensare all'esempio classico della televisione, laddove ogni venticinquesimo di secondo viene cambiata un'intera immagine statica per ottenere effetti di movimento secondo un meccanismo al quale siamo più che abituati, ma del quale non ci rendiamo nemmeno conto.

Nel mondo dei computer abbiamo invece a che fare con tempi completamente inferiori: un microprocessore (ad esempio il 286) che viaggi a 10 MHz di clock ha un tempo di ciclo pari a 100 nsec (1 nsec è pari ad un miliardesimo di secondo) e perciò un'istruzione di 10 clock impiega appena 1 µsecondo ad essere eseguita: in uno slice di tempo pari a 55 msec (1/18 di secondo) di tali istruzioni ne esegue 55000...

Comunque sappiamo che le istruzioni più semplici e più comunemente usate non richiedono, in media, mai più di 5 cicli di clock per cui abbiamo mediamente centomila istruzioni eseguite in uno slice di tempo.

Eseguendo perciò per ogni slice tante istruzioni di un programma, altrettante di un altro programma nello slice successivo, è facile ingannare l'utente il quale avrà la sensazione di contemporaneità e continuità dei processi che si svolgono.

Abbiamo proprio ora citato un nome che useremo più volte nel seguito: processo. Con tale termine si intende in pratica uno dei «programmi» (termine che ora poniamo tra virgolette) che vengono eseguiti secondo una ben determinata partizione di tempo (time-sharing), sotto la supervisione da parte di un nucleo centrale (il Kernel), che funge da «controllore», scheduler (si veda la figura 1).

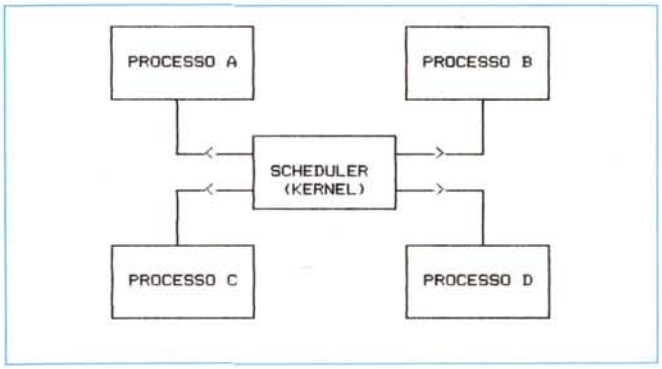


*Figura 1 - Rappresentazione, schematica di un sistemino multi-tasking nel quale uno scheduler posto all'interno di un nucleo centrale (Kernel) controlla l'esecuzione dei quattro processi A, B, C e D, facendoli eseguire ognuno per uno slice di tempo, uno di seguito all'altro.*

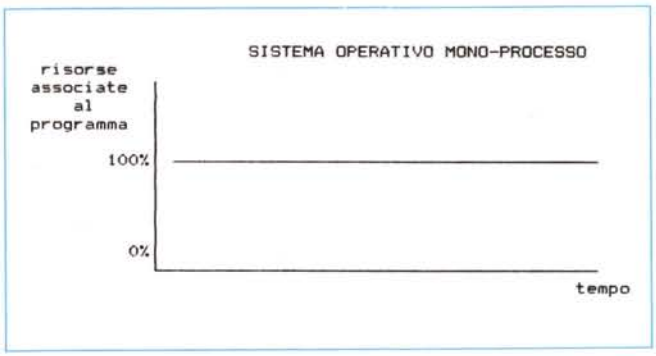


*Figura 2 - In questo diagramma temporale vediamo che, istante per istante, tutte le risorse (CPU, memoria, ecc.) vengono concesse all'unico programma in esecuzione.*

Possiamo vedere successivamente, in forma di diagrammi temporali, il susseguirsi degli eventi, e cioè l'andamento temporale dell'utilizzazione delle risorse, in un sistema mono-processo, nel nostro PC (sotto MS-DOS) e sempre nel nostro PC, ma sotto TPMT.

In particolare il primo ed il secondo caso si riferiscono ad uri generico computer nel quale solo un'applicazione alla volta può girare: nella figura 2 abbiamo indicato quei computer («dedicati») in cui gira soltanto un programma che non viene mai interrotto dall'esterno, fatto questo alquanto raro. In figura 3 invece vediamo il caso più vicino alla realtà e cioè il nostro PC sotto MS-DOS, nel quale, ad intervalli regolari, il programma che sta girando viene interrotto dalla routine di servizio dell'interrupt generato dal timer interno, che

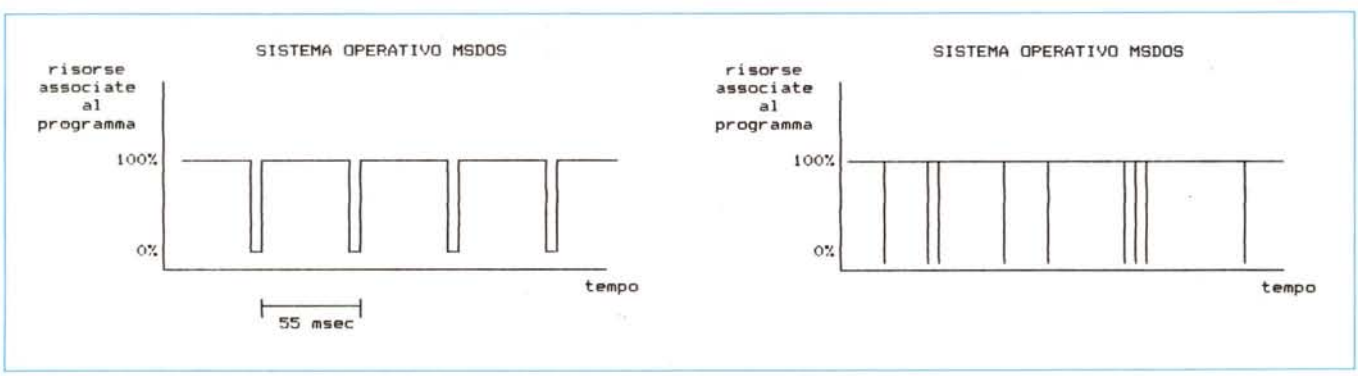


*Figura 3 - In realtà invece (anche nei nostri personal), ad intervalli regolari il controllo passa ad una routine di clock che effettua l'aggiornamento dell'ora e di alcune altre variabili di sistema: la durata di tale routine è in effetti esagerata per motivi grafici, dal momento che invece è di piccola entità.*

*Figura 4 - Per essere ancor più aderenti alla realtà, abbiamo riportato un esempio di diagramma temporale in cui abbiamo tentato di mostrare (senza scale temporali definite) che il programma in esecuzione viene interrotto in maniera alquanto caotica da una serie di interrupt (timer, tastiera, porte seriali, controller di dischi, ecc.), le cui routine di servizio devono essere comunque molto brevi.*

consente al sistema di avere l'ora aggiornata e di effettuare altre operazioni sempre legate al tempo.

Dicevamo che neanche questo è un caso molto reale, in quanto in realtà oltre all'interrupt del timer ne sono presenti tanti altri, quali quello generato ogni volta che si preme un tasto della tastiera, quello utilizzato per scrivere sullo schermo video, quello generato dal controller delle unità a dischi nel caso di richiesta d'uso dei dischi stessi, quello generato dal controller della porta seriale (USART) allorché riceve un carattere dalla porta stessa ed alla quale ad esempio abbiamo collegato il nostro mouse, e tanti altri interrupt ancora, che in generale sono suddivisi in «interrupt hardware», generati cioè da componenti elettronici esterni alla CPU, ed «interrupt software» generati viceversa di programmi.

Ecco perciò il significato della figura 4 dove si è cercato, per quanto possibile a livello semi-grafico, di rappresentare la caoticità di presenza di interrupt all'interno di un certo intervallo di tempo: all'unico processo viene in genere garantita un'alta percentuale di utilizzazione delle risorse del sistema.

Questo grazie al fatto che si stabilisce che ogni routine di servizio di un certo interrupt duri un tempo brevissimo, tale da non impedire ad altri interrupt di sopraggiungere ed essere correttamente risolti da parte della CPU.

In figura 5, infine, vediamo in teoria quanto succede nel caso del TPMT: più processi vengono eseguiti uno dopo l'altro, per una durata di tempo per ognuno pari ad 1/18 di secondo.

Come vedremo meglio nel seguito, in realtà il passaggio da un processo all'altro non è istantaneo, ma bensì richiede un certo tempo, per cui se vogliamo un andamento temporale ancor più vicino alla realtà dobbiamo fare riferimento alla figura 6, nella quale abbiamo ancora una volta previsto interrupt casuali all'interno di ciascuno slice di tempo.

## *Altre nozioni utii*

Proseguendo nell'analisi degli elementi fondamentali che incontreremo nelle prossime puntate, affrontiamo ora il problema legato al salvataggio di informazioni nel passaggio tra un processo e l'altro.

Abbiamo detto dunque che lo scheduler, ad intervalli di tempo prefissati, attiva un processo e lo mantiene in esecuzione per tutta la durata di uno slice: per fare questo deve necessariamente andare ad interrompere il processo che in quel momento era in esecuzione, salvandone lo «stato» (in particolare i valori contenuti nei registri), in modo tale da poterlo ripristinare, nell'istante in cui tale processo verrà rieseguito, il tutto come se per il processo in questione non fosse accaduto nulla di particolare.

Questo fatto del salvataggio dello stato è di fondamentale importanza in quanto basta ovviamente trascurare un registro per ottenere un indesiderato stravolgimento nel funzionamento del processo, non appena questo verrà riattivato.

Tra l'altro l'importante è salvare i registri in una zona sicura, laddove non sia possibile avere interferenze ed ingerenze da parte di altri processi che volontariamente o per errore di programmazione possono alterare le informazioni in essa contenute.

Ecco che perciò per ogni processo deve essere definito uno stack locale, dove poter salvare, innanzitutto, i registri all'atto del «passaggio di consegne» tra un processo e l'altro (tale termine si indica in gergo con «task-switching») e poi dove poter appoggiare le informazioni nel normale svolgimento del processo stesso.

Già sappiamo che nel 286 (e lo stesso accade ovviamente nel 386), lavorando in modo protetto, bisogna definire, tra le tante altre cose, anche uno stack segment, «locale» al processo o task che dir si voglia, che rimane inattacabile dall'esterno grazie ad appositi meccanismi di protezione interni alla CPU stessa.

Siccome noi stiamo parlando di un sistemino multi-tasking orientato all'8086, ecco che dobbiamo prevedere questo stack (del quale avremo la possibilità di definire l'ampiezza, come vedremo) che verrà posto nell'heap e come tale sarà gestito dall'heap-manager interno del TP stesso, che per sua natura garantisce una già buona protezione verso l'esterno: ma dato che si tratta di una protezione software (contrapposta alla protezione hardware del 286 e 386) sappiamo già che ci sono infiniti modi per raggirarla...

Comunque il TPMT è stato costruito in modo tale da non darsi subito la classica zappa sui piedi, nel senso che se proprio si vuole raggirarlo, allora bisogna crearsi delle routine apposite... cosa che a noi non interessa certo.

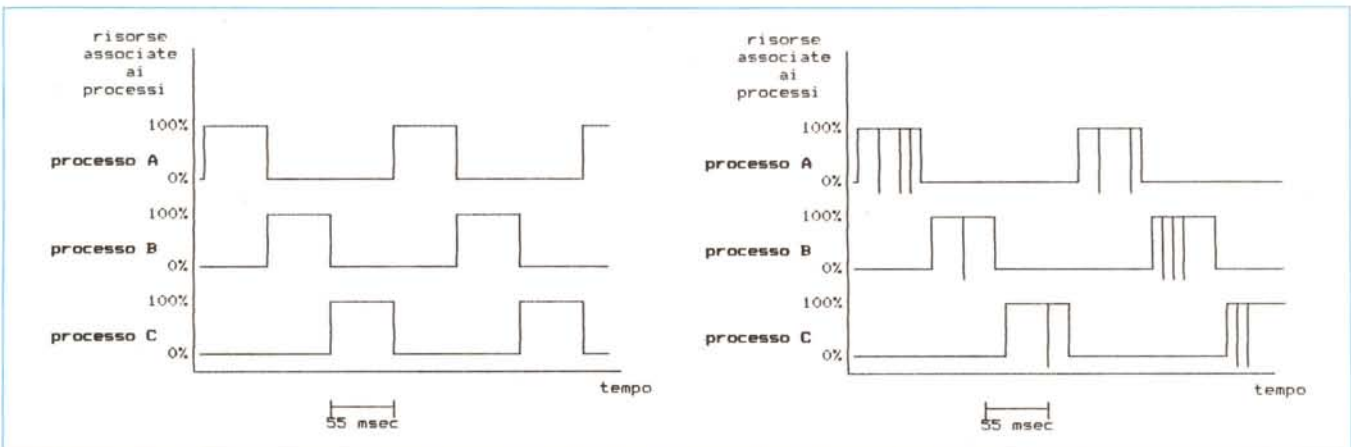


*Figura 5 - In un sistema operativo multi-tasking a partizioni di tempo, i processi (nel nostro caso A, B e C) assumono il 100% delle risorse (CPU, memoria, ecc.) per un intervallo di tempo pari allo slice, che nel caso del TPMT è pari a 55 msec (1/18 di secondo).*

*Figura 6 - Ecco quello che succede in realtà in un sistema dove gira il TPMT: i processi vengono eseguiti in successione, ognuno per uno slice di tempo, ma durante lo slice possono essere interrotti in modo alquanto casuale da interrupt di vario genere.*

## Le risorse

Passiamo ora ad un altro concetto importante, quello delle risorse: abbiamo citato più volte questo termine, sottintendendo concetti che viceversa è meglio chiarire. In particolare con tale termine intendiamo innanzitutto la memoria (utilizzabile istante per istante da tutti i processi) ed inoltre il video, la tastiera, le unità a dischi, la stampante, le porte seriali, ecc: in generale prendono il nome di «risorse di sistema» in quanto risiedono nel sistema (o nelle sue immediate vicinanze, come la stampante, il modem, il mouse), e perciò devono poter essere utilizzate dai processi senza problemi e soprattutto senza conflitti e, ancora una volta, interferenze.

Facciamo un esempio chiarificatore del cosiddetto problema della «condivisione delle risorse» tra processi, senza però tirare in ballo il classico ed arcinoto esempio della stampante che più processi devono usare (problema che si risolve in parte con gli spooler di stampa).

Il nostro esempio è un po' più calato all'interno del TP e riguarda la gestione del video in modalità grafica: sappiamo che per tracciare un segmento giallo tra i punti di coordinate (100,100) e (200,200) si esegue il semplice frammento di programma dato da:

```
setcolor(yellow);
line(100,100,200,200);
```

Ora queste istruzioni sono state costruite per poter funzionare in un ambiente mono-processo ed in particolare, non appena esegue la setcolor, il TP setta un'apposita locazione di memoria, posta all'interno di una zona di memoria che il TP stesso usa per memorizzare tutte le sfaccettature dello stato del programma in corso, per far sapere, alle istruzioni successive che lo richiedano, che il colore con cui debbono accendere i pixel è proprio il giallo.

Questo non basta più in un sistema multi-tasking e dall'esempio ci accorgiamo subito del perché: supponiamo dunque di avere due processi che (è superfluo dire «in istanti differenti, incorrelati», dal momento che in multitasking non c'è sincronizzazione tra processi, a meno di non volerla esplicitamente...) devono tracciare, l'uno, una serie di cerchi di raggio random e di colore rosso e l'altro una serie di rettangoli sempre random, ma di colore verde.

I due processi avranno dunque al loro interno frammenti di codice in TP rappresentati in tabella A.

```
...
for i := 1 to 1000 do
 begin
  setcolor(red);
  circle(random(639),random(479),random(200));
 end;
...

...
for i := 1 to 1000 do
 begin
  setcolor(green);
  x0 := random(639);    { valori per la VGA }
  x1 := random(639);
  y0 := random(479);
  y1 := random(479);
  rectangle(x0,y0,x1,y1);
 end;
...
```

*Tabella A*

Dal momento che i due processi in esame possono essere interrotti in ogni luogo ed in ogni istante, può capitare (ed infatti capita senz'altro!!!) che il primo processo sia interrotto subito dopo la setcolor(red) (con il che il solerte TP memorizza nel suo stato interno che i pixel dovranno essere di colore rosso), dopodiché il secondo processo esegue le sue istruzioni, arrivando dunque alla fatidica («critica») setcolor(green) (e qui il solerte TP non può fare altro che settare come colore dei prossimi pixel il verde) e magari già disegnando dei rettangoli verdi, dopodiché deve cedere il controllo all'altro processo.

Quello che succede ora è che il primo processo esegue la circle, la quale trova come colore dei pixel proprio il verde, lasciato dall'altro processo: il risultato è che alcuni cerchi saranno verdi e viceversa alcuni rettangoli saranno rossi... Tra l'altro questo è effettivamente capitato nei primi esperimenti fatti su programmi di prova: in modo apparentemente inspiegabile si aveva il cambiamento di alcuni colori di certi oggetti; è bastata però una semplice analisi per scoprire che il problema era dunque nell'errata condivisione delle risorse.

Questo è stato solo un esempio di fatti veramente accaduti e le sue conseguenze non sono state certo catastrofiche: è inimmaginabile quello che può succedere se le risorse in conflitto sono più importanti e critiche. Per risolvere dunque questo problema, bisogna sfruttare un meccanismo di accesso alle risorse, che prende in gergo un nome alquanto colorito (i soliti americani...), che però rende bene l'idea di quanto succede: stiamo parlando dei cosiddetti semafori di condivisione delle risorse.

## I semafori

Con tale termine si intende, semplificando al massimo, un flag posto «a guardia» della risorsa: all'inizio dei tempi tale flag è resettato (si dice che il «semaforo è verde»).

Ogni processo che vuole accedere a tale risorsa deve eseguire una certa primitiva (cioè una funzione appartenente al Kernel) di «passaggio del semaforo» (in inglese passlock) che consiste banalmente nel test del flag: se tale flag è resettato (semaforo verde, la prima volta che si accede a tale risorsa) allora tale semaforo viene posto subito a rosso (settato) in modo tale che il prossimo processo che deve fare accesso alla risorsa trovi appunto il semaforo rosso.

Nel caso in cui tale semaforo sia rosso (flag settato) il processo deve rimanere lì in loop d'attesa fino allo sblocco della risorsa da parte del processo che l'aveva in gestione.

Ecco che perciò il processo che sta utilizzando la risorsa, al termine dell'utilizzazione della risorsa stessa, deve rimettere il semaforo a verde (resettarlo), per mezzo dalla primitiva freelock, per riabilitare l'accesso alla risorsa agli altri processi che ne hanno bisogno: se non facesse così, gli altri processi non potrebbero più essere eseguiti. Garantiamo che questo fatto capita più volte di quanto non si pensi...

Detto questo concludiamo questa puntata alquanto pesante a causa della parecchia teoria contenuta: nella prossima termineremo l'analisi teorica dei concetti che ci serviranno nel seguito (analisi volutamente semplificata, altrimenti tanto vale leggersi uno dei numerosi trattati sul multi-tasking), per poi iniziare l'analisi dell'effettiva implementazione delle singole primitive, fatto che richiederà un'ottima conoscenza del TP, alla quale si cercherà di sopperire con altre disquisizioni teoriche...