Appunti di Sistemi Operativi

Enzo Mumolo

e-mail address :mumolo@units.it
web address :www.units.it/mumolo

$\surd$ Introduzione ai processi - concorrenza

Indice

Capitolo 1

La gestione dei Processi

Un processo puó essere definito come un programma in esecuzione, cioé la sequenza delle ativitá computazionali di un algoritmo descritto con un programma scritto in un linguaggio di programmazione. Naturalmente un programma puó generare infiniti processi, a seconda dei dati che vengono forniti in ingresso al programma stesso.

Vedremo che un processo richiede un ambiente per il suo funzionamento. Ogni ambiente é indipendente dagli altri. Da questo punto di vista i processi possono essere visti come delle sfere (che rappresentano l'ambiente di un processo) che si muovono su un piano senza in nessun modo interferire l'uno con l'altro. Compito del Kernel é di gestire questa indipendenza.

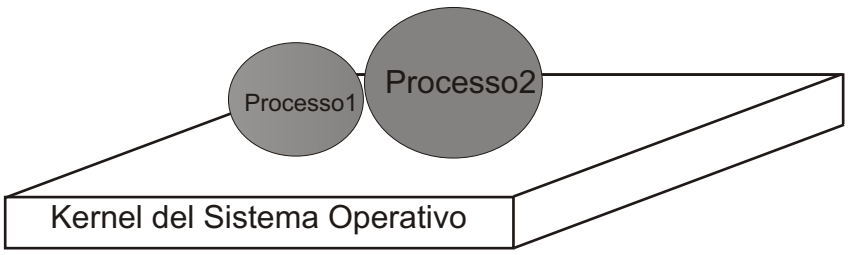

Figura 1.1 Modello di processi concorrenti

L'ambiente di un processo contiene sia il codice eseguibile che lo stack e delle variabili (**variabili d'ambiente**) che servono al funzionamento del processo stesso.

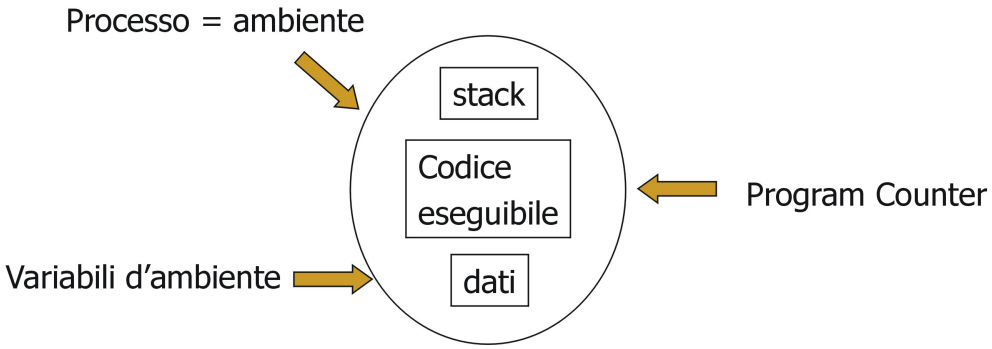

Figura 1.2 L'ambiente di un processo

1.1 Una realizzazione della schedulazione tra processi

Il compito fondamentale di ogni sistema operativo é quello di fornire un ambiente adatto per l'esecuzione dei programmi utente, e di fornire agli stessi tutte le caratteristiche richieste. I processi sono descritti mediante strutture dati chiamate in generale PCB, che rappresentano tutte le informazioni dei processi. Tali strutture dati sono accodate in una lista, come rappresentato nella seguente figura:

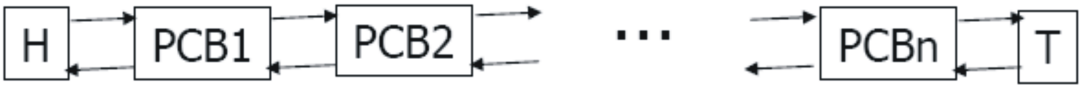

Figura 1.3 Lista dei descrittori dei processi

É compito del sistema operativo estrarre un descrittore attraverso gli algoritmi di schedulazione e farli eseguire assegnando loro la CPU. In questo modo, la funzione di un sistema operativo - quella di eseguire processi - puó essere riassunta dallo pseudocodice di fig.1.2.

Secondo questo schema, il sistema é gestito da una serie di interruzioni che producono la esecuzione a divisione di tempo: ad ogni interruzione del timer lo schedulatore seleziona un descrittore di processo al quale viene assegnata la CPU e che viene rimesso in coda una volta che l'esecuzione é terminata. Questa esecuzione puó essere descritta nella figura 1.3.

Il **kernel** é il cuore del sistema operativo Unix e risiede permanentemente in memoria centrale come ogni processo in esecuzione (o parte di esso). La seguente figura fornisce un diagramma a blocchi dell'architettura del kernel.

Il sistema operativo Unix mette a disposizione le **chiamate di sistema** (system call) che permettono ai processi di eseguire operazioni che altrimenti sarebbero loro proibite. Una chiamata di sistema e' semplicemente una richiesta di servizi al kernel.

L'interfaccia per le chiamate di sistema e l'**interfaccia di libreria** rappresentano le linee di divisione tra i programmi utente ed applicativi ed il livello kernel. L'interfaccia per le chiamate di sistema si occupa di cambiare la modalitá di esecuzione di un processo da **modalita' utente** (running user mode) a **modalita' sistema** (running kernel mode) ogni volta che intercetta una chiamata di sistema (system call).

Il meccanismo con il quale vengono attivate le chiamate di sistema é illustrato in fig.1.2. Una chiamata di sistema genera una interruzione che mette il processore in modalitá sistema (kernel mode) nella quale vengono eseguite le chiamate di sistema. In generale, le interruzioni possono essere di tio software o di tipo hardware. Entrambe sono gestite con il meccanismo illustrato in fig.1.2. In particolare, le chiamate di sistema sono prodotte dalle istruzioni 'trap' che producono iterruzioni software.

```
Sched()
{
   while(true)
      GestisceCodaProcessi();
}
GestisciCodaProcessi()
{
   SelezionaPCB();
   EstraiPCB();
   StartTimer();
   EseguiPCB();
}
GestioneInterruptTimer()
{ SalvaContesto(); //in PCB
   Sched();
}
```

Figura 1.4 Pseudocodice di un sistema operativo a divisione di tempo

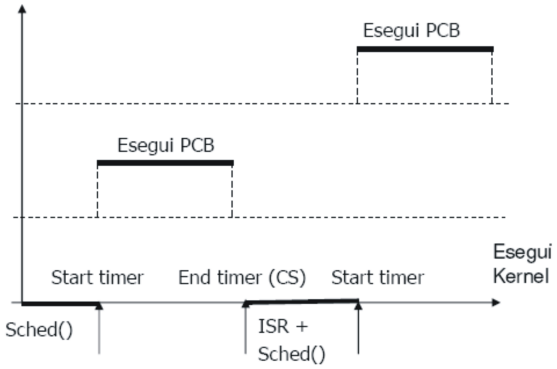

Figura 1.5 Esecuzione dei processi a divisione di tempo

La protezione del sistema si realizza costruendo questi confini in maniera molto precisa: in genere vengono progettati in modo da poter essere attraversati solo con l'uso delle **trappole** software (le **trap**); ovvero se mi trovo in un livello e voglio accedere a quello sottostante *devo volerlo fare* e invoco le trappole: questo serve per evitare problemi accidentali. Ma cosa sono le trappole?

Le trappole rappresentano il meccanismo principale attraverso cui si cambia di livello: esse non sono nient'altro che delle *interruzioni software* che provocano:

a. il salvataggio sullo stack dello stato del processore (del program counter);

b. il cambiamento della **modalità** di esecuzione del processore.

I processori hanno almeno due modalità di esecuzione: **user** e **system**. Non in tutte le modalità si possono eseguire le stesse istruzioni: per esempio se il processore è in modalità user si hanno delle restrizioni su alcune funzioni assembler. Per quanto riguarda la struttura onion skin, questa prevede sei modalità di esecuzione – come si può vedere dalla figura **??** – e sono quelle consentite dal processore (ogni processore infatti può offrire o meno determinate modalità).

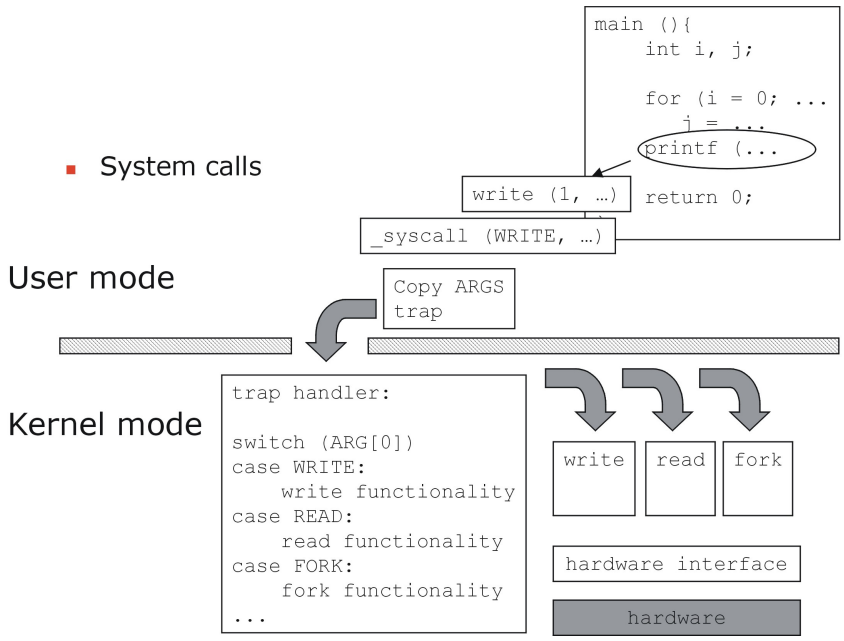

Figura 1.6 Il meccanismo delle chiamate di sistema

Il kernel oltre a mettere a disposizione le interfacce per le chiamate di sistema possiede anche tutto il software in grado di controllare direttamente l'hardware del calcolatore, la gestione dell'I/O, la gestione del file system e la gestione dei processi. Il **sottoinsieme di gestione del file system** gestisce i file e la memoria secondaria, alloca lo spazio per i file, amministra lo spazio libero su nastro o disco, controlla l'accesso ai file e ricerca i dati per conto di ogni utente. Il **sottoinsieme di controllo del hardware** e' responsabile della gestione delle interruzioni (interrupt) e delle comunicazioni con la macchina. I device di I/O a blocchi, cioe' i nastri ed i dischi, possono interrompere la CPU durante l'esecuzione di un processo ed il kernel puo' riavviare il processo interrotto dopo aver gestito l'interruzione.

Il **sottoinsieme di gestione del file system** puo' accedere ai dati contenuti nei file utilizzando la **buffer cache** cioe' un meccanismo di buffering, che regola il flusso dei dati tra il kernel ed i device di I/O a blocchi, cioe' i nastri ed i dischi. Il meccanismo di buffering interagisce infatti con i device di I/O a blocchi per avviare il trasferimento dei dati da e verso il kernel. Il **sottoinsieme di gestione del file system** puo' interagire anche direttamente con i device di I/O a caratteri, cioe' le stampanti, lo schermo video del terminale, la tastiera del terminale e i dispositivi di rete, senza l'intervento di alcun meccanismo di buffering. I device driver sono memorizzati, come gia' osservato, sempre nel direttorio **/dev** e quindi sono esterni al kernel. Il **modulo dei processi di I/O** e' un'istanza dei device driver in esecuzione e quindi controlla l'operato dei device di I/O.

I processi interagiscono con il **sottoinsieme di gestione del file system** per mezzo di un insieme di particolari chiamate di sistema, come **open**( ) che apre un file in lettura oppure in scrittura, **close**( ) che chiude un file aperto, **read**( ) che legge da un file aperto, **write**( ) che scrive in un file aperto, **stat**( ) che richiede gli attributi di un file, **chown**( ) che cambia l'utente proprietario di un file e **chmod**( ) che cambia i permessi di accesso ad un file.

Il **sottoinsieme del kernel di gestione dei processi** risulta diviso in tre moduli. Il **modulo di schedulazione** detto anche **scheduler della CPU** che decide a quali dei processi pronti in memoria centrale bisogna assegnare il controllo della CPU. Lo scheduler della CPU gestisce quindi la schedulazione dei processi in modo da eseguirli a turno finche' lasciano volontariamente libera la CPU in attesa di una risorsa oppure il kernel decide che il tempo loro concesso e' gia' passato. Lo scheduler della CPU scegliera' allora il processo a priorita' piu' alta e lo fara' eseguire. Il

processo appena interrotto ripartira' invece quando tornera' ad essere il processo di priorita' piu' alta. L'algoritmo realizzato dallo scheduler della CPU e' comunque fortemente influenzato dalle tecniche di gestione della memoria. Il **modulo di comunicazione tra processi** si occupa invece della sincronizzazione dei processi, della comunicazione tra i processi e della comunicazione tra i processi e la memoria centrale. Ci sono diverse forme di comunicazione tra i processi che vanno dalla segnalazione asincrona degli eventi alla trasmissione sincrona di messaggi tra i processi. Una trasmissione asincrona non aspetta un messagio di conferma, mentre una trasmissione sincrona attende sempre un messagio di conferma. Il **modulo della gestione della memoria** si occupa della gestione della memoria centrale e controlla l'allocazione della memoria centrale. Ad esempio, durante la compilazione di un programma, il compilatore genera un insieme di indirizzi di memoria virtuale, che rappresentano gli indirizzi di memoria virtuale delle variabili, delle strutture dati e delle funzioni del programma stesso. Il compilatore pero' genera questi indirizzi di memoria virtuale come se nessun altro processo venisse eseguito simultaneamente sulla macchina fisica. Quando il programma viene invece eseguito, il kernel, tramite il **modulo della gestione della memoria**, alloca in memoria centrale tutto lo spazio necessario e compie la traduzione degli indirizzi di memoria virtuale in indirizzi fisici della macchina. Possiamo osservare inoltre che affinche' un processo possa essere eseguito, almeno una parte di esso deve essere contenuta in memoria centrale in quanto la CPU non puo' eseguire un processo che risiede interamente in memoria secondaria. Tuttavia la memoria centrale e' una risorsa preziosa e spesso non puo' contenere tutti i processi. Se il sistema operativo Unix in qualunque momento non dovesse avere memoria centrale sufficiente per tutti i processi il **modulo della gestione della memoria** decide quali processi possono risiedere, almeno in parte, in memoria centrale e gestisce quei segmenti di spazio di indirizzi di memoria virtuale dei processi che non sono piu' memorizzati in memoria centrale. Oltre a questo il **modulo della gestione della memoria** controlla la quantita' di memoria centrale disponibile e puo' periodicamente trasferire i processi da e verso un device di I/O a blocchi chiamato **swap device**, che tipicamente e' un disco oppure un'area del file system. Lo **swap device** viene dunque utilizzato per la memorizzazione temporanea dei processi che non possono restare in memoria centrale. Quando un sistema e' sovraccarico oppure dispone di memoria centrale insufficiente, viene utilizzato a rotazione lo **swap device** per consentire l'esecuzione di tutti i processi. Purtroppo l'utilizzo frequente della **swap device** comporta sempre un aumento considerevole dell'inefficenza del sistema.

Il **modulo della gestione della memoria** si occupa dunque di scambiare i processi tra la memoria centrale e lo **swap device** in modo da permettere a tutti i processi le migliori condizioni di esecuzione. Inizialmente tutti i sistemi operativi Unix trasferivano interi processi tra la memoria centrale e lo **swap device** e non traferivano indipendentemente parti di un processo ad eccezione delle aree di testo condivise. Questa modalita' di gestione della memoria centrale e' detta **swapping** (scambio) ed era utilizzata per la diffusissima serie Digital PDP-11 dove la dimensione massima di un processo era di 64 kbyte. Anche i moderni sistemi operativi Unix prevedono la modalitá di gestione della memoria centrale detta **swapping** e cioe' il trasferimento di un processo dalla memoria centrale nello **swap device**, quando il kernel ha bisogno di spazio in seguito al verificarsi, per esempio, di una chiamata di sistema per la gestione dei processi **fork**( ), che crea un nuovo processo senza liberare la memoria occupata dal processo chiamante. In tal caso il processo 0 del kernel detto **swapper** realizza l'**algoritmo di swapping** cioe' si occupa della gestione dello spazio sullo **swap device**, del trasferimento del processo a priorita' piu' bassa (tra quelli che sono in memoria centrale) dalla memoria centrale allo swap device e del trasferimento del processo a priorita' piu' alta (tra quelli che sono nello swap file) dallo swap device in memoria centrale.

1.2 La Comunicazione tra processi (Inter Process Communicatons o IPC)

Il meccanismo della comunicazione e' descritto nella seguente figura:

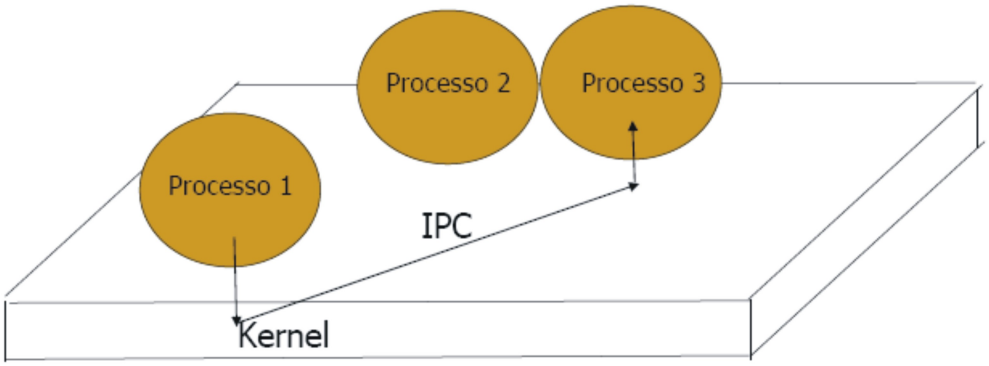

Figura 1.7 Interazione tra processi tramite ipc

La comunicazione tra processi é realizzata mediante alcune primitive (realizzate medante **chiamate di sistema**che sono state introdotte nella distribuzione Unix System V che é stata rilasciata dai Laboratori Bell nel 1983. I meccanismi introdotti con quella versione di Unix sono stati poi utilizzati e realizzati nelle versioni seguenti e sono state standardizzati con gli standard Posix.

Le primitive sono realizzate secondo una filosofia comune, descritta dalle seguenti caratteristiche.

- ogni primitiva contiene una tabella i cui elementi sono relative alle istanze del meccanismo, descritte mediante una chiave numerica
- ogni primitiva consiste in un certo numero di chiamate di sistema. La prima chiamata di sistema di ogni primitiva crea una istanza del meccanismo di comunicazione oppure si aggancia ad una istanza pre-esistente. A questa chiamata di sistema viene fornita una chiave numerica che, se corrispondente ad una istanza pre-esistente, si aggancia ad essa. Altrimenti, se la chiave é IPC_PRIVATE viee crata una nuova istanza e viene assegnata una chiave non utilizzata. Questa chiamata di sistema ritorna un descrittore della primitiva che viene usata nelle seguenti chiamate di sistema in modo simile alle chiamate di gestione dei file.
- ogni elemento della tabella contiene i permessi d'uso da parte degli utenti, l'identificatore dell'utente e del processo
- ogni primitiva contiene una seconda chiamata di sistema che controlla lo stato della comunicazione o puó chiuderla.
- ogni primitiva contiene una chiamata di sistema che effettua la comunicazione desiderata.

Le primitive di IPC sono:

1. **invio di segnali tra processi**
2. **scambio tra processi imparentati tramite PIPE**
3. **scambio tra processi anche non imparentati tramite FIFO**
4. **passaggio di messaggi tra processi**

5. **scambio informazioni tramite memoria condivisa**
6. **controllo concorrenza mediante semafori**
7. **invio di informazioni tra processi tramite socket**

Capitolo 2

La concorrenza

2.1 Introduzione

Le motivazioni principali della concorrenza sono da un lato Funzionali, cioé legate al vantaggio di poter avere diverse operazioni simultanee sul calcolatore, cosa che consente di risolvere agevolmente problemi pratici (si pensi alla possibilitá di avere un editor attivo per controllare i parametri di un programma simultaneamente in esecuzione). D'altro lato le Prestazioni del sistema di elaborazone possono aumentare in virtú della sovrapposizione tra operazioni di I/O ed esecuzione. Infine la Potenza espressiva delle applicazioni viene esaltata dal fatto che molte attivitá sono inerentemente concorrenti e quindi possono essere descritte molto piú agevolmente usando la concorrenza.

Pe quanto riguarda il Costo della concorrenza, esso vá identificato non solo nella complessitá del sistema operativo, ma anche e sopratutto nei problemi nei quali puó incorrere un processo concorrente, cioé Mutua esclusione, determinatezza e Stallo dei processi.

Il meccanismo del **context switch** realizza la concorrenza, cioé realizza quel meccanismo che consente a piú processi di eseguire simultaneamente. Naturalmente la simultaneitá nel caso di una CPU non é reale perché i processi possono eseguire solo uno alla volta; tuttavia i problemi e le soluzioni sono le stesse che si avrebbero nel caso di piú CPU (nel qual caso si parla di parallelismo).

Nel caso di una sola CPU l'esecuzione procede per intersfogliatura (interleaving) delle istruzioni. Nel caso ci siano piú processi l'esecuzione comprende anche delle sovrapposizioni.

Figura 2.1 Sequenza di esecuzione per una CPU

I termini P1, P2 etc sono i processi in esecuzione e P11, P12 etc sono singole unitá computazionali del processo. Nella figura si vede come la sequenza che viene eseguita é ottenuta dalla concatenazione non prevedibile di unitá computazionli.

Elenchiamo ora altri concetti fondamentali della concorrenza.

- Sezione critica : é una sezione di codice che accede ad una risorsa condivisa con un'altro processo. La sezione critca non puó essere eseguita se un altro processo si trova all'interno della sua sezione critica sulla stessa risorsa.
- Stallo (deadlock): situazione di blocco di due processi nella quale ciascun processo aspetta che l'altro faccia qualcosa prima di eseduire.
- Livelock (blocco vivo): é una situazione di blocco di due processi nella quale ciascuno riceve delle richieste dall'altro processo ma non l'esecuzione corrisondente non porta niente di utile. I processi sono bloccati.
- Mutua esclusione (mutual exclusion): descrive la situazione nella quale una risorsa condivisa tra piú processi puó essere usata solo da un processo alla volta. Non ci possono essere piú processi che usano simultaneamente quella risorsa.
- Contesa (race condition): la situazione nella quale piú processi accedono una risorsa condivisa e il risultato dipende dall'ordine di accesso.
- Morte per inedia (starvation): la situazione nella quale non c'e' un blocco ma una attesa indefinita.
- Atomicitá: condizione nella quale una sequenza di istruzioni o una procedura non viene interrotta dal context switch, ma esegue dall'inizio alla fine senza interferenza

La descrizione della concorrenza data nei capitoli precedenti é stata fatta dando maggiore enfasi al sistema operativo, cioé alle sue strutture dati e agli algoritmi. In questo capitolo l'enfasi viene spostata sull'utente, cioé al programmatore che deve costruire programmi concorrenti.

Il primo problema riguarda la sequente domanda: se il SO in Time sharing trova diversi processi nella coda dei PCB li esegue in concorrenza, ma come faá l'utente a generare questo processo?

Piú in generale, come fá l'utente a fare in modo che delle unitá computazionali siano concorrenti?

Per fissare le idee, facciamo un semplice esempio:

Figura 2.2 Esempio di processo che legge, elabora e scrive da un nastro

Il primo DAT contenga record sequenziali da elaborare con la CPU; i risultati vadano poi scritti sul secondo DAT. La sequenza ideale di lettura,esecuzione e scrittura potrebbe essere:

L1-E1-S1-L2-E2-S2-L3-E3-S3-...

E' ovvio che non è possibile leggere il secondo record finchè non si è finito di leggere il primo, ma la struttura del processo potrebbe venir modificata; a tale proposito, ricordiamo alcuni concetti giá noti che serviranno per avere le idee più chiare:

- **Processo** = sequenza di stati di esecuzione
- **Programma** = algoritmo, può dar luogo a molti processi

Cerchiando i processi singoli [1]che costituiscono il processo in esecuzione, quest'ultimo appare composto dalla concatenazione di sottoprocessi, ovvero di task:

Figura 2.3 Elaborazione sequenziale dei tre processi

Si possono introdurre dei vincoli temporali di sincronizzazione sui task: ad esempio imponiamo di non poter iniziare il task L2 finchè non sia finito L1:

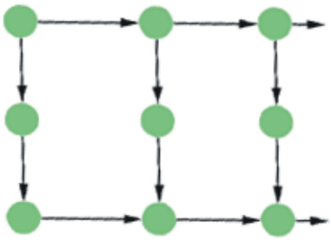

Figura 2.4 Elaborazione sequenziale dei tre processi

In altri termini, i pallini sulla riga piú in alto rappresentano i task di lettura, quelli nella riga intermedia i task di esecuzione e quelli nella riga in basso sono i task di scrittura. Il secondo record sul nastro puó essere letto solo dopo che é stato letto il primo record, essendo il nastro sequenziale. Ugualmente, l'elaborazione del primo record puó iniziare solo dopo che é stato letto il primo record.

Sui task di **elaborazione** nella figura precedente **non** ci sono i vincoli temporali: allora Ei e Ej possono diventare **concorrenti**. La concorrenza può esistere solo quando **non ci sono vincoli** tra i task.

Questa situazione però è prettamente teorica perché in realtà esistono **sempre** dei vincoli tra i task.[2]

La struttura visualizzata in figura precedente rappresenta **un grafo** e si può riorganizzare come segue:

Se i task fossero realmente **indipendenti**,ognuno di essi avrebbe una struttura sequenziale: in tal caso la presenza di più processi indipendenti significherebbe presenza di processi **paralleli**. Introducendo invece vincoli temporali tra task di processi indipendenti, si potrebbe introdurre un ordinamento parziale nel grafo, che altrimenti appare come segue:

NB: In questa discussione, saremo interessati ad un ambiente parzialmente ordinato con task concorrenti.

Non è necessario che i task siano tutti concorrenti e la concorrenza stessa si può avere solo tra task che non siano collegati da frecce.

I **collegamenti tra i task possono essere di diverso tipo**, principalmente:

- **A)** COOPERAZIONE: i singoli task si possono passare delle informazioni
- **B)** CONTESA: la stessa risorsa può essere richiesta in contemporanea da due task, rendendo così necessaria la presenza di un **arbitro**

[1] e dunque pensati come atomici

[2] Altrimenti si potrebbe usare l'elaborazione parallela

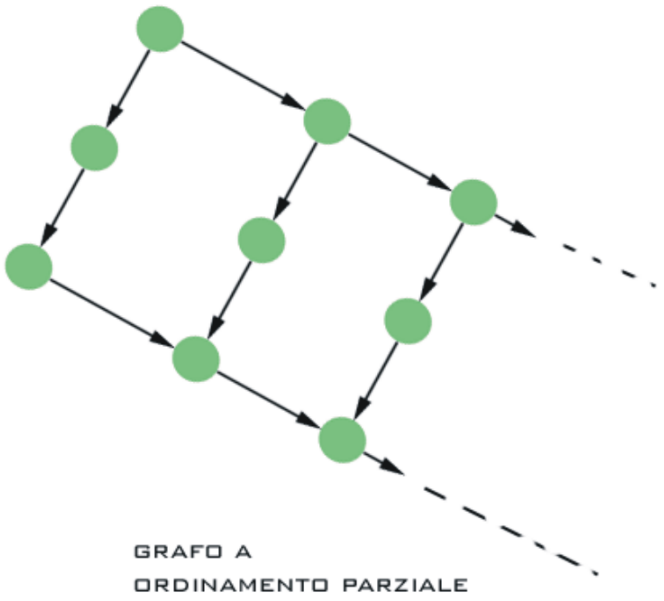

Figura 2.5 Elaborazione sequenziale dei tre processi

- **C)** INTERFERENZA: task che interferiscono in maniera non desiderata dal programmatore [3]

- **D)** SINCRONIZZAZIONE. Ad un certo istante, piú processi concorrenti devono attendere l'un l'altro in modo tale da realizzare una determinata operazione. La sincronizzazione non é legata solo alla protezione di una risorsa condivisa, ma anche al rispetto di un ordine di funzionamento.

I vincoli di sincronizzazione, che coinvolgono strutture semaforiche utilizzate con le primitive del Kernel, vengono introdotti per **risolvere** i problemi che nascono nei casi B) e C).

Per avere programmazione concorrente è necessario che l'ambiente la supporti. Allora:

- Da qualche parte ci devono essere uno scheduler ed il clock di timesharing. Essi possono essere implementati:
  Via HW, attraverso l'uso del timer di context switch. Un esempio di schedulazione é stato visualizzato all'inizio di questo capitolo.
  Via SW, come ad esempio in java

- Devono essere presenti dei metodi per esprimere le biforcazioni (del grafo a ordinamento parziale). Ad esempio la prima operazione di lettura L1 vista [4] fa nascere due task: E1 e L2. Ciò sottolinea come siano necessari **strumenti linguistici per la concorrenza**

I processi non condividono lo spazio di indirizzamento! Ma: se i processi devono cooperare? Le Soluzioni alla comunicazioni tra processi vanno sotto il nome di IPC (InterProcess Communications). Questa situazione puó essere modellata com rappresentato in fig.3.2. Quando un processo vuole comunicare con un altro, lo puó fare attraverso una chiamata di sistema, cioé attraverso l'esecuzione di un particolare algoritmo di nucleo. Queste comunicazoni attraverso le IPC sono quindi

[3] sono errori di programmazione
[4] nel sistema batch dell'esempio

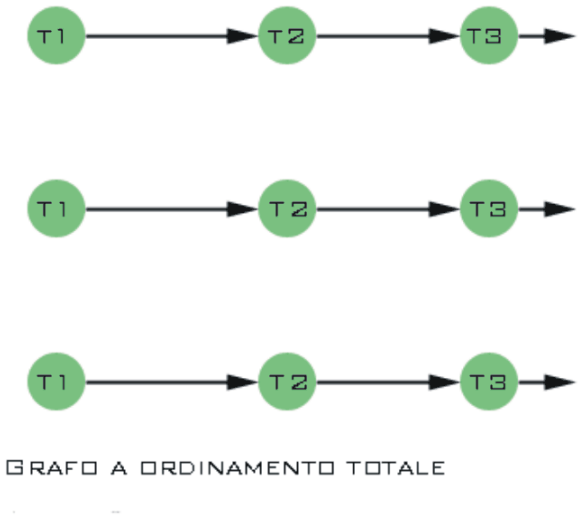

Figura 2.6 Processi indipendenti

sostanzialmente lente. Un altro problema é che il cambio di contesto é piuttost pesante perché deve essere coinvolto tutto un ambiente.

Una soluzioni a questi problemi é quella di aver dei processi che condividono lo spazio di indirizzamento, e che quindi vivono nello stesso ambiente. Questi sono i Thread (codice eseguito nello stesso spazio di indirizzamento). Il fatto che i processi vivono nello stesso ambiente porta al fatto che questo modo di generare la concorrenza é particolarmente veloce.

2.1.1 I thread

Ci sono due tipi di programmazione concorrente:

1. la programmazione concorrente a **processi concorrenti**;
2. la programmazione concorrente a **thread concorrenti** ('94).

Analizziamone le differenze.

Il processi concorrenti si possono vedere come delle sfere non compenetrabili: ogni processo è mondo a sè stante con il suo ambiente e con il suo spazio d'indirizzamento (ovvero con il suo indirizzo virtuale minimo e indirizzo virtuale massimo entro i quali i processi possono esistere). I processi concorrenti sono qualcosa di estremamente complesso switchare: passare da un processo all'altro significa cambiare completamente ambiente. La programmazione a thread cerca di semplificare tutto questo facendo sì che all'interno del processo ci siano diverse sequenze di esecuzione: ad esempio un programma composto da diverse procedure "azionate" in parallelo anzichè sequenzialmente. Le procedure di ogni singolo processo diventano dei thread. Il vantaggio sta nel fatto che per switchare da un thread all'altro non devo switchare tutto l'ambiente ma solamente un **program counter**, che è quello che seleziona ogni singolo istante ciò che dev'essere eseguito (devo cambiare un puntatore, i dati a questo punto sono condivisi da tutti i thread in esecuzione).

Tipicamente ci sono due context switch: uno a livello di sistema e uno a livello utente. I thread all'interno di un processo sono switchati (schedulati) a livello utente, mentre è il sistema operativo

a livello di kernel che si occupa della schedulazione dei processi nella loro interezza. Se pensiamo a un programma in java (che contiene al suo interno la possibilità di generare un insieme dui thread concorrenti) la commutazione fra thread viene fatta completamente a livello utente: questo rappresenta un vantaggio in quanto una schedulazione e livello utente avviene con maggior facilità.

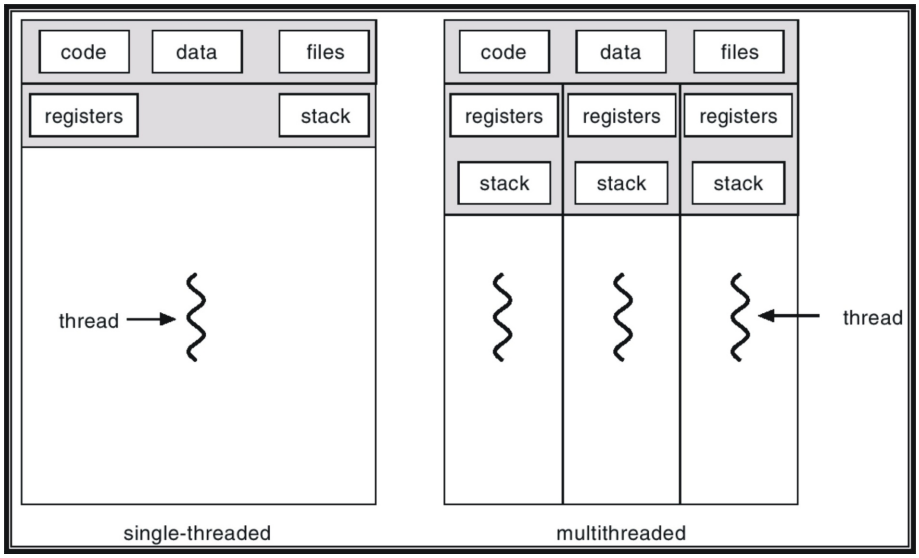

Figura 2.7 (a) Tre processi ognuno di un thread (b) Un processo con tre thread.

I thread ormai sono la modalità di programmazione concorrente per eccellenza perché garantiscono una maggior semplicità ed efficienza (anche se comportano tutta una serie di complicazioni a livello implementativo): ad esempio in molti sistemi operativi le interruzioni a livello thread vengono effettuate direttamente dall'applicazione (anche perché spesso il kernel non vede thread ma solo processi).

Un'altra osservazione: il blocco di un thread non blocca il processo (un thread bloccato sull'attesa di dati che non arrivano viene semplicamante commutato): questa caratteristica fa sì che i thread vengano utilizzati in particolar modo per la programmazione dei server (spesso i server accedono a uno o più dischi che possono essre bloccati: non è pensabile naturalmente che l'intero server si blocchi).

A proposito delle complicazioni: il fatto che i dati sono condivisi se da un lato mi evita l'uso degli IPC per far comunicare due processi, dall'altro introduce la possibilità che i dati possano essere "sporcati" dai vari thread: viene introdotta quindi la necessità di un meccanismo di sincronizzazione per l'uso delle risorse condivise.

Concludendo ci sono almeno tre vantaggi con l'utilizzo dei thread:

- commutazione molto più efficiente;
- il blocco di un thread non provoca il blocco di un intero processo;
- i thread vedono lo stesso ambiente, quindi i dati vengono visti da tutti i thread.

Esempio

Per mostrare un esempio di come si possono utilizzare i thread consideriamo i browser per il Web. Molte pagine Web contengono un buon numero di piccole immagini. Per ogni immagine il browser deve stabilire una diversa connessione verso il sito relativo alla pagina e richiedere l'immagine. Una gran quantità di tempo viene persa per stabilire e chiudere tutte le connessioni necessarie. Se il browser comprende più thread è possibile richiedere molte immagini allo stesso tempo, migliorando sostanzialmente le prestazioni nella maggior parte dei casi.

In generale i thread sono gestiti con Librerie a livello utente (come in Java), a Livello Kernel (Mach), oppure a Livello kernel+utente (Solaris).

I vantaggi dei thread, oltre alla velocitá detta, sono legatei al fatto che le chiamate bloccanti non bloccano l'intero processo ma solo un thread.

In sintesi, possiamo dire che un Processo é una entità computazionale, dotato di Identificatori del processo, Risorse e limiti (memoria, disco, tempo di CPU),spazio di indirizzamento virtuale, mentre un Thread é una entità computazionale che condivide, con altri Threads, lo spazio di indirizzamento, le risorse, i limiti del processo.

Modelli di implementazione dei thread: Thread a livello utente, piú veloci da creare, con Context Switch piú veloce dei precedenti. Lo svantaggio di un kernel a singolo thread é che se un thread esegue una system call bloccante, tutti gli altri thread che condividono lo stesso spazio di indirizzamento si bloccano. Naturalmente, nei Thread a livello kernel il kernel controlla i singoli thread e li schedula indipendentemente, con il conseguente vantaggio che anche se il kernel è a singolo thread, se un thread esegue una system call bloccante gli altri thread non si bloccano. Metodi misti:

- Da molti a uno (Solaris2)
- Da uno a uno (come Linux, WinNT, Win2000)
- Da molti a molti (come Solaris2, Irix, HP-UX)

Nel caso da molti a molti le chiamate di sistema sono disponibili tramite **wrappers** alle system calls bloccanti: prima che si blocchino l'esecuzione é passata ad un'altro thread

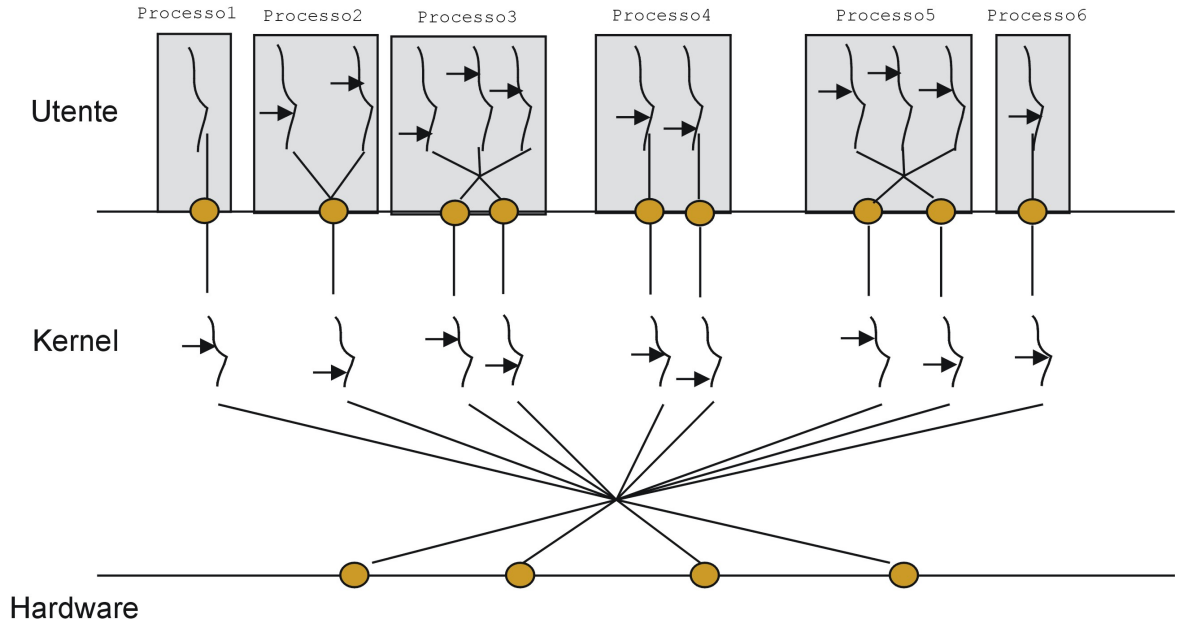

Figura 2.8 Implementazione dei thread

I Thread in Solaris 2, sono offerti al programmatore in diverse modalitá: a livello utente, a livello del nucleo e a livello intermedio. A livello utente contengono identificatore, registri, pila, priorità. A livello del nucleo contengono copie dei registri del nucleo, puntatore a LWP, informazioni di scheduling e priorità. A livello intermedio (LWP) contengono un insieme di registri relativi ai thread che eseguono, risorse di memoria e informazioni di contabilizzazione.

I Thread in Windows 2000: sono realizzati secondo il modello uno a uno. Contemgono: identificativo, insieme di registri, pila utente, pila nucleo.

I Thread in Linux sono presenti dalla versione 2.2 Sono invocati con la chiamata di sistema clone: processo distinto che condivide lo spazio di indirizzi del processo chiamante. Sono oggetti di sistema. Il sistema non distingue tra processi e thread.

In conclusione possiamo dire che la creazione di un task cooncorrente da parte di un utente puó essere effettuata o tramite una chiamata di sistema che crea un nuovo PCB e lo mette in coda assieme agli altri, o mediante delle chiamate di libreria utente che creano un thread.

Tuttavia questi metodi non sono i soli.

2.2 Costrutti linguistici per la programmazione concorrente

Un **costrutto linguistico** é una istruzione fornita da un linguaggio ad alto livello. Spesso la realizzazione di queste istruzioni richiede alcune chiamate di sistema, ma questo riguarda il creatore del linguaggio.

2.2.1 Coroutine

Probabilmente il costrutto piú semplice é quello delle **CO-ROUTINE**. Le coroutine sono un meccanismo che consente al programmatore di stabilire l'ordine di attivazione. La programmazione diviene dunque **quasi concorrente**. Il meccanism é simile ai thread, nel senso che vengono attivate delle procedure all'interno dello stesso ambiente di programmazione, con la differenza che la schedulazione delle procedure non é lasciata ai meccanismi del sistema operativo ma é responsabilitá del programmatore. Il passaggio tra una coroutine e l'altra viene realizzata con la primitiva **resume(SP)** che ha il compito di passare dalla coroutine corrente alla coroutine caratterizzata dal puntatore SP_dest (stack pointer destinazione). In figura sono rappresentate tre coroutine, chiamate A, B e C, e mediante la resume il programmatore gestisce la schedulazione tra esse.

Ogni coroutine richiede uno stack separato per memorizzare gli indirizzi di ritorno, nella figura gli indirizzi di ritorno dopo le resume sono indicati con ind, ind2, ind3, ind4. Assunto che A, B e C rappresentino i puntatori agli stack delle tre coroutine, che inizialmente puntano agli indirizzi iniziali delle tre routine, e che lo stack pointer della routine attualmente in esecuzione si chiama SP_actual, il funzionamento della primitiva **resume(SP_dest)** é il seguente:

- preleva l'indirizzo puntato dal puntatore SP_dest e mettilo in addr, cioé addr=(SP_dest)
- incrementa SP_actual
- scrivi l'ndirizzo di ritorno *ind* nella locazione puntata da SP_actual, cioé (SP_actual)=ind
- continua l'esecuzione dall'istruzione addr (cioé PC=addr dove PC é il Program Counter)

In questo modo il funzionamento del codice é descritto dalle seguenti figure.

2.2.2 Cobegin-Coend

Una secondo modo per eseguire in concorrenza dei thread utilizza il costrutto **cobegin - coend**. La scrittura

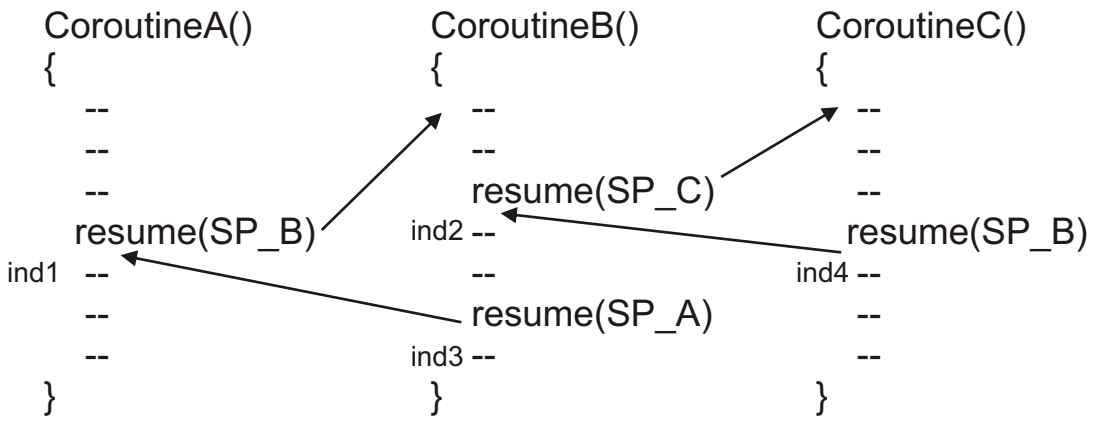

Figura 2.9 Programmazione quasi concorrente con le coroutine

```
cobegin
    P1(); P2(); P3();
coend;
```

esegue in concorrenza i processi P1, P2 e P3. L'istruzione `coend` rappresenta il punto di sincronizzazione dei tre processi, nel senso che l'esecuzione della istruzione situata dopo `coend` aspetta che siano terminati tutti i tre processi.

Bisogna peó fare attenzione al fatto che non tutti i vincoli di precedenza sono descrivibili con questo costrutto. Esempio: il grafo di thread concorrenti visualizzat nella prossima figura:

é realizzabile con questo codice:

```
begin
   A;
   cobegin
      begin A; F; end;
      begin
            B;
            cobegin
               D; E;
            coend;
      end;
    coend;
    G;
end;
```

mentre il prossimo grafo non é realizzabile con `cobegin - coend`.

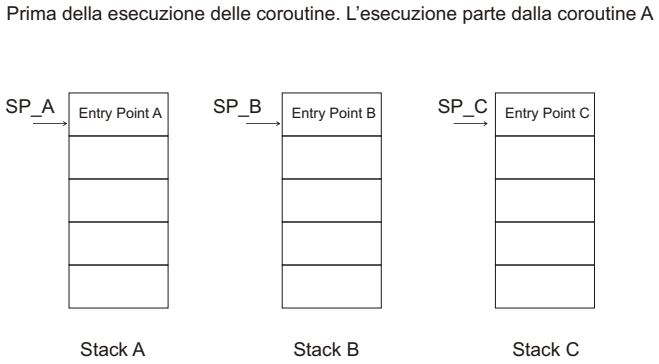

Figura 2.10 Possibile gestione degli stack all'inizio della esecuzione delle tre coroutine

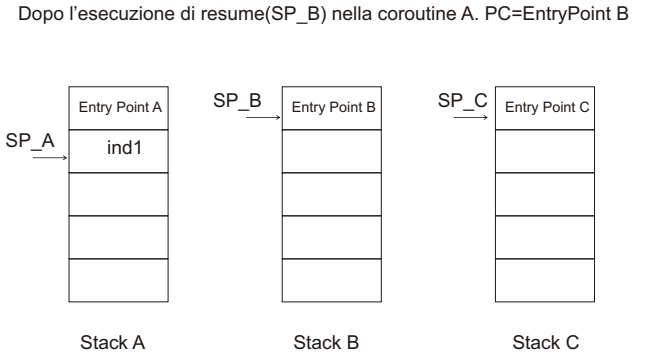

Figura 2.11 Possibile gestione degli stack dopo la prima resume

É necessario usare un altro meccanismo per realizzare questo grafo.

2.2.3 Il costrutto fork-join

Questo meccanismo é fornito dal costrutto linguistico `fork-join`. La chiamata `fork` é simile alla `call`. Il programma che chiama la `fork`, tuttavia, continua la sua esecuzione, perché l'effetto della chiamata é di creare un nuovo processo, come rappresentato nella seguente figura:

Il costrutto `fork` é anche chiamato chiamata asincrona di procedura. I due o piú processi creati possono essere raccordati con il costrutto `join`. La funzione della `join` é di aspettare la terminazione della esecuzione di tutti i flussi di esecuzione collegati.

Un modo per realizzare queste primitive é mediante la definizione di un tipo di dato, diciamo `processo`, che é realizzato dal linguaggio. In questo modo la fork restituisce un valore di tipo `processo` e la join usa la variabile di tipo `processo` per specificare con quale processo intende sincronizzarsi. In questo modo la scrittura di queste primitive potrebbe essere del tipo:

```
processo P;

P = fork(routine);
```

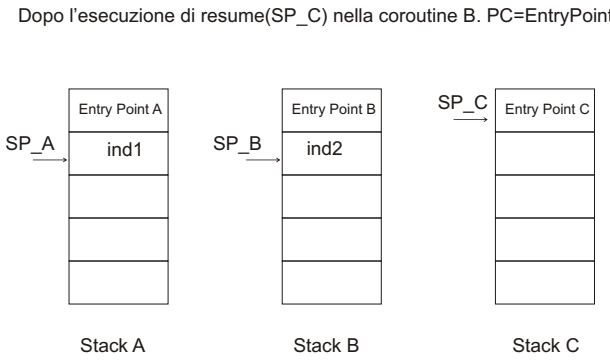

Figura 2.12 Possibile gestione degli stack dopo la seconda resume

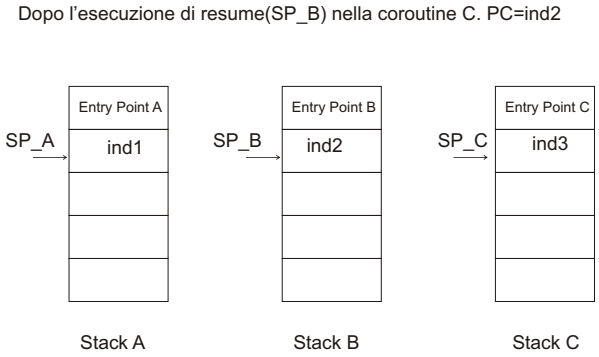

Figura 2.13 Possibile gestione degli stack dopo la terza resume

```
join P;
```

Tuttavia, nel seguito di questo capitolo e per motivi di semplicita' si userá la notazione semplificata `fork(A); join A;` supponendo quindi che l'entri point del processo sia anche il suo identificatore.

Facendo riferimento alla figura precedente possiamo indicare che i due processi B e C possono essere raccordati nel modo descritto nella figura precedente.

Riprendendo ora il grafo alle precedenze non realizzabile con cobegin-coend, vediamo come puó essere invece realizzato con fork-join.

```
A();
fork(B); C(); //mentre esegue B esegue C

join B; //aspetto che finisca B
// a questo punto sono terminati C e B
fork(F); fork(D); E();  // eseguono concorrentemente F, D, E
```

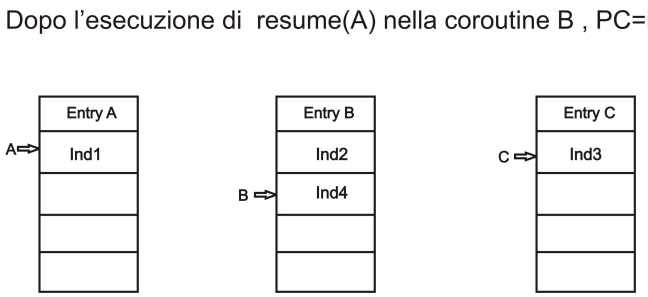

Figura 2.14 Possibile gestione degli stack dopo la quarta resume

```
join(F); join(D);  //aspetto la terminazione di F e D G();
// a questo punto e' terminato G ed E
H();
```

Da tenere presente che usando le primitive fork-join sono possibili molte diverse soluzioni: ad esempio l'esecuzione del grafo ora visto puó essere realizzata con diverse sequenze di esecuzione. La differenza tra una sequenza e l'altra sta nel diverso tempo di calcolo dovuto al fatto che le diverse routine hanno un tempo di allocazione diverso. Il problema di minimizzare il tempo di esecuzione complessivo diventa quindi un problema di schedulazione, che viene ora trascurato. In ogni caso, deve essere garantito che il risultato finale sia lo stesso per ogni possibile sequenza di esecuzione. Questo é un problema di determinatezza al quale é dedicato un prossimo capitolo .

In queste considerazioni abbiamo assunto l'esistenza di questi costrutti; non é stata data nessuna informazione sul modo nel quale sono costruite le primitive stesse. Questo dipende ovviamente dal modo con cui é realizzato il linguaggio che offre queste primitive, assieme alle caratteristiche del sistema operativo sul quale esegue il linguaggio. In Unix la `fork(A)` é realizzata con la coppia delle chiamate di sistema fork-exec mentre la join con la chiamata di sistema `wait`.

2.3 Programmazione concorrente in Java

Un modo per generare thread é attraverso l'uso del linguaggio Java. In Java ogni thread é un oggetto, creato come istanza della classe java.lang.Thread. La classe Thread contiene tutti i metodi per gestire i threads. L'utente implementa il metodo run(). Uno dei metodi piú importanti é il metodo start() che lancia il thread utilizzando il metodo run definito dall'utente. Ogni istanza di Thread deve quindi essere associata ad un metodo run. Ci sono due metodi per realizzare un thread: Implementando l'interfaccia Runnable Estendendo la classe java.lang.Thread e sovrascrivendo il metodo run(). Un thread termina quando

- Finisce
- Viene eseguito il metodo stop() del thread
- Scatta una eccezione

Lo scheduling e' effettuato tramite le priorita' (metodo setPriority()). A paritá di prioritá la schedulazione é effettuata mediante Round-Robin (si veda la parte dello scheduling).

I seguenti codici Java illustrano la creazione di un thread in Java mediante implementazione della classe **runnable**.

```
import java.io.*; public class PingPong{  //il main crea e lancia i
thread
```

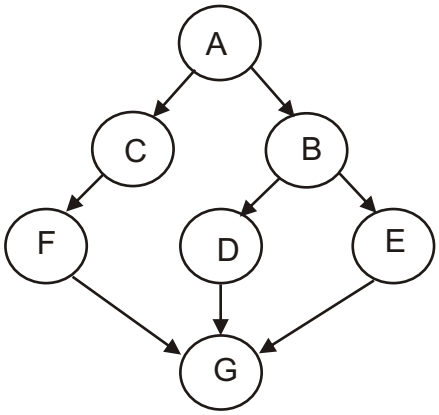

Figura 2.15 Un grafo alle precedenze

```
      public static void main(String[] a){
          Ping c=new Ping(); Pong t=new Pong();
          Thread th1=new Thread(c); th1.start();
          Thread th2=new Thread(t); th2.start();
      }
} class Ping implements Runnable{
   public void run(){
     while(true) {
       try{ Thread.sleep(800); } catch(InterruptedException e) {}
       System.out.println("Ping");
     }
   }
} class Pong implements Runnable{
   public void run(){
     while(true) {
```

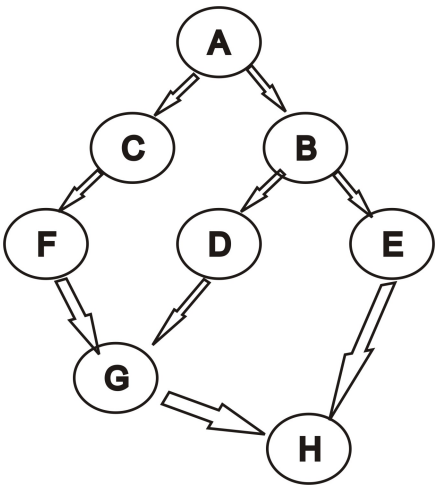

Figura 2.16 Un altro grafo alle precedenze non realizzabile con cobegin-coend

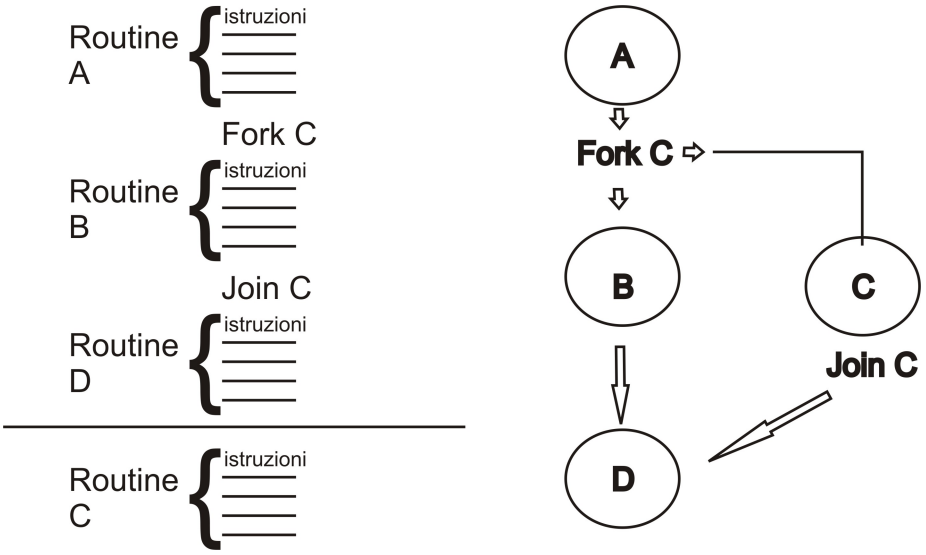

Figura 2.17 Il meccanismo della fork

```
        try{ Thread.sleep(990); } catch (InterruptedException e){}
        System.out.println("\tPong");
      }
    }
}
```

I seguenti codici Java illustrano la creazione di un thread in Java mediante estensione della classe **Thread**.

```
import java.io.*; public class PingPong2{ // il main crea e lancia i
thread
    public static void main(String[] a){
                Ping2 c=new Ping2(); c.start(); Pong2 t=new Pong2(); t.start();
    }
} class Ping2 extends Thread{
    public void run(){
```

```
                while(true) {
                        try{ Thread.sleep(800); } catch(InterruptedException e) {}
                        System.out.println("Ping");  }
    }
} class Pong2 extends Thread{
    public void run(){
                while(true) {
                    try{ Thread.sleep(990); } catch (InterruptedException e){}
                    System.out.println("\tPong");     }
    }
}
```

Vediamo ora una possibilitá per realizzare la condivisione di variabili tra Threads. Ovviamente se i Threads sono scritti nella stessa classe, i parametri sono condivisi. Se i thread non sono scritti nella stessa classe, é possibile condividere una classe (che puó contenere strutture dati e metodi) passando il suo indirizzo.

In altri termini, tutti i thread devono condividere un oggetto, che contiene i dati e i metodi condivisi. La condivisione viene effettuata mediante definizione del puntatore all'oggetto in ciascun Thread, e mediante l'inizializzazione del puntatore all'oggetto. L'oggetto in realtá viene allocato nella classe principale (quella che contiene il main). Il codice riportato nel seguito rappresenta un esempio di due Threads chiamati pi e po, che si scambiano 5 reali, con un ritardo (genera sequenzialitá quindi in questo caso non ci sono problemi di mutua esclusione).

Questa é la classe condivisa:

```
import java.io.*; public class z{
        float b[]= new float[10];

        void put(int i, float f){  b[i]=f;  }

        float get(int i){ return(float)b[i];  }
}
```

Questo il thread principale (main):

```
public class pth{
    public static void main(String[] a){
                    z buf=new z();

                    pi c=new pi(buf); po t=new po(buf);
                c.start(); t.start();
    }
}
```

Infine, il codice dei due thread:

```
public class pi extends Thread{
    z buf;
    pi(z buf){  this.buf=buf;   }  //costruttore
    public void run(){
            while(true) {
                try{  Thread.sleep(800);} catch(InterruptedException e) {}
                System.out.print("leggo ");
```

```
                for (int i=0; i<5; i++) System.out.print( "+buf.get(i));
                System.out.println();
            }
    }
}

public class po extends Thread{
    z buf;
    Random r=new Random();
    po(z buf){  this.buf=buf;  }  //costruttore
    public void run(){
        while(true) {
               try{ Thread.sleep(990);} catch (InterruptedException e){}
               System.out.print("\tscrivo ");
               for(int i=0; i<5; i++) {
                        buf.put(i,r.nextFloat());
                        System.out.print(" "+buf.get(i));
               }
               System.out.println();
         }
    }
}
```

2.3.1 Supporti Java per la concorrenza

Gli stati di un thread in Java sono riportati nella figura seguente:

Alcuni metodi della classe Thread:

- public void start() //lancia il thread
- public void run() //esegue il codice
- public final void stop() //distrugge il thread
- public final void suspend() //sospende il thread
- public final void resume() //riattiva il thread
- public static void sleep(long n) //sospende il thread per n ms
- public final void setPriority(int priority) //modifica la prioritá
- public final int getPriority() //ottiene la prioritá corrente
- public static void yield() //rischedula
- public final native boolean isAlive() //esce con true se il thread è vivo

Un Thread di Java inizia chiamando il metodo start() e termina con il metodo stop().

Riguardo il diagramme degli stati di un thread, nello stato NonRunnable il thread é sospeso, cosa che puó avvenire per: attesa del completamento di una operazione di I/O o del periodo di una sleep(), chiamata del metodo suspend() o wait(). Viceversa il risveglio puó aversi per la fine della operazione di I/O o dell'intervallo di sleep(), chiamata dei metodi notify(), notifyAll() o resume().

La schedulazione dei thread é a prioritá, che é un valore che inizia con quello del thread padre ma puó essere variata con il metodo setPriority().

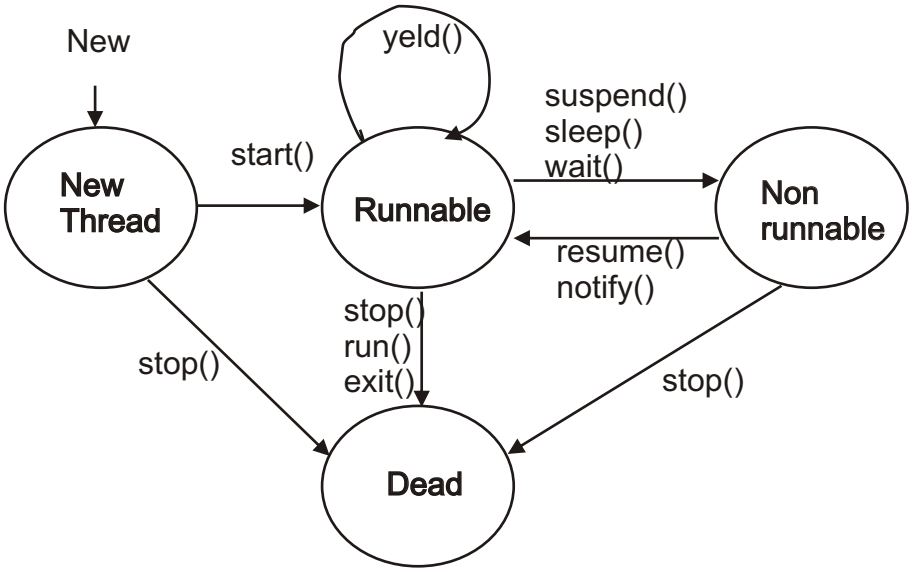

Figura 2.18 Stati di un Thread Java

I metodi per la sincronizzazione

Se il prgrammatore vuole sospendere un thread, deve chiamare il metodo wait all'interno di un metodo sincronizzato. La sospensione provocata da wait() é preceduta dal rilascio del blocco della sincronizzazione per consentire l'esecuzione di altri thread. Il thread sospeso viene messo in coda d'attesa sulla risorsa critica. Il metodo Notify() riattiva un thread della coda e il metodo notifyAll() riattia tutti i tread.

2.4 Alcuni problemi della concorrenza

La concorrenza non ha solo vantaggi, naturalmente. A parte una ovvia ma superabile complicazione della architettura del calcolatore, sia dal punto di vista hw che sw, i veri problemi sono nell'uso della concorrenza da parte dell'utente che consistono nel fare in modo che non ci siano interferenze dannose tra i vari processi concorrenti per quanto riguarda le risorse condivise tra i processi. I processi concorrenti sono esguiti in interleaving, cioé l'esecuzione di un processo puó essere interrotta in qualsiasi istante, causando dei possibili malfunzionamenti del programma. Nel seguito si elencheranno alcuni di questi possibili problemi, mostrando con un esempio il problema stesso.

2.4.1 Problemi di mutua esclusione

Si pensi alla tabella di Spooling, in cui ci sono due puntatori, IN e OUT, con la funzione di inserite da parte dei processi l'indirizzo dei dati da stampare in Tab[IN] e di fornire alla stampante l'indirizzo dei dati da stampare in Tab[OUT]. Supponiamo che due processi che devono stampare facciano le seguenti operazioni:

```
P1                                P2
```

```
---                                    ---
---                                    ---
//produci dati da stampare             //produci dati da stampare
//e mettili all'indirizzo ind          //e mettili all'indirizzo ind1
Tab[IN]=ind;                           Tab[IN]=ind1;
IN++;                                  IN++;
---                                    ---
---                                    ---
```

Il demone di Spooling, poi, stamperá i dati in un secondo momento. Le istruzioni Tab[IN]=ind1; IN++; in realtá corrispondono alle seguenti operazioni:

```
P1                                     P2
---                                    ---
---                                    ---
Tab[IN]=ind1;                          Tab[IN]=ind2;
temp=IN                                temp1=IN
temp=temp+1;                           temp1=temp1+1;
IN=temp;                               IN=temp1;
---                                    ---
---                                    ---
```

Supponiamo che ci sia una interruzione dopo Tab[IN]=ind1; e che l'esecuzione continui in P2, la quale sovrascrive Tab[IN] mettendoci ind2 sovrascrivendo cosí ind1. Se l'esecuzione continua con temp1=IN e subito dopo c'é il ritorno in P1, che incrementa IN. Tornando a P2 IN viene incrementato e sovrascritto correttamente. Ma se P2 continua fino a IN=temp1 e poi ritorna a P1, IN viene incrementato ancora mentre c'e' solo l'indirizzo ind2. Quindi si e' perso ind1 e IN e' incrementato due volte. Altri comportamenti sbagliati si possono avere interrompendo e ritornando in altri istanti. Questo é un tipico problema di mutua esclusione, nel senso che le isruzioni Tab[IN]=ind1; IN++; dovrebbero essere eseguite senza interruzioni.

2.4.2 Problemi di Sincronizzazione

Si consideri un semplicissomo esempio di programmazione parallela: vogliamo fare l'operazione (a+b)*(c+d) in modo parallelo, cioé facendo (a+b) in parallelo a (c+d). Ovviamente solo quando sono fatte le due somme possiamo fare il prodotto dei due termini, non prima. Se il programmatore non dispone di uno strumento per sincronizzare questo flusso operativo, il risultato non é corretto.

2.4.3 Problemi di Stallo dei processi

Si consideri la possibilitá si risolvere il problema dello spooling con un'altra variabile condivisa, chiamata *blocca*. Questa variabile viene usata nel modo seguente:

```
P1                                     P2
---                                    ---
---                                    ---
//produci dati da stampare             //produci dati da stampare
//e mettili all'indirizzo ind          //e mettili all'indirizzo ind1
while(blocco==1) {};                   while(blocco==1) {};
blocco=1;                              blocco=1;
Tab[IN]=ind;                           Tab[IN]=ind1;
IN++;                                  IN++;
```

```
blocco=0;                          blocco=0;
---                                ---
---                                ---
```

Se la variabile é inizializzata a 0, il primo processo entra e mette subito blocco a 1, bloccando cosí l'altro processo. Se peró c'é una interruzione dopo il while, puó entrare anche l'altro processo perché blocco é ancora a 0. Quindi non ho risolto il problema. Se blocco é a 1, entrambi i processi sono bloccati sul while e sono in stallo: ciascuno aspetta che l'altro processo metta la variabile a 0 per poter proseguire ma non ci arriverá mai.

2.4.4 Problemi di determinatezza

Si consideri i due processi paralleli, dove a, b, c, d sono inizializzate a 1:

```
P1                                 P2
---                                ---
a=b+c;                             d=2a;
---                                ---
```

Il risultato dipende dalla sequenza di esecuzione: se esegue prima P1 e poi P2, a=2, d=4. Se esgue prima P2 e poi P1, a=2, d=2. Se esegue prima P1 e si interrompe durante la somma, a=1, d=2. Cioé l'uscita non é determinata ma dipende da come eseguono i processi.