Programma: descrizione statica di un processo.

Processo: scaturisce con l'esecuzione di un programma da parte di un esecutore.

Risorse: si dividono in *prerilasciabili* (che possono essere sottratte ad un processo senza comprometterne l'esecuzione) e *non-prerilasciabili* (la cui sottrazione al processo che le sta utilizzando comporta il fallimento del processo stesso)

Molteplicita' di una risorsa: numero di processi che possono usare concorrentemente la risorsa. Risorse di molteplicita' finita necessitano di un accesso controllato. L'accesso si struttura in:

- un gestore della risorsa
- un protocollo di accesso alla risorsa

Vantaggi della programmazione concorrente:

- Migliorare la comprensione di un SO che regola diverse attività parallele
- Sfruttare le prestazioni ottenibili da architetture multi-processore, dato che un programma sequenziale non giova di una architettura parallela
- Migliorare la reattività delle applicazioni all'input dell'utente durante lunghe operazioni di I/O o di elaborazione
- La maggiore naturalezza con la quale si possono scrivere alcune tipologie di applicazioni (server, robotica, giochi, simulazioni di attività concorrenti)

Un **processo sequenziale** definisce un ordinamento totale sulle istruzioni.

Un **processo parallelo** definisce un ordinamento parziale sulle istruzioni, cioe' su alcune istruzioni l'esecutore è libero di scegliere quali iniziare prima e/o di eseguirle contemporaneamente.

Una sequenza di esecuzione ammissibile e' una sequenza di eventi che rispetta i vincoli espressi dal diagramma delle precedenze. Ad un certo diagramma delle precedenze corrispondono molteplici sequenze di esecuzione ammissibili.

La **sequenza di interleaving** e' la sequenza scelta dall'esecutore (in caso di istruzioni indivisibili ed unico esecutore).

Costrutti adottati per esprimere attivita' concorrenti:

- **fork** - Crea un processo figlio di quello attuale mediante attivazione del programma specificato. Restituisce l'identificatore del processo appena creato. Padre e figlio proseguono la loro esecuzione in modo indipendente.
- **join** - Il processo corrente rimane bloccato fintantoche' non termina quello specificato.

Con fork e join si puo' tradurre qualsiasi diagramma delle precedenze.
Vantaggi: espressivita' - flessibilita'
Svantaggi: poco astratti - non impongono nessuna struttura al programma concorrente.

cobegin - coend : esegue N istruzioni concorrentemente
Non sono sufficientemente espressive da esprimere qualunque diagramma delle precedenze.

Rango di una istruzione i : insieme delle aree di memoria alterate da i
Dominio di una istruzione i : insieme delle aree di memoria lette da i

Due istruzioni i e j possono essere eseguite concorrentemente se:

- rango(i) intersezione rango(j) = $\emptyset$
- dominio(i) intersezione rango(j) = $\emptyset$

- rango(i) intersezione dominio(j) = $\emptyset$

Quando un insieme di istruzioni soddisfa le condizioni di Bernstein, il loro esito complessivo sara' sempre lo stesso, indipendentemente dalla sequenza di interleaving adottata dall'esecutore.

In caso di violazione si ha interferenza.

Si ha interferenza quando:

- ci sono due o piu' flussi di esecuzione
- almeno un flusso di esecuzione esegue scritture

L'interferenza causa errori temibili in quanto dipendenti dalla particolare sequenza di interleaving scelta dall'esecutore, che e' al di fuori del controllo del programmatore.

Assunzione di progresso finito - Unica assunzione che possono fare i programmatori in merito alle velocita' relative dei processi e dei loro esecutori: tutti i processori hanno una velocita finita e non nulla.

Due situazione invaldanti l'assunzione di progresso finito:

- **Starvation** - quando un processo rimane in attesa di un evento che pure si verifica infinite volte. Un sistemi di processi che garantisce contro questa evenienza di definisce fair.
- **Deadlock** - quando due o piu' processi rimangono in attesa di eventi che non potranno piu' verificarsi a causa di condizioni cicliche nella richiesta e nel possesso di una risorsa.

Due processi possono essere:

- Disgiunti
- Interagenti a livello competitivo - due o piu' processi richiedono l'uso di una risorsa comune riusabile con molteplicita' finita
- Interagenti a livello cooperativo - due o piu' processi cooperano per raggiungere un obiettivo comune

In presenza di competizione e' necessario gestire i possibili fenomeni di interferenza.

La strategia piu' opportuna per gestire l'interferenza dipende largamente da:

- tollerabilita' degli effetti
- rilevabilita' dei fenomeni di interferenza
- recuperabilita' degli effetti dell'interferenza

Strategie:

- conseguenze inaccettabili -> evitare ogni forma di interferenza
- conseguenze trascurabili -> ignorare le interferenze
- conseguenze rilevabili e controllabili -> rilevare ed evitare
- conseguenze rilevabili e recuperabili -> rilevare e ripetere

Condizioni di Coffman per lo stallo: affinche si manifesti uno stallo, tutte le seguenti condizioni devono verificarsi:

- condizione di mutua esclusione - tutte le risorse sono seriali
- condizione di incrementalita' delle richieste - i processi richiedono le risorse una alla volta
- condizione di non prerilasciabilita' - le risorse, una volta assegnate, non possono essere prerilasciate
- condizione di attesa circolare

Dato un grafo di allocazione delle risorse, lo stallo e' in atto quando il grafo contiene dei cicli.

Tecniche di gestione dello stallo:

- **Fuga dallo stallo**: si rifiutano richieste di risorse che causano o potrebbero causare uno stallo - Strategia conservativa: ci si mantiene in uno stato sicuro (non si e' in stallo - si e' in grado di soddisfare in un tempo finito tutte le richieste non ancora evase).
  Svantaggi:
  - i processi devono specificare in anticipo il numero di risorse che intendo richiedere
  - vengono rifiutate richieste che non necessariamente conducono allo stallo. Possibile sotto utilizzazione delle risorse
- **Riconoscimento e risoluzione** dello stallo: si verifica dal grafo di allocazione delle risorse se uno stallo e' in atto e, in tal caso, si adottano determinate tecniche - Strategia ottimistica: una volta rilevato lo stallo, si risolve:
  - eliminando uno dei processi che lo causano, provocando il rilascio delle risorse
  - con tecniche di rollback, se supportate
- **Prevenzione dello stallo**: si invalidano una o piu' delle condizioni di Coffman
  - serialita' delle risorse: condizione intrinseca di una risorsa, quindi non invalidabile
  - incrementalita' delle richieste: invalidabile imponendo che le risorse vengano chieste in modo atomico tutte in una volta. Approccio "tutto o niente" semplice, con due svantaggi: sotto utilizzazione delle risorse, necessita' di cooperazione tra processi.
  - non prerilasciabilita': invalidabile imponendo che, quando un processo si vede negata una risorsa, rilascia tutte le risorse che gia possedeva e ricomincia. Svantaggi: costi intrinseci del prerilascio forzato, degrado prestazionale, non applicabile a risorse non prerilasciabili.
  - attesa circolare: invalidabile costruendo un ordinamento totale sulle risorse e obbligando i processi a chiedere le risorse nell'ordine stabilito. Principale svantaggio e' la possibile sotto utilizzazione delle risorse.

Problema della mutua esclusione - Contesto: dati due processi P e P' e una risorsa R, si deve garantire che:
- in ogni istante R e' libera oppure assegnata ad uno e un solo tra P e P' (mutua esclusione)
- ciascuno dei due processi deve sempre poter ottenere l'uso della risorsa (condizione di fairness)

La prima condizione richiede la serializzazione della risorsa, la seconda condizione impone che nessun processo rimanga in attesa della risorsa per un tempo indefinito. Questo deve essere garantito in modo algoritmico, non probabilistico.

Regione o sezione critica: frammento di programma che utilizza la risorsa R durante la cui esecuzione l'esecutore vuole accedere alla risorsa con la garanzia di esserne l'unico utilizzatore.

Risoluzione del problema di mutua esclusione (per raffinamenti successivi):

- semaforo di basso livello (un bit) associato alla risorsa R: 0 = risorsa libera - 1 = processo in sezione critica. Uso di spin lock.
  Svantaggi:
  - attesa attiva
  - non garantisce la serialita' a causa della non atomicita' delle primitive LOCK e UNLOCK
- semaforo di basso livello (un bit) associato alla risorsa R: 0 = risorsa libera - 1 = processo in sezione critica. Uso di spin lock con disabilitazione delle interruzioni.
  Svantaggi:
  - attesa attiva
  - valida solo su sistemi uniprocessore

    - la disabilitazione delle interruzioni penalizza la reattivita' del sistema
- semaforo di basso livello (un bit) associato alla risorsa R: 1 = risorsa libera - 0 = processo in sezione critica. Uso di spin lock con supporto specifico dell'hardware tramite istruzione TestAndSet atomica. Svantaggi:
    - attesa attiva
    - non c'e' garanzia esplicita di fairness

- primitive P(S) e V(S) di Dijkstra, dove S e' definito semaforo di alto livello: 0 = risorsa occupata - >0 = risorsa libera. Si disciplinano i processi consumatori di eventi ad esplicitare il proprio interesse verso una risorsa, e i processi produttori di eventi a notificare attivamente i corrispondenti processi consumatori.
  Condizioni necessarie:
    - primitive P e V indivisibili. Soluzione: uso di spinlock che disciplinano l'accesso alla risorsa S. E' necessario solo nei sistemi multiprocessore. Nei sistemi uniprocessore basterebbe la semplice disabilitazione delle interruzioni, perche' le attese attive sono brevi e tollerabili. Nella pratica si disabilitano le interruzioni anche nei sistemi multiprocessore, cosicche' non possa verificarsi una interruzione un attimo prima che un processo rilasci il semaforo di basso livello, lasciando in attesa attiva lo spin lock di un altro processo su un altro processore.
    - fairness nella gestione della coda dei processi in attesa (generalmente FIFO)

Le primitive di Dijkstra (i semafori) risolvono ogni problema di sincronizzazione.

Vantaggi dei semafori:
- espressivi: permettono di risolvere qualunque problema di sincronizzazione

Svantaggi:
- non strutturati
- difficili da testare
- richiedono variabili comuni

Regione critica - Semantica: garantire l'indivisibilita' della sezione critica su una risorsa R in modo semplice.

Implementazione:
- un semaforo binario Sr associato alla variabile condivisa R
- P(Sr) prima di ogni sezione critica su R
- V(Sr) dopo ogni sezione critica su R

Regioni critiche condizionali - Semantica: il processo, appena entrato in sezione critica, valuta una condizione (che coinvolge la variabile condivisa). Se e' vera, esegue le istruzioni nella regione critica. Altrimenti esce dalla sezione critica e si mette in attesa passiva in una coda Q<cond> associata alla condizione specificata: l'uscita dalla sezione critica e' necessaria per permette ad altri processi di agire sulla risorsa ed eventualmente cambiare la condizione su cui il processo e' in attesa.

La condizione è *rivalutata* dopo l'esecuzione di ogni regione critica sulla stessa variabile e se vera, viene risvegliato il primo dei processi in attesa dentro la coda associata Q<cond>.

Le regioni critiche, sebbene riescano nello scopo di aumentare il livello di astrazione rispetto ai semafori, hanno due grosse limitazioni:
- efficienza - la loro implementazione pone problemi di efficienza per la necessità di rivalutare le condizioni all'uscita di ogni regione critica
    - possibile soluzione: alcuni processi sanno quando una condizione attesa da altri processi diventa vera; si dovrebbe cercare metterli in comunicazione con i processi interessati
- coesione - le regioni critiche su una variabile condivisa possono essere sparse in tutto il programma; per conoscere e comprendere tutti i modi in cui viene utilizzata può essere necessario esaminare grandi porzioni di codice
    - possibile soluzione: utilizzare paradigmi che aumentino la coesione tra le risorse e le operazione

per manipolarle

Monitor: nascono dall'adattamento del concetto di regione critica al concetto tipo di dato astratto. Un tipo di dato astratto T racchiude in una unita' logicamente coesa la rappresentazione di un dato di tipo T e tutte le operazioni lecite su tale dato.

Il monitor esporta le OP_1 … OP_n che costituiscono le uniche operazioni permesse sui dati di tipo *<nome_monitor>*.

Può servire una particolare procedura per inizializzare le variabili condivise delle istanze di monitor non appena create, ovvero per porre la risorsa nello stato iniziale voluto.
Le variabili globali del monitor rappresentano la stato della risorsa comune: sono condivise ed accessibili unicamente tramite le OP_1 … OP_n .

Le primitive OP_1 … OP_n vengono richiamate in sezione critica sull'istanza del monitor stesso: l'esecuzione delle operazioni avviene in modo mutuamente esclusivo con quella di possibili chiamate concorrenti ad operazioni sulla stessa istanza di monitor.

I monitor rispetto alle regioni critiche:

- risolvono il problema della scarsa coesione
- consentono di risolvere elegantemente problemi di competizione
- non consentono di risolvere facilmente i problemi di cooperazione per la mancanza di meccanismi espliciti di sincronizzazione tra processi
  - le regioni critiche invece consentivano la risoluzione dei problemi di sincronizzazione ma con problemi di efficienza a causa della rivalutazione delle condizioni

La sincronizzazione viene resa piu' efficiente e simplice adottando le primitive **wait** e **signal** di Hoare.

L'idea è di associare una *variabile condizione* agli eventi o condizioni di interesse su cui sincronizzarsi: si tratta di condizioni su una risorsa condivisa (una istanza di monitor) per la quale competono diversi processi.
Tali processi durante le loro sezioni critiche modificano lo stato della risorsa e finiscono per alterare la valutazione della condizione: i processi che attendono l'evento associato possono esplicitare il proprio interesse emettendo una wait sulla variabile condizione, mentre i processi che sono nella posizione di conoscere quando tali eventi si verificano notificano i processi interessati emettendo una signal sulle variabili condizione associate all'evento.

Alla var. condizione è associata una coda di attesa. Quando un processo esegue una

- *<variabile-condizione>*.**wait -** viene bloccato, inserito nella coda di attesa per quella variabile condizione e rilascia il monitor
- *<variabile-condizione>*.**signal -** se nessun altro processo è in attesa sulla variabile condizione prosegue il proprio avanzamento; altrimenti:
  - viene bloccato e rilascia il monitor
  - il primo dei processi in attesa sulla variabile condizione viene risvegliato per rientrare in competizione sul monitor

Per evitare ambiguità ed inefficienze, è preferibile che la signal sia sempre l'ultima istruzione eseguita in una procedura del monitor, altrimenti il processo che la esegue va in competizione per il monitor con il processo che ha appena risvegliato.

Con i monitor e' possibile risolvere qualsiasi problema di sincronizzazione (e' possibile infatti implementare in modo semplice le primitive P e V con i monitor dotati delle primitive di Hoare).

Implementazione dei monitor tramite semafori:

- si associa un semaforo binario MUTEX ad ogni istanza di monitor
- il corpo di ciascuna procedura viene racchiusa tra **P**(MUTEX) e **V**(MUTEX)

- nell'ipotesi semplificativa che le signal vengano sempre eseguite come ultima istruzione di una procedura, ad ogni variabile condizione $X_{<cond>}$ si associa:
  - un contatore $C_{<cond>}$ inizializzato a 0 che conta il numero di processi bloccati sulla <cond>
    - serve ad evitare segnalazioni inutili
  - un semaforo binario $S_{<cond>}$ inizializzato a 0
    - serve per la sincronizzazione sulla variabile condizione

Vantaggi dei monitor - il principale vantaggio è il maggior livello di astrazione che deriva dal loro utilizzo, da cui poi seguono:

- la pulizia concettuale
- l'incapsulamento delle porzioni di codice concorrente
- la possibilità di dimostrazioni formali di correttezza

Svantaggi:

- limitazioni sul parallelismo
  - il meccanismo automatico di mutua esclusione attuata dai monitor talvolta si rileva troppo conservativo, come mostrato nell'esempio della mailbox
- problema delle chiamate annidate
  - non è possibile annidare direttamente od indirettamente chiamate della medesima istanza di monitor
- presuppongono la disponibilità di memoria comune