Contents

Fork & Join

```
fde_d = fork <program>
```

Crea un processo figlio del processo attuale, successivamente la loro vita è separata.

```
join fde_id
```

Il processo corrente rimane bloccato fino a quando non termina il fde con id specificato.

Condizioni di Bernstein

Se queste condizioni non vengono violate, allora sicuramente non si avrà interferenza.

- scritture(a) $\cap$ scritture(b) = $\emptyset$
- scritture(a) $\cap$ letture(b) = $\emptyset$
- letture(a) $\cap$ scritture(b) = $\emptyset$

Assunzione di progresso finito

Tutti i processi virutali hanno una velocità finita non nulla.

Fairness

Si ha un programma che rispetta le proprietà di fairness quando non c'è né starvation, né stallo. Si dice che c'è starvation, quando un fde rimane affamato, ovvero non riesce a ottenere mai le risorse (difficile da prevenire o individurare). Lo stallo invece si ha quando due o più fde rimangono in attesa di eventi che non potranno mai verificarsi.

Condizioni di Coffman (stallo)

Condizione di mutua-esclusione: le risorse condivise sono seriali.

Questa di condizione non si può invalidare, perché il punto è proprio che stiamo gestendo risorse

Condisione di incrementalità delle richieste: i processi richiedono le risorse una alla volta.

Questa condizione dice che un fde non richiede in blocco tutte le risorse che gli servono, ma ne richiede atomicamente una alla volta. Questa di condizione è facilmente invalidabile, basta che ciascun fde richieda in blocco tutte le risorse di cui necessita. Il problema è il sottoutilizzo delle risorse.

Condizione di non-prerilasciabilità.

Questa condizione dice che un fde, una volta che ha richiesto le risorse A e B, non può pre-rilasciarle, fin tanto che non ha terminato la sua esecuzione, o meglio fin tanto che non ha finito di farci quello per cui gli servivano. Questa di condizione anche è facilmente invalidabile, basta che quando un fde richiede la risorsa A e B; successivamente quando richiede C, se questa risorsa gli viene negata perché non disponibile (vedi caso di stallo dei pulmini), allora basta che pre-rilascia A e B, (nel caso dei pulmini corrisponderebbe a fare marcia in dietro). Questa condizione anche può comportare un calo delle performance.

Condizione di attesa circolare.

Se si genera il grafo che descrive le dipendenze tra fde e risorse, si ha una ciclo. Invalidare la condizione di attesa circolare è possibile. Basta che si prendano tutte le risorse condivise e gli assegni un ordinamento gerarchico totale. Questo significa che siano A, B, C e D le risorse, ciascun flusso di esecuzione potrà (ad esempio), richiedere queste risorse con un ordinamento preciso. Ovvero dovrà richiedere prima A, poi B, poi C e poi D. Chiaramente se C non gli serve non la richiede.

Algoritmo del Banchiere

Fuga dallo stalo, mediante strategia conservativa. Ovvero si ammette la possibilità di rifiutare (posticipare) alcune richieste.

Domanda orale (stallo)

Programma complesso, non vuoi studiarlo tutto, ma vuoi prevenire lo stallo. Come fai?

Flussi in competizione

Due o più fde hanno un loro obiettivo, ma devono utilizzare risorse condivise e con molteplicità finita (seriale = molteplicità = 1)

Flussi in cooperazione

Due o più fde hanno un obiettivo comune e devono usare delle risorse condivise e seriali.

Problema della mutua esclusione

Siano dati due flussi di esecuzione e sia R una risorsa. Il problema consiste nel garantire che R sia sempre assegnata a uno solo dei due flussi, garantendo la fairness (no stallo o starvation).

Le soluzioni di questo problema prevedono la definizione di un protocollo di utilizzo della risorsa:

1. Richiesta della risorsa
2. Uso della risorsa -> Sezione critica
3. Rilascio della risorsa

Spink Lock (versione 1, BUGGATA)

Questa è una prima versione buggata del semaforo. Qui il problema è che nella LOCK a un certo punto viene fatto il controllo su X. Quindi si dice che se X = 1 (ovvero se è verde), allora deve uscire dal ciclo infinito e settare il semaforo su rosso. Il problema è che il controllo della variabile e il settaggio della variabile X, sono due operazioni separate, il che significa che qualcuno intanto potrebbe settare il semaforo a rosso.

```
// richiesta
LOCK(X):
    begin
        loop begin
```

```
            if X = 1 exit; // aspetta che sia verde
        end

        // in questo punto, il semaforo ormai
        // si è deciso che verrà settato a verde
        // ma ancora non è stato settato.
        // quindi qualcun altro potrebbe pensare
        // che il semaforo sia verde, e quindi
        // non si avrebbe mutua esclusione
        // perché due flussi di esecuzione
        // riuscirebbero a usare la stessa risorsa.

        X = 0; //poni il semaforo a verde
    end

// riascio
UNLOCK(X):
    begin
        X = 1; // poni il semaforo a verde
    end
```

Spink Lock (versione 2, BUGGATA)

```
// richiesta
LOCK(X):
    begin
        loop begin
            1. << disabilita le interruzioni >>
            if X = 1 exit; // aspetta che sia verde
            2. << abilita le interruzioni >>
            3. << ritardo >>
        end
        X = 0; //poni il semaforo a verde
        4. << abilita le interruzioni >>
    end

// riascio
UNLOCK(X):
    begin
        5. << disabilita interruzioni >>
        X = 1; // poni il semaforo a verde
        6. << abilita interruzioni >>
    end
```

Disabilitare le interruzioni significa che da quel momento in poi l'esecuzione della

procedura non può essere interrotta eseguendo un'altra istruzione di un'altra procedura. Praticamente da quel momento in poi, tutto ciò che viene eseguito, sono le istruzioni di questa procedura. A questo punto:

Qui disabilito le interruzioni e dopo effettuo il controllo. Se il controllo non viene superato, vengono immediatamente riabilitate le interruzioni. Se invece il test viene superato, allora si fa la exit, si evita l'istruzione 2 e 3 e si setta il semaforo a rosso e poi si riabilitano le interruzioni.

Perché viene aggiunto il ritardo?

Perché disabilitare le interruzioni costa molto e in questo modo si abilitano e si interrompono meno volte. Praticamente invece che continuare costantemente il valore di X, si controlla ad esempio ogni 5 secondi.

Questa soluzione è comunque BUGGATA, perché?

Perché questa cosa funziona se si sta utilizzando un computer con un unico processore. Ma con computer con più processori, si disabilitano le interruzioni sul processo corrente, ma sugli altri no, quindi un fde su un altro processore, quando faccio il controllo del semaforo che è verde, potrebbe (prima che esso venga messo a rosso), rubare la risorsa, non garantendo dunque la mutua esclusione.

Spink Lock (versione 3, un po' meno BUGGATA, ma non fair)

```
// richiesta
LOCK(x):
begin
    loop begin
        TestAndSet(X);
        if (X era 1) exit; // aspetta che sia verde
        <ritardo>
    end
    X = 0;
end

// rilascio
UNLOCK(x):
    begin
        Set(X); // poni il semaforo a verde
    end
```

TestAndSet e Set sono due istruzioni macchina, che l'hardware ci garantisce essere indivisibili (su tutti i processori). Quindi abbiamo risolto tutto. Il punto è che senza un piccolo ma decisivo aiuto da parte dell'hardware, la mutua esclusione non si riesce a garantire. C'è un problema però, questi spin lock non ci garantiscono fairness. Questo perché? Perché se N fde fanno la TestAndSet concorrentemente, il processore sceglierà a chi dare la risorsa, di solito viene data priorità sulla base del numero del processore. Quindi gli fde su un processore con numero alto potrebbero rimanere affamati.

Semafori di Dikstra

```
P(S):
    << disabilita le interruzioni >>
    LOCK(SX)
    if S = 0
        then << poni il fde in stato di attesa passiva di una V(S) >>;
    else
        // 1. se non venisse utilizzata la LOCK e l'UNLOCK
        // un altro fde potrebbe fare qui la P(S)
        // e nel caso in cui S si trovasse a 1
        // non si avrebbe più mutua esclusione.
        S--;
    UNLOCK(SX)
    << abilita le interruzioni >>

V(S):
    << disabilita le interruzioni >>
    LOCK(SX)
    if << esiste un fde in attesa di una V(S) >>
        then << risveglia uno di questi fde >>;
    else
        S++;
    UNLOCK(SX)
    << abilita le interruzioni >>
```

Prima il semaforo era binario, dunque o 0 rosso o 1 verde, perché la molteplicità della risorsa è pari a uno. Qui invece la molteplicità della risorsa è anche maggiore di 1, quindi S può assumere tutti i valori da 0 a N, dove N è la molteplicità della risorsa.

Ad ogni semaforo S è associata una coda Q_s. Quando un fde trova la risorsa occupata, si mette in coda ad attendere passivamente di essere risvegliato da una V(S). Qui viene usata una disciplina FIFO.

Perché si fa LOCK e UNLOCK?

Perché dopo che sulla P(S) si controlla S, prima di decrementare un altro fde potrebbe prendersi la risorsa e quindi non si avrebbe mutua esclusione.

Perché conviene disabilitare le interruzioni anche nei sistemi multi-processore?

Perché in questo caso le attese attive sono di brevissima durata (il tempo di fare un controllo e un decremento/incremento). Di fatto qui il semaforo di basso livello non sta proteggendo la risorsa, ma il semaforo di alto livello.

Non abilitare le interruzioni potrebbe invece portare a una situazione in cui l'attesa attiva sulla LOCK diventa particolarmente lunga, in quanto tra un LOCK e una UNLOCK potrebbe verificarsi una interrupt.

Ecco la situazione:

```
CPU( i )                    CPU( j )            i <> j
fde(P)
P(SX)
LOCK(SX)
 ..   interrupt  fde(Z)     fde(Q)
 ..               ..        P(S)
 ..               ..        LOCK(SX)
 ..
 ..
 ..
 ..
UNLOCK(SX)
```

I semafori di Dijkstra garantiscono fairness?

No, perché non la garantisce il semaforo di basso livello utilizzato per la sua implementazione.

I cinque filosofi mangiatori di spaghetti

Cinque filosofi trascorrono la vita alternando, ciascuno indipendentemente dagli altri, periodi in cui pensano, a periodi in cui mangiano degli spaghetti.

Per raccogliere gli spaghetti ogni filosofo necessita delle due forchette poste rispettivamente a destra e a sinistra del proprio piatto. Trovare una strategia che consenta a ciascun filosofo di ottenere sempre le due forchette richieste.

In questo contesto i flussi di esecuzione sono i 5 filosofi, mentre le forchette rappresentano le risorse condivise.

Prevediamo che ogni filosofo segua un protocollo di questo tipo:

```
loop
    <<pensa>>;
    <<impossessati delle due forchette>>;
    <<mangia>>;
    <<rilascia le due forchette>>;
end
```

Soluzione con stallo

Le forchette sono modellate con una variabile condivisa. In particolare abbiamo 4 semafori, ovvero un semaforo per ogni forchetta. Questo ci servirà per far si che ciascuna forchetta venga utilizzata da un filosofo (fde) alla volta.

```
var F: shared array[0..4] of semaforo;

// Qui viene definito il tipo filosofo.
// Vedi dopo.
type filosofo = concurrent procedure (I: 0..4);
    begin loop
        <<pensa>>;

        // Mettiti in attesa passiva
        // di una V sulla forchetta i (quella a destra)
        // o prendila se è disponibile.
        P(F[ i ]);

        // Stessa cosa di sopra per la forchetta
        // a sinistra.
        P(F[ (i+1) mod 5 ]);

        <<mangia>>;

        // Sveglia il filosofo accanto se è affamato
        // o libera la risorsa
        V(F[ (i+1) mod 5 ]);
        V(F[ i ]);

    end

// Qui si dichiarano 5 variabile di tipo filosofo
var A, B, C, D, E: filosofo;
```

```
// Qui si dichiara un array di dimensione 4
// di tipo semaforo. Tutti i semafori hanno come
// valore di default
var F[4]: semaforo;

begin
    cobegin A(0) || B(1) || C(2) || D(3) || E(4) coend
end
```

Soluzione conservativa, senza stallo ma con starvation

```
type filosofo = concurrente procedure (I: 0..4);
    begin
        <<pensa>
        prendi_forchette( i );
        <<mangia>>;
        posa_forchette( i );
    end

var A, B, C, D, E: filosofo;
var J: 0..4;

// d'ora in poi questi semafori rappresentano la coppia
// di forchette dell'iesimo filosofo.
// 01, 12, 23, 34, 40
var F[4]: semaforo;

// FP[ i ] è verso se e solo se l'i-esima coppia di
// Forchetta Prenotata
var FP[4]: boolean;

// vero se e solo se l'i-esimo filosofo è Affamato
var A[4]: boolean;

// un semaforo binario R per accedere allo stato
// corrente (vettori A e FP) in mutua esclusione.
var R: semaforo;

begin
    INIZ_SEM(R,1);
    for J <- 0 to 4 do INIZ_SEM(F[J], 0);
    for J <- 0 to 4 do FP[J] <- false;
    for J <- 0 to 4 do A[J] <- false;
    cobegin A(0) || B(1) || C(2) || D(3) || E(4) coend
end
```

```
concurrent procedure prendi_forchette(I:0..4)
begin
    // Faccio la P su R. R è un semaforo binario ed è verde quando
    // né A né FP è posseduto da un fde.
    P(R);

    // Filosofo corrente affamato.
    A[I] <- true;

    // Controlla che il filosofo corrente sia affamato
    // e che la forchetta alla sua sinistra e alla sua destra
    // non sia stata prenotata. Se tutto questo è vero
    // allora setta l'i-esima coppia di forchette su true (FP)
    // inoltre fa la V(F[ i ]). La V(F[ i ]) viene fatta
    // perché di default il semaforo è rosso, quindi la risorsa
    // deve effettivamente essere liberata.
    // nel caso in cui il test non andasse a buon fine
    // il filosofo si metterebbe semplicemente i attesa
    // che la sua coppia di forchette venga rilasciata dai
    // filosofi che la stanno utilizzando in questo momento.
    test(I);

    // Rilascia le risorse A e FP a chi le ha richieste.
    V(R);

    // Richiedo la i-esima coppia di forchette.
    // qui vengono svegliati dai test di un filosofo accato
    // oppure se la coppia di forchette è libera vengono subito prese.
    P(F[I]);
end

concurrent procedure posa_forchette(I:0..4)
begin
    P(R);

    // la coppia di forchette on è più prenotata
    // e il filosofo non è più affamato
    FP[ I ] <- false;
    A[ I ] <- false;

    // controllo che i filosofi alla sinistra e alla destra
    // siano affamati e che non abbiamo una delle due forchette
    // accanto occupate. In quel caso li sveglio con la V(F[i])
    test( (I-1) mod 5 );
    test( (I+1) mod 5 );
```

```
    V(R);
END

concurrent procedure test(I:0..4)
begin
    if(A[I] and not(FP[ (I-1) mod 5]) and not(FP[ (I+1) mod 5 ]))
    begin
        FP[ I ] <- TRUE;
        V(F[ I ]);
    end
end
```

Questa soluzione non è ancora fair; due filosofi infatti potrebbero mettersi d'accordo per non far mangiare un terzo. Una soluzione consiste nel violare una delle condizioni di Coffman, in particolare quella di attesa circolare. Questo è possibile ottenerlo prevedendo la richiesta incrementale delle tue forchette e la rpesenza di un filosofo mancino, che quindi acquisisca le forchette nell'ordine opposto agli altri.

```
loop
    <<pensa>>;
    <<impossessati delle due forchette>>;
    <<mangia>>;
    <<rilascia le due forchette>>;
end

concurrent program CINQUE_FILOSOFI_MANGIATORI;

type filosofo = concurrent procedure (I: 0..4, mancino = false);
    begin
        if(mancino)
            P(F[(i-1) % 5]);
            P(F[i]);
        else
            P(F[i]);
            P(F[(i-1) % 5]);
    end

var A, B, C, D, E: filosofo;

J: 0..4;

var F: shared array[0..4] of semaforo;

begin
    for J <- 0 to 4 do INIZ_SEM(F[J], 1);
    cobegin A(0, true) || B(1) || C(2) || D(3) || E(4) coend
```

```
end
```

Il barbiere dormiente

In un negozio lavora un solo barbiere, ci sono N sedie, per accogliere i clienti in attesa ed una sedia di lavoro. Se non ci sono clienti, il barbiere si addormenta sulla sedia di lavoro. Quando arriva un cliente, questi deve svegliare il barbiere, se addormentato, od accomodarsi su una dellesedie in attesa che finisca il taglio corrente, Se nessuna sedia è disponibile preferisce non aspettare e lascia il negozio.

Qui ci sono due flussi: quello del barbiere e quello de clienti.

“‘ type barbiere = concurrent procedure; begin /* … */ end type

cliente = concurrent procedure; begin /* … */ end

var Dormiente: barbiere;

var CLIENTE array[0..NUM_CLIENTI] of cliente

// C approssimato al cliente, semaforo in stile cooperativo // B approssimato al barbiere, semaforo in stile cooperativo // MX, semaforo binario, stile competitivo che protegge la // variabile in_attesa. var C, MX, B: semaforo;

// Sedie disponibili var N = 5;

// Clienti in attesa var in_attesa: intero;

begin

```
INIZ_SEM(MX,1);
INIZ_SEM(C,0);
INIZ_SEM(B,0);

fork Dormiente;

// Cobegin e coend parametrizzato con il numero di clienti
for J <- 0 to NUM_CLIENTI do fork CLIENTE[J];
for J <- 0 to NUM_CLIENTI do join CLIENTE[J];

join Dormiente;
```

end “‘

```
concurrent program BARBIERE_DORMIENTE;
type barbiere = concurrent procedure;

    begin loop
```

```
    // cicla e attende l'arrivo di un cliente
    // per manifestare il proprio interesse all'arrivo
    // di un cliente, il barbiere fa la P(C)
    P(C);

    // a questo punto, se arriva un cliente occorre
    // aggiornare in mutua esclusione i clienti in attesa
    // che dovrà essere decrementato di uno
    P(MX);

    // decrementa i clienti in attesa di uno
    in_attesa--;

    // a questo punto il barbiere segnala la propria disponibilità
    V(B);

    // e infine mette a verde il semaforo in mutua esclusione
    V(MX)

    <<taglia capelli>>;


    end
end;
type cliente = concurrent procedure;

    begin

    <<raggiungi il negozio>>

    // mutua esclusione di in_attesa
    P(MX);

    // controlla se ci sono posti a sedere
    // se non ci sono posti a sedere, il cliente se ne va via
    if (in_attesa < N) begin
        in_attesa++;

        // Il cliente segnala al barbiere e gli dice,
        // che c'è un cliente.
        V(C); // sveglia il barbiere se dorme

        V(MX);
        // <---
        P(B); // aspetta che il barbiere finisca
```

```
        <<siedi per il taglio>>;
    end
    else V(MX); // neanche un posto a sedere: meglio ripassare
end
```

Produttore e Consumatore

```
                    buffer
(P) inserimenti --> |||||| --> estrazioni (C)
```

Uno dei problemi di cooperazione tra più flussi più semplice in assoluto consiste nel far scambiare un messaggio tra due flussi diversi. Quindi il produttore consumatore modella questo genere di problema.

In questo problema i due flussi in esame sono P (produttore) e C (consumatore). Questi due flussi devono scambiarsi un messaggio. Per scambiarsi questo messaggio usano un buffer. Dunque il produttore inserisce il messaggio nel buffer e il consumatore lo estrae.

In questo caso le sequenze di interleaving problematiche sono quelle in cui ad esempio il consumatore si mette a leggere dal buffer quando non c'è niente da leggere, o magari quelle in cui il produttore scrive sul buffer scrivendo un vecchio messaggio che ancora non è stato letto ecc…

Bisogna dunque sincronizzare i due semafori. In particolare occorre sincronizzarsi su due eventi diversi, a cui associamo due diversi semafori binari.

Soluzione seriale prod-cons-prod-cons...

- DEPOSITATO: ad 1 se e solo se un messaggio è stato depositato ed è prelevabile
- PRELEVATO: ad 1 se e solo se il buffer è vuoto e pronto ad accogliere un nuovo messaggio

```
begin
    INIZ_SEM(PRELEVATO, 1)
    INIZ_SEM(DEPOSITATO, 0)
    cobegin PROD || CONS coend
end

concurrent procedure PROD
loop begin
    <produci un messaggio in M>
    P(PRELEVATO)
    BUFFER <- M
```

```
    V(DEPOSITATO)
end

concurrent procedure
CONS loop begin
    P(DEPOSITATO)
    M <- BUFFER
    V(PRELEVATO)
    <consuma il messaggio in M>
end
```

Esercizi

1. Cosa accadrebbe se PRELEVATO venisse inizializzato a 0 e DEPOSITATO a 1?
2. Il consumatore preleverebbe dalla coda leggendo dal buffer un "messaggio vuoto".

Implementazione a Buffer Circolare

La soluzione precedentemente vista fa si uso di semafori, ma in realtà è per lo più seriale. Questo perché un produttore non può produrre fin tanto che un consumatore non ha consumato. Una possibile (e unica) sequenza di interleaving sarà: prod-cons-prod-cond-…

Questa cosa accade perché produttore e consumatore sono fortemente accoppiati. In particolare sono accoppiati a causa del buffer di dimensione unitaria.

Nella generalizzazione classica questo problema viene superato, in particolare qui la dimensione del buffer non è capace di ospitare un unico messaggio, ma N messaggi. In questo modo il produttore può produrre messaggi anche se il consumatore non li ha consumati.

Inoltre il numero di produttori e di consumatori è varibile. In particolare ci sono P produttore e C consumatori.

```
concurrent program PRODUTTORI_CONSUMATORI;

// dimensione del buffer, e.g. N = 5
N = 5;

// definisco il tipo messaggio
type messaggio = ...;

// definisco indice per gestire buffer
indice = 0..N-1;
```

```
// definisco il BUFFER, un array di grandezza N di messaggi
var BUFFER: array[indice] of messaggio;

// indice utilizzato per gli inserimenti dai produttori
// in particolare è l'indice della prima cella utile per inserire
D: indice;

// indice utilizzato per gli inserimenti dai consumatori
// in particolare è l'indice della prima cella utile per leggere
T: indice;

// semafori competitivi in mutua esclusione
USO_T, USO_D: semaforo_binario;

// semafori cooperativi di molteplicità pari a N
PIENE, VUOTE: semaforo;

concurrent procedure PROD_i /* generico produttore */

concurrent procedure CONS_j /* generico consumatore */

begin
    INIZ_SEM(USO_D,1);
    INIZ_SEM(USO_T,1);
    INIZ_SEM(PIENE,0);
    INIZ_SEM(VUOTE,N);
    T <- 0;
    D <- 0;
    cobegin ... || PROD_i || CONS_j || ... coend
end

concurrent procedure PROD_i //generico produttore
var M: messaggio;

loop

    <produci un messaggio in M>

    // attendo che almeno uno spazio nel buffer venga liberato
    P(VUOTE);

        // richiedo l'uso esclusivo di D, l'indice per gli inserimenti
        P(USO_D);

            // aggiungo nel buffer il messaggio
```

```
            BUFFER[D] <- M;

            // definisco l'indice di D, che verrà utilizzato dal
            // successivo produttore
            D <- (D+1) mod N;

        // rilascio l'indice D
        V(USO_D);

    // notifico che un messaggio è stato aggiunto nel buffer
    V(PIENE);

end

//generico consumatore
concurrent procedure CONS_j

// dichiaro una variabile M di tipo messaggio
var M: messaggio;
loop

    // attendo che un messaggio venga inserito nel buffer
    P(PIENE);

        // richiedo l'uso esclusivo dell'indice T dell'estrazioni
        P(USO_T);

            // estraggo il valore del messaggio in posizione T
            M <- BUFFER[T];

            // inizializzo il valore di T che dovrà utilizzare
            // il successivo consumatore
            T <- (T+1) mod N;

        // rilascio T
        V(USO_T);

    // notifico che uno spazio è stato liberato nel buffer
    V(VUOTE);

    <consuma il messaggio in M>
end
```

Regioni critiche

```
var R: shared T;

region R do
    <lista istruzioni> // sezione critica con uso di R
end region
```

Ciò che sta dentro la sezione critica è indivisibile.

La regione critica è uno strumento pensato per la competizione.

Per risolvere problemi non competitivi ma cooperativi? Esistono le regioni critiche condizionali.

```
var R: shared T;
region R do
    when <cond_su_R> do <lista istruzioni>
end region
```

La condizione parla di R e sta dentro la sezione critica, perché quando valuti la condizione, devi essere tranquillo che nessuno ti cambi lo stato della risorsa sotto i piedi.

Region con semafori

P(R) if() if(&&) V(Q) else V(R) else V(R) P(Q_) ————————

Monitor

Mette d'accordo il paradigma orientato agli oggetti e quello concorrente. Il costrutto delle regioni critiche condizionali è stato un primo tentativo, ma aveva dei problemi, in particolare le performance, infatti ogni volta che un flusso entra e esce da una regione critica, occorre svegliare tutti per verificare la condizione.

Hoare ha introdotto le primitive wait e signal. Rappresenta un tentativo di unire il tipo di dato astratto con le regioni critiche condizionali.

Istanza di monitor ~= istanza di classe

```
type <nome_monitor> = monitor
<dichiarazioni di tipi e costanti globali>;
var <variabili condivise>;
entry procedure OP1 (<lista parametri>) begin
    <dichiarazioni locali>;
    <corpo di OP1>;
end
```

```
// ...

// è come se fosse un metodo pubblico
entry procedure OPn (<lista parametri>) begin
    <dichiarazioni locali>;
    <corpo di Op_n>;
end

<eventuali procedure locali al monitor>;
<procedura di inizializzazione delle var. condivise>;

end monitor;
```

Le primitive introdotte da Hoare sono wait e signal e cercano di astrarre il concetto di semaforo cooperativo.

```
// sono interessato a un certo evento (e.g. non_pieno, non_vuoto)
<variabile-condizione>.wait

// posso dire che si è verificato un certo evento
<variabile-condizione>.signal
```

Il legame che esiste tra wait, signal e il monitor qual è ?

Quando faccio la wait viene bloccato il flusso e inserito in una coda. Oltre a questa sospensione viene rilasciato l'uso del monitor. Perché? Perché il monitor rappresenta l'istanza della risorsa alla quale sei interessato. Quindi devi sospendere, dare agli altri la possibilità di lavorarci sopra e sperabilmente attendere che qualche altro fde faccia qualche cosa che sblocchi la tua condizione.

Nella signal, se nessun altro fde è in attesa di quella variabile condizione si prosegue normalmente, altrimenti se c'era qualcuno in attesa (a fare una wait), allora viene rilasciato il monitor, per dare l'occasione a quelli che attendevano l'evento di poter lavorare.

Quindi non solo viene segnalato che è accaduto qualche cosa, ma gli dai anche l'uso esclusivo di una risorsa, dove la risorsa è il monitor.

Ci sono tante code in ballo. C'è una coda per gestire i flussi di esecuzione, una per ogni condizione.

La signal generalmente deve essere l'ultima cosa che che deve essere eseguita in un flusso di esecuzione. Questo perché? Perché se faccio una signal significa che sono dentro una entry procedure e se non è l'ultima cosa che faccio, allora non viene rilasciato il monitor e come fa a lavorare l'altro flusso se io continuo a usare il monitor.

Nella signal_and_wait faccio subito context switch, mentre nella signa_and_continue no. Nella signal_and_continue, la condizione che hai fatto, potrebbe essere

invalidata.

In Java la semantica utilizzata è la signal_and_continue.

Implementazione dei semafori con wait e signal

Implementiamo P e V con due procedure.

```
type semaforo = monitor

    var S: 0..LAST;
    S_POSITIVO: condition; //variabile condizione per S>0

    entry procedure P begin

        while (S=0) do S_POSITIVO.wait; end
        S <- S - 1;
    end

    entry procedure V begin
        S <- S + 1;
        S_POSITIVO.signal;
    end begin

    S <- 0; end

end monitor;
```

Implementazione dei monitor tramite semafori