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) scritture(b) = Ø
- scritture(a) letture(b) = Ø
- letture(a) scritture(b) = Ø

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 semaforo, 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

concurrente procedure prendi_forchette(I:0..4)
begin
    P(R);
```

```
    A[I] <- true;
    test(I);
    V(R);
    P(F[I]);
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
```