# CAPITOLO 14 – CONCORRENZA: MUTUA ESCLUSIONE E SINCRONIZZAZIONE

## CONCORRENZA, MUTUA ESCLUSIONE, SINCRONIZZAZIONE

Premesso che la trattazione e comprensione di un sistema operativo l'abbiamo basata sui concetti di processo e di thread che evidentemente sono un'astrazione di quello che accade in una macchina, come si è iniziato a dire innanzi, i vari processi cercheranno di assicurarsi le risorse del sistema per poter essere eseguiti.
Questo capitolo si occupa della concorrenza tra processi. Nella trattazione affronteremo il problema della mutua esclusione e della sincronizzazione.

## Tipi di ambiente in cui si sviluppa la CONCORRENZA

- Multiprogrammazione:
  - gestione di più processi su un singolo processore
- Multiprocessing:
  - gestione di più processi su più processori
- Processi distribuiti
  - cluster

La concorrenza tra processi consiste nella gestione di più processi in esecuzione contemporaneamente su una macchina. I tre tipi fondamentali di ambienti in cui è sviluppata la gestione di processi concorrenti sono l'ambiente multiprogrammato, l'ambiente multiprocessore ed i cluster.
In un ambiente multiprogrammato il gestore di processi del sistema permette l'esecuzione di più processi su una macchina a singolo processore.
In un ambiente multiprocessore il gestore di processi è in grado di far eseguire più processi su tutti i processori disponibili.
In un cluster ( computer distribuiti) i processi sono eseguiti su tutti i nodi del cluster (collegati tra loro tramite una rete), ed il gestore di processi tiene traccia di tutti i processi che sono in esecuzione su ogni nodo del cluster.

## Origini della Concorrenza

- Competizione tra processi per ottenere (e condividere) le Risorse:
  - CPU;
  - memoria;
  - canali di I/O;
  - files;
  - ecc ...
- In applicazioni complesse la comunicazione tra processi è fondamentale

La concorrenza è una caratteristica fortemente radicata nei sistemi operativi, e comprende diversi aspetti di progettazione, fra cui la competizione per le risorse e la comunicazione tra processi:
La competizione per le risorse ha luogo innanzitutto nei sistemi multiprogrammati a singolo processore, poiché tutti i processi per poter essere eseguiti hanno necessità di accedere al processore, ed il sistema operativo alloca loro la cpu eseguendo un algoritmo di schedulazione. La seconda risorsa fondamentale per cui competono i processi è la memoria, in quanto questa non ha dimensione infinita. Il sistema operativo deve quindi gestire la memoria disponibile e fornirla ai processi che ne fanno richiesta, cercando di massimizzare l'utilizzo e minimizzare i tempi di attesa.
Inoltre più processi possono essere in competizione per l'accesso ad un canale di input/output, come può essere un disco fisso, una stampante o anche un singolo file. Il sistema operativo deve quindi minimizzare i danni che possono essere causati da un accesso incontrollato alle risorse di sistema, ad esempio mettendo in uno stato di attesa un processo con poca priorità, e cercando nel migliore dei modi di soddisfare tutte le richieste.
La comunicazione tra processi è fondamentale in applicazioni complesse formate da più sottosistemi o moduli.

## Contesti della concorrenza

- Applicazioni multiple: per condividere dinamicamente il tempo di calcolo tra applicazioni;
- Applicazioni strutturate:
  - gestione della progettazione modulare;
  - programmazione strutturata;
- Struttura del sistema operativo.

La concorrenza appare principalmente in tre contesti diversi:
Applicazioni multiple: è il caso di macchine monoprocessore multiprogrammate, in cui la potenza di calcolo della CPU viene dinamicamente divisa tra tutti i processi attivi.
Applicazioni strutturate: è il caso delle applicazioni composte da più moduli, ognuno dei quali svolge una funzione specifica. Ogni modulo inoltre può essere implementato come un processo o come un thread, facendo apparire al sistema operativo una singola applicazione come un insieme di processi concorrenti.
Struttura del sistema operativo: anche la progettazione dei sistemi operativi gode dei vantaggi della progettazione modulare e strutturata, e nei moderni sistemi ogni sottosistema è implementato come un processo a sé stante. Il sistema operativo diventa quindi esso stesso un insieme di processi.

**Meccanismi per risolvere i problemi della CONCORRENZA**

MUTUA ESCLUSIONE
- Meccanismi software
  - Busy waiting
  - Meccanismi forniti dal S.O.
  - Meccanismi forniti dal compilatore
    - Approccio basato su Semafori
    - Approccio basato su Monitor
    - Approccio basato su scambio di messaggi
- Meccanismi hardware

Esistono modi sia software che hardware di risolvere i problemi della concorrenza. Il principale modo software usa la tecnica del busy waiting ma esistono anche sia modi offerti direttamente dal S.O. che dal compilatore oltre che approcci diversi di risoluzione dei problemi della concorrenza come l'uso di semafori, l'uso di monitor e lo scambio di messaggi tra processi concorrenti.

**Problemi generali della CONCORRENZA**

- VANTAGGI

- benefici sulla esecuzione nonostante il sovraccarico.
- migliore utilizzazione delle risorse.

- PROBLEMI

1. difficile condivisione di risorse globali (nell'uso delle variabili globali ci possono essere operazioni critiche (es. perdita di coerenza));
2. difficile gestione ottimale delle risorse da parte del S.O. (es. canale di I/O che viene sospeso);
3. trovare un errore di programmazione diventa difficile.

Esempio per il caso 2): %

La possibilità di eseguire più processi contemporaneamente presenta sia dei vantaggi che degli svantaggi:
Il principale vantaggio risiede nella miglior efficienza di esecuzione dei processi, nonostante i cambi di contesto tra i vari processi in esecuzione comportino un notevole sovraccarico della macchina. Questo vantaggio ha luogo poiché statisticamente ogni processo durante la sua vita non consuma il cento per cento di tutte le risorse della macchina, ma, al contrario, passa una considerevole quantità di tempo in attesa che le sue richieste vengano soddisfatte, sia dall'utente sia dall'hardware. Durante questa attesa il sistema operativo può quindi dare il controllo delle risorse ad altri processi, ottimizzandone l'utilizzo e riducendo i tempi complessivi di esecuzione.
Ma la concorrenza tra processi presenta anche diversi problemi, i quali derivano principalmente dall'impossibilità di prevederne la velocità di esecuzione, poiché dipende sia dalla tipologia del processo, sia dal carico del sistema e da come questo gestisce la schedulazione delle risorse. Inoltre, data la generalità dei sistemi operativi, essi non possono conoscere gli scopi per cui un processo utilizza uno spazio di memoria, ma solo controllare i permessi di accesso. Si presentano quindi le seguenti difficoltà:

- La condivisione di variabili globali è pericolosa, soprattutto nei casi in cui esse vengono utilizzate per scopi di sincronizzazione: se due processi scrivono "contemporaneamente" sulla stessa variabile, l'ordine in cui avvengono le operazioni diviene critico ed è spesso fonte di problemi.
- Per il sistema operativo è difficile gestire l'assegnazione delle risorse in maniera ottimale, sempre a causa dell'incognito tempo di esecuzione: se esso decide di allocare un canale di I/O ad un processo che necessita di parecchio tempo per completare le sue operazioni, il risultato è una scelta inefficiente, poiché tutti gli altri processi bisognosi di quel canale si trovano impossibilitati a completare le loro operazioni, anche se questi richiedono meno tempo del processo bloccante.

Infine, gli errori di programmazione dovuti ad una cattiva sincronizzazione sono difficili da individuare poiché la loro riproducibilità è insita nella particolare sequenza di eventi che hanno scatenato l'errore: sequenza sempre diversa a seconda del contesto in cui lavora il processo.

**Esempio di problema del caso 1: PROCEDURA di ECHO**

```
procedure echo;
var out, in : character ;
begin
    input (in,tastiera) ;
    out := in;
    output (out, schermo);
end.
```

*Cosa succede se in multi-programmazione questa procedura viene condivisa?*
*Si risparmia spazio in memoria ma:*
*- se due processi concorrenti non completano la procedura correttamente un "in" si perde ed il successivo viene sullo schermo due volte.*
*Soluzione: %*

Un esempio di incoerenza causata da un cattivo utilizzo delle variabili globali è presentato in questa slide.
La procedure echo è una semplice funzione che legge un carattere da tastiera, lo memorizza in una variabile e ne visualizza il valore sullo schermo.
La procedura è inoltre parte di una libreria che viene utilizzata da un utente di un sistema multiprogrammato:

ogni applicazione dell'utente potrà utilizzarla per risparmiare spazio in memoria (dovuto alla condivisione del codice), ma questo può provocare, ad esempio, che un carattere si perda ed un altro venga stampato sullo schermo due volte. Si consideri ad esempio questa sequenza, in cui sono impegnati due processi concorrenti, P1 e P2:

- Il processo P1 chiama la procedura echo e viene interrotto subito dopo la lettura dell'input. A questo punto il carattere letto (x) e' conservato nella variabile globale in.
- Il processo P2 viene attivato e chiama la procedura echo, eseguita fino alla fine, con la lettura di un carattere (y) e scrittura sullo schermo.
- Il processo P1 viene riattivato, ma la variabile in è stata sovrascritta dalla precedente esecuzione del processo P2, quindi il carattere x è stato perso ed il carattere y è stato stampato due volte.

Il problema risiede tutto nella variabile globale "in" condivisa, perciò accessibile da tutti i processi. Se un processo viene interrotto prima che abbia completato le sue operazioni con quella variabile, un altro processo può alterarne il contenuto e portare quindi a delle incoerenze.

Soluzione:

Proteggere la procedura condivisa.
(1 solo processo alla volta può usare echo)

Esempio di applicazione:

P1 usa echo;
P2 chiede echo (ma non la ottiene)
P1 completa echo;
P2 ottiene echo.

Lo stesso problema e la stessa soluzione si può usare nei sistemi multiprocessore

La soluzione al problema della procedura echo consiste nel proteggere le variabili globali condivise da procedure di libreria, in modo che un solo processo alla volta possa utilizzarle.
Ad esempio, nel nostro caso si potrebbe ipotizzare una protezione dell'intera procedura, impedendone l'utilizzo da parte di più di un processo alla volta.
Se la procedura echo è protetta, la sequenza precedente si tramuta in:

- Il processo P1 chiama echo e viene interrotto prima che possa stampare il carattere letto
- Il processo P2 viene attivato e richiede la procedura echo, ma poiché essa è in uso da P1 (bloccato), anche P2 viene bloccato in attesa che echo diventi disponibile
- P1 viene riattivato e completa l'esecuzione di echo.
- Quando P1 termina l'esecuzione di echo, il blocco su P2 viene rimosso e quando esso viene riattivato, ottiene la procedura echo.

Anche in ambienti multiprocessore la crucialità risiede tutta nella oculata condivisione delle risorse: in un sistema monoprocessore la causa è dovuta all'interrompibilità dei processi, mentre in un sistema multiprocessore vi è il problema causato da due o più processi eseguiti contemporaneamente che cercano di accedere simultaneamente alla stessa risorsa.
La soluzione in entrambi gli ambienti risulta la medesima: il controllo d'accesso alle risorse condivise.

**Problemi di progettazione dovuti alla concorenzza e soluzioni.**

1) Il S.O. deve tenere traccia dei processi attivi (PCB e code);
2) Il S.O. deve allocare e de-allocare le risorse:
   - tempo di elaborazione;
   - memoria;
   - file;
   - dispositivi di I/O
3) Il S.O. deve proteggere i dati (sistemi di protezione);
4) Il risultato deve essere indipendente dalla velocità di esecuzione dei processi. Per assicurare ciò occorre studiare come avviene l'interazione tra processi

In un sistema operativo che supporti la concorrenza, nascono numerose implicazioni e problemi di progettazione, presentati in questa slide con le relative possibili soluzioni:

- Il sistema operativo deve sempre tenere traccia dei processi attivi e del loro stato. Ciò è realizzabile mediante tabelle contenenti i PCB (process control block) per tutti i processi attivi. Ogni PCB contiene tutte le informazioni relative ad un processo.

Il sistema operativo deve allocare e deallocare diverse risorse per ogni processo attivo, tra cui le più importanti:

- Il tempo di elaborazione, gestito dallo schedulatore.
- La memoria, eventualmente con meccanismi di memoria virtuale.
- I file.
- I canali di input/output.

Il sistema operativo deve inoltre proteggere i dati e le risorse di ogni processo da interferenze (volontarie o meno) da parte di altri processi. Questo richiede una interazione tra i sottosistemi che gestiscono la memoria, i file system ed i dispositivi di I/O.
I risultati che un processo produce devono essere indipendenti dalla sua velocità di esecuzione. Per assicurare quest'ultimo punto è necessario studiare l'interazione tra processi.

**INTERAZIONE FRA PROCESSI**

| Grado di conoscenza | Relazione | Influenza che un processo ha sugli altri | Possibili problemi di controllo |
|---|---|---|---|
| Processi che non si vedono tra loro | Competizione | • I risultati di un processo non dipendono dalle informazioni ottenute dagli altri<br>• Il tempo di esecuzione dei processi può cambiare | • Mutua esclusione<br>• Stallo (risorse riutilizzabili)<br>• Starvation |
| Processi che vedono gli altri processi indirettamente (es. oggetti condivisi) | Cooperazione tramite condivisione | • I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri<br>• Il tempo di esecuzione dei processi può cambiare | • Mutua esclusione<br>• Stallo (risorse riutilizzabili)<br>• Starvation<br>• Coerenza dei dati |
| Processi che vedono gli altri processi direttamente (usano primitive di comunicazione) | Cooperazione tramite comunicazione | • I risultati di un processo possono dipendere dalle informazioni ottenute dagli altri<br>• Il tempo di esecuzione dei processi può cambiare | • Stallo (risorse non riutilizzabili)<br>• Starvation |

**Tabella 5.1** *Interazione fra processi*

Per descrivere l'interazione tra processi, distinguiamo tre diverse situazioni, classificate in base al grado di conoscenza che un processo ha dell'esistenza degli altri processi. Esaminiamo di seguito i gradi possibili:

• Dei processi che non si vedono tra loro sono processi indipendenti, non disegnati per collaborare, come si verifica con la multiprogrammazione di processi indipendenti. L'indipendenza provoca competizione, la quale deve essere gestita dal sistema operativo: due processi in questa categoria possono richiedere accesso allo stesso disco, file o stampante, potenzialmente provocando conflitti. Il ruolo del sistema operativo è quello di gestire gli accessi. Ogni processo indipendente non ha influenza sugli altri processi, ed i risultati di uno non dipendono da eventuali informazioni possedute da un altro processo. Questo comporta che i tempi di esecuzione sono altrettanto indipendenti, poichè nessun processo deve attendere risposte da altri processi. Su questi processi infine deve essere effettuata la mutua esclusione, e vi è la possibilità di stalli o di starvation.

• I processi che vedono gli altri processi indirettamente non conoscono necessariamente i dettagli sugli altri processi attivi, ma condividono con essi l'accesso ad una risorsa. Tali processi non sono in competizione, ma effettuano una cooperazione mediante condivisione, poichè condividono un oggetto comune.

• I processi che vedono gli altri processi direttamente sono progettati per lavorare insieme, e possono comunicare tra loro. Essi effettuano quindi una cooperazione mediante comunicazione.

Nelle slide seguenti esamineremo in dettaglio queste tre classi di processi concorrenti.

**Processi che non si vedono tra loro sui possibili problemi di controllo:**

- Mutua esclusione

Una stampante è una "risorsa critica" (perché?):
(la parte di programma che usa una risorsa critica si chiama "sezione critica").

Nella mutua esclusione le sezioni critiche vengono gestite usando appositi comandi: entracritica ed escicritica. (vedasi esempio slide successiva).

- Deadlock
- Starvation

I processi che non si vedono tra loro sono in competizione per l'utilizzo di una o più risorse. In questa slide esaminiamo i possibili problemi di controllo di un gruppo di processi in competizione: la mutua esclusione, il deadlock e la starvation.

- Si supponga che due o più processi richiedano l'accesso ad una risorsa non condivisibile, come una stampante. Poichè l'utilizzo di una risorsa come questa impone una "sessione di lavoro" sulla risorsa, questa non può essere interrotta durante l'esecuzione. È necessario quindi garantire che un solo processo alla volta possa accedere alla stampante, chiamata risorsa critica. Inoltre, la parte di programma che fa uso della risorsa è detta sezione critica. L'accesso alla sezione critica è effettuato mediante una funzione, che qui chiameremo EntraCritica(). L'uscita è invece garantita da una seconda funzione, chiamata EsciCritica().

- Una volta che si è garantita la mutua esclusione, nascono altri due problemi di controllo: il deadlock e la starvation. Il deadlock (stallo) avviene ad esempio quando due processi, ciascuno in possesso di una risorsa, attendono reciprocamente il rilascio della risorsa posseduta dall'altro processo, restando bloccati fin quando non la ottengono. I due processi sono quindi in una condizione di stallo e attenderanno "per sempre".

L'ultimo problema di controllo è la starvation (morte per fame). Essa si verifica quando in un gruppo di processi in attesa di una risorsa, solo alcuni di essi riescano ad ottenerla, negando l'accesso ad alcuni processi del gruppo, nonostante non ci sia stallo.

**Programma di rappresentazione astratta del meccanismo della mutua esclusione**

```
program mutuaexclusione;
const n = . . . ; (*numero di processi *);
procedure P(i: integer);
begin
   repeat
      entracritica(R);
      < sezione critica >;
      escicritica(R);
      < resto del programma >
   forever
end;
begin (* programma principale *)
   parbegin
      P(1);
      P(2);
      . . .
      P(n)
   parend
end.
```

Una rappresentazione astratta del meccanismo della mutua esclusione è rappresentata in questa slide:
l'ipotetica procedura P() per l'accesso ad una risorsa non condivisibile è formata da una sezione critica e da una sezione non critica. Essa viene chiamata per ogni processo che abbia bisogno della risorsa cui P() fa riferimento.
Prima dell'ingresso nella sezione critica, viene richiamata EntraCritica(), la quale esegue il lavoro di controllo di accesso, e blocca il chiamante se necessario. Una volta ottenuto l'accesso e terminate le operazioni nella sezione critica, la funzione EsciCritica() provvede a liberare la risorsa e renderla disponibile per un nuovo processo. Il resto del programma non critico, può essere multiprogrammato senza rischi.

**Cooperazione tramite condivisione - Esempio del problema della coerenza**

```
-) Prima esecuzione concorrente:
       P1  a:=a+1;
           b:=b+1;
       P2  a:=a*2;
           b:=b*2;

-) Seconda esecuzione concorrente:
       P1  a:=a+1;
       P2  b:=b*2;
       P1  b:=b+1;
       P2  a:=a*2;

Risultato: perdita della coerenza!
Soluzione: Dichiarare critiche le sezioni P1 e P2.
```

La cooperazione tramite condivisione riguarda quei processi che intergiscono tra loro senza essere a conoscenza l'uno dell'altro. In questa situazione, il problema principale è quello della perdita della coerenza dei dati condivisi, quali possono essere variabili, file o basi di dati. I meccanismi di controllo devono quindi garantire l'integrità dei dati condivisi.
In particolare, le operazioni di scrittura sui dati devono essere mutualmente esclusive, ed essere quindi eseguite in sezioni critiche.
Un esempio di perdita della coerenza è visualizzato nella slide: a e b sono due variabili di un programma gestionale condivise tra due processi P1 e P2. a e b sono legate dalla relazione a = b, quindi ognuno dei due

processi quando modifica un valore esegue la stessa modifica sull'altro per preservare la relazione.
Questo è il caso illustrato nella prima esecuzione concorrente, in cui il processo P2 esegue le sue operazioni dopo la terminazione del processo P1.
Ma nella seconda esecuzione concorrente, il processo P2 inizia le sue operazioni prima del termine di quelle di P1: il risultato è la perdita della coerenza, poiché la relazione non è più verificata, anche se i processi hanno eseguito correttamente le operazioni richieste per preservarla.
La soluzione a questo problema è dichiarare critica la sezione di modifica dati, garantendo la mutua esclusività delle operazioni di modifica.

**Cooperazione fra Processi tramite comunicazione**

I messaggi possono essere scambiati:
- tramite primitive del Kernel;
- tramite istruzioni del linguaggio di programmazione.

IN QUESTO CASO:
- NON E' NECESSARIO GARANTIRE MUTUA ESCLUSIONE;
- PUO' VERIFICARSI STARVATION;
- PUO' VERIFICARSI STALLO.

Al contrario delle prime due tipologie di gruppi di processi, quelli che cooperano tramite comunicazione sono collegati tra loro per uno scopo comune: la comunicazione è un mezzo per sincronizzare o coordinare le attività svolte da ogni singolo processo.
Tipicamente la comunicazione è basata su uno scambio di messaggi, tramite:

- Primitive del kernel del sistema operativo.
- Istruzioni o funzioni del linguaggio di programmazione.

In questo caso, poiché lo scambio di messaggi non comporta condivisione, non è necessario garantire la mutua esclusione, ma può verificarsi la starvation e lo stallo:

- La starvation può verificarsi quando in un gruppo di processi (P1, P2, P3) un gruppo (P1, P2) vuol comunicare con un singolo processo (P3) e solo uno o alcuni membri del gruppo (P1) riescono a comunicare con successo, gli altri membri del gruppo (P2) restano bloccati indefinitamente.
- Lo stallo si può verificare invece quando due processi sono reciprocamente in attesa di una comunicazione dall'altro, bloccandoli entrambi indefinitamente.

**I Meccanismi per la mutua esclusione devono soddisfare i seguenti requisiti.**

- Un solo processo alla volta deve accedere alla sezione (o risorsa) critica;
- Ogni processo deve poter accedere dopo un tempo finito di attesa in coda (pertanto ogni processo deve restare nella sezione critica per un tempo finito) o nullo (se non ci sono altre richieste);
- Nessun processo deve aspettare infinitamente (no stallo o starvation);
- Non ci devono essere limitazioni sulla velocità di esecuzione.

E' infine ovvio che una volta specificato un meccanismo di accesso tutti i processi devono accedere alla risorsa tramite quel meccanismo

Compreso il principio teorico alla base della mutua esclusione, analizziamo i requisiti di un meccanismo che la implementi.
1.
Un solo processo alla volta deve avere accesso alla sezione (o risorsa) critica, dalla definizione stessa di mutua esclusione.
2.
Ogni processo in attesa di entrare in sezione critica deve avere la possibilità di accederci in un tempo finito di attesa, e questo implica che un processo non può restare indefinitamente nella sezione critica.
Inoltre se non ci sono processi in sezione critica, al primo processo che faccia richiesta di entrare deve essergli concesso senza attesa.
3.
Nessun processo deve aspettare indefinitamente quando chiede accesso alla sezione critica: questo comporta che non devono verificarsi ne stalli ne starvation.
4.
Il meccanismo non deve essere limitato ad un sistema con predeterminate caratteristiche di velocità di esecuzione: esso deve essere sempre valido con qualsiasi velocità di processore e qualsiasi numero di processori.
Infine, per efficacemente usare la mutua esclusione una volta definito il meccanismo, è necessario che tutti i processi debbano accedere alle risorse critiche tramite esso, altrimenti la progettazione ed implementazione risulta vanificata.

**I meccanismi per la mutua esclusione possono essere realizzati:**

- O con approcci software, cioè direttamente dai processi senza ausilio del Sistema Operativo o del linguaggio di programmazione; (uso di semafori)
- O usando particolari istruzioni di macchina; (si vedano i Monitor)
- O col supporto del Sistema Operativo o del linguaggio di programmazione. (uso di messaggi)

I meccanismi per la mutua esclusione possono essere implementati con diverse metodologie operative:

- Tramite approcci software, implementando direttamente nel codice del programma le primitive per l'entrata e l'uscita dalle sezioni critiche, senza l'aiuto dell'hardware, del sistema operativo o del linguaggio di programmazione (ad esempio utilizzando i semafori utente).
- Con l'aiuto dell'hardware e di istruzioni macchina predisposte alla mutua esclusione (ad esempio i monitor).

Con il supporto del sistema operativo (semafori kernel o messaggi) o del linguaggio di programmazione, il quale mette a disposizione strumenti per la gestione della mutua esclusione.

La differenza sostanziale tra i semafori utente e quelli kernel risiede nella partecipazione del sistema operativo alla loro realizzazione: un semaforo utente è costruito con degli spazi di memoria condivisi che indicano lo stato della sezione critica (libera od occupata). I semafori kernel sono invece implementati con l'ausilio del sistema operativo, il quale fornisce delle primitive atomiche, la cui esecuzione non può venire interrotta da un cambio di contesto.

**MUTUA ESCLUSIONE: supporto software**

Presentiamo adesso due implementazioni software di meccanismi di mutua esclusione: l'algoritmo di Dekker e l'algoritmo di Peterson.

**Approcci software alla "MUTUA ESCLUSIONE" - Algoritmo di DEKKER**


L'algoritmo di Dekker viene qui presentato per gradi, con quattro possibili soluzioni le quali via via risolvono meglio il problema della mutua esclusione. Questo tipo di esposizione ha il vantaggio di mostrare gli errori più comuni commessi durante lo sviluppo di programmi concorrenti.

Primo tentativo

Qualsiasi approccio alla mutua esclusione deve basarsi su un meccanismo fondamentale di esclusione a livello hardware: il più comune ed immediato consiste nell'effettuare un accesso alla volta, mentre l'altro processo è in attesa. Il primo tentativo si basa sull'attesa attiva (busy waiting) e sul protocollo dell'igloo come metafora, per l'esclusione degli accessi contemporanei.

I due frammenti di codice e l'immagine espongono in maniera immediata questa metafora:

- I due processi, Processo 0 e Processo 1 condividono una variabile globale, turno.
- Ogni processo è rappresentato da un eschimese, e il cartello indicatore vicino all'igloo indica la sezione critica.
- La variabile globale è invece rappresentata da una lavagna situata all'interno dell'igloo.
- Ad ogni processo è assegnato un identificatore univoco (0, 1)
- Quando il Processo 0 vuole eseguire la sezione critica, esso (entrare nell'igloo) controlla il valore della variabile globale (guarda la lavagna) e se il valore conservato nella variabile corrisponde al proprio identificatore allora entra nella sezione critica. Dopo aver eseguito le operazioni e prima di uscire dalla sezione critica, esso cambia il valore della variabile con l'identificatore dell'altro processo (lo scrive sulla lavagna).

Se, al contrario, nella variabile è scritto l'identificatore dell'altro processo (Processo 1), il Processo 0 continua a controllare il valore della variabile aspettando che esso cambi diventando uguale al suo identificatore.
Questo meccanismo è detto di attesa attiva ( busy waiting), poiché il processo in attesa non può fare nulla di produttivo finché non abbia il permesso di entrare nella sezione critica, quindi consuma (molto) tempo di CPU per il controllo del valore della variabile turno per un repentino intervento.
Questa soluzione garantisce la mutua esclusione, ma ha due punti deboli:
Il processo più lento stabilisce la velocità di avanzamento di entrambi i processi. Infatti poiché si osserva un'alternanza stretta per l'uso della sezione critica: se Processo 0 la utilizzasse una volta all'ora ma Processo 1 volesse utilizzarla 1000 volte all'ora, sarebbe forzato a mantenere il ritmo di Processo 0.
Se uno dei due processi si blocca, l'altro allo stesso modo rimane bloccato per sempre, poiché non vi è più l'opportunità di un'alternanza tra i due. Vi è quindi uno stallo.

**Approcci software alla "MUTUA ESCLUSIONE" - Algoritmo di DEKKER**


- Secondo tentativo

Il problema fondamentale del primo tentativo risiede nella mancanza di controllo dello stato di entrambi i processi, conservando nella variabile globale "turno" solo l'identificatore del processo autorizzato ad entrare in sezione critica.
È al contrario molto utile conservare informazioni sullo stato di entrambi i processi, in maniera che se uno di essi dovesse fallire l'altro potrebbe comunque entrare nella sua sezione critica.
Questa filosofia è illustrata in figura: ogni processo possiede il suo igloo, e può guardare la lavagna dell'altro, senza modificarla.
Sulle lavagne ora non vi è più scritto l'identificatore del processo autorizzato ad entrare in sezione critica, bensì è indicato se il processo proprietario della lavagna è o meno nella sua sezione critica. Ad esempio, se sulla lavagna del Processo 1 è scritto "vero", significa che esso è in sezione critica, se vi è scritto "falso" non lo è.
Quando un processo richiede l'entrata in sezione critica, controlla periodicamente la lavagna dell'altro processo, fin quando non vede scritto "falso": a questo punto esso si reca nel suo igloo e scrive "vero" sulla propria lavagna. Quando esce, esso deve scrivere invece "falso".
In questo caso la variabile condivisa non è più un intero, ma un array di boolean tanti quanti sono i processi in gioco, come mostrato nella slide.
Il programma per i due processi allo stesso modo è diverso, il primo attende che il flag del secondo assuma il valore false, tramite un'attesa attiva. In seguito esso setta il proprio flag a true, esegue la sua sezione critica e risetta il suo flag a false.
Come detto in precedenza, questa soluzione garantisce la possibilità per un processo di operare sempre anche se l'altro si è bloccato fuori dalla sua sezione critica, poiché in tal caso il flag di questi sarà sempre false.

- Ma se uno dei due processi fallisce all'interno della sua sezione critica, l'altro è bloccato per sempre e si ha nuovamente uno stallo.

- Infine, questa soluzione non garantisce neppure una corretta mutua esclusione, in quanto entrambi, vedendo il flag dell'altro a false, possono entrare contemporaneamente in sezione critica. Si consideri ad esempio questa sequenza:

- Il sistema operativo manda in esecuzione il Processo 0, il quale vede il flag di Processo 1 come false, supera il controllo ed entra in sezione critica.

- Il sistema operativo subito dopo manda in esecuzione Processo 1, prima che Processo 0 possa settare il suo flag a true, e anche Processo 1 entra in sezione critica.

Il codice quindi non è corretto, è necessario un terzo tentativo.

## Approcci software alla "MUTUA ESCLUSIONE" - Algoritmo di DEKKER

Approccio per gradi per mostrare gli errori più comuni della programmazione concorrente


3° tentativo:

```
var flag: array [0..1] of boolean;

Processo 0                  Processo 1
...                         ...
flag [0] := true;           flag [1 ]:= true;
while flag[1] do            while flag[0] do
{nulla};                    {nulla};
<sezione critica>;          <sezione critica>;
flag [0] := false;          flag [1]:= false;
...                         ...
```

- se un processo fallisce entro la sua sezione critica o prima di aver messo false nel suo flag, l'altro è bloccato, non lo è se fallisce fuori.
- la mutua esclusione è garantita ma se entrambi mettono a true il flag si ha lo stallo.

• Terzo tentativo

Il fallimento del secondo tentativo era dovuto alla possibilità per un processo di cambiare il proprio stato dopo che l'altro processo lo avesse letto, ma prima che questi sia entrato in sezione critica.

Il terzo tentativo elimina questa possibilità, spostando il settaggio del proprio flag di esclusione prima del controllo del flag altrui.

Questo approccio soffre comunque del problema di stallo del precedente, in quanto se un processo fallisce all'interno della sua sezione critica o prima di aver settato il suo flag a false, l'altro viene bloccato per sempre. Al contrario, se un processo fallisce all'esterno della sua sezione critica, l'altro non è bloccato.

La mutua esclusione in questo caso è garantita, ma questo approccio soffre inoltre di un altro problema di stallo, poiché se entrambi i processi mettono a true il loro flag quando nessuno dei due ha ancora eseguito il while, allora ognuno penserà che l'altro sia entrato in sezione critica, causando uno stallo.

## Approcci software alla "MUTUA ESCLUSIONE" - Algoritmo di DEKKER

Approccio per gradi per mostrare gli errori più comuni della programmazione concorrente (4° tentativo) Mutua Cortesia

```
Processo 0                  Processo 1
...                         ...
flag [0]:= true;            flag [1] := true;
while flag [1]  do;         while flag [0] do;
    begin                       begin
      flag [0] := false;          flag [1]:=false;
      <breve pausa>;              <breve pausa>;
      flag [0] := true;           flag [1] := true;
    end;                        end;
<sezione critica>;          <sezione critica>;
flag [0] := false;          flag [1] := false;
...                         ....
```

Siamo vicini alla soluzione ma se entrambi passano da (true,true) a (false,false) a (true,true) la mutua esclusione resta legata alla velocità di ciasuno e ciò non va bene.

•

Quarto tentativo

Nel terzo tentativo ogni processo modifica il proprio stato senza conoscere lo stato dell'altro processo. Si ha stallo poiché ogni processo si ostina ad entrare nella propria sezione critica e non esce da quella situazione.

Si può invece provare a rendere ogni processo più accomodante: ciascuno mette il proprio flag a true per indicare il proprio desiderio di entrare in sezione critica, ma si è anche pronti a rimettere i flag a false per lasciare spazio all'altro processo.

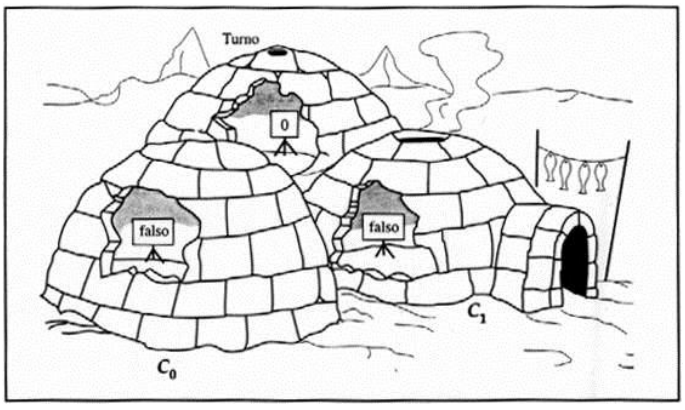
Questa è detta mutua cortesia. Come si vede dal frammento di codice, ogni processo controlla il flag dell'altro, attendendo che assuma il valore false. Quando ciò avviene, il proprio flag viene resettato a false, si esegue una breve pausa e si risetta il proprio flag a true. Infine si riesegue il ciclo.

La cortesia consiste nella breve pausa, durante la quale l'altro processo può entrare nella propria sezione critica.

In questa soluzione la mutua esclusione è garantita, e viene anche eliminata l'attesa attiva, poiché viene effettuata una pausa passiva all'interno del ciclo.

Ma questo approccio presenta un altro problema: se i processi hanno la stessa velocità di esecuzione, si potrebbe transitare in una situazione in cui entrambi i flag passano da true, true a false, false e di nuovo a true, true e così via. Questo "eccesso di cortesia" porterebbe ad un susseguirsi di cambi di flag senza fornire alcun risultato produttivo. Questo non è stallo, poiché un cambiamento nella velocità relativa di uno dei due processi sbloccherebbe la situazione, ma questa aleatorietà nel blocco rende anche questo tentativo inaccettabile.

## ALGORITMO DI DEKKER




La soluzione corretta consiste nel creare un algoritmo con tre igloo, come mostrato in figura.

**Algoritmo di DEKKER**

La soluzione corretta si ottiene combinando l'algoritmo della "mutua cortesia" con la variabile "turno" del I° tentativo.

Infatti vedremo nella prossima slide, in cui è riportato l'algoritmo di DEKKER, che P0 mette flag [0] a true e va a controllare flag [1] che trovandolo true deve vedere se effettivamente è il turno di P1 e se è così allora deve rimettere false in flag [0] e fino a che il turno resta 1 deve far nulla. Non appena il turno diventa 0 pone flag [0] a true ed entra nella sezione critica.
Da cui uscendo porrà turno a 1 e flag [0] false.

E' l'algoritmo di Dekker di cui una breve spiegazione è presentata in questa slide.
L'algoritmo si ottiene combinando la variabile turno del primo approccio con l'algoritmo della mutua cortesia del quarto. In questo modo si ha un igloo "arbitro" in cui è conservata la lavagna della variabile turno e i due igloo privati dei due processi.
Quando P0 vuole entrare in sezione critica, mette il proprio flag (flag[0]) a true e controlla il flag di P1 e la variabile turno. Se trova il flag di P1 a true ed il turno ad uno, sa che il processo uno è in sezione critica e setta nuovamente il proprio flag a false, continuando a controllare la variabile turno, fin quando questa non raggiunge valore zero. A questo punto esso setta nuovamente il suo flag a true ed entra in sezione critica.
È inoltre evidente che all'uscita dalla sezione critica il processo P0 dovrà reimpostare turno ad uno ed il proprio flag a false, per permettere al processo P1 di richiedere la sezione critica.
Infine, se P0 trova il flag dell'altro processo false, entra direttamente in sezione critica senza bisogno di controllare turno.
Il codice dell'algoritmo di Dekker è nella slide seguente.

**ALGORITMO DI DEKKER**

```
var flag: array [0 .. 1] of boolean;
    turno: 0 .. 1;
procedure P0;
begin
    repeat
        flag [0] := true;
        while flag [1] do if turno = 1 then
            begin
                flag [0] := false;
                while turno = 1 do { nulla };
                    flag [0] := true
            end;
        < sezione critica >;
        turno := 1;
        flag [0] := false;
        < resto del programma >
    forever
end;
procedure P1;
begin
    repeat
        flag [1] := true;
        while flag [0] do if turno = 0 then
            begin
                flag [1] := false;
                while turno = 0 do { nulla };
                    flag [1] := true
            end;
        < sezione critica >;
        turno := 0;
        flag [1] := false;
        < resto del programma >
    forever
end;
begin
    flag [0] := false;
    flag [1] := false;
    turno := 1;
    parbegin
        P0; P1
    parend
end.
```

Il codice illustra i concetti teorici espressi precedentemente, le due procedure P0 e P1 rappresentano i due processi concorrenti, ed il costrutto parbegin .. parend nella procedura principale del programma ha l'effetto di sospenderne l'esecuzione ed iniziare quella delle due procedure come due processi concorrenti.

**Algoritmo di PETERSON**

- L'algoritmo di Peterson si articola su un rapporto incrociato di concessione anticipata del turno all'altro processo e di verifica dello stato dell'altro processo finchè quest'ultimo diventi falso, solo allora potrà entrare nella sezione critica.
- Uscendo dalla sezione critica il flag del

I l secondo approccio software alla mutua esclusione presentato è l'algoritmo di Peterson: esso al contrario dell'algoritmo di Dekker è una soluzione semplice ed elegante, dalla facile dimostrazione.
Le variabili utilizzate sono le stesse dell'algoritmo di Dekker: l'array globale flag indica la posizione dei vari processi rispetto alla mutua esclusione, e la variabile globale turno risolve i conflitti.
Come nel primo approccio di Dekker, ogni processo ha un suo identificatore univoco, e la variabile turno assume il valore dell'identificatore del processo che ha il permesso di entrare in sezione critica.
L'algoritmo di Peterson impone che un processo prima di entrare in sezione critica debba settare il proprio flag a true e concedere anticipatamente il turno all'altro processo, ed attendere sia che il flag dell'altro processo diventi false, sia che il turno

processo chiamante verrà messo a false.

- E' evidente che l'algoritmo di Peterson garantisce la mutia esclusione.

corrisponda al proprio identificatore. In questo caso, il processo entra in sezione critica ed all'uscita imposta il proprio flag a false.
La mutua esclusione è garantita, ed inoltre un processo non può monopolizzare la sezione critica, poiché prima dell'ingresso deve dare una possibilità all'altro di entrare, impostando turno con l'identificatore di quest'ultimo.

**ALGORITMO DI PETERSON**

```
var flag: array [0 .. 1] of boolean;
    turno: 0 .. 1;
procedure P0;
begin
  repeat
    flag [0] := true;
    turno := 1;
    while flag [1] and turno = 1 do { nulla };
    < sezione critica >;
    flag [0] := false;
    < resto del programma >
  forever
end;
procedure P1;
begin
  repeat
    flag [1] := true;
    turno := 0;
    while flag [0] and turno = 0 do { nulla };
    < sezione critica >;
    flag [1] := false;
    < resto del programma >
  forever
end;
begin
  flag [0] := false;
  flag [1] := false;
  turno := 1;
  parbegin
    P0; P1
  parend
end.
```

La slide illustra un codice di esempio che implementa l'algoritmo di Peterson per due processi, rappresentati come solito dalle due procedure P0 e P1, e parbegin .. parend rappresentano il consueto costrutto per interrompere il programma principale e avviare le due procedure come processi concorrenti.