PCD Module 2.2

Il problema della sezione critica

N processi, ognuno esegue in un loop infinito una sequenza di istruzioni che può essere divisa in 2 sottosequenze, CS e non-CS (NCS):

```
loop {
  NCS
  entry protocol
  CS
  exit protocol
  NCS
}
```

Ogni sezione critica è una sequenza di istruzioni che accede ad un oggetto condiviso.

Proprietà delle CS

Obiettivo: progettare *entry* e *exit protocol* che soddisfino:

- **mutex**: le istruzioni dai 2+ processi non devono essere sovrapposte
- **libero da deadlock**: se un processo prova di entrare in CS, un processo deve per forza avere successo
- **libero da starvation individuale**: se un qualsiasi processo prova ad entrare in CS, quel processo deve aver successo

Ogni soluzione proposta deve soddisfare anche la **proprietà progressa** delle CS: una volta che un processo inizia l'esecuzione di istruzioni nella CS, deve finire. Ciò non è richiesto nella NCS.

Problemi concreti basati sulle CS

Il problema della CS serve per sviluppare modelli che eseguono computazioni complesse, ma occasionalmente hanno bisogno di accedere a dati o risorse condivise da più processi (es. chiosco check-in aeroporto).

Algoritmi vs. Meccanismi

CS può essere risolto tramite:

- **meccanismi**, es. semafori o blocchi (piu avanti)
- **algoritmi** (questo modulo), utilizzando solo istruzioni base atomiche (load, store)

CS di due processi: tentativi

- 1° tentativo: `await turn`, istruzione che attende la fine dell'istruzione `turn`. Implementabile tramite busy-wait loop.
    - Studio della correttezza dell'algoritmo, riduzione stati
    - Starvation non soddisfatta
- 2° tentativo: variabili globali che indicano quando un processo è nella sua CS
    - mutex non soddisfatta, raggiungibile a causa della nonatomicità di pre e post protocol.
- 3° tentativo: si considera `await` parte della CS, si spostano gli assignment prima dell' `await`
    - mutex risolta ma possono accadere deadlock (livelock)
- 4° tentativo: richiedere che un processo si arrenda ad entrare nella CS se vede che è contesa da altri processi
    - deadlock risolta, ma c'è possibilità di starvation in caso di perfetta sovrapposizione

Algoritmi

Algoritmo di Dekker

Combo 1° + 4° tentativo. Si richiede che il *diritto di insistere ad entrare* sia esplicitamente passato tra processi per mezzo della variabile `turn`. È corretto: soddisfa mutex ed è libero da deadlock e starvation.

Algoritmo di Peterson

Più conciso, collassa le due istruzioni `await` in una con una condizione composta.

Istruzioni atomiche composte

Il problema della CS può essere semplificato se si sfruttano istruzioni atomiche più complesse fornite dalla macchina concorrente, es. *test-and-set*, *exchange*, *fetch-and-add*, *compare-and-swap*. Ad esempio, *test-and-set* è un'istruzione atomica definita come l'esecuzione di `test-and-set(x,r): <r:=x, x:=1>` senza possibilità di sovrapposizione. Notazione tipica: `< ... >` tra gruppi di istruzioni.

Blocchi

Sfruttando istruzioni atomiche composte si può facilmente realizzare un meccanismo di blocco basilare.

Due fasi: acquisizione e rilascio del blocco. Pg. 21

Istruzioni atomiche in Java: `synchronized`

Istruzioni atomiche possono essere implementate in Java mediante l'uso di `synchronized`. In questo caso le sequenze non si sovrappongono.

Soluzione con ticket: l'Algoritmo di Bakery (tie-breaker)

Introduzione di un ticket per stabilire il turno di un processo. Possibili overflow aritmetici di `turn` e `num`.