TERMINI IMPORTANTI

Operazione Atomica

funzione o operazione con uno o più linee di codice che non può essere divisa. Sarà garantito che essa verrà eseguita come gruppo, senza che l'esecuzione possa venire messa in pausa durante.
E' ISOLATA rispetto ad altre operazioni atomiche

Sezione Critica

Numero di linee di codice che hanno necessità di accedere a risorse comuni, ma mentre è in esecuzione nient'altro è nella corrispondente sezione di codice

Deadlock

Situazione nella quale 2 o più processi sono in attesa l'uno dell'altro per proseguire

Mutua Esclusione

La richiesta che, mentre un processo è in una sezione critica e accede a risorse comuni, che nessun altro possa essere in un'altra sezione critica accedente alle stesse

Esecuzione CONCORRENTE

Indispensabile per l'uso di tutti i giorni degli OS, in modo che processi e thread si eseguano in maniera corretta

→ Terminologia dei problemi slide 2.4

DIFFICOLTÀ TECNICHE

- Garantire un accesso e scrittura di dati concorrenziali
- Gestione errori non facile, possibilità di non determinismo

↓

RACE CONDITION:

Condizione che si verifica quando c'è accesso concorrente ad uno stesso set di dati, ma l'output dipende dall'ordine di esecuzione

Nel caso di un accesso concorrente alla stessa risorsa (es. io, file, dock...) ci sono 3 DIVERSI PROBLEMI.

MUTUA ESCLUSIONE: è la richiesta che, quando un processo è in un'esecuzione critica, nessun altro processo abbia esecuzioni critiche da accedo alla stessa cosa

DEAD LOCK: problema in cui due processi si bloccano a vicenda, aspettando che uno liberi l'altro

STARVATION problema che consiste nel fatto che lo scheduler posone indefinitivamente l'esecuzione, senza mai sceglierlo

È importante per un os garantire che gli output di un processo o programma siano INDIPENDENTI dalla velocità di processamento

Per evitare questo, assieme anche al fatto che processi diversi possano competere per lo stesso set di dati, bisogna dover GARANTIRE la mutua esclusione

Requisiti per la mutua esclusione

- Deve essere obbligata
- Un processo in una sezione non critica che si ferma non deve interferire con altri
- Non ci devi essere deadlock o starvation
- Un processo deve poter accedere alla sezione critica se nessun'altro lo sta usando
- Non ci: deve essere dipendenza tra velocità di processamento o numero di processi
- Un processo rimane in una sezione critica per un TEMPO FINITO

Come fare?

Ci sono 2 modi, sostanzialmente. Ogni processo dovrà poter comunicare l'un con l'altro in modo che ci sia mutua esclusione, senza l'aiuto di istruzioni dal SISTEMA OPERATIVO o dalla CPU. SOFTWARE APPROACH

Oppure si possono sfruttare delle istruzioni macchina che possono garantire, per esempio, l'atomicità di operazioni. HARDWARE APPROACH

HARDWARE APPROACH

INTERRUPT DISABLING:

In un sistema uniprocessor, non ci sarà mai esecuzione concorrente. Infatti un processo continuerà ad essere eseguito fino a che non eseguirà un processo OS oppure finché non verrà INTERROTTO.
È possibile quindi garantire mutua esclusione disabilitando temporaneamente gli interrupt, e riabilitandoli una volta fuori dalla sezione critica.
Buon sistema ma:

- La performance concorrenziale sarà peggiorata
- in un sistema multiprocessore esso non funziona.

Soluzione:

ISTRUZIONI MACCHINA SPECIFICHE

In un sistema multiprocessore più processi accedono a una memoria condivisa. Non ci sono interrupt.
Come detto è necessario che se un processo richiede accesso in memoria allora nessun altro lo può avere.
Soluzione sta nel costruire istruzioni macchina che possono ATOMICAMENTE 2 azioni

- compare_and_swap
- Exchange

COMPARE & SWAP

È chiamata anche istruzione di scambio, e funziona a questo modo

```
int compare_and_swap (int * word, int testval, int newval)
{
    int oldval = *word;
    if (oldval == testval)
        *word = newval;
    return oldval;
}
```

la funzione è semplice. Compara il valore nella memoria word. Se è uguale ad un valore di test allora gli cambia il valore a newval, altrimenti no.
Il valore di ritorno è sempre il vecchio valore, quindi c'è stato uno scambio solo se il valore di ritorno è UGUALE al valore di test

Viene spesso usata per concorrenza.

```
int bolt;
void P(int i){
 while (true){
   while (compare_and_swap(bolt, 0, 1) == 1)
     //nothing
   //sezione critica;
   bolt = 0;
   //continua;
   }
}

void main(){
  bolt = 0;
  parallel begin (P(1), P(2), ..., P(n));
}
```

Una variabile bolt è inizializzata a 0. L'unico processo che può entrare nello stato critico è quello che trova bolt uguale a ZERO. Tutti gli altri entrano in uno stato BUSY WAITING, ovvero attendono senza fare nulla se non VERIFICARE che possano accedervi. Solo quando la C&S è zero vuol dire che bolt era zero. Una volta finito lo stato critico, bolt ritorna a zero

EXCHANGE

L'istruzione scambia il contenuto di un registro con uno in memoria. Sia l'IA-32 che IA-64 supportano l'istruzione XCHG.

```
void exchange (int *register, int *memory)
{
    int temp = *memory;
    *memory = *register;
    *register = temp;
}
```

Nel codice sottostante si può vedere come la mutua esclusione venga rispettata con exchange.
Anche questo caso, una variabile bolt è inizializzata a ZERO, e ogni processo usa la variabile keyi inizializzata ad uno.

L'unico processo che può entrare nello stato critico è quello che trova bolt = 0.
Esclude tutti gli altri processi dallo stato critico impostando bolt su 1, fino a che non termina lo stato critico tornando ad 1

```
int bolt;
void P(int i){
 while (true){
  int keyi = 1;
  do exchange (&keyi, &bolt);
  while (keyi!=0);
// critical section
  bolt = 0
// fine
}

void main(){
  bolt = 0;
  parallel begin (P(1), P(2), ..., P(n));
}
```

È importante NON comparare direttamente su bolt

Alcune proprietà

VANTAGGI

- Facilmente applicabile anche a sistemi multiprocesso
- Facilmente verificabile
- Può supportare più sezioni critiche aggiungendo per ogni sezione una variabile diversa

SVANTAGGI

- BUSY WAITING: il processo che è in attesa del suo turno consuma comunque tempo di processamento e risorse della CPU
- STARVATION: Quando un processo si toglie dalla sezione critica si cerca un nuovo processo da eseguire. La decisione è arbitraria e un processo potrebbe attendere INDEFINITIVAMENTE
- DEADLOCK possibile, se priorità è differente.

Si USANO ALTRI MECCANISMI, e.g. semafori

SEMAFORI

È una VARIABILE su cui ci si può scrivere SOLO in tre operazioni, in teoria:

1) Può essere inizializzato ad un valore intero NON negativo
2) il semWait decrementa il valore
3) il semSignal incrementa il valore

Non si può leggere o modificare direttamente il valore!

Conseguenze

Non c'è modo in un sistema uniprocessore di sapere, dopo un invio di semWait, se il processo è bloccato o no

Non c'è modo di sapere quale processo verrà ripreso subito dopo un semSignal, se 2 processi sono concorrenti

Il valore del semaforo inizializzato è NON NEGATIVO: se il valore è maggiore di zero, esso equivale a dire il numero di processi che possono dare il wait e continuare l'esecuzione

SEMWAIT() serve per dire che il processo sta richiedendo accesso ad una risorsa o ad una sezione critica, decrementa il semaforo

SEMSIGNAL() è utilizzata per rilasciare una risorsa o una sezione critica. Aumenta il valore del semaforo consentendo ad un altro processo in wait di continuare

Monitor

- Sono tipo semafori, ma per i costrutti orientati ad oggetti, e sono più facili da controllare

tutte le variabili locali sono accessibili solo dai processi del monitor.

Un processo per volta può essere eseguito nel monitor

Sfrutta variabili di condizione

Accessibile solo nel monitor, modificabili solo da

cwait(c)

csignal(c)

[serve di più]

MESSAGE PASSING

Serve a far interagire 2 processi secondo:

SINCRONIZZAZIONE: esempio, risorsa liberata, di nuovo accessibile

COMUNICAZIONE: esempio, output del pezzo di codice

funzionano con sistemi multiprocessore, distribuiti e uniprocessore, estremamente flessibile.

Funziona con 2 PRIMITIVE:

SEND (destination, messaggio)

RECEIVE (source, messaggio)