Monitor

Tipo astratto con variabili e procedure, garantisce mutua esclusione per il loro accesso

I monitor funzionano per i thread

Dentro al monitor vi sono le variabili usate in condivisione

Il monitor garantisce che un thread alla volta possa accedervi.

Questo vincolo di accesso è meccanismo di sincronizzazione è fornito da

- coda di attesa interna
- variabili di condizione | wait, signal > operazioni sulle var di condizione.

↓

Variabili speciali

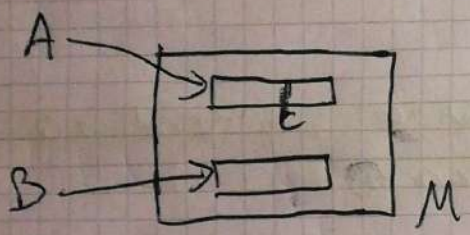
Per ogni variabile c'è una coda di attesa per i thread che si sono legalmente bloccati

le variabili e le operazioni sono molto semplici, fare signal su un thread non bloccato non dà problemi, e gli interbloccamenti (race cond.) non si verificano.

Le sezioni critiche sono gestite all'interno dei metodi usando strumenti come semafori mutex.

Problema

Hyp: A e B lavorano con un monitor M
A entra in M e fa wait su C



Ora B può entrare, B vede le condizioni per risvegliare A.
Fa signal su C

Ma così c'è un A che B in M.
Per risolvere il problema basta sapere che la signal non fa solo il risveglio
La signal può avere varie semantiche

- Hoare (teorico): vale come "signal and Wait" cioè risveglia A e va a dormire
- Mesa (Java): "signal & continue" il risveglio avviene effettivamente solo alla fine del lavoro di B.

  Svantaggio: la risveglia i thread non nei momenti propizi
- Compromesso: "Signal & return"

  Si forza il programmatore a fare la signal alla fine del thread

In questo modo si evita il deadlock tra la chiamata signal e la sua effettiva realizzazione

NB: Se faccio signal su un non dormiente non c'è problema, non ci si pone il problema di "memorizzare" le sveglie

Produttore - consumatore coi monitor

```
Monitor PC_monitor
    condition full, empty
    int count = 0

function insert (item)
    if count = N then wait(full)
    insert_item (item)
    count++
    if count = 1 then signal(empty)

function remove()
    if count = 0 then wait(empty)
    removed = remove_item()
    count--
    if count = N-1 then signal(full)
```

```
function producer()
    while (true)
        item = produce_item()
        PC_monitor.insert(item)

function consumer()
    while(true)
        item = PC_mon_remove()
        consume(item)
```

Dato che insert e remove hanno mutua esclusione, non abbiamo problemi legati alle corse critiche.

Scambio messaggi tra processi

Tra thread è inutile usare i messaggi, meglio memoria condivisa, scambiamo messaggi i processi.

Si usano le primitive | send(dest, msg)
receive(src, msg)

La receive potrebbe essere bloccante se il documento è dove essere

Il problema dello scambio di messaggi è estendibile anche in caso ci siano più macchine con le librerie MPI

In realtà anche la send può diventare bloccante se lo scambio è tramite buffer e il buffer è pieno.

Metodi indirizzamento msg: diretto o mailbox

diretto → invia msg a Luca

Mailbox → lascia msg nella casella | quindi si può evitare stare lì

Problema: lo scambio di messaggi è meno efficiente rispetto alla memoria condivisa.

Produttore - Consumatore con messaggi

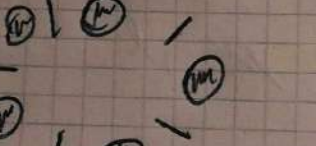
```
function producer()
  while (TRUE)
    item = produce_item()
    build_msg(m, item)
    send(consumer, msg)
```

```
function consumer()
  while (true)
    receive(producer, msg)
    item = extract_msg(msg)
    consume(item)
```

Problema dei 5 filosofi

Descrive il problema dell'accesso ad un numero limitato di risorse da parte di processi in concorrenza

Abbiamo 5 filosofi, 5 piatti e 5 forchette



Un filosofo per mangiare deve usare entrambe le forchette accanto (forchette)

Dunque dobbiamo gestire le risorse e i processi (filosofi) che le vogliono usare

Soluzione 1

```
int N = 5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

] - potrebbero essere bloccati

Problema: se ogni filosofo prende la fork a dx ci si blocca

Soluzione 2

Come in Sol 1 però la take_fork non è bloccante.
Se ha preso una sola forchetta e l'altra è bloccata, rilascia quella che ha preso e ci prova.
C'è la remota possibilità che si crei uno stallo attivo.

Soluzione 3

Facciamo come in Sol 2 un nuovo tentativo aspettando però un tempo random

Soluzione 4

Usare un mutex sul tavolo → Permettiamo a un solo filosofo alla volta di mangiare

Soluzione basata su semafori slide

Soluzione basata su monitor slide

```
int N=5; int THINKING=0
int HUNGRY=1; int EATING=2
int state[N]
semaphore mutex=1
semaphore s[N]={0,...,0}
```

```
function philosopher(int i)
    while (true) do
        think()
        take_forks(i)
        eat()
        put_forks(i)
```

```
function take_forks(int i)
    down(mutex)
    state[i]=HUNGRY
    test(i)
    up(mutex)
    down(s[i])

function put_forks(int i)
    down(mutex)
    state[i]=THINKING
    test(left(i))
    test(right(i))
    up(mutex)
```

```
function left(int i)  = i-1 mod N
function right(int i) = i+1 mod N

function test(int i)
    if state[i]=HUNGRY and state[left(i)]!=EATING and state[right(i)]!=EATING
        state[i]=EATING
        up(s[i])
```

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2

monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            wait(self[i])

    function put_forks(int i)
        state[i] = THINKING;
        test(left(i));
        test(right(i));

    function test(int i)
        if ( state[left(i)] != EATING and state[i] = HUNGRY
         and state[right(i)] != EATING )
            state[i] = EATING
            signal(self[i])

function philosopher(int i)
    while (true) do
        think()
        dp_monitor.take_forks(i)
        eat()
        dp_monitor.put_forks(i)
```