Corso di Programmazione Concorrente

I Monitor

Valter Crescenzi
crescenz@dia.uniroma3.it
http://crescenzi.inf.uniroma3.it

Sommario

- Monitor e Tipi di Dato Astratto
  - sintassi & semantica
- Le primitive wait & signal di Hoare
  - Utilizzo delle primitive
  - Tre semantiche alternative
- *Mailbox*
- *Barbiere Dormiente*
- Implementazione tramite semafori
- Monitor: Vantaggi & Svantaggi
- Monitor e POO

I Monitor (1)

- Nascono dall'adattamento del concetto di regione critica al concetto di *tipo di dato astratto*

- Un tipo di dato astratto *T* racchiude in una *unità logicamente coesa:*
  - la rappresentazione di un dato di tipo *T*
  - tutte le operazioni lecite su dati di tipo *T*

- Possono esistere diverse *istanze* dello stesso monitor cosi come possono esistere diverse istanze della medesima risorsa

I Monitor (2)

- L’idea è che la rappresentazione del tipo di dato astratta costituisce la risorsa comune condivisa alla quale vengono allegate tutte le operazioni che permettono di manipolarla in sezione critica

- Sono lo strumento grazie al quale i costrutti per la specifica di codice concorrente trovano una collocazione elegante nell'ambito dei linguaggi di programmazione orientati agli oggetti

Una Possibile Sintassi per i Monitor

type <*nome_monitor*> = **monitor**
    <*dichiarazioni di tipi e costanti globali*>;
var <*variabili condivise*>;
entry procedure $\mathbf{OP_1}$ (<*lista parametri*>) **begin**
    <*dichiarazioni locali*>;
    <*corpo di* OP_1>;
end
…
entry procedure $\mathbf{OP_n}$ (<*lista parametri*>) **begin**
    <*dichiarazioni locali*>;
    <*corpo di* Op_n>;
end

<*eventuali procedure locali al monitor*>;
<*procedura di inizializzazione delle var. condivise*>;
end monitor;

Operazioni e Variabili di Monitor

- Il monitor esporta le OP_1 … OP_n che costituiscono le uniche operazioni permesse sui dati di tipo *<nome_monitor>*
- Le operazioni OP_i vengono richiamate su una istanza di monitor con questa sintassi
  - *<nome istanza>*.OP_i(*<parametri attuali>*)
  - può servire una particolare procedura per inizializzare le variabili condivise delle istanze di monitor non appena create, ovvero per porre la risorsa nello stato iniziale voluto
    - n.b. equivalente ai “costruttori”
- Le variabili del monitor
  - rappresentano la stato della risorsa comune
  - sono condivise ed accessibili *unicamente* tramite le OP_1…OP_n da tutti i f.d.e. che usano la stessa istanza

Semantica dei Monitor

- Le OP_1 … OP_n vengono richiamate in sezione critica sull'istanza del monitor stesso
- L'esecuzione delle OP_i avviene in modo mutuamente esclusivo con quella di possibili chiamate concorrenti ad operazioni sulla stessa *istanza* di monitor

Le Primitive **wait** e **signal** di Hoare (1)

- I monitor rispetto alle regioni critiche
  - risolvono il problema della scarsa coesione
  - consentono di risolvere elegantemente problemi di competizione
  - *non* consentono di risolvere facilmente i problemi di cooperazione per la mancanza di meccanismi espliciti di sincronizzazione tra f.d.e.
    - le regioni critiche condizionali invece consentivano la risoluzione dei problemi di sincronizzazione ma risultano inefficienti a causa della rivalutazione delle condizioni
- Per superare questi limiti bisogna introdurre nei monitor le primitive **wait** e **signal** di C.A.R. Hoare per la sincronizzazione tra f.d.e.

Le Primitive **wait** e **signal** di Hoare (2)

- L'idea è di associare una *variabile condizione* agli eventi o condizioni di interesse su cui sincronizzarsi
  - sostanzialmente un modo per assegnare un nome esplicito a degli eventi di interesse (cfr. regioni critiche condizionali)
  - si tratta di condizioni su una risorsa condivisa (una istanza di monitor) per la quale competono diversi f.d.e.
  - tali f.d.e. durante le loro sezioni critiche modificano lo stato della risorsa e finiscono per alterare la valutazione delle condizioni
  - i f.d.e. che attendono l'evento associato possono esplicitare il proprio interesse emettendo una **wait** sulla variabile condizione
  - i f.d.e. che sono nella posizione di conoscere quando tali eventi si verificano notificano i f.d.e. interessati emettendo una **signal** sulla variabile condizione associata

wait e **signal**: Semantica (1)

- Alla var. condizione è associata una coda di attesa
- Quando un f.d.e. esegue una

  <*variabile-condizione*>.**wait**
  - viene bloccato ed inserito nella coda di attesa
  - rilascia il monitor

  <*variabile-condizione*>.**signal**
  - se nessun altro f.d.e. è in attesa sulla variabile condizione prosegue il proprio avanzamento; altrimenti:
  - viene bloccato e rilascia il monitor
  - il primo dei f.d.e. in attesa sulla variabile condizione viene risvegliato per rientrare in competizione sul monitor

wait e **signal**: Semantica (2)

- Attenzione a non confondere la coda di attesa associata alla variabile condizione con altre code
  - ad es. quella utilizzata dai semafori con cui viene spesso realizzata la mutua esclusione nei monitor
- Per evitare ambiguità ed inefficienze, per il momento ipotizziamo che **signal** sia *sempre* l'ultima istruzione eseguita in una procedura del monitor
  - altrimenti il f.d.e. che la esegue va in competizione per il monitor con il f.d.e. che ha appena risvegliato >>

Utilizzo di **wait** e **signal**

- Per utilizzarle basta associare una variabile condizione $X_{<cond>}$ per ogni condizione <*cond*> necessaria all'avanzamento dell'esecuzione in una procedura del monitor ed inserire

  while (**not** <*cond*>) **do**

  $X_{<cond>}$.**wait**;

  end;

- E' poi compito del programmatore individuare i punti in cui <*cond*> può diventare vera e forzare la segnalazione agli altri f.d.e. tramite la $X_{<cond>}$.**signal**;

Esercizi

Esercizio: Elencare le motivazioni dietro la seguente affermazione:

“per attendere un condizione con $X_{<cond>}$.**wait** conviene *sempre* utilizzare un'istruzione **while** piuttosto che **if** per proteggere la valutazione della condizione”.

Diverse Semantiche di **wait** e **signal**

- *signal_and_continue*:
  - il flusso che esegue la **signal** continua indisturbato mantenendo l'uso esclusivo del monitor sino alla naturale terminazione della entry-procedure corrente
- *signal_and_wait*:
  - il flusso che esegue la **signal** viene sospeso e rilascia il monitor per dare la possibilità ai flussi interessati di reagire immediatamente alla segnalazione
- *signal_and_return*:
  - le due semantiche di sopra collassano in quanto si richiede esplicitamente che una **signal** sia sempre collocata come ultima istruzione eseguita in ogni entry-procedure

Diverse Semantiche **wait** e **signal**: Vantaggi & Svantaggi

- *signal_and_continue*:
  - Minor numero di context-switch
  - Possibilità di invalidare la condizione segnalata prima di terminare la entry-procedure e rilasciare il monitor
- *signal_and_wait*:
  - Maggior numero di context-switch
  - Migliore reattività agli eventi segnalati ed impossibilità di invalidarli prima di rilasciare il monitor
- *signal_and_return*:
  - Impone un vincolo sulla collocazione delle **signal** il cui soddisfacimento ad ogni modo risulta naturale:
    - Le segnalazioni si fanno dopo un cambio di stato, che si ottiene alla fine dell'esec. di una entry-procedure
  - Le due semantiche di sopra collassano
  - Risulta semplificata l'implementazione tramite semafori

Implementazione dei Semafori con **wait** e **signal** (1)

- Si supponga di voler implementare le primitive **P**(S) e **V**(S) facendo uso dei monitor con le primitive wait e signal
- La condizione su cui sincronizzarsi è S>0
  - gli associamo una variabile condizione S_POSITIVO
  - la coda associata alla variabile condizione svolgerà, per l'occasione, il ruolo della coda associata al semaforo
- Basterà invocare la **P**(S) e **V**(S) rispettivamente con S.**P** e S.**V**
- L'esempio fornisce la prova che con i monitor è possibile risolvere qualsiasi problema di sincronizzazione

Implementazione dei Semafori con **wait** e **signal** (2)

```
type semaforo = monitor
   var          S: 0..LAST;
                S_POSITIVO: condition; //variabile condizione per S>0
   entry procedure P begin
                while (S=0)  do S_POSITIVO.wait; end
                S ← S - 1;
   end
   entry procedure V begin
                S ← S + 1;
                S_POSITIVO.signal;
   end
   begin
                S ← 0;
   end
end monitor;
```

Il Concetto di *Sezione Critica* rivisitato

Per gestire problemi di cooperazione, bisogna rivisitare il concetto di sezione critica

- In precedenza: una seq. di istruzioni eseguita da f.d.e. per accedere ad una risorsa $\mathcal{R}$ (seriale) con la garanzia di possederla esclusivamente e senza perderne il possesso durante tutta la durata della sezione critica
- D'ora in poi meglio precisare: durante la sezione critica è possibile che il f.d.e. perda temporaneamente l'uso della risorsa purché il flusso stesso rimanga sospeso durante il rilascio (altrimenti verrebbe meno la *cond. di mutua esclusione*)

Legame tra Problemi di Competizione e di Cooperazione

- Ogni problema di cooperazione *nasconde* un sottostante problema di cooperazione: per sincronizzarsi su un *evento* inerente una risorsa condivisa è necessario possederla
- sia per segnalare l'evento
  - bisogna valutare una condizione sulla risorsa per fare la segnalazione
- sia per reagire alla segnalazione
  - bisogna valutare una condizione sulla risorsa per accertare che l'evento notificato sia effettivamente accaduto
- Alcuni dettagli delle API testimoniano questo legame indissolubile >>

Produttori / Consumatori con i Monitor (Mailbox)

- Le operazioni da implementare sono:
  - **DEPOSITA(**M**:** *messaggio***)** per i produttori
  - **PRELEVA(ref** R**:** *messaggio***)** per i consumatori
- Gli eventi su cui sincronizzarsi sono
  - buffer non pieno (variabile condizione NON_PIENO)
  - buffer non vuoto (variabile condizione NON_VUOTO)
- Rispetto alla soluzione con i soli semafori, l'accesso esclusivo agli indici risulta inutile perché l'uso esclusivo del buffer è garantito dalla semantica dei monitor

type *mailbox*

```
type mailbox(messaggio) = monitor
    sia   N: …;
    type indice=0..N-1;
    var   BUFFER: array[indice] of messaggio;
          T,D: indice;
          NON_PIENO,NON_VUOTO: condition;
          K: integer; //numero messaggi presenti
```

```
entry procedure
        DEPOSITA(M: messaggio)
begin
  while (K=N) NON_PIENO.wait;
  BUFFER[D] ← M;
  D ← (D + 1) mod N;
  K ← K + 1;
  NON_VUOTO.signal;
end
```

```
entry procedure
        PRELEVA(ref M1: messaggio)
begin
  while (K=0) NON_VUOTO.wait;
  M1 ← BUFFER[T];
  T ← (T + 1) mod N;
  K ← K - 1;
  NON_PIENO.signal;
end
```

```
    begin
        K ← 0; T ← 0; D ← 0;
    end
end monitor;
```

Commenti alla mailbox con Monitor

- Rispetto alla soluzione con i semafori, la soluzione basata sui monitor è significativamente più semplice ed elegante
- Tuttavia:
  - impone limitazioni nel parallelismo non strettamente necessarie:
    - Un produttore ed un consumatore non possono operare concorrentemente neanche se il buffer non è ne vuoto ne pieno
    - Due produttori non possono depositare contemporaneamente in due posizioni diverse del buffer
    - Due consumatori non possono prelevare contemporaneamente da due posizioni diverse del buffer
  - vengono emesse
    - **signal** dai produttori senza che ci siano consumatori in attesa di consumare
    - **signal** dai consumatori senza che ci siano produttori in attesa di produrre

Esercizio del Barbiere Dormiente (1)

Esercizio: L'uso delle **wait** e **signal** nell'implementazione della mailbox vista non è ottimale. Si può infatti obiettare che non è necessario che ogni produttore emetta una **signal** se non c'è alcun consumatore che lo attende. Analogamente, non è necessario che un consumatore emetta ogni volta una **signal** se non c'è alcun produttore in attesa di essere risvegliato.

Per ovviare a questi inconvenienti Dijkstra ha proposto una versione, a cui ha attribuito il nome "sleeping barber" (barbiere dormiente).

(...continua...)

Esercizio del Barbiere Dormiente (2)

(continua dalla slide precedente)

Essa prevede che la variabile K possa assumere qualsiasi valore intero con il seguente significato (valido prima dell'inizio e dopo la fine delle procedure DEPOSITA e PRELEVA):

- $K<0$: il BUFFER è vuoto e $-K$ consumatori sono in attesa di un messaggio
- $0 \leq K \leq N$: vi sono K messaggi nel BUFFER e non esistono f.d.e. in attesa (né produttori né consumatori)
- $K>N$: il BUFFER è pieno e vi sono $K-N$ produttori in attesa di inserire

Soluzione al Barbiere Dormiente

```
entry procedure
        DEPOSITA(M: messaggio)
begin
      if (K<0) then begin
        K ← K + 1;
        D ← (D + 1) mod N;
        BUFFER[D] ← M;
        NON_VUOTO.signal;
        end
  else if (K<N) then begin
        K ← K + 1;
        D ← (D + 1) mod N;
        BUFFER[D] ← M;
  end
  else /* K≥N */ begin
        K ← K + 1;
        NON_PIENO.wait;
        D ← (D + 1) mod N;
        BUFFER[D] ← M;
  end
```

```
entry procedure
        PRELEVA(ref M1: messaggio)
begin
      if (K≤0) then begin
        K ← K - 1;
        NON_VUOTO.wait;
        T ← (T + 1) mod N;
        M1 ← BUFFER[T];
        end
  else if (K≤N) then begin
        K ← K - 1;
        T ← (T + 1) mod N;
        M1 ← BUFFER[T];
  end
  else /* K>N */ begin
        K ← K - 1;
        T ← (T + 1) mod N;
        M1 ← BUFFER[T];
        NON_PIENO.signal;
end
```

Implementazione dei Monitor Tramite Semafori (1)

- Si associa un semaforo binario MUTEX ad ogni istanza di monitor
- Il corpo di ciascuna procedura viene racchiusa tra **P**(MUTEX) e **V**(MUTEX)
- Nell'ipotesi semplificativa che le **signal** vengano sempre eseguite come ultima istruzione di una procedura (semantica *signal_and_return*), ad ogni variabile condizione $X_{<cond>}$ si associa:
  - un contatore $C_{<cond>}$ inizializzato a 0 che conta il numero di f.d.e. bloccati sulla <cond>
    - serve ad evitare segnalazioni inutili
  - un semaforo binario $S_{<cond>}$ inizializzato a 0
    - serve per la sincronizzazione sulla variabile condizione

Implementazione dei Monitor Tramite Semafori (2)

- **wait** e **signal** si possono quindi implementare come segue:

$X_{<cond>}$.**wait**: **begin**
    $C_{<cond>} \leftarrow C_{<cond>}+1$;
    V(MUTEX)**;** //libera il monitor
    P($S_{<cond>}$); //attende una $X_{<cond>}$.**signal**
    $C_{<cond>} \leftarrow C_{<cond>}-1$;
end;

$X_{<cod>}$.**signal**: **begin** *(al posto della* V(MUTEX) *finale)*
    if $C_{<cond>}>0$ **then** V($S_{<cond>}$)
    else V(MUTEX)
end;

Monitor: Vantaggi & Svantaggi

- Vantaggi
  - il principale vantaggio è il maggior livello di astrazione che deriva dal loro utilizzo, da cui poi seguono
    - la pulizia concettuale
    - l'incapsulamento delle porzioni di codice concorrente
    - la possibilità di dimostrazioni formali di correttezza
- Svantaggi
  - limitazioni sul parallelismo
    - il meccanismo automatico di mutua esclusione attuata dai monitor talvolta si rileva troppo conservativo, come mostrato nell'esempio della mailbox
  - problema delle chiamate annidate >>
    - l'impl. tramite semafori non permette la nidificazione (diretta od indiretta) delle chiamate sulla medesima istanza di monitor
  - presuppongono la disponibilità di memoria comune

Monitor & POO

- I monitor hanno trovato piena applicazione con i linguaggi che meglio supportano il paradigma di programmazione orientata agli oggetti
  - La tipologia di risorsa comune su cui regolare gli accessi concorrenti è modellata da una *classe*
  - La risorsa è un oggetto istanza di tale classe
  - Le operazioni sulla risorsa corrispondono ai *metodi* (pubblici) della classe

Esercizi

Esercizio: Risolvere il problema dei 5 filosofi usando i monitor con le seguenti operazioni PRENDI(i), RILASCIA_DESTRA(i), RILASCIA_SINISTRA(i)

Esercizio: Usare i monitor per risolvere il seguente problema di competizione:

<<N f.d.e. $P_1,\ldots,P_N$ condividono una risorsa seriale R. All'atto della richiesta ciascun f.d.e. specifica una priorità *p* (0=max priorità) che gli consente di avere la precedenza su altri f.d.e. aventi priorità minore>>.

Esercizi

Esercizio: Utilizzando come riferimento la possibile implementazione del costrutto di Monitor con semafori, ideare una tecnica di implementazione delle regioni critiche condizionali tramite semafori.

*Suggerimento: In effetti l'unica sostanziale differenza tra i due costrutti è la mancanza di un nome esplicito per le condizioni delle regioni critiche; vedere una regione critica condizionale come un monitor con semantica signal_and_return ed un'unica var. cond. CAMBIO_STATO che sarà "segnalata" alla fine del corpo di ogni clausola **when** ed "attesa" al suo inizio; quindi riusare l'implementazione con semafori di un simile monitor.*