Corso di Linguaggi di Programmazione

Lezione 21 e 22

Alberto Ceselli
alberto.ceselli@unimi.it

Dipartimento di Informatica
Università degli Studi di Milano

21 Maggio 2013

Concorrenza a livello di sottoprogrammi

Coroutines

- Il meccanismo di controllo dei sottoprogrammi è di tipo *master–slave*
- Modello **simmetrico di controllo:** ogni sottoprogramma può
  - interrompere la propria esecuzione, chiamando un altro sottoprogramma
  - *riprendere* la propria esecuzione, su richiesta di un altro sottoprogramma
  - nel secondo caso, il suo stato è preservato nel periodo di non-esecuzione
- Esempi di applicazioni: simulazioni (es gioco delle carte), applicazioni tipo client–server.

Coroutines

```
int i = 0; // variabile globale

unit client {
   int stop_value = ...;
   ...
   while (i != stop_value) {
      ...
      resume give_me_next;
      ...
}

unit give_me_next {
   int step() {
   ...
   };
   ...
   for ( ; ; ) {
      i += step();
      resume client;
      ...
}

main { resume client; }
```

Coroutines

- Questo tipo di flusso di controllo è detto **quasi-concorrente**.
- Le istruzioni `resume` sono un esempio di istruzione di **sincronizzazione**.
- Esistono linguaggi con istruzioni di questo tipo, es. SIMULA e MODULA 2.

Coroutines

Alcuni dettagli sull'implementazione:

- Ogni sottoprogramma deve essere in grado di accedere alle variabili globali
- Ogni sottoprogramma conserva *sempre* il suo frame nello stack (anche quando non è in esecuzione)
- Ad ogni `resume`, il puntatore all'istruzione corrente viene salvato nel frame del chiamante
- Se si vuole rendere possibile la chiamata a sottoprogrammi all'interno delle coroutines, è necessario avere frame dinamici.

Guardie

Guardia: costrutto che garantisce che una certa porzione di codice sia eseguita solo quando una certa condizione è verificata

- Può essere utilizzata per consentire l'accesso ad una porzione di codice ad un solo task per volta.

Semafori

Un *semaforo* (Dijkstra '65) è un'implementazione di una guardia

- **Semaforo:** struttura dati che consiste in
  - Un intero (`counter`)
  - Una coda di *descrittori di task*.
- **Descrittore di task:** struttura dati che memorizza tutte le informazioni relative allo stato di esecuzione di un task.

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)
- Problema: come implemento effettivamente P() e V()?

Semafori

Su un semaforo possono essere effettuate due operazioni:

- richiesta di accesso P() : se `counter > 0`, decrementa `counter` (e consenti l'accesso), altrimenti poni il task in stato waiting
- notifica di rilascio V() : se la coda dei descrittori è vuota, incrementa `counter`, altrimenti risveglia uno dei task nella coda dei descrittori (poni in coda ready)
- Problema: come implemento effettivamente P() e V()?
- Soluzione: ho bisogno di *istruzioni atomiche* di tipo *test–and–set* (hardware o sistema operativo).

Semafori

Esempio: accesso mutuamente esclusivo ad una variabile condivisa utilizzando semafori
Esempio: cooperation synchronization su un buffer condiviso utilizzando semafori (pag 506 Sebesta)
Esempio: competition synchronization su un buffer condiviso utilizzando semafori (pag 507 Sebesta)

Valutazione

- semplice, potente e flessibile
- basso livello (un analogo del "goto" per programmi concorrenti)
- *unsafe* sia per cooperation che per competition
- non c'è modo di controllarne staticamente la correttezza (dipende interamente dalla semantica del programma)

Esercizi:

- Implementare una simulazione della Cena dei Filosofi utilizzando:
  - semafori
- Rendere “thread safe” il buffer FIFO, esercizio delle lezioni precedenti

Monitor

- idea: sfruttare i concetti di *data abstraction* anche in questo contesto
- *incapsulare* strutture dati (o ADT) condivisi.
- simili strutture ("ADT sincronizzati") vengono detti **monitor**.
- in un monitor ci può essere **al più** un task attivo (ready o running)

Concurrent Pascal

Concurrent Pascal:

- task $\rightarrow$ `process` (che è un *tipo*)

```
type process_name = process (formal parameters)
   -- local declarations --
   -- process body --
end
```

- una variabile di tipo `process` può essere allocata staticamente o dinamicamente
- la dichiarazione crea semplicemente il codice, l'allocazione avviene tramite l'istruzione `init`

```
init process_variable_name (actual parameters)
```

ed il task diventa ready.

Concurrent Pascal

- monitor gestiti in modo simile ad ADT

```
type monitor_name = monitor (parametri formali)
   -- dichiarazione variabili condivise --
   -- definizione procedure ‘‘locali’’ (private) --
   -- definizione procedure ‘‘esportate’’ (public) --
   -- inizializzazione --
end
```

- procedure “esportate” sono identiche a quelle locali, ma marcate con la parola chiave `entry`

Synchronization

- Competition synchronization: *safe* ed automaticamente garantita dal monitor.
- Cooperation synchronization: deve essere gestita con costrutti a parte.
- Concurrent Pascal ha un tipo queue, che ammette due operazioni
  - `delay(queue)`: mette il chiamante in attesa sulla coda specificata e libera il monitor
  - `continue(queue)`: disconnette il chiamante dal monitor; se la coda specificata è non vuota, riattiva un processo della coda specificata (altrimenti non ha effetto)
- N.B. `delay()` e `continue()` hanno un comportamento *diverso* da P() e V().

Valutazione

- Meccanismo più rigido, ma più sicuro
- (meno parallelismo)
- Gestione safe di competition synchronization
- Stessi problemi di P() e V() per cooperation synchronization

Esercizio:

Implementare un buffer condiviso, utilizzando monitor e garantendo cooperation e competition synchronization (pseudo-codice – concurrent_pascal –)

- codice monitor ("server")
- codice produttore
- codice consumatore