

Programmazione concorrente

Non un diverso paradigma di programmazione, ma una feature aggiuntiva a paradigmi preesistenti

- Programmazione concorrente all'interno di
 - linguaggi imperativi, ad oggetti,
 - dichiarativi logici, funzionali
(avvantaggiati da meccanismo di valutazione non sequenziale
mancanza di stato)
 - Erlang
 - Haskell
- Argomento vasto, implementato in molti modi diversi
tenteremo un catalogazione dei vari aspetti della programmazione concorrente

- sfruttare l'evoluzione dell'hardware:
 - negli ultimi anni i miglioramenti della prestazioni ottenuti principalmente con aumento parallelismo (numero dei core)
- alcuni problemi si descrivono meglio in maniera concorrente

divido un compito in sottocompiti da svolgere in parallelo e in maniera indipendente

 - esempio: un programma browser
 - un thread per ogni parte della pagina da visualizzare
 - le immagini non bloccano la visualizzazione del testo
- gestire hardware distribuito:
 - applicazione in esecuzione su più calcolatori collegati via web
 - sistema di controllo di un'auto, un impianto industriale

Concorrenza fisica, a livello di macchina fisica:

- esecuzione simultanea di istruzioni
- diversi modi, gradi di granularità:
 - pipeline
 - superscalari (parallelismo a livello di istruzione)
 - istruzioni vettoriali
 - multicore (parallelismo a livello di processi)
 - GPU (computazione vettoriale (SIMD))
 - multicomputer
 - reti di calcolatori

A livello di linguaggi di programmazione

- programmazione parallela
 - si cerca di parallelizzare l'esecuzione di un singolo problema
calcolo scientifico, algoritmi di simulazione
 - algoritmi paralleli
- programmazione multithreaded:
 - più thread o processi attivi contemporaneamente e che girano su
una stessa macchina fisica
 - Es: thread in Java
 - Modello della memoria:
 - a memoria condivisa
 - a scambio di messaggi
- programmazione distribuita:
 - programmi concorrenti, eseguiti su macchine separate
 - es: multicomputer con memoria distribuita oppure
reti di calcolatori con varie architetture e topologie
 - non si assume la memoria condivisa

esempi di programmazione distribuita, con una precisa struttura

- SOC (Service Oriented Computing): paradigma di programmazione basata sulla composizione di componenti (servizi) che interagiscono
 - servizi: funzionalità base messe a disposizione, descritti in un catalogo
 - consumatori di servizi, posso essere utenti o altri servizi
 - network
 - comunicazione mediante scambio messaggi, linguaggio specifico
 - debolmente accoppiati
 - mancanza di stato
 - interoperabilità: servizi possono essere implementati con sistemi diversi

- stesse idee dei SOC ma meno strutturate, con una granularità molto più fine
 - anche servizi elementari (spazio disco, CPU ecc.)
- insieme di tecnologie (infrastrutture, modelli di comunicazione ecc.) che permettono lo sviluppo di applicazioni distribuite su web
- accesso on demand, semplice, a risorse configurabili distribuite sulla rete
- problemi di sicurezza

Separazione tra i vari livelli di parallelismo

Parallelismo logico e fisico non corrispondono

esempi:

- pipeline, superscalari: parallelismo sono fisico, invisibile a linguaggio di programmazione
- un programma multithread può essere eseguito in vari modi casi possibili:
 - ogni thread logico eseguito su core distinto
 - tutti i thread eseguiti su di un unico core (interleaving)
 - molti core ciascuno con molti thread in esecuzione
 - ...
- distinguibili solo in termini di prestazione, stesso insieme di possibili risultati

- parallelismo attraverso chiamate di libreria:
 - uso un linguaggio sequenziale,
 - lancio threads con chiamate a funzioni di libreria
funzione che riceve come argomento il codice da eseguire in parallelo
 - es C e POSIX thread
- estensioni, supportate dal compilatore, di un linguaggio sequenziale
 - Fortran e OpenPM, con direttive al compilatore (pragma)
- linguaggi di programmazione con costrutti per la concorrenza

Molti modi diversi di strutturare la programmazione concorrente.

In tutti esistono più thread (esecuzioni attive)
si distingue su come avviene:

- la comunicazione: i thread si scambiano informazioni
- la sincronizzazione: i thread regolano la velocità relativa
 - attesa attiva
 - scheduler (rilascio della CPU)

Altri aspetti:

- come avviene la creazione di nuovi thread

- Thread (del controllo): l'esecuzione di una sequenza di comandi, una specifica computazione, con spazio di indirizzamento comune
- Processo (pesante): generico insieme di istruzioni in esecuzione, con il proprio spazio di indirizzamento.
un processo può essere costituito da più thread diversi
- Terminologia spesso non univoca (thread, processi leggeri, task..)

Creazione di nuovi thread:

- `fork/join`: primitive per lanciare un nuovo processo e concludere l'esecuzione
- `co-begin`:

```
co-begin
  stm_1
  ...
  stm_n
end
```

- `parallel loops`: vengono definiti cicli `for` con indicazione al compilatore (`#pragma`) di poter eseguire in parallelo i diversi loop
- `early replay`: una procedura restituisce il controllo al chiamante prima della sua conclusione, chiamato e chiamante in esecuzione contemporanea
- ...

Memoria condivisa:

- accesso in lettura/scrittura ad una zona comune di memoria
- presuppone l'esistenza di una memoria comune (almeno concettualmente)
 - possibili differenze tra livello logico e fisico
memoria comune simulata a livello di sistema operativo, librerie

Scambio di messaggi:

- uso di primitive esplicite di invio (send) e ricezione (receive) di messaggi
- un'opportuna struttura di comunicazione (canale) deve fornire un percorso logico fra mittente (send) e destinatario (receive)

Blackboard: modello intermedio fra i precedenti.

- i processi condividono una stessa zona di memoria (blackboard).
- comunicazione mediante send e receive al/dal blackboard

Meccanismi di sincronizzazione

- Sono i meccanismi che permettono di controllare l'ordine relativo delle varie attività dei processi.
- Essenziali per la correttezza, [race condition](#).
- Esempio: due processi devono incrementare una variabile condivisa

processo P1 in marcato con +, processo P2 non marcato.

```
+ leggi x in reg1;  
leggi x in reg2;  
incrementa reg2;  
scrivi reg2 in x;  
+ incrementa reg1;  
+ scrivi reg1 in x;  
x è incrementata solo di 1
```

Race condition non necessariamente negative

- programmi concorrenti intrinsecamente non-deterministici
 - diverso ordine di esecuzioni possibili
 - spesso le varie alternative sono accettabili
 - se qualcuna va esclusa, i meccanismi di sincronizzazione permettono di farlo

Con memoria condivisa:

mutua esclusione dati, regioni critiche del codice, non sono accessibili contemporaneamente a più processi

sincronizzazione su condizione si sospende l'esecuzione di un processo fino al verificarsi di una opportuna condizione

Con scambio di messaggi

- di solito impliciti usando le primitive di **send** e **receive** posso ricevere un messaggio solo dopo il suo invio

Come implementare la sincronizzazione:

- attesa attiva (busy waiting o spinning)
 - ha senso solo su multiprocessori
- sincronizzazione basata sullo scheduler (auto-)sospensione del thread

Mutua esclusione mediante attesa attiva: lock

- Utile un'operazione **test-and-set(B)** atomica, non interrompibile altrimenti sono necessari algoritmi più complessi
 - in programmazione concorrente è importante definire le **azioni atomiche**

Struttura processo:

```
process Pi {  
    sezione non critica;  
    acquisisci_lock(B);  
    sezione critica;  
    rilascia_lock(B);  
    sezione non critica;  
}
```

Mutua esclusione mediante attesa attiva: lock

- Acquisizione lock:
 - lock variabile booleana
 - true bloccata, false libera
 - si sfrutta un'istruzione read and write atomica
 - test and set
 - possibile anche senza test and set
 - algoritmi più complessi, diverse alternative

```
void acquisisci_lock(ref B: bool) {  
    while test_and_set(B) do skip;  
}
```

- Rilascio lock

```
void rilascia_lock(ref B: bool) {  
    B = false;  
}
```

Algoritmi alternativi

- Algoritmi più complessi che non usano `test_and_set`
- Problemi di **fairness**
 - algoritmo precedente non assicura la fairness
 - nel caso di più richieste, l'assegnazione fatta arbitrariamente
 - un processo potrebbe dover attendere all'infinito
 - algoritmi più complessi assicurano la fairness
- Problemi per l'uso della memoria e dei bus nei sistemi multiprocessori:
 - `test_and_set` operazione costa per il bus, blocca l'uso della memoria
 - la versione **test and test_and_set(B)** migliora le prestazioni

```
void acquisisci_lock(ref B: bool) {  
    while B do skip;  
    while test_and_set(B) do skip;  
}
```

Meccanismo di coordinazione tra processi che non implementa mutua esclusione

- processo in attesa che una condizione globale non venga soddisfatta.

- Esempio:

un compito comune suddiviso tra più processi

ogni processo attende che tutti gli altri terminato prima di proseguire allo stadio successivo

possibile implementazione:

- contatore condiviso
- conta il numero di processi che devono terminare
- uso sezioni critiche per aggiornare il contatore

L'algoritmo di mutua esclusione presentato prima si basa sull'attesa attiva

- il processo bloccato continua ad usare tempo di CPU
 - non viene sospeso esplicitamente
 - conveniente per attese brevi, ma inefficiente su attese lunghe
 - poco sensato in sistemi singolo processore

- Il processo che deve essere posto in attesa, rilascia la CPU
- I processi sospesi sono gestiti dallo scheduler dei processi (nucleo s.o.)

Possibili costrutti per la sincronizzazione con scheduler

- semafori
- monitor

- Semaforo: tipo di dato con
 - l'insieme dei valori, costituito dai numeri interi ≥ 0
 - due operazioni atomiche chiamate P e V
- sem s
- P(s): accesso semaforo
 - se $s > 0$ s decrementata (operazione atomica)
 - se $s = 0$ processo sospeso
- V(s): rilascio semaforo
 - s incrementata (op. atomica)
 - si attiva lo sblocco di eventuali processi in attesa
 - fairness
 - FIFO

Esempio di uso dei semafori: filosofi a cena

Problema di sincronizzazione:

- proposto da Dijkstra - Hoare
- illustra la possibilità di **deadlock**
 - tutti i processi in attesa di un evento
 - sistema bloccato



Esempio: filosofi a cena

Soluzione:

- le forchette in posizione dispari richieste per prime
- un volta ottenuta una forchetta dispari, un filosofo può essere bloccato solo da un filosofo mangiante
- problema: se tutti fanno richiesta, quale percentuale di filosofi riesce sicuramente a mangiare?

Modelizzazione:

- filosofi – processi
- forchette sono risorse condivise
 - mutua esclusione gestita tramite semafori

Esempio: filosofi a cena

```
sem forchette[5];
for (i=1, i<=5, i+=1) {forchette[i] = 1};

process Filosofo[i]{          % i = 1, 3, 5
    while true {
        pensa;
        P(forchette[i]);    // acquisisce forchetta di destra
        P(forchette[i-1])  // acquisisce forchetta di sinistra
        mangia;
        V(forchette[i]);    // rilascia forchetta di destra
        V(forchette[i-1])  // rilascia forchetta di sinistra
    }
}
```

Esempio: filosofi a cena

```
process Filosofo[i]{           % i = 2 , 4
    while true {
        pensa;
        P(forchette[i-1]); // acquisisce forchetta di sinistra
        P(forchette[i])    // acquisisce forchetta di destra
        mangia;
        V(forchette[i-1]); // rilascia forchetta di sinistra
        V(forchette[i])    // rilascia forchetta di destra
    }
}
```

```
Filosofo[1] || Filosofo[2] || Filosofo[3] ||
Filosofo[4] || Filosofo[5]
```

- Più astratti e strutturati dei semafori
 - semafori:
 - semplici contatori condivisi
 - la gestione corretta di risorse condivise affidata al programmatore
- oggetto (classe, modulo) condiviso tra più thread,
 - contiene al suo interno le risorse
 - procedure pubbliche per l'uso di risorse condivise, non si accede alle risorse direttamente
 - variabili permanenti (realizzano lo stato);
 - procedure
 - `init` procedure di inizializzazione
- monitor: oggetti thread-safe
 - la mutua esclusione è garantita, un solo thread alla volta può avere accesso ai metodi
 - non necessariamente vero per oggetti generici
- terminologia
 - monitor **componenti passivi** (eseguiti se invocati)
 - thread **componenti attivi**

Variabili condizionali

- Componente aggiuntive al monitor, rappresentano lo disponibilità, o meno, di una risorsa condivisa
- permetto ad un thread di:
 - entrare nel monitor
 - esaminare lo stato della risorsa
 - e se necessario
 - mettersi in attesa che lo stato cambi
 - liberando l'accesso al monitor
- più thread all'interno del monitor
 - al più uno attivo
 - gli altri in attesa
- esempio:
 - monitor gestisce una coda di lavoro
 - thread consumer: prelevano elementi
 - thread producer: inseriscono elementi
 - sospesi in caso di coda piena o vuota

Costrutti per variabili condizionali

- dichiarazione

`cond namevar`

vista come coda dei thread in attesa

- interrogazione

`empty(namevar)`

- sospensione

`wait(namevar)`

nel caso risorsa non disponibile, thread in attesa,
altri thread possono 'entrare' nel monitor

- rilascio, risveglia un processo in attesa
`signal(namevar)`

Due alternative:

- segnala e continua
- segnala e sospendi

Comunicazione mediante scambio di messaggi

Esistono due tipi di comunicazione,
con meccanismi di sincronizzazione diversi:

- **sincrona**: invio e ricezione di un messaggio allo stesso tempo
 - send e receive (concettualmente) allo stesso tempo
 - send (e receive) bloccante
 - più semplice da implementare
 - coda canale di lunghezza uno
 - errori gestiti immediatamente
 - più difficile evitare i deadlock, (deadlock prone)
- **asincrona**: invio e ricezione di un messaggio in momenti diversi
 - send e receive i momenti diversi
 - send non bloccante,
 - necessario implementare una coda di messaggi inviati
 - receive bloccante
 - l'invio di messaggi risveglia (potenzialmente) processi

Possibile simulare la comunicazione sincrona usando quella asincrona

- dopo aver spedito il messaggio si attende (receive) un acknowledgement da parte del ricevente

E viceversa

- si crea un **processo buffer intermedio**,
 - riceve messaggi dal mittente, li inserisce in un coda
 - contemporaneamente, cerca di spedirli al ricevente

- **nomi espliciti di processi**

- calcolo CSP, processi memoria separata
- primitive di comunicazione
 - Proc1 ! espressione invio, si valuta l'espressione per ottenere il dato
 - Proc2 ? variabile ricezione, si assegna alla variabile il dato

- **porte**

- meccanismo usato nelle reti protocollo TCP
 - si definiscono degli identificatore di porta
 - ogni porta associata ad un tipo di comunicazione, servizio
 - si specifica (indirizzo IP) - numero porta (numero di porta)
 - stessa connessione fisica - più connessioni simboliche
- usato anche in linguaggi di programmazione,

Esempio in ADA

- definite nelle interfacce di un processo task

```
task TypeTask is
    entry portaIn (dato : in integer);
    entry portaOut (dato : out integer);
end TypeTask
```

- usate all'interno del codice

```
task body TypeTask is
    ...
    accept portaIn(dato : in integer) do ... end portaIn
    ...
    accept portaOut(dato : out integer) do ... end portaOut
    ...
end TypeTask
```

- posso definire un processo Pr di tipo TypeTask
- inviarli dati con il comando Pr.portaIn(mioDato)

- canali

- simili alle porte ma più generale astratto
- concettualmente qualsiasi meccanismo di comunicazione tra processi
 - non si fanno ipotesi sul numero di processi coinvolti, o sulla direzione dei dati

Meccanismi di uso:

Dichiarazione

```
channel nomeCh (Type)
```

- processi che condividono un canale possono comunicare tra loro
- nel π -calcolo il nome di un canale può essere scambiato tra processi
 - cambia la geometria della comunicazione

Invio dati

```
send NomeCanale(dati)
```

- aggiunge un messaggio alla coda del canale
- bloccante solo nella comunicazione sincrona

Ricezione

```
receive NomeCanale(var)
```

- preleva un messaggio dalla coda del canale
- sempre bloccante (nel caso di coda vuota)

Verso:

- monodirezionali,
- bidirezionali

Numero di connessioni

- link (un destinatario - un mittente)
- input port (più mittenti - un destinatario)
- mailbox (più mittenti - più destinatari)

Sincronizzazione

- sincroni
- asincroni

Esempio (interazione client server)

```
channel richiesta (int client, char dati);
channel risposta1 (char ris);
channel risposta2 (char ris);

process Client1{
    char valori;
    char risultati;
    .... // Definizione dei valori
    send richiesta(1, valori);
    receive risposta1(risultati);
    .... // Uso dei risultati
}

process Client2{
    char valori;
    char risultati;
    .... // Definizione dei valori
    send richiesta(2, valori);
```


Esempio (interazione client server)

```
process Server{
  int cliente;
  char valori;
  char risultati;
  .... // Inizializzazione
  while true {
    receive richiesta (cliente, valori);
    if cliente = 1 then
      { ... // Elaborazione valori
        send risposta1(risultati);
      }
    if cliente = 2 then
      { ... // Elaborazione valori
        send risposta2(risultati);
      }
  }
}
```

Alternative:

- passare il nome di un canale
 - al posto dell'identificativo di processo
 - permette comunicazione privata: il client invia un nome privato di canale su cui ricevere la risposta
 - permette al server di gestire un numero arbitrario di processi, in modo semplice
- receive bloccante, per evitare i blocchi:
 - primitive per controllare lo stato della coda di input

Ulteriore meccanismo di comunicazione:

- sistemi distribuiti,
- client-server

Si invoca su una macchina remota una procedura:

- comunicazione: passaggio dei parametri, risultato
- meccanismi per rendere pubblici i nomi, parametri delle procedure remote
 - tipicamente moduli
 - `call NomeModulo.NomeProc (parametriAttuali)`
- nella macchina remota meccanismi per attivare un processo all'arrivo della richiesta
- disponibile in Java

- Tramite **stub**
- Stub: processo locale che implementa la comunicazione remota
comunicazione nascosta, confinata negli stub
 - viene chiamato un stub locale
 - crea messaggio per un stub su macchina remota (procedura dati)
 - stub remoto, invoca procedura, restituisce risultati

Terminologia: 'stub' indica anche una dichiarazione di procedura incompleta

- nel rendez-vous esiste già un processo attivo sul destinatario che gestisce la risposta
 - disponibile in Ada, esempio 'port' precedente

```
task body TypeTask is
  ...
  accept portaIn(dato : in integer) do ... end portaInf
  ...
  accept portaOut(dato : out integer) do ... end portaInf
  ...
end TypeTask
```

- introduco comandi imperativi non deterministici
- oltre al non-determinismo indotto dalla concorrenza
- Dijkstra guarded command

```
guardia -> comando
```

```
[]
```

```
guardia -> comando
```

```
[]
```

```
...
```

```
[]
```

```
guardia -> comando
```

- evito di imporre scelte

$x \leq y \rightarrow \max = y$

[]

$y \leq x \rightarrow \max = x$

- guardia: condizione booleana + test sullo stato di un canale
- utile per gestire la concorrenza
 - evito di dover definire uno scheduler

Esempio: server con due client

- server che deve gestire richieste di lettura e scrittura provenienti da due client
- soluzione deterministica => ordinamento nella gestione dei messaggi => possibile deadlock

```
channel lettura (int dati);
channel scrittura (int dati);

process ClientLettura{
    int risultati;
    ....
    send lettura(risultati);
    ....           // Uso di risultati
}
```


Esempio

```
process ClientScrittura{
```

```
    int valori;
```

```
    ....           // Definizione di valori
```

```
    send scrittura(valori);
```

```
    ....           }
```

```
process Server{
```

```
    int v;
```

```
    int r;
```

```
    while true{
```

```
        receive lettura(v) -> servi richiesta lettura
```

```
        []
```

```
        receive scrittura (r) -> servi richiesta scrittura
```

```
    } }
```

- scelta pseudo non-deterministica tra le guardie abilitate,
- problemi di fairness

Meccanismo astratto per la definizione di thread - sottoprocessi

P || Q || R

- indica l'esecuzione parallela di P, Q, R
- visto in precedenza
 - cobegin
 - nell'esempio dei filosofi
- implementazione
 - true concurrency: più processi eseguiti allo stesso istante
 - interleaving: esecuzione alternate delle istruzioni nei vari processi