Concurrency:
esempi vari di programmazione concorrente

Luca Manini *

2014

1 Introduzione

Appunti per le lezioni di TPS di quarta, per la parte di programma che riguarda la concorrenza. Nel corso vedremo alcuni esempi classici ed alcuni esempi più "realistici". Siccome il corso vuole essere anche molto "pratico", per ciascun esempio ci saranno più implementazioni complete di argomenti della linea di comando, logging e testing. Questo file raccoglie la parte più "teorica" del corso.

2 Algoritmi, flusso di esecuzione, concorrenza e parallelismo

Concorrenza e parallelismo sono due termini usati spesso come se fossero quasi sinonimi, ma in realtà indicano due concetti molto diversi, anche se in stretta relazione tra loro.

La concorrenza è un aspetto strutturale (quindi statico) di un problema, o di una soluzione ad un problema, in particolare di una sua soluzione algoritmica che per noi è sinonimo di "programma". La concorrenza è un modo di "decostruire" il problema (e quindi spesso di "costruire" una sua soluzione) ed è anche un metodo per "costruire" dei sistemi mettendo insieme più parti che possono essere eseguite "indipendentemente".

Il parallelismo è un aspetto legato all'esecuzione (quindi dinamico) di un algoritmo ed è legato alla possibilità di avere più esecutori (e quindi più flussi di esecuzione) contemporanei. La concorrenza è un modo di "ottimizzare" l'esecuzione di una soluzione ad un dato problema.

Una soluzione concorrente può essere una "buona soluzione" anche se poi non si può avere il parallelismo ma solo per la sua "intelligenza" o per la sua chiarezza. Se poi si può avere anche il "parallelismo" (di cui la concorrenza è un prerequisito) allora tanto meglio (di solito).

Per illustrare il concetto concorrenza è utile ricordare cos'è di "controllo di flusso" (controllo del flusso di esecuzione).

Nelle parole di Bob Pike (video *Concurrency is not parallelism*) la concorrenza serve a "gestire più cose nello stesso momento" (*dealing with a lot o things at once*) mentre il parallelismo è "fare più cose nello stesso mome" (*doing a lot of things at once*).

2.1 Algoritmo

Un algoritmo è una descrizione di una serie di operazioni che, se eseguite, portano da un certo risultato. Queste operazioni verranno eseguite da un "esecutore" in un certo ordine, ci sarà quindi un "flusso di esecuzione". Tutti linguaggi di programmazione, in un modo o nell'altro, forniscono delle "strutture" per specificare, nel "testo" del programma (che è una struttura lineare e statica), come queste istruzioni devono essere eseguite (non necessariamente nell'ordine in cui compaiono nel testo). Le possibilità sono: sequenza, scelta, ripetizione.

La sequenza non richiede costrutti particolare, semplicemente si tratta di eseguire le varie parti del programma nell'ordine in cui compaiono nel testo.

La scelta (tipicamente espressa dalla struttura `if` nelle sue varianti) permette di specificare uno o più blocchi di istruzioni che non verranno eseguite tutte in sequenza ma che sono in alternativa tra loro. Alcune parti del testo non saranno quindi sempre eseguite.

La ripetizione (tipicamente espressa con le strutture `for` e `while` o altre simili o con la ricorsione) permette di ripetere l'esecuzione di un blocco di istruzioni senza doverle ripetere nel testo.

Queste tre strutture permettono di descrivere un qualsiasi algoritmo e quindi tutti i linguaggi che le contengono sono in un qualche senso "equivalenti" (equipotenti).

A queste tre strutture si aggiunge di solito la possibilità di definire e di eseguire delle "funzioni" (o procedure, dal punto di vista del flusso di esecuzione la differenza non è significativa). Definire una funzione significa in pratica "isolare" un pezzo di codice e di dargli un nome (oltre al fatto di definire e nominare dei parametri, ma ciò non è importante in questo contesto). La funzione può essere poi "chiamata" da un altro punto del codice (e quindi in un qualsiasi momento dell'esecuzione) con l'effetto di deviare il flusso di esecuzione passando ad eseguire il codice della funzione per poi "tornare" al punto di deviazione.

2.2 Flusso di esecuzione

Il flusso di esecuzione è quindi il passaggio dall'esecuzione di una "istruzione" a quello dell'istruzione "successiva" (l'ordine in cui le istruzioni vengono eseguite), seguendo il codice e le regole di "traffico" descritte prima.

Una delle qualità di un codice scritto bene è la facilità con cui dalla lettura del codice si può "immaginare" (seguire nella propria testa) un ipotetico flusso di esecuzione (uno dei tanti possibili a seconda, ad esempio, dei dati forniti al programma o da altre condizioni "esterne").

Nella programmazione "normale" (non concorrente) il flusso di esecuzione è univoco (al netto della variazione dei dati) nel senso che eseguendo più volte il programma

"nella stessa situazione" ottengo non solo lo stesso risultato (se il programma è corretto!) ma l'esecuzione si è ripetuta sempre nello stesso modo, seguendo sempre lo stesso flusso.

Come vedremo ciò non è più vero nella programmazione concorrente/parallela e ciò è un grosso problema in quanto rende molto più difficile sia "immaginare" i vari possibili flussi di esecuzione sia "controllare" il programma eseguendo delle "suite" di test.

2.3 Concorrenza

2.4 Parallelismo

3 Supporto software

La maggior parte dei linguaggi (tra le eccezioni `Erlang` e `go`) non offre un supporto diretto per esprimere l'eventuale struttura concorrente del codice e nemmeno per gestirne l'esecuzione "parallela". Di solito tutto ciò è delegato a librerie che forniscono degli strumenti (strutture dati e funzioni) per gestire i vari esecutori (ticipamente funzioni eseguite in *thread* o processi) e per controllare l'accesso alle risorse condivise.

3.1 Processi

I processi sono le strutture usate dal sistema operativo per gestire l'esecuzione di più programmi mettendo di volta in volta a disposizione di ciascuno le risorse disponibili (CPU, memoria, dispositivi di I/O etc.).

Per facilitare la programmazione e per evitare "interferenze" tra i vari processi, il sistema operativo "isola" i processi l'uno dall'altro in modo molto deciso: ciascun processo ha la propria area di memoria per codice e dati e non può influenzare più di tanto l'esecuzione degli altri processi.

Se per ottenere il parallelismo si punta quindi sul *multiprocessing*, delegando a processi diversi le varie "attività", la condivisione delle informazioni, fondamentale per la collaborazione, risulta piuttosto problematica.

Uno dei vantaggi del *multiprocessing* rispetto al *multithreading* e che una soluzione *multiprocessing* si presta molto di più alla distribuzione delle attività su macchine diverse.

3.2 Thread

Anche i *thread* sono uno strumento per l'esecuzione di programmi ma diversi *thread* possono essere eseguiti all'interno di uno stesso processo potendo quindi condividere varie risorse, tra le quali la memoria. È possibile quindi utilizzare, ad esempio, delle variabili "globali" visibili a tutti i *thread*. In questo caso la comunicazione tra gli esecutori delle varie attività è molto facile! Anche troppo facile in realtà in quanto ci si deve occupare del controllo dell'accesso condiviso ad una stessa risorsa, con i relativi problemi di sincronizzazione.

3.3 Coroutine (*future* e parenti)

Un'altra possibilità, recentemente perseguita da molti linguaggi, è quella di utilizzare dei meccanismi ancora più leggeri dei *thread* chiamati in modo diverso a seconda dei linguaggi e degli "ambienti". Nel passato, molti *framework Python*, specialmente quelli rivolti allo sviluppo di server per protocolli di rete (per esempio i server *web*) hanno sviluppato varie soluzioni sfruttando varie funzionalità del linguaggio tra le quali le *coroutine* e i *generatori*. Nelle versioni più recenti di *Python* tutti i vari meccanismi sono stati "uniformati" ed è stata introdotta una libreria (`asyncio`) dedicata alla gestione asincrona dell'I/O.

4 Meccanismi di base

I due problemi principali, fondamentalmente molto simili, che si devono affrontare nello sviluppo di applicazioni concorrenti sono la sincronizzazione tra i vari processi e il controllo di accesso a risorse condivise. Nel corso degli anni sono state inventate (e reinventate) moltissime soluzioni che si basano su un certo numero di "strumenti" (meccanismi): semafori, *lock*, code, messaggi (XXX). Molti di questi meccanismi sono "equivalenti" nel senso che avendone a disposizione uno (normalmente fornito dal sistema operativo), si possono "costruire" gli altri. Di tutti questi meccanismi "di base", i più facili da capire sono i semafori e i *lock* e quindi ci occuperemo solo di quelli.

C'è da notare che questi meccanismi sono molto difficili da usare in modo corretto per cui vengono usati in realtà solo nella scrittura dei sistemi operativi. Nello sviluppo di applicazioni è molto meglio usare meccanismi di livello più alto come le code e i messaggi. Ciò nonostante fare un po' di esercizio con semafori e *lock* non può che fare bene!

4.1 Semafori

Un semaforo può essere visto come una variabile intera con le seguenti caratteristiche particolari:

1. ha sempre un valore non negativo;
2. non è possibile conoscerne il valore;
3. è possibile incrementarne il valore (di una unità) usando una funzione, chiamata a seconda dei casi (*signal*, *release*, *up*), questa funzione "ritorna" sempre istantaneamente;
4. è possibile "provare" a decrementarne il valore (di una unità) usando una funzione chiamata a seconda dei casi (*wait*, *acquire*, *down*); siccome la variabile non può mai assumere un valore negativo, se la variabile vale già zero queste funzioni "bloccano" l'esecuzione fintanto che un altro *thread* non incrementa la variabile rendendone quindi possibile il decremento.

I semafori sono quindi come dei contatori con la caratteristica che le due funzioni di accesso, che prevedono un controllo e una modifica, sono "atomiche".

4.2 Lock

I *lock* sono molto simili ai semafori con la differenza che hanno solo due valori possibili. Il loro scopo è di controllare che un solo *thread* acceda abbia accesso in un dato momento ad una risorsa.

4.3 Code (*queue*)

4.4 Messaggi

5 Esempi

5.1 basic fork/exa (C e Python)

Semplici esempi di fork:

1. fork di un singolo processo senza attesa del figlio da parte del padre: codice in python e in C e documentazione in pdf.
2. fork di più processi, anche qui senza attesa del figlio: codice in C e documentazione in pdf.
3. fork di più processi, questa volta con attesa dei figli: codice in C e documentazione in pdf.
4. fork di un singolo processo con attesa del figlio e ritardi personalizzabili sui vari processi per vedere la creazione di zombie: codice in Python e in C e documentazione in pdf.

5.2 char-count (C e Python)

Conteggio caratteri di più file implementato in vari modi:

1. in C con processo singolo usando una funzione esterna,
2. in Python con fork di una funzione e comunicazione via pipe;
3. in C con fork di più processi;
4. nello stesso modo ma con le *wait* tutte alla fine;
5. in C con i *thread* usando la libreria *pthread*.

5.3 Dining philosophers

Classico problema del "filosofi a cena". Soluzione in C "à la Tanenbaum" e poi anche in Python. Documentazione qui.

5.4 Sleeping barber

Classico problema del "barbiere che dorme" (*sleeping barber*). Soluzione in C "à la Tanenbaum" e poi anche in Python con uso delle librerie `threading`, `logging` e `getopt`.

5.5 read/write

Una serie di versioni più o meno complicate del problema dei *reader* e *writer* (o *consumer producer*).

Single reader/writer:

1. versione molto semplice in Python con variabili globali a *go-go*;
2. versione più completa con opzioni;
3. versione come la precedente ma molto commentata;
4. interessantissima versione single producer single consumer senza buffer (singola variabile globale) ma con la possibilità di variare i "ritardi" dei due processi e stampare una tabella dei tempi (istanti) di partenza, stop e wait dei due processi.

Multi reader/writer:

1. versione minimalista procedurale;
2. versione procedurale ma con commenti e opzioni;
3. versione OOP;
4. versione procedurale con *queue*:
5. versione OOP con *queue*;
6. versione minimalista in C.

Documentazione:

1. generica sul problema;
2. specifica su come sviluppare le versioni single reader/writer;
3. specifica su come sviluppare le versioni multi reader/writer;

5.6 Primes

Calcolo dei numeri primi da 2 fino ad un certo massimo. In varie versioni:

1. processo singolo (per controllo) in C e in Python;
2. con multiprocessing e *queue* in Python.

5.7 Tickets

Primo esempio (l'altro è `Ice store`) del mitico Jeff Cain di Stanford. Vendita di un insieme di biglietti da parti di più venditori. Il tutto in Python usando il multithreading. Documentazione specifica qui.

5.8 Ice store

Mega esempio di Jeff Cain. Distribuzione di coni gelato con multipli clienti, multipli camerieri, controllo dei coni uno per uno e passaggio finale alla cassa. Il tutto riscritto in Python.

Varie versioni:

1. versione liscia, più vicina possibile all'originale;
2. versione più ricca, ma ancora vicina all'originale;
3. versione più pythonica;

Documentazione specifica qui.

5.9 On-net

Esempio di calcolo distribuito, per la fattorizzazione di numeri interi. Anche questa volta molte versioni diverse:

1. solito esempio di echo server (con client);
2. prima versione single process con le tre funzioni di fattorizzazione (stupida, furba e ricorsiva);
3. versione locale ma multithreading;
4. come sopra ma con code (doc);
5. come sopra ma con *sync manager*;
6. versione client/server con socket un client alla volta;
7. come sopra ma con select e connessioni multiple.

Documentazione qui.