UNIVERSITA' DEGLI STUDI DI PADOVA
INGEGNERIA INFORMATICA

Tesina:

La Concorrenza in Ruby

Laureando:
Federica Moro - 580784IF

Relatore:
Prof. Federico Filira

Data:
28.09.2010

Anno Accademico: 2009-2010

Indice

Introduzione **ix**

1 Introduzione a Ruby **1**
1.1 Breve panoramica su Ruby . . . 2
1.1.1 Ruby è orientato agli oggetti . . . 2
1.1.2 Nessuna Dichiarazione . . . 3
1.1.3 Duck Typing . . . 3
1.1.4 Blocchi ed Iteratori . . . 4
1.1.5 Espressioni ed Operatori . . . 5
1.1.6 Metodi . . . 6
1.1.7 Regular Expression . . . 6
1.1.8 Alcune particolarità di Ruby . . . 6
1.1.9 Versioni di Ruby . . . 7
1.1.10 Cos'è Shoooes! . . . 10

2 Metaprogrammazione **11**
2.1 Oggetti e classi . . . 11
2.2 Singleton . . . 11
2.3 Ereditarietà . . . 13
2.4 Moduli e Mixin . . . 13
2.4.1 Moduli come spazio dei nomi . . . 13
2.4.2 Moduli come mixin . . . 14

3 Multiprogrammazione **17**
3.1 Introduzione alla multiprogrammazione . . . 17
3.2 Il multithreading in Ruby . . . 18
3.2.1 Il main thread . . . 18
3.2.2 Eccezioni . . . 19
3.3 Utilizzo delle variabili . . . 19
3.3.1 Variabili private . . . 19
3.3.2 Thread e variabili locali . . . 20
3.4 Scheduling . . . 21
3.4.1 Le priorità . . . 21
3.4.2 Preemption . . . 21
3.5 La gestione dei thread . . . 21
3.5.1 Stati di interrogazione dei thread . . . 22
3.5.2 Alterazione dello stato: mettere in pausa, risvegliare e uccidere i thread . . . 22
3.6 Liste e Gruppi di thread . . . 23

3.7 Thread Exclusion e Deadlock . . . 23
3.7.1 Deadlock . . . 24
3.8 Queue e SizedQueue . . . 25
3.9 Variabili di condizione e code . . . 25
3.10 I thread e il Garbage Collector . . . 26

4 Meccanismi per la sincronizzazione 29
4.1 Interazione tra processi . . . 29
4.1.1 Mutua esclusione . . . 29
4.1.2 Cooperazione . . . 30
4.2 Thread e dipendenze dalla piattaforma . . . 30
4.3 Global Interpreter Lock . . . 32
4.3.1 Inside Ruby's Interpreter: Threading in Ruby 1.9.1 . . . 33
4.3.2 Parallelismo tra processi . . . 35
4.4 Thread e attività CPU o IO bound . . . 36
4.5 Oltre ai thread . . . 40
4.5.1 Fiber . . . 41
4.5.2 Actors . . . 41
4.5.3 Eventmachine . . . 41

5 Costrutti per la sincronizzazione 43
5.1 Semaforo . . . 43
5.1.1 Lock e Unlock . . . 44
5.1.2 Semaforo binario . . . 45
5.1.3 Semaforo numerico . . . 46
5.1.4 Semaforo privato . . . 46
5.2 Regione critica . . . 47
5.3 Monitor . . . 47
5.3.1 Confronto con il Monitor di Java . . . 48
5.3.2 Monitor di Hoare . . . 49
5.4 Rendez-vous . . . 50

6 Esempi 51
6.1 Problema del Produttore-Consumatore... . . . 51
6.1.1 ... utilizzando un semaforo numerico . . . 51
6.1.2 ... utilizzando una regione critica . . . 51
6.1.3 ... utilizzando un monitor . . . 52
6.2 5 filosofi a cena: Ruby vs Java . . . 52
6.3 Un server Internet in 30 righe di codice . . . 55
6.4 Implementare una coda distribuita . . . 56

Appendice: Shoooes! 59
A.1 Concetti fondamentali . . . 59
A.2 Semplice applicazione per visualizzare .txt . . . 64

Bibliografia ragionata 66

Elenco delle figure

1.1 Fumetto tratto da "_why's (poignant) guide to Ruby". 2
1.2 Vignetta umoristica sulla definizione statica delle variabili. 4
1.3 Loghi delle differenti versioni di Ruby 7
1.4 Logo della piattaforma MRI 7
1.5 Logo di JRuby 8
1.6 Logo di MacRuby 8
1.7 Logo di MagLev 9
1.8 Logo di IronRuby 9
1.9 Logo di Rubinius in 2D 10
1.10 Alcuni esempi di interfaccia grafica realizzabile con Shoes. 10

2.1 Modello dell'oggetto per una classe semplice 12
2.2 Modello dell'oggetto per una classe Singleton 12
2.3 Come vengono inclusi i moduli 14

4.1 Cooperazione tra produttore e consumatore con risorse limitate 30
4.2 Confronto tra Ruby 1.8, Ruby 1.9 e JRuby 32

5.1 Schema per la realizzazione delle operazioni di wait e signal su un semaforo 43
5.2 Acquisizione e rilascio del *lock* associato ad un oggetto in Java. 49

6.1 I dieci punti essenziali di Shoes. Tratto da "Nobodyknowsshoes" 59
6.2 Gli stack assomigliano al domino 60
6.3 I flow assomigliano a una scatola di fiammifieri 60
6.4 Stack e flow permettono di ottenere diverse impaginazioni. 61
6.5 Esempio di semplice applicazione. 61
6.6 Esempio di come appare in una finestra una casella per l'immissione di testo. 62
6.7 Clear spiegato da whytheluckystiff. 63

Elenco delle tabelle

1.1 Tabella riassuntiva dei prefissi che denotano la tipologia di variabile 3

3.1 Stato dei thread e valori restituiti 22

4.1 Risultato dei test effettuati utilizzando thread per un'attività CPU-bound . 37
4.2 Risultati dei test effettuati utilizzando processi per un'attività CPU-bound 38
4.3 Risultati dei test effettuati utlizzando thread per attività IO-bound 40

Introduzione

La tesina si propone di esplorare la gestione della concorrenza in Ruby, linguaggio di programmazione ideato e sviluppato nella metà degli anni '90 dal giapponese Yukihiro Matsumoto.

Negli anni le **performance dei processori** sono aumentate principalmente attraverso l'aumento della **velocità di clock**, ma oggi l'ulteriore aumento in questa direzione porterebbe a chip che consumano potenze inadeguate, senza contare la produzione eccessiva di calore. Inoltre, una maggiore velocità di clock non sempre porta a processori più veloci a causa del collo di bottiglia dovuto agli accessi in memoria.

La nuova strada che è stata intrapresa è quella dei **processori multi-core**. Il problema che questo approccio ha portato alla luce è il fatto che il software per PC generalmente è scritto per eseguire un flusso di istruzioni per volta. Si ha pertanto la necessità di creare software che faccia più cose allo stesso tempo, anche se più difficile da scrivere, per sfruttare al meglio le potenzialità dell'hardware.

In alcuni casi, un'architettura multi-core porta a piccoli benefici se l'applicazione stessa non è scritta in modo da bilanciare il carico di lavoro sui diversi core disponibili. Scrivere codice multithread spesso richiede un'attenta **gestione della concorrenza** che può portare facilmente a sottili errori di programmazione difficili da debuggare. I passi fondamentali da tenere a mente nella scrittura di software composto da più flussi esecutivi sono:

- Partizionamento: in fase di progetto è necessario suddividere i compiti che possono venir eseguiti in parallelo. L'obiettivo è quello di definire un grande numero di piccoli task in modo da portare a termine un problema anche di grosse dimensioni attraverso la risoluzione di problemi più piccoli.

- Comunicazione: I task generati al passo precedente sono pensati per essere eseguiti in modo concorrente, ma in generale, non sono indipendenti. Deve quindi essere possibile il trasferimento di dati da un task all'altro. Il flusso dell'informazione deve essere specificato in questa fase del progetto.

- Agglomerazione: in questa fase, si esaminano le decisioni prese nei passi precedenti per ottenere un algoritmo efficiente su qualche classe di computer che supportano l'esecuzione in parallelo. In particolare, il progettista deve considerare quali task sia utile agglomerare, in modo da ottenere un minor numero di task, anche se di dimensioni maggiori. È necessario anche determinare se convenga o meno replicare dati o segmenti di codice in modo che i task possano operare in modo più efficiente.

- Mapping: in questa fase finale, il progettista deve specificare quali operazioni vengono eseguite da ciascun task. Il problema di tener traccia di quello che fanno i diversi task non si pone nel caso di processori uni-core o su computer a memoria condivisa che dispongono di uno scheduling automatico dei task. D'altro canto, nel

caso di server, i processori multi-core sono ideali poichè permettono agli utenti di connettersi simultaneamente e avere thread d'esecuzione indipendenti.

Diverse versioni di Ruby permettono di sfruttare i **thread nativi** del sistema operativo e non **green thread**. I green thread emulano sistemi in cui è possibile il multithreading senza utilizzare le potenzialità fornite dal sistema operativo. Sono gestiti nello spazio utente anzichè nello spazio kernel, e permettono di lavorare in ambienti che non hanno il supporto per i thread nativi. Su un processore multicore, l'implementazione basata su thread nativi permette automaticamente di suddividere il lavoro sui diversi processori, mentre normalmente ciò non è valido per un'implementazione basata su green thread.

La tesina si propone di esplorare i diversi aspetti della concorrenza in Ruby, affrontando gli argomenti per passi successivi.

Nel *primo capitolo* vengono presentati i **concetti fondamentali per programmare in Ruby**, senza i quali non è possibile comprendere gli argomenti esposti. Vengono anche presentate le varie **Virtual Machine disponibili per Ruby**, mettendone in luce differenze e potenzialità. Questo è fondamentale per capire in che modo venga gestita la concorrenza in Ruby: implementazioni diverse determinano modelli diversi di gestione della concorrenza.

Nel *secondo capitolo* ci si sofferma principalmente su **Moduli e Mixin**. Se i primi possono essere un concetto più o meno noto per chi ha una certa familiarità con altri linguaggi di programmazione, i Mixin sono un concetto leggermente diverso, che se sfruttato in modo appropriato può risultare particolarmente efficace.

Nel *terzo capitolo* ci si sofferma sulle caratteristiche dei thread in Ruby. Vengono riportati in modo esaustivo i metodi della **classe Thread** e alcuni esempi di utilizzo di tali metodi.

Nel *quarto capitolo* viene esposto il cuore dell'argomento in esame. Vengono presentati i **meccanismi di sincronizzazione** che caratterizzano le diverse piattaforme Ruby, con un occhio di riguardo per le versioni 1.8, 1.9 e JRuby. Vengono presentati i risultati di test che mettono in luce quando sia preferibile l'utilizzo dei thread o processi. Nel paragrafo 4.5 ("Oltre ai thread"), vengono introdotti Fiber, Actor e Eventmachine basata su Reactor.

Nel *quinto capitolo* vengono presentati i **costrutti di sincronizzazione** quali in particolare semafori, monitor e regioni critiche. Di particolare interesse è il confronto tra il Monitor in Ruby e il Monitor di Java (v. par. 5.3.1, "Confronto con il monitor di Java").

Nel *sesto capitolo* vengono proposti alcuni **esempi** che riassumono i concetti esposti nei capitoli precedenti. Il problema del produttore-consumatore viene implementato utilizzando i diversi costrutti di sincronizzazione. Nel paragrafo 6.2 viene confrontata l'implementazione in Ruby del problema dei 5 filosofi a cena con quella in Java. Nei due paragrafi successivi vengono presentati due esempi di applicazioni reali dei concetti esposti, quali l'implementazione di un server web o di una coda distribuita.

Nell'*appendice* viene presetato **Shoooes!**, che permette di creare facilmente interfacce grafiche per applicazioni Ruby. Viene riportato come è stato possibile implementare una **semplice applicazione che permette di aprire un file .txt**, in cui sarà possibile eseguire ricerche ed eventualmente eseguire le parti di codice Ruby presenti.

Capitolo 1

Introduzione a Ruby

Yukihiro Matsumoto, conosciuto come Matz, è il creatore di Ruby. Citando le sue parole:

''I knew many languages before I created Ruby, but I was never fully satisfied with them. They were uglier, tougher, more complex, or more simple than I expected. I wanted to create my own language that satisfied me, as a programmer. I knew a lot about the language's target audience: myself. To my surprise, many programmers all over the world feel very much like I do. They feel happy when they discover and program in Ruby. Throughout the development of the Ruby language, I've focused my energies on making programming faster and easier. All features in Ruby, including object-oriented features, are designed to work as ordinary programmers (e.g., me) expect them to work. Most programmers feel it is elegant, easy to use, and a pleasure to program.''

La filosofia che Matz ha seguito per la progettazione di Ruby è spesso riassunta in:

''Ruby is designed to make programmers happy.''

Ruby è un potente linguaggio open-source completamente orientato agli oggetti. È un linguaggio portabile, infatti funziona sia in ambienti UNIX che Microsoft, e particolarmente user-friendly. È stato ideato per permettere al programmatore di concentrarsi quanto più possibile sul problema che si trova ad affrontare, e non alla sintassi del linguaggio.

Ruby è un linguaggio relativamente giovane, che dispone già di numerose librerie, in continua espansione. Permette pertanto di affrontare diversi problemi di programmazione quali:

- *Text processing: classi specifiche rendono efficiente l'elaborazione di testi;*
- *CGI programming: Ruby fornisce tutto ciò che serve per la programmazione di CGI (Common Gateway Interface), incluse classi per la manipolazione di testi, una libreria CGI, interfacce per DataBase, e anche eRuby (embedded Ruby) un mod_ruby per Apache;*

- *Network programming: la programmazione di reti può avvalersi di classi specificamente strutturate in Ruby;*
- *GUI programming: sono disponibili alcuni ambienti di sviluppo con interfaccia grafica come ad esempio Ruby/Tk e Ruby/Gtk;*
- *XML programming: Ruby, grazie alle elevate potenzialità di elaborazione dei testi e potendosi avvalere del UFT-8 (Unicode Transformation Format, 8 bit) rende la programmazione in XML veramente semplice. Inoltre è disponibile un'interfaccia alla libreria expat XML parser;*
- *Prototyping: grazie alla sua alta produttività Ruby è spesso usato per sperimentare nuove soluzioni, che a volte vanno a risolvere i colli di bottiglia su alcuni programmi scritti in C;*
- *Programming education: Ruby per la sua sinteticità è particolarmente indicato per la didattica, in quanto con poche righe di codice permette di creare applicazioni anche complesse.*

Nel seguito vengono esposti concetti fondamentali da tener presente per programmare in Ruby. Gli argomenti non vengono trattati in modo esaustivo, ma sono necessari per la comprensione dei capitoli successivi, in particolare per gli esempi proposti. Per approfondire gli argomenti trattati o altri aspetti della programmazione in Ruby si consiglia di leggere ***Programming Ruby 1.9*** *di D. Thomas (ed. The Pragmatic Programmers' Guide), che tratta ogni argomento in modo chiaro, oppure* ***_why's (poignant) guide to Ruby****, scritta da _whytheluckystiff, una sorprendente ed eccentrica guida, che rende lo studio divertente.*

Figura 1.1: Fumetto tratto da "_why's (poignant) guide to Ruby".

1.1 Breve panoramica su Ruby

1.1.1 Ruby è orientato agli oggetti

Come già accennato in precedenza, **Ruby è completamente orientato agli oggetti**. Qualsiasi valore è un oggetto, anche i semplici numeri o *true*, *false* e *nil* (*nil* è un oggetto speciale che indica l'assenza di valore; è la versione in Ruby di *null*). Riportiamo nel seguito degli esempi di invocazione del metodo che restituisce il nome della classe di appartenenza dell'oggetto:

```
0 1.class       # => Fixnum
1 0.0.class     # => Float
2 true.class    # => TrueClass
3 false.class   # => FalseClass
4 nil.class     # => NilClass
```

Tabella 1.1: Tabella riassuntiva dei prefissi che denotano la tipologia di variabile

Tipologia	Prefisso	Esempio
Variabile Globale	$	$global
Variabile d'istanza	@	@istance
Variabile locale	lettera minuscola o _	local
Costante	lettera maiuscola	Cost
Variabile di classe	@@	@@class_var

Si tenga presente che in Ruby i commenti sono preceduti dal simbolo # e che in genere per convenzione se a tal simbolo si fa seguire la freccia =>, ciò che segue indica il valore restituito dal metodo invocato su tale riga di codice.

In Ruby, l'uso di parentesi nell'invocazione di un metodo è opzionale, e generalmente vengono omesse, specialmente se il metodo che si vuole utilizzare non richiede argomenti. Il fatto che qui vengano omesse potrebbe portare a confondere l'invocazione del metodo *class* con il riferimento al nome di un campo o di una variabile appartenenti ad un oggetto. In realtà, **Ruby non permette l'accesso allo stato interno di un oggetto da un oggetto esterno. Qualsiasi tipo di accesso deve essere mediato da un metodo d'accesso**, come ad esempio il metodo *class* riportato sopra.

1.1.2 Nessuna Dichiarazione

Il tipo di variabile in un programma Ruby può essere riconosciuto dal prefisso del suo nome. A differenza di Perl, **il prefisso denota la tipologia della variabile**, ciò rende inutile la dichiarazione del suo tipo. Inoltre, questo aumenta la leggibilità, poiché con un unico colpo d'occhio è possibile identificare il tipo di una variabile.

1.1.3 Duck Typing

Come già detto, in Ruby non si dichiara il tipo delle variabili, tutto è un qualche tipo di oggetto. Il fatto che le variabili siano di tipo dinamico, porta a dei vantaggi in fase di programmazione.

Non c'è da stupirsi che tutto funzioni correttamente. Se si utilizza una variabile per qualche scopo, si hanno delle buone possibilità che verrà usata per lo stesso scopo se si accede ad essa qualche linea di codice dopo. Il fatto che non si verifichino errori, sta anche nel fatto che il codice Ruby è generalmente composto da metodi corti, che vengono testati man mano dal programmatore. Questo comporta il fatto che le variabili hanno una visibilità ridotta all'interno del codice, e nel caso dovessero verificarsi errori banali, essi possono subito venir individuati e corretti, senza che essi si propaghino all'interno del codice.

Programmando nei linguaggi tradizionali, si è portati a pensare che il *tipo* di un oggetto sia la sua *classe*: tutti gli oggetti sono istanze di qualche classe, e tale classe è il tipo dell'oggetto. Questo però non è sempre vero, nemmeno in linguaggi come Java. A volte il tipo è un sottoinsieme della classe, e a volte gli oggetti implementano

più tipi. In Ruby, la classe non è mai (o quasi) il tipo dell'oggetto. Invece, **il tipo è determinato da quello che l'oggetto sa fare**. Questo fatto viene detto *duck typing*: **se un oggetto cammina come una papera e si esprime come una papera, allora l'interprete Ruby è felice di trattarlo come se fosse una papera**.

In conclusione, per scrivere codice utilizzando la filosofia del duck typing, l'unica cosa da tenere a mente è che il tipo di un oggetto è determinato da quello che quest'ultimo sa fare, non dalla sua classe.

"Now! *That* should clear up
a few things around here!"

Figura 1.2: Vignetta umoristica sulla definizione statica delle variabili.

1.1.4 Blocchi ed Iteratori

La possibilità di invocare un metodo su un intero è un fatto che i programmatori Ruby sfruttano di frequente:

```
0 3.times { print "Ruby! " }    # Stampa "Ruby! Ruby! Ruby! "
1 1.upto(9) {|x| print x }      # Stampa "123456789"
```

times e *upto* sono metodi implementati dagli oggetti di tipo intero. Sono dei metodi speciali noti come iteratori, che si comportano come dei cicli. Il codice presente tra le parentesi graffe (o a volte tra *do* e *end*) è associato al metodo invocato e determina il corpo del ciclo. L'uso di blocchi ed iteratori è un'altra caratteristica peculiare di Ruby; **anche se il linguaggio supporta l'utilizzo del comune ciclo while, si è soliti scrivere un ciclo utilizzando costrutti che sono in effetti chiamate a metodi**.

Gli interi non sono gli unici valori che dispongono di iteratori. Gli array (e oggetti "enumerabili" simili) definiscono un iteratore chiamato *each*, che invoca il blocco associato una volta per ogni elemento contenuto nell'array. In ogni invocazione del blocco viene passato un singolo elemento dall'array:

```
0 # Questo è un array
1 a = [3, 2, 1]
2 # Uso [ ] per impostare e accedere agli elementi dell'array
3 a[3] = a[2] - 1
4 # each è un iteratore. Il blocco ha un paramentro 'elt'
5 a.each do |elt|
6 # Stampa "4321"
7   print elt+1
8 end
9 # Questo blocco era delimitato da do/end al posto di { }
```

Il linguaggio fornisce altri utili iteratori. Di seguito vengono riportati alcuni esempi:

```ruby
# Questo è un array
a = [1,2,3,4]
# Eleva al quadrato gli elementi: b è [1,4,9,16]
b = a.map {|x| x*x }
# Selezione gli elementi pari: c è [2,4]
c = a.select {|x| x%2==0 }
# Calcola la somma degli elementi => 10
a.inject do |sum,x|
  sum + x
end
```

Gli hash, come gli array, sono strutture dati fondamentali in Ruby. Come dice il nome, si basano su hashtable e servono per mappare in modo arbitrario oggetti-valore con oggetti-chiave. Anche gli hash come gli array utilizzano le parentesi quadre per impostare ed accedere ai valori. Invece di utilizzare un indice numerico, è necessario inserire la chiave tra le parentesi quadre per accedere al valore corrispondente. Anche la classe *Hash* dispone dell'iteratore *each*. Questo metodo invoca il blocco associato al codice una volta per ogni coppia chiave/valore nell'hash, e (a differenza di quanto avveniva per la classe Array), passa sia chiave che valore come parametri del blocco:

```ruby
h = {                           # Un hash che mappa nomi di interi in cifre
  # La "freccia" mostra le associazioni: chiave=>valore
  # i due punti indicano l'utilizzo di un Simbolo
  :one => 1,
  :two => 2
}
h[:one]                         # => 1.  Accede ad un valore data la chiave
h[:three] = 3                   # Aggiunge una nuova coppia all'hash
h.each do |key,value|           # Itera attraverso le coppie chiave/valore
  print "#{value}:#{key}; "     # Trasforma le variabili in stringhe
end                             # Stampa "1:one; 2:two; 3:three; "
```

Gli hash in Ruby possono utilizzare qualsiasi oggetti come chiave, ma normalemente si utilizzano oggetti Simbolo. I simboli sono oggetti immutabili. È il compilatore Ruby che si preoccupa di associare ad ogni simbolo un valore diverso, in modo da non confondere due simboli all'interno del codice.

La possibilità di associare all'invocazione di un metodo un blocco è una caratteristica fondamentale e molto potente di Ruby. È molto utile utilizziare i blocchi anche se invocati solo una volta (e non ripetutamente come avviene quando si vuole ottenere un ciclo). Ad esempio:

```ruby
# Apre il file data.txt e passa lo stream al blocco
File.open("data.txt") do |f|
  line = f.readline             # Legge dal file
end                             # Il flusso viene automaticamente chiuso

t = Thread.new do         # Esegue questo blocco in un nuovo thread
  File.read("data.txt") # Legge un file in background
end
# I contenuti del file sono disponibili come valore del thread
```

1.1.5 Espressioni ed Operatori

La sintassi di Ruby è expression-oriented. Strutture di controllo del flusso come ad esempio *if* che vengono chiamati "costrutti" in altri linguaggi di programmazione, sono a tutti gli effetti delle espressioni in Ruby. Assumo dei valori come le altre espressioni, ed è possibile scrivere del codice come il seguente:

```ruby
minimum = if x < y then x else y end
```

Anche se **ogni "costrutto" in Ruby è a tutti gli effetti un espressione**, non tutti possono restituire valori utili. Ad esempio, i cicli *while* e le definizioni di metodi, normalmente restituiscono il valore *nil*.

Come nella maggior parte dei linguaggi, le espressioni in Ruby sono costituite da valori e operatori. Di seguito vengono riportati degli esempi di comuni espressioni:

```
1 + 2                    # => 3: addizione
1 * 2                    # => 2: moltiplicazione
1 + 2 == 3               # => true: == test di uguaglianza
2 ** 1024                # 2^1024: gli interi hanno dimensione arbitraria
"Ruby" + " rocks!"       # => "Ruby rocks!": concatenazione di stringhe
"Ruby! " * 3             # => "Ruby! Ruby! Ruby! ": ripetizione di stringhe
"%d %s" % [3, "rubies"]  # => "3 Rubies": formattazione
max = x > y ? x : y      # operatore condizionale
```

Molti degli operatori in Ruby sono implementati come metodi, e le classi possono definire (o ridefinire) tali metodi.

1.1.6 Metodi

I metodi vengono definiti utilizzando la parola chiave *def*. Il valore restituito dal metodo corrisponde al valore dell'ultima espressione valutata nel corpo del metodo stesso:

```
def square(x)    # Definisce un metodo di nome square con un parametro x
  x*x            # Restituisce il quadrato di x
end              # Fine del metodo
```

Nel capitolo seguente (cap. 2) vengono approfonditi aspetti importanti sulla definizione dei metodi, introducendo ad esempio i singleton.

1.1.7 Regular Expression

Ruby fornisce una grande varietà di regular expression per la massima flessibilità, soprattutto per le stringhe, ad esempio:

```
  "stringa con escape\n"
  'senza escape\n'
  `echo command output pseudo string`
  %q!perl style single quote!
  %Q!perl style double quote\n!
  %w(word list)
  /regular expression/
  %r!another regex form!
```

Le regular expression in Ruby sono molto simili a quelle in Perl, ad eccezione di alcune che richiedono il simbolo % all'inizio.

1.1.8 Alcune particolarità di Ruby

Ogni linguaggio di programmazione possiede alcune caratteristiche che invogliano i programmatori ad utilizzarlo. Di seguito vengono descritte due caratteristiche di Ruby magari inaspettate per programmatori abituati ad altri linguaggi.

Le stringhe in Ruby sono oggetti che possono essere modificati, cosa che potrebbe sorprendere i programmatori Java in particolare. L'operatore []= permette di modificare i caratteri di una stringa o di inserire, cancellare o sostituire sottostringhe. L'operatore << permette di concatenare caratteri alla fine di una stringa, e la classe *String* mette a disposizione molti altri metodi che permettono di modificare una stringa in place.

In Ruby le espressioni di condizione, che determinano quale ramo di codice eseguire (per il costrutto *if-else*) o se continuare un ciclo (ad esempio per il costrutto *while*),

spesso valutano se la condizione è vera o falsa, ma questo non è necessario. Il valore *nil* è trattato nello stesso modo di *false*, e qualsiasi altro valore è lo stesso di *true*. Questo potrebbe sorprendere i programmatori C che si aspettano che 0 produca lo stesso risultato di *false*, e i programmatori JavaScript che si aspettano che la stringa vuota sia lo stesso di *false*.

1.1.9 Versioni di Ruby

Le caratteristiche di Ruby che l'hanno reso popolare, sono le stesse che hanno permesso la sua diffusione su diverse piattaforme: JVM, Objective-C, Smalltalk VM e Microsoft's DLR. In pochi anni, il Ruby VM space si è evoluto in varie direzioni: MRI, JRuby, IronRuby, MacRuby, Rubinius, MagLev, REE e BlueRuby (solo per citare alcuni nomi).

Figura 1.3: Loghi delle differenti versioni di Ruby

Nel seguito presentiamo una veloce panoramica sulle versioni più conosciute di Ruby.

MRI e YARV

Figura 1.4: Logo della piattaforma MRI

MRI (Matz's Ruby Interpreter) è la piattaforma originale, creata da Matz, utilizzata da Ruby 1.8. Negli anni sono stati apportati significativi miglioramenti, che hanno aumentato in modo consistente le performace (inizialmente Ruby veniva considerato 'lento'). La versione più recente di Ruby (1.9) utilizza una piattaforma chiamata **YARV**, che ha sostituito MRI.

Quando un programma viene analizzato, viene creato un **albero della sintassi**. Questo rappresenta in modo gerarchico tutte le parti (identificatori, parole chiavi, operatori, ecc.) del linguaggio. MRI iterava direttamente attraverso quest'albero quando eseguiva un programma. Anche se questo è più semplice da implementare, è inefficiente sia per l'utilizzo della memoria che per il tempo d'esecuzione nel lungo periodo. YARV, dopo aver creato l'albero della sintassi, modifica il programma originale, ottenendo una struttura più efficiente. Tale programma viene eseguito su una macchina virtuale (simile ad un emulatore), senza dover attraversare l'albero della sintassi più di una volta. Questa modifica a portato Ruby, che era considerato

uno dei linguaggi di programmazione più lenti largamente utilizzato, a raggiungere delle buone prestazioni.

JRuby: Ruby e la JVM

Tra tutte le diverse "alternative" per le VM di Ruby, JRuby è uno dei progetti più maturi sia in termi di compatibilità che di diffusione. Combinando gli aspetti migliori della JVM, come un Garbage Collector generazionale[1], concorrenza reale, e interazione trasparente con tutte le librerie Java, tutto utilizzando la sintassi di Ruby, non c'è da sorprendersi che JRuby stia avendo successo. È veloce, può essere utilizzato con **Ruby on Rails**[2], e presto sarà compatibile con Ruby 1.9.

Figura 1.5: Logo di JRuby

JRuby, come del resto le altre piattaforme Ruby, ha subito di recente diversi miglioramenti. Il JRuby team, che attualmente è composto da 7 membri attivi (3 dei quali hanno lasciato la Sun per lavorare per la Engine Yard, una delle maggiori compagnie nell'ecosistema di Ruby on Rails), e molti altri collabratori, hanno aggiunto un gran numero di nuove funzionalità a JRuby 1.4.0 e corretto oltre 300 bug rispetto la versione 1.3.1.

MacRuby: Objective-C, LLVM e Ruby

Figura 1.6: Logo di MacRuby

MacRuby permette l'accesso alle librerie dei sistemi OSX o alla Cocoa API attraverso la VM Ruby.

MacRuby 0.4 è fornito con il supporto ai 64-bit, sonde (probe) in DTrace[3], e diversi miglioramenti all'API HotCocoa. Oltre a questo, il progetto è passato da YARV a una nuova VM basata su LLVM compiler infrastructure[4], che ha portato numerosi benefici, tra i quali la concorrenza reale.

[1] È stato osservato che in molti programmi gli oggetti creati più recentemente sono anche quelli con più probabilità di diventare rapidamente irraggiungibili (noto come aka Ephemeral o Generational GC). L'ipotesi generazionale divide gli oggetti in generazioni.

[2] Spesso chiamato RoR o semplicemente Rails, è un framework open source per applicazioni web scritto in Ruby. I suoi obiettivi sono la semplicità e la possibilità di sviluppare applicazioni di concreto interesse con meno codice rispetto ad altri framework. Il tutto con necessità di configurazione minimale.

[3] La funzione Dynamic Tracing (DTrace) offre capacità di osservazione pervasiva e sicura sui sistemi di produzione per velocizzare lo sviluppo e l'ottimizzazione di applicazioni.

[4] La Low Level Virtual Machine (macchina virtuale di basso livello), generalmente nota come LLVM, è una infrastruttura di compilazione, scritta in C++, progettata per l'ottimizzazione di programmi in fase di compilazione, di linking, di esecuzione e di non utilizzo.

MacRuby è a tutti gli effetti un compilatore Ruby. È possibile scrivere un applicazione HotCocoa, utilizzando i thread nativi POSIX, o sfruttando i vantaggi proposti dal Grand Central Dispatch della Apple (una tecnologia che ottimizza l'utilizzo dei processori multi-core), compilarla e distribuirla come binario a qualsiasi utente OSX.

Con un team composto da 7 membri e una comunità in crescita (RubyOnOSX), MacRuby sta velocemente diventando uno dei più promettenti progetti open-source per Apple. In teoria, sarebbero sufficienti piccole modifiche per rendere MacRuby utilizzabile per applicazioni per IPhone.

MagLev: Smalltalk VM e Ruby

Figura 1.7: Logo di MagLev

Si può pensare a MagLev come ad un database distribuito che esegue codice Ruby al suo interno. Centinaia di VM concorrenti su centinaia di nodi possono accedere agli stessi dati utilizzando la semantica ACID (deriva dall'acronimo inglese: Atomicity, Consistency, Isolation, Durability).

La VM offre un persistence layer, che è una delle funzionalità più interessanti, ma MagLev supporta anche altri modelli di persistenza che permettono di utilizzare anche ActiveRecord con MySQL, ecc.

Ruby Enterprise Edition

Lanciata a metà del 2008, REE è una versione di Ruby 1.8.7 ottimizzata principalmente per server. L'utilizzo della memoria e le performance in generale sono significativamente maggiori rispetto a MRI Ruby.

IronRuby: Ruby on .NET

Figura 1.8: Logo di IronRuby

IronRuby è un'implementazione .NET di Ruby che sfrutta DLR (Microsoft's Dynamic Language Runtime), aprendo diverse porte per le possibilità d'utilizzo di Ruby. Ad esempio, il fatto che Ruby sia eseguito su DLR significa che esiste la possibilità di eseguirlo con Silverlight[5].

[5]Silverlight è un ambiente di runtime sviluppato da Microsoft per piattaforme Windows e Mac che consente di visualizzare, all'interno del browser, Rich Internet applications, ovvero applicazioni multimediali ad alta interattività.

Rubinius: Ruby scritto (quasi) completamente in Ruby

Rubinius è un progetto iniziato nel 2006 per implementare quanto più possibile Ruby utilizzando codice Ruby. La VM è stata riscritta in C++, è stato aggiunto un compilatore just-in-time (JIT), per aumentare le performance, ed è possibile eseguire un gran numero di gem, ossia programmi/librerie scritte in Ruby.

Il progetto non è ancora maturo (è troppo presto per parlare di performance), ma la combinazione del compilatore JIT e dell'infrastruttura LLVM determina la presenza in Rubinius di una concorrenza reale, e un gran numero di possibilità per l'ottimizzazione del codice (un vantaggio portato direttamente dal fatto di riscrivere Ruby utilizzando Ruby).

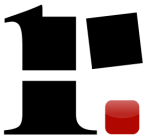

Figura 1.9: Logo di Rubinius in 2D

Rubies everywhere!

Parlare delle diverse VM per Ruby mette in evidenza come il linguaggio sia in espansione in diversi settori, ma soprattutto come l'utilizzo di Ruby possa creare un ponte di collegamento con le altre communities.

1.1.10 Cos'è Shoooes!

Shoooes! è un potente toolkit grafico che permette di creare in modo semplice e rapido interfacce grafiche per applicazioni Ruby. Per maggiori informazioni si fa riferimento al **Manuale di Shoes** scaricabile assieme al programma all'indirizzo shoooes.net e all'appendice riportata nel seguito (v. cap. 6.4).

Figura 1.10: Alcuni esempi di interfaccia grafica realizzabile con Shoes.

Capitolo 2

Metaprogrammazione

Fondamentale per scrivere codice è basarsi su diversi livelli di astrazione. Linguaggi di programmazione come Ruby, permettono molti livelli di astrazione.

La metaprogrammazione permette però di spaziare in astrazioni che non fanno parte del linguaggio. In effetti, è possibile ottenere un linguaggio che è specifico per risolvere il problema che si sta considerando. Ruby rende semplice la metaprogrammazione, e per questo motivo i programmatori Ruby esperti utilizzano tale strumento per semplificare il loro codice.

Nel seguito, verranno esaminati i principi che rendono possibile la metaprogrammazione in Ruby. Fondamentale per la comprensione dei capitoli successivi è il capire il funzionamento dei ***moduli*** *e dei* ***mixin****.*

2.1 Oggetti e classi

Le classi e gli oggetti sono certamente centrali in Ruby, ma a prima vista potrebbero sembrare dei concetti leggermente confusi. Un oggetto in Ruby è costituito da tre componenti: un insieme di flag, qualche variabile d'istanza, e una classe associata. Una classe in Ruby è essa stessa un oggetto della classe *Class*. Una classe contiene tutto quello che è contenuto in un oggetto, più un insieme di definizioni di metodi e un riferimento ad una superclasse (che è essa stessa una classe).

2.2 Singleton

Vengono chiamati *singleton method* quei **metodi che appartengono ad un singolo oggetto**. Per definire un metodo di questo tipo è sufficiente scrivere dopo *def* un'espressione che valuta l'oggetto, seguita da un punto e il nome che si vuole assegnare al metodo. Tale metodo sarà disponibile solo all'oggetto che viene valutato. Ad esempio:

```
0 o = ''message''     # Una stringa è un oggetto
1 def o.printme     # Definisce un metodo di tipo singleton sull'oggetto
2   puts self
3 end
4 o.printme         # Invoca il singleton
```

Altri esempi di metodi singleton sono *Math.sin* e *File.delete*. Infatti, *Math* è una costante che si riferisce ad un modulo oggetto, mentre *File* è una costante che fa riferimento a un oggetto Classe.

Per capire in che modo vengono fatti i riferimenti, si considera il semplice esempio:

```
0 animal = ''cat''
1 puts animal.upcase
```

che stampa a vidio la stringa "CAT". In fig. 2.2 viene riportato il modello dell'oggetto *animal*.

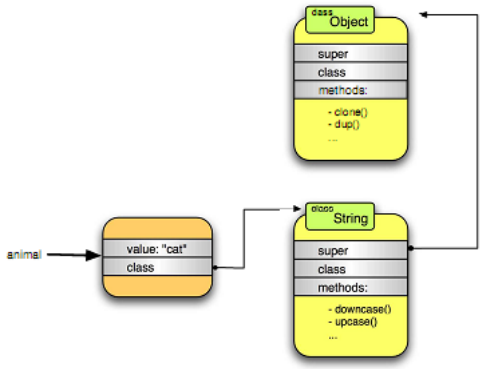

Figura 2.1: Modello dell'oggetto per una classe semplice

Se viene definito un singleton sulla stringa a cui fa riferimento *animal*:

```
0 def animal.speak
1     puts ''The #{self} says miaow''
2 end
```

Il modello dell'oggetto si trasforma come in fig. 2.2.

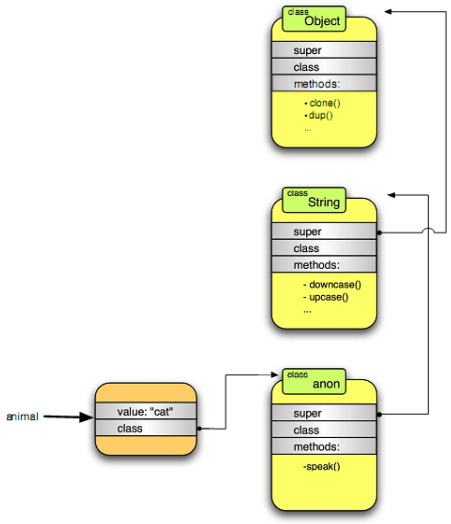

Figura 2.2: Modello dell'oggetto per una classe Singleton

2.3 Ereditarietà

La maggior parte dei linguaggi orientati agli oggetti permette il meccanismo dell'ereditarietà, ossia la possibilità di creare nuove classi con comportamento simile a classi già esistenti. Per far ciò, è sufficiente specificare, al momento della creazione della classe, che vogliamo estendere (ossia ereditare i metodi da) una classe esistente. La sintassi da utilizzare è la seguente:

```
0 class nomeSottoClasse < nome Superclasse
1 ...
2 end
```

In realtà, anche senza specificare una superclasse (ossia la classe da cui vogliamo ereditare i metodi), la nuova classe che creiamo sarebbe una sottoclasse, in quanto ogni classe è sottoclasse della classe *Object*. In Ruby 1.9, la cima nella gerarchia delle classi è la classe *BasicObject*, una classe semplicissima che non contiene metodi. Questa viene seguita da *Object*, ed è quest'ultima classe che viene implicitamente estesa al momento della creazione di una nuova classe. Anche in Ruby è possibile sovrascrivere i metodi ereditati dalla superclasse, definendo all'interno della sottoclasse il metodo che si ha la necessità di sovrascrivere.

2.4 Moduli e Mixin

I moduli sono delle strutture che contengono metodi, variabili di classe e costanti, ma a differenza delle classi, un modulo non può essere istanziato e non può diventare una sottoclasse. Infatti, non è possibile stabilire una gerarchia d'ereditarietà tra moduli. Un oggetto della classe *Class* è un'istanza di tale classe, come un oggetto modulo è un'istanza dalla classe *Module*. *Class* è una sottoclasse di *Module*: **tutte le classi sono moduli, ma non è vero il contrario**. Le classi possono essere usate come spazio dei nomi, ma non possono essere usate come mixin, mentre i moduli possono svolgere entrambi i ruoli.

2.4.1 Moduli come spazio dei nomi

Spesso risulta utile raggruppare metodi pur non avendo la necessità di creare una nuova classe. Per prevenire la collisione nello stesso spazio dei nomi è bene inserire tali metodi in un modulo, nel modo seguente:

```
0 module NomeModulo
1     def self.metodo1
2         #...
3     end
4     def self.metodo2
5         #...
6     end
7 end
```

Generalmente i moduli vengono indicati con l'iniziale maiuscola. Un modulo può contenere anche delle costanti. Ad esempio:

```
0 module NomeModulo
1     COST = 3
```

All'esterno del modulo si farà riferimento a tale costante con *NomeModulo::COST*, mentre all'interno dei metodi del modulo si potrà far riferimento ad essa semplicemente utilizzando *COST*. È possibile creare dei moduli nidificati contenenti altre

classi o moduli. Questo non ha effetti particolari, è sufficiente prestare attenzione al campo di visibilità delle variabili.

2.4.2 Moduli come mixin

Il secondo utilizzo dei moduli è più potente. Se un modulo definisce metodi d'istanza, anziché metodi di classe, tali metodi possono essere "mixati" in altre classi. Per citare qualche esempio, *Enumerable* e *Comparable* sono moduli mixin. Per mixare un modulo in una classe, si usa la parola chiave *include*:

```
0 class Classe
1     include Modulo
2 end
```

Concettualmente, i moduli mixin ricordano le interfacce in Java o C#. **Come un'interfaccia, un modulo permette a classi diverse di condividere un insieme comune di metodi. Una differenza fondamentale sta però nel fatto che un'interfaccia è un concetto astratto, mentre un modulo contiene l'implementazione dei suoi metodi.** Anche se ogni classe è un modulo, la parola chiave *include* deve necessariamente essere seguita da un modulo definito con *module*. È possibile includere un modulo per semplificare la scrittura di chiamata ad un metodo, ad esempio:

```
0 Math.sin(0)       #Math è uno spazio dei nomi
1 include Math
2 sin(0)        #Adesso l'accesso al metodo è più semplice
```

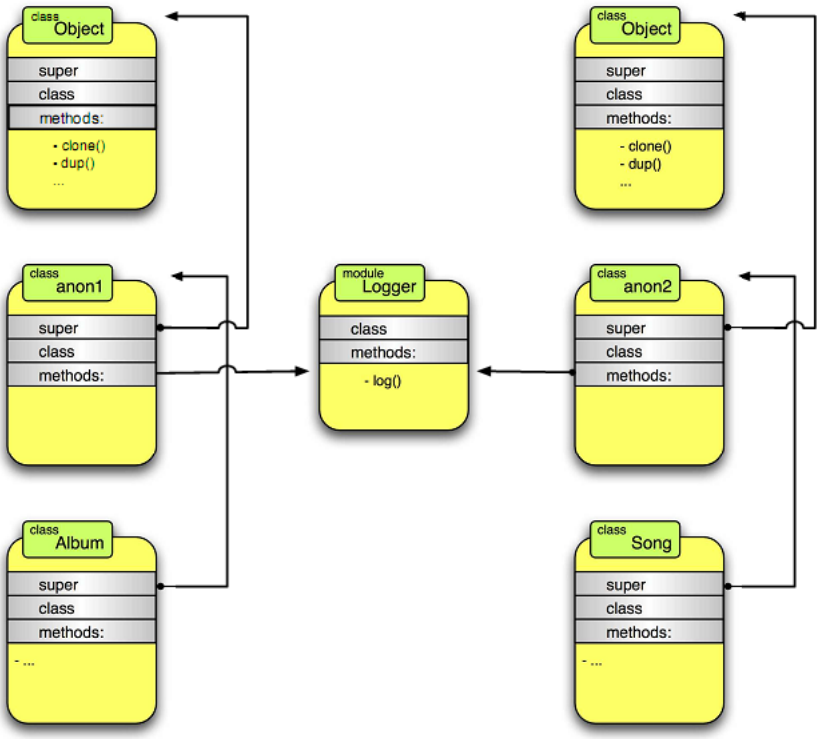

Figura 2.3: Come vengono inclusi i moduli

Per capire in che modo i moduli vengono inclusi, si considera l'esempio:

```
0  module Logger
1      def log(msg)
2          STDERR.puts Time.now.strftime(''%H:%M:%S: '') + ''#{self} (#{msg})''
3      end
4  end
5  class Song
6      include Logger
7  end
8  class Album
9      include Logger
10 end
11 s = Song.new
12 s.log(''created'')
```

Viene definito il modulo *Logger* contenente il metodo *log*, due classi (*Song* e *Album*) in cui viene mixato il modulo creato sopra. Viene poi creata un'istanza della classe Song, a cui è permesso utilizzare il metodo *log*. Tale codice produce infatti: "13:26:13: #<Song:0x0a323c> (created). Come riportato in fig. 2.3, il modulo che viene mixato viene effettivamente aggiunto alla superclasse della classe considerata, e ciò avviene per ciascuna classe in cui venga incluso il modulo.

Capitolo 3

Multiprogrammazione

*Ruby mette a disposizione principalmente due vie per eseguire parti di codice differenti apparentemente "allo stesso tempo". I **Fiber**, introdotti a partire da Ruby 1.9, permettono di sospendere l'esecuzione di una parte di programma per eseguire qualche altra parte. In alternativa, si possono dividere le operazioni in diversi task all'interno del programma, utilizzando thread multipli, oppure dividere i task tra differenti programmi, usando processi multipli.*

*Nel seguito di questo capitolo non si parlerà oltre dei fiber, mentre ci si soffermerà sul **multithreading**. Verranno quindi introdotti i metodi principali che permettono di interagire con i thread, mentre nel capitolo successivo si approfondiranno in dettaglio i meccanismi di sincronizzazione offerti da Ruby.*

3.1 Introduzione alla multiprogrammazione

I programmi tradizionali hanno un singolo flusso esecutivo: le istruzioni che costituiscono il programma vengono eseguite sequenzialmente finché il programma stesso non giunge al termine. Un programma multithread è composto da più thread, che possono essere eseguiti in parallelo su processori multi-core, anche se le istruzioni all'interno di ciascun thread vengono eseguite sequenzialmente. Spesso (su single-core, macchine a processore singolo, ...) i thread multipli non vengono effettivamente eseguiti in parallelo, ma il parallelismo viene solo simulato alternando l'esecuzione dei singoli thread.

I programmi compute-bound possono trarre vantaggio dal multithreading solo se sono presenti CPU multiple che eseguano computazioni in parallelo. Molti programmi non sono comunque completamente compute-bound. Molti, come ad esempio i browser web, spendono la maggior parte del tempo nell'attesa di file di I/O. Programmi di questo tipo vengono detti IO-bound, e tali programmi possono sfruttare il multithreading anche in presenza di una singola CPU. Un browser web può scaricare un'immagine in un thread mentre un altro thread è in attesa della prossima immagine dalla rete.

Ruby rende semplice la scrittura di programmi multithread fornendo la classe *Thread*. Per avviare un nuovo thread, basta associare un blocco con la chiamata *Thread.new*. Un nuovo thread verrà creato per eseguire il codice nel blocco, e il thread originale riprenderà la propria esecuzione in modo concorrente al thread appena creato con l'istruzione immediatamente successiva al blocco:

```
0 #qui è in esecuzione il thread #1
1 thread.new {
2     # qui è in esecuzione il thread #2
3 }
4 # qui esegue il codice il thread #1
```

3.2 Il multithreading in Ruby

Per creare nuovi thread, oltre a *Thread.new*, si può utilizzare anche il sinonimo *Thread.start* o *Thread.fork*. Non è necessario avviare un thread dopo averlo creato; inizia la sua esecuzione automaticamente non appena le risorse della CPU divengono disponibili. Il valore di un'invocazione *Thread.new* è un oggetto Thread. La classe *Thread* definisce diversi metodi di interrogazione e manipolazione del thread quando questo è in esecuzione. Un thread esegue il codice nel blocco associato alla chiamata di *Thread.new* e dopo termina l'esecuzione. Il valore dell'ultima espressione nel blocco è il valore del thread, e può essere ottenuto chiamando il metodo *value* dell'oggetto *Thread*. Se il thread è giunto a completamento, allora il metodo value restituisce subito il valore, altrimenti il metodo attende finché il thread non abbia completato la sua esecuzione. Di seguito riportiamo un esempio di codice che utilizza i thread per leggere il contenuto di più file in parallelo:

```
0 # Questo metodo si aspetta come parametro un array di nomi di file.
1 # Restituisce un array di stringhe costituito dal contenuto dei file.
2 # Il metodo crea un thread per ogni file.
3 def readfiles(filenames)
4   # Crea un array di thread dall'array di file.
5   # Ogni thread inizia a leggere un file.
6   threads = filenames.map do |f|
7     Thread.new { File.read(f) }
8   end
9
10  # Adesso crea un array di contenuti dei file chiamando il metodo
11  # value per ogni thread.
12  threads.map {|t| t.value }
13 end
```

Il metodo di classe *Thread.current* restituisce l'oggetto *Thread* che rappresenta il thread corrente. Questo permette che un thread manipoli se stesso. Il metodo di classe *Thread.main* restituisce l'oggetto Thread che rappresenta il main thread (il thread iniziale dell'esecuzione che è iniziato con l'esecuzione del programma).

3.2.1 Il main thread

Il main thread è speciale: l'interprete Ruby termina l'esecuzione quando il main thread termina. Questo avviene anche se il main thread ha creato altri thread che sono ancora in esecuzione. Bisogna essere certi, per questo motivo, che il main thread non termini prima di altri thread ancora in esecuzione, altrimenti quest'ultimi non terminano la loro esecuzione come ci si aspetterebbe. Un metodo per assicurare questo fatto è scrivere il main thread come un ciclo infinito. Un altro metodo è di esplicitare l'attesa del termine dei thread di cui ci si vuole assicurare il completamento corretto, ad esempio chiamando il metodo *value*. Se il valore dei thread non ha importanza, si può attende il completamento corretto anche utilizzando il metodo *join*. Il metodo seguente attende che tutti i thread, diversi dal main thread e dal thread corrente (che qui dovrebbero essere la stessa cosa), terminino:

```
0 # Attendi che tutti i thread (diversi dal main thread e dal thread
1 # corrente) terminino.
2 # Assumi che nessun nuovo thread venga eseguito durante l'attesa.
```

```
3 def join_all
4   main = Thread.main         # il main thread
5   current = Thread.current   # il thread corrente
6   all = Thread.list          # tutti i thread al momento in esecuzione
7   # chiamo join su ciascun thread
8   all.each {|t| t.join unless t == current or t == main }
9 end
```

3.2.2 Eccezioni

Se viene riscontrata un'eccezione nel main thread, e non viene gestita da nessuna parte, l'interprete Ruby stampa un messaggio ed interrompe l'esecuzione. Negli altri thread, le eccezioni che non vengono gestite causano il termine dell'esecuzione del thread. Di default, comunque, questo non provoca la stampa di un messaggio d'errore da parte dell'interprete e l'uscita dal programma. Se un thread t termina a causa di un'eccezione ed un altro thread s chiama *t.join* o *t.value*, allora l'eccezione che si era presentata in t si propaga ad s. Se non si vuole che una eccezione non gestita in qualche thread provochi questo, si può usare il metodo di classe *Thread.abort_on_exception=*:

```
0 Thread.abort_on_exception= true
```

Se invece si vuole che un'eccezione non gestita causi l'uscita dell'interprete, bisogna utilizzare il metodo di istanza con lo stesso nome:

```
0 t = Thread.new {...}
1 t.abort_on_exception = true
```

3.3 Utilizzo delle variabili

Poiché i thread sono definiti dai blocchi, possono avere accesso a tutti i tipi di variabile (variabili locali, variabili d'istanza, variabili globali, e così via) che sono visibili al blocco:

```
0 x = 0
1
2 t1 = Thread.new do
3   # Questo thread può interrogare e modificare la variabile x
4 end
5
6 t2 = Thread.new do
7   # Anche questo thread ha accesso a x
8   # E può interrogare e modificare gli stessi t1 e t2.
9 end
```

Quando due o più thread leggono e scrivono la stessa variabile in modo concorrente, bisogna essere certi che lo facciano nel modo corretto.

3.3.1 Variabili private

Variabili definite all'interno del blocco di un thread sono variabili private del thread non visibili agli altri thread. Questa è una semplice applicazione delle regole di visibilità delle variabili in Ruby. Spesso si vuole che un thread abbia la propria copia privata di una variabile in modo che il suo comportamento non venga influenzato da eventuali modifiche a tale variabile. Si consideri il seguente codice, che provvede a creare tre thread che stampano (rispettivamente) i numeri 1, 2 e 3:

```
0 #Esempio di codice imprevedibile
1 n = 1
2 while n <= 3
3   Thread.new { puts n }
4   n += 1
5 end
```

In alcune circostanze, in alcune implementazioni, questo codice lavorerà come ci si aspetterebbe stampando 1, 2 e 3. In altre circostanze o altre implementazioni, non lo farà. È perfettamente possibile che tale codice stampi 4, 4 e 4, ad esempio. Ciascun thread legge una copia salvata della variabile n, e il valore di questa variabile cambia quando il ciclo viene eseguito. Il valore stampato dal thread dipende da quando tale thread viene eseguito in relazione agli altri thread. Per risolvere questo problema, è necessario passare il corrente valore di n al metodo *Thread.new*, e assegnare il corrente valore di questa variabile a un parametro del blocco. I parametri del blocco sono privati al blocco, e questo valore privato non è condivisibile tra i thread:

```
0 n = 1
1 while n <= 3
2   # Ogni thread ha una copia privata del valore corrente di n in x
3   Thread.new(n) {|x| puts x }
4   n += 1
5 end
```

Nel caso non risulti chiaro come avviene l'associazione tra parametro del metodo e parametro del blocco far riferimento a 1.1.4. Si noti che un altro modo di risolvere questo problema è quello di usare un iteratore al posto di un ciclo while. In questo caso, il valore di n è privato per i blocchi esterni e non varia mai durante l'esecuzione del blocco:

```
0 1.upto(3) {|n| Thread.new { puts n }}
```

3.3.2 Thread e variabili locali

Alcune speciali variabili globali di Ruby sono locali ai thread: hanno valori differenti in thread differenti. Esempi sono *$SAFE* e *$.* Questo significa che se due thread stanno eseguendo in modo concorrente la ricerca per delle espressioni regolari, vedranno valori differenti per *$,* e l'esecuzione di un thread non influenzerà il risultato dell'altro. La classe *Thread* può comportarsi come un hash. Definisce i metodi d'istanza [] e []= che permettono di associare arbitrariamente valori con qualsiasi simbolo. (Se si sceglie di utilizzare una stringa, verrà convertita automaticamente in un simbolo. Contrariamente a un vero hash, la classe *Thread* consente solo simboli come chiavi.) I valori associati con questi simboli si comportano come variabili locali al thread. Non sono privati come le variabili locali ad un blocco perché qualsiasi thread può guardare il valore di una variabile in qualsiasi altro thread, però non sono variabili condivise, perché ogni thread può avere la propria copia. Ad esempio, si supponga di aver creato thread per il download di file da un web server. Il main thread provvederà a monitorare il progresso del download. Per permettere questo, ciascun thread dovrà fare:

```
0 Thread.current[:progress] = bytes_received
```

Il main thread potrebbe determinare i byte totali scaricati con un codice come questo:

```
0 total = 0
1 download_threads.each {|t| total += t[:progress] }
```

Insieme ai metodi [] e []=, la classe *Thread* definisce anche un metodo *key?* per verificare se una data chiave esiste o meno per un thread. Il metodo *keys* restituisce invece un array di simboli che rappresentano le chiavi che sono state definite per il thread. Il codice precedente può essere migliorato, tenendo presente quali thread non sono ancora in esecuzione e quelli che non hanno ancora definito la chiave *:progress*:

```
0 total = 0
1 download_threads.each {|t| total += t[:progress] if t.key?(:progress)}
```

3.4 Scheduling

Se non è possibile eseguire processi in parallelo, il parallelismo viene simulato dividendo il tempo di utilizzo della CPU tra i thread (scheduling). A seconda dell'implementazione della piattaforma, lo scheduling viene affidato all'interprete Ruby oppure viene gestito dal sottostante sistema operativo.

3.4.1 Le priorità

Il primo fattore che influenza lo scheduling dei thread è la loro priorità: i thread ad alta priorità hanno precedenza su quelli a bassa priorità. Più precisamente, ad un thread verrà assegnata la CPU solo se non c'è un thread a priorità più alta in attesa d'essere eseguito. Si può settare ed interrogare la priorità di un oggetto *Thread* con i metodi *priority=* e *priority*. Si noti che non è possibile settare la priorità di un thread prima che questo inizi la propria esecuzione. Un thread può, comunque, aumentare o diminuire la propria priorità come prima azione che esegue. Un thread eredita la propria priorità dal thread che l'ha generato. Il main thread inizialmente ha sempre priorità 0.

3.4.2 Preemption

Quando più thread aventi la stessa priorità competono per l'utilizzo della CPU, spetta allo scheduler decidere quando, e per quanto tempo, ogni thread possa rimanere in esecuzione. Esistono scheduler che permettono la preemption e altri che non la permettono. Se un compute-bound thread con un lungo tempo d'esecuzione (ad esempio, uno che non fa richieste di IO nel corso del blocco) è in esecuzione su uno scheduler non preemptive, gli altri thread correranno il rischio di starvation. Per evitare questa situazione, questo tipo di thread dovrebbe periodicamente chiamare *Thread.pass* per richiedere allo scheduler di assegnare la CPU ad un altro thread.

3.5 La gestione dei thread

Un thread di Ruby può essere in uno di cinque stati possibili. I due più interessanti stati possibili sono *runnable* e *sleeping*. Un thread è *runnable* se è attualmente in esecuzione, o se è pronto ad essere eseguito non appena ci sono risorse della CPU a sufficienza. Un thread è *sleeping* quando è in attesa per IO, o si è arrestato. Tipicamente i thread alternano questi due stati. Un thread può terminare normalmente o in modo anormale con un'eccezione. Per rappresentare queste due situazioni esistono due stati (anche se nelle nuovo versioni di Ruby verranno eliminati). Infine,

Tabella 3.1: Stato dei thread e valori restituiti

Stato del thread	Valore restituito
Runnable	"run"
Sleeping	"sleep"
Aborting	"aborting"
Terminated normally	False
Terminated with exception	Nil

c'è uno stato di transizione. Un thread che è stato "ucciso", ma che non ha ancora terminato la sua esecuzione, è nello stato di *aborting*.

3.5.1 Stati di interrogazione dei thread

La classe thread definisce svariati metodi d'istanza per testare lo stato di un thread. *alive?* restituisce true se un thread è *runnable* o *sleeping*. *stop?* restituisce true se un thread è in qualsiasi stato che non sia *runnable*. Infine, il metodo *status* restituisce lo stato del thread. Ci sono cinque possibili valori che verranno restituiti a seconda dei cinque possibili stati, riportati nella tabella seguente.

3.5.2 Alterazione dello stato: mettere in pausa, risvegliare e uccidere i thread

I thread vengono creati nello stato runnable. Un thread può mettersi in pausa da solo (entrando nello stato *sleeping*) chiamando il metodo *Thread.stop*. Questo è un metodo di classe che opera sul thread corrente, non esiste un metodo d'istanza equivalente, quindi un thread non può costringere un altro thread a mettersi in pausa. Chiamare *Thread.stop* è effettivamente la stessa cosa di chiamare *Kernel.sleep* senza argomenti: il thread si mantiene in pausa permanete (oppure nel frattempo si sveglia, come spiegato di seguito). Quindi se si chiama *Kernel.sleep* con un argomento i thread si mettono temporaneamente nello stato *sleeping*. In questo caso, si risvegliano automaticamente e ritornano nello stato runnable dopo (approssimativamente) il numero di secondi specificati passati. Anche chiamare metodi per l'IO all'interno del blocco causa l'attesa del thread finché l'operazione di IO non viene completata (in effetti, è l'attesa inerente alle operazioni di IO che rende il multithreading utile anche su sistemi composti da una singola CPU). Un thread che si è messo in pausa con *Thread.stop* o *Kernel.sleep* può risvegliarsi (anche se il tempo di attesa non è ancora scaduto) con i metodi d'istanza *makeup* e *run*. Entrambi i metodi fanno passare il thread dallo stato *sleeping* allo stato *runnable*. Il metodo *run* invoca anche lo scheduler. Questo determina il rilascio da parte del thread corrente della CPU, e potrebbe portare in esecuzione il thread appena risvegliato. Il metodo *makeup* risveglia il thread specificato senza cedere la CPU. Un thread può passare dallo stato runnable allo stato terminated semplicemente terminando il suo blocco o lanciando un'eccezione. Un altro modo per un thread di terminare normalmente è di chiamare *Thread.exit*. Un thread può far terminare forzatamente un altro thread invocando il metodo d'istanza *kill*. *terminate* e *exit* sono sinonimi per *kill*. Questo

metodo mette il thread "ucciso" nello stato terminated normally. Il metodo *kill!*, come *terminate!* e *exit!*, fa terminare brutalmente il thread. È possibile lanciare un'eccezione all'interno di un altro thread con il metodo d'istanza *raise*. Se il thread non può gestire l'eccezione che si è imposta ad esso, entrerà nello stato *terminated with exception*. Uccidere un thread è una cosa pericolosa da fare, in quanto non si hanno molte vie per sapere se il thread sta modificando lo stato di dati salvati sul sistema. Uccidere un thread con i metodi che hanno il punto esclamativo è ancora più pericoloso perché non si sa se così facendo si lasceranno file, socket o altre risorse aperte. Se un thread deve poter essere terminato a comando, è meglio controllarne periodicamente lo stato attraverso una variabile in modo da poterlo far terminare in modo sicuro quando la variabile assume un determinato valore.

3.6 Liste e Gruppi di thread

Il metodo *Thread.list* restituisce un array di oggetti *Thread* che rappresentano tutti thread *running* o *sleeping*. Quando un thread termina la sua esecuzione, viene rimosso dall'array. Tutti i thread apparte il main thread sono creati da un altro thread. I thread possono per questo motivo, essere organizzati in una struttura ad albero, in cui ogni thread ha un genitore ed un insieme di figli. La classe *Thread* non mantiene questa informazione, comunque, anche se i thread vengono considerati degli oggetti autonomi, sono subordinati al thread creatore. Se si vuole impostare qualche ordine in un sottoinsieme di thread, è possibile creare un oggetto *ThreadGroup* ed aggiungere thread ad esso:

```
0 group = ThreadGroup.new
1 3.times {|n| group.add(Thread.new { do_task(n) }}
```

I nuovi thread vengono inizialmente posti nel gruppo a cui appartiene il padre. Utilizzando il metodo d'istanza *group* è possibile sapere a quale *ThreadGroup* appartiene un thread, mentre usando il metodo list di *ThreadGroup* si ottiene un array contenente i thread del gruppo. Come il metodo di classe *Thread.list*, anche il metodo d'istanza *ThreadGroup.list* restituisce solo i thread che non sono ancora terminati. Si può utilizzare il metodo *list* per definire metodi operanti su tutti i thread di un gruppo. Ad esempio, si potrebbe scrivere un metodo che imposta la stessa priorità a tutti i membri del gruppo. La caratteristica della classe *ThreadGroup* che la rende più utile di un semplice array di thread è il metodo *enclose*. Quando un gruppo di thread viene "chiuso", i thread che ne fanno parte non possono venirne rimossi, come non è possibile aggiungere nuovi thread. I thread che fanno parte del gruppo possono però creare nuovi thread e questi entrano automaticamente a far parte del gruppo. Un *ThreadGroup* di questo tipo può essere utile quando si esegue codice Ruby poco affidabile sotto la variabile *$SAFE* e si vuole tener traccia dei thread generati dal codice.

3.7 Thread Exclusion e Deadlock

Se due thread condividono l'accesso agli stessi dati, e almeno uno di questi thread modifica tali dati, bisogna essere certi che nessuno possa vedere tali dati in uno stato inconsistente. Per questo è necessaria la **mutua esclusione**, di cui si parlerà in modo più approfondito nel capitolo seguente (v. par. 4.1.1). Ciascuno thread

che vuole accedere a dati condivisi deve prima eseguire un'operazione di *lock* su tali dati. Il *lock* è rappresentato da un oggetto *Mutex* (abbreviazione di mutua esclusione). Per fare il *lock* su un *Mutex*, è necessario invocare il suo metodo *lock*. Quando l'operazione che poteva creare inconsistenze è stata eseguita, si chiama il metodo *unlock*. Il metodo *lock* blocca quando chiamato il *Mutex* se esso è già nello stato *lock*, e non ritorna finché il chiamante non riceve effettivamente il *lock*. Se ciascun thread che deve far accesso ai dati condivisi fa il *lock* e l'*unlock* sul *Mutex* correttamente, nessun thread vedrà mai i dati in uno stato inconsistente. **Mutex** è una classe del core di Ruby 1.9 e fa parte della libreria standard *thread* in Ruby 1.8. Invece di utilizzare i metodi *lock* e *unlock* esplicitamente, è più comune usare il metodo synchronize e associargli un blocco. *synchronize* fa il *lock* sul *Mutex*, esegue il codice nel blocco, e fa l'*unlock* al termine, cosicché le eccezioni probabilmente verranno gestite correttamente.

3.7.1 Deadlock

Quando si utilizzano oggetti *Mutex* bisogna prestare attenzione nell'evitare deadlock, la condizione che si verifica quando tutti i thread sono in attesa che venga rilasciata una risorsa asseganta ad un altro thread. Poiché tutti i thread sono in stallo, non possono uscire dalla mutua esclusione. Un classico scenario in cui può verificarsi stallo è in presenza di allocazione parziale delle risorse. Si supponga di avere due thread e due oggetti *Mutex*. Il primo thread fa il *lock* sul Mutex 1 e poi attende di fare il *lock* sul Mutex 2. Nel frattempo, il secondo thread fa il *lock* sul Mutex 2 e poi cerca di farlo sul Mutex 1. Nessuno dei due thread può acquisire il lock che gli serve, così entrambi si bloccano per sempre:

```
0  # Classico deadlock: due thread e due lock
1  require 'thread'
2
3  m,n = Mutex.new, Mutex.new
4
5  t = Thread.new {
6    m.lock
7    puts "Thread t locked Mutex m"
8    sleep 1
9    puts "Thread t waiting to lock Mutex n"
10   n.lock
11 }
12
13 s = Thread.new {
14   n.lock
15   puts "Thread s locked Mutex n"
16   sleep 1
17   puts "Thread s waiting to lock Mutex m"
18   m.lock
19 }
20
21 t.join
22 s.join
```

Il modo per evitare queste spiacevoli situazione è di fare il *lock* delle risorse sempre nello stesso ordine (**allocazione gerarchica delle risorse**). Si osservi che lo stallo è possibile anche senza l'utilizzo di oggetti *Mutex*. Chiamare *join* su un thread che chiama *Thread.stop* provocherà lo stallo per entrambi i thread, anche se ci fosse un terzo thread in grado di svegliare il thread bloccato. Bisogna pertanto tenere a mente che alcune implementazioni di Ruby possono riconoscere semplici situazioni di stallo come questa e terminare con un errore, ma questo non è garantito.

3.8 Queue e SizedQueue

La libreria standard *thread* definisce le strutture dati *Queue* e *Sized-Queue*, specificatamente per programmi concorrenti. Queste implementano code thread-safe di tipo FIFO e seguono un modello di programmazione del tipo produttore/consumatore. Secondo questo modello, un thread produce valori di qualche tipo e li deposita in una coda con il metodo *enq* (enqueue) o con *push*. Un altro thread, "consuma" tali valori, rimuovendoli dalla coda con *deq* (dequeue) secondo i suoi bisogni. È possibile utilizzare per questo anche i metodi *pop* o *shift*. La caratteristica chiave della struttura *Queue* che la rende utilizzabile in programmi concorrenti è che il metodo deq si blocca in attesa se la coda è vuota finché il produttore non aggiunge un nuovo elemento. Le classi *Queue* e *SizedQueue* implementano la stessa API di base, ma la variante *SizedQueue* ha una dimensione massima. Se la coda è piena per inserire un nuovo valore è necessario attendere finché non si libera spazio in coda. Come per le altre collezioni di oggetti in Ruby, si può determinare il numero di elementi che contengono con i metodi *size* o *length*, e si può determinare se una coda è vuota con il metodo *empty?*. È necessario specificare la dimensione massima della coda di tipo *SizedQueue* quando si chiama *SizedQueue.new*. È però possibile interrogare o modificare la massima dimensione della coda anche dopo la sua creazione con i metodi *max* e *max=*.

3.9 Variabili di condizione e code

È importante notare che il metodo deq della classe *Queue* può andare in stallo. Normalmente, si pensa solo a situazioni di stallo dovute a metodi di IO (o per la chiamata dei metodi *join* o lock su *Mutex*). Nella programmazione concorrente, comunque, è necessario in alcuni casi avere un thread in attesa finché non si verifica una certa condizione (fuori dal controllo del thread). Nel caso della classe *Queue*, la condizione è che lo stato della coda non sia vuoto: se la coda è vuota, allora un thread consumatore dovrà attendere finche un thread produttore non chiami il metodo *enq* sulla coda. Una *ConditionVariable* rende più evidente il fatto che un thread possa rimanere in attesa del verificarsi di una condizione dovuta ad un altro thread. Come *Queue*, anche *ConditionVariable* fa parte della libreria standard *thread* e viene creata con *ConditionVariable.new*. Un thread viene posto in attesa sulla condizione con il metodo *wait*, mentre viene risvegliato con *signal* (*signal* risveglia il primo thread in fila in attesa del lock). È possibile risvegliare tutti i thread in attesa con *broadcast*. C'è un particolare pericoloso nell'utilizzo delle variabili di condizione: per far funzionare le cose correttamente, il thread in attesa deve passare un Mutex bloccato al metodo *wait*. Questo mutex viene temporaneamente sbloccato mentre il thread è in attesa, è verrà ribloccato quando il thread si risveglia. Di seguito, viene presentata una classe che a volte è utile. La classe *Exchanger* permette a due thread di scambiarsi alcuni valori arbitrari. Di seguito è riportato un esempio di utilizzo di tale classe:

```
0 require 'thread'
1
2 class Exchanger
3   def initialize
4     # Queste variabili conterranno i due valori da scambiare
5     @first_value = @second_value = nil
6     # Questo Mutex protegge l'accesso al metodo Exchange.
7     @lock = Mutex.new
8     # Questo Mutex ci permette di determinare se siamo il primo o il secondo
```

```
 9     # thread a chiamare il metodo Exchange.
10     @first = Mutex.new
11     # Questa ConditionVariable permette al primo thread di aspettare
12     # la chiamata del secondo thread.
13     @second = ConditionVariable.new
14   end
15
16   # Scambia questo valore con il valore passato dall'altro thread.
17   def exchange(value)
18     @lock.synchronize do       # Solo un thread alla volta può chimare questo
19                                # metodo
20       if @first.try_lock       # Se siamo il primo thread
21         @first_value = value   # Salva l'argomento del primo thread
22         # Quindi aspetta il secondo thread.
23         # Questo sblocca temporaneamente il Mutex mentre aspettiamo, così
24         # anche il secondo thread può chiamare il metodo
25         @second.wait(@lock)    # Aspetta il secondo thread
26         @first.unlock          # Pronto per il prossimo scambio
27         @second_value          # Restituisce il valore del secondo thread
28       else                     # Altrimenti, siamo il secondo thread
29         @second_value = value  # Salva il secondo valore
30         @second.signal         # Dice al primo thread che siam qui
31         @first_value           # Restituisce il valore del primo thread
32       end
33     end
34   end
35 end
```

3.10 I thread e il Garbage Collector

In Java, un oggetto *thread*, come qualsiasi altro oggetto, è un candidato per il *garbage collector* quando non esiste più una variabile che fa riferimento ad esso.

Il *garbage collector* non può liberare la memoria occupata da un thread se esso non ha terminato la propria esecuzione o se è stato "stoppato". Questo perché si continua ad avere il riferimento all'oggetto *TimerThread* anche dopo che è stato chiamato il metodo *stop()*. Perché tale *thread* possa essere rimosso, sarebbe necessario rimuovere manualmente il riferimento ad esso, dopo che il metodo *stop()* è stato chiamato. Comunque, questo servirebbe solo per liberare la memoria occupata dal *thread*. Il sistema rilascia automaticamente tutte le risorse impegnate dal thread nel corso della sua esecuzione, dopo che esso ha completato la sua esecuzione o se è stato invocato su di esso il metodo *stop()* anche se non cancelliamo i riferimenti all'oggetto.

Anche in Ruby è presente un *garbage collector*, che si comporta in modo simile a quello presente in Java: gli oggetti non sono candidati per il *garbage collector* se esiste ancora un riferimento ad essi. In alcune situazioni, risulta però utile una flessibilità maggiore. Per questo è possibile utilizzare quelle che vengono chiamate **weak reference**, che si comportano come i normali riferimenti all'oggetto, tranne per il fatto che permettono di spostare nel *garbage collector* l'oggetto in questione, anche se esiste ancora un riferimento a quest'ultimo. Se si cerca di accedere ad un oggetto, che non è più presente in memoria e a cui si faceva riferimento con una *weak reference*, viene lanciata l'eccezione *WeakRef::RefError*. Riportiamo nel seguito un esempio di codice:

```
0 require 'weakref'
1 # Generiamo un sacco di piccole stringhe.
2 # Probabilmente la prima finirà nel garbage collector
3 refs = (1..10000) maps {|i| WeakRef.new("#{i}")}
4 puts "L'ultimo elemento è #{refs.last}"
5 puts "Il primo elemento è #{refs.first}"
```

Che produce:

```
0 L'ultimo elemento è 10000
1 prog.rb:6:in '<main>': Invalid Reference - probably recycled
2  (WeakRef::RefError)
```

Il *GC module* mette a disposizione dei metodi per interagire col *garbage collector* di Ruby. Alcuni dei metodi messi a disposizione sono disponibili anche attraverso il modulo *ObjectSpace*.

Capitolo 4

Meccanismi per la sincronizzazione

Entriamo ora nel cuore dell'argomento. In questo capitolo verrà infatti approfondita la questione dell'interazione tra processi e verranno presentati i meccanismi di sincronizzazione che Ruby mette a disposizione per la gestione della concorrenza.

*Dopo un **approfondimento sulla mutua esclusione**, ci si sofferma sulle forme elementari con cui vengono rappresentati, in un sistema di elaborazione, i meccanismi di sincronizzazione e di scambio delle informazioni tra i processi. Vengono confrontate le diverse versioni di Ruby, prestando particolare attenzione alla versione 1.8, 1.9 e JRuby. Nel paragrafo 4.4 verranno messe a confronto le performance di un programma CPU-bound eseguito attraverso l'utilizzo di thread o processi, e verrà poi testato un programma IO-bound multi-thread. L'ultimo paragrafo introduce Fiber, Actors e la Eventmachine basata su Reactor, di cui si sente sempre più spesso parlare in ambito della gestione della concorrenza in Ruby.*

4.1 Interazione tra processi

I più semplici fenomeni di parallelismo che devono essere gestiti in un insieme di processi concorrenti sono fondamentalmente di due tipi:

- La mutua esclusione, quando scopo della sincronizzazione è di consentire ad un processo di operare su un insieme di dati (o risorse), comuni con altri processi, controllando l'interferenza;
- La cooperazione, quando i processi contribuiscono ad un obiettivo comune, per esempio quando un processo deve attendere dati che devono ancora essere prodotti da un altro processo.

4.1.1 Mutua esclusione

Il problema della mutua esclusione nasce quando più di un processo opera su una variabile comune (modificandola). Tale variabile viene detta **sezione critica**. La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle sezioni critiche non si sovrappongano nel tempo. Non viene però imposto nessun vincolo sull'ordine con il quale le operazioni vengono eseguite.

Detto in altri termini, le sezioni critiche corrispondenti nei vari processi rappresentano la zona mutuamente esclusiva, ovvero il meccanismo di sincronizzazione deve operare in modo che non più di un processo alla volta si trovi nella propria sezione critica.

Per ottenere una soluzione il più possibile generale, il protocollo che disciplina la sezione critica, oltre al vincolo fondamentale sull'accesso esclusivo, deve soddisfare le seguenti condizioni:

- un processo non attende quando chiede di entrare nella sezione critica se questa è libera;
- se la sezione critica viene liberata dal processo che la occupava e vi sono processi in attesa di entrarvi, ad uno di questi si deve concedere l'ingresso;
- se un processo P ha già 'chiesto' di entrare nella sezione critica, è fissato un limite massimo al numero di processi diversi da P che possono entrare nella sezione critica prima di P;
- la soluzione deve essere valida con qualunque velocità di evoluzione dei processi.

4.1.2 Cooperazione

Come esempio di cooperazione tra più processi, sarà presentato il problema che va sotto il nome di Produttore-Consumatore. Si tratta di un modello molto generale, applicabile a quei sistemi nei quali alcuni processi svolgono, in una determinata fase, la funzione di generatori di dati, e altri processi di consumatori/elaboratori di quei dati, quando disponibili.

In figura 4.1 viene riportata la rete di Petri rappresentante la cooperazione con risorse limitate tra un produttore e un consumatore.

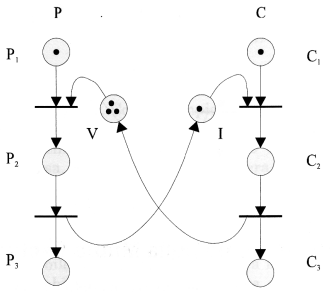

Figura 4.1: Cooperazione tra produttore e consumatore con risorse limitate

4.2 Thread e dipendenze dalla piattaforma

Sistemi operativi diversi implementano i thread diversamente, come le differenti implementazioni di Ruby (vedi 1.1.9) costruiscono i thread di Ruby sui thread del sistema operativo in modi differenti. Il linguaggio di programmazione Ruby e lo specifico modello dei thread utilizzato da una specifica implementazione di Ruby può creare una certa confusione. Attualmente esistono circa 11 differenti implementazioni di Ruby, che gestiscono i thread in maniera molto diversa tra loro. (Sfortunatamente,

non tutte queste implementazioni sono a tutti gli effetti stabili, ma è un campo di ricerca che sta portando a ottimi risultati.)

L'implementazione standard in C di **Ruby 1.8**, ad esempio, **utilizza un unico thread nativo su cui eseguire tutti i thread in Ruby**. Se i thread vengono implementati in questo modo, come avviene in Ruby 1.8, vengono chiamati green thread. In pratica, i green thread non vengono mai eseguiti in parallelo, anche se in presenza di CPU multi-core. Comunque, un numero qualsiasi di thread C (come ad esempio i thread POSIX) possono venir eseguiti in parallelo a thread Ruby, come le librerie esterne in C, o le estensioni in C della MRI, che creano i loro propri thread e che quindi possono essere eseguite in parallelo.

Ruby 1.9 è diverso: basato su YARV, **alloca un thread nativo per ciascun thread in Ruby**. YARV implementa i thread Ruby come thread POSIX o Windows NT, **ma utilizza un Global Interpreter Lock (GIL)** per assicurare che solo un thread Ruby alla volta possa essere eseguito. È comunque valido il discorso fatto per Ruby 1.8: parti di codice C possono venir eseguite in parallelo. In futuro, il GIL verrà sostituito da lock più flessibili, per permettere a più parti di codice di venir eseguite in parallelo.

JRuby, l'implementazione Java di Ruby, **mappa ciascun thread in Ruby in un thread in Java**. Comunque, l'implementazione ed il comportamento dei thread in Java dipende a sua volta dalla implementazione della Java Virtual Machine. Le moderne implementazioni di Java permettono processi paralleli su CPU multi-core.

IronRuby implementa i thread Ruby come thread nativi, che in questo caso sono thread CLR. Non viene imposto nessun lock addizionale (nessun GIL), quindi possono venire eseguiti in parallelo, se il CLR lo supporta.

I thread di **Ruby.Net** sono analoghi a quelli di IronRuby.

Rubinius implementa i thread Ruby come green thread all'interno della propria Virtual Machine. Rubinius non può (al momento) eseguire thread in parallelo, ma è possibile eseguire diverse istanze della VM in diversi thread POSIX in parallelo, all'interno di un processo Rubinius. Poiché i thread sono implementati in Ruby, vengono considerati come tutti gli altri oggetti, e possono essere inviati a differenti VM in differenti thread POSIX, permettendo in questo modo un'esecuzione concorrente. (Il modello qui descritto è lo stesso della BEAM Erlang VM usato per la concorrenza SMP.)

MacRuby sfrutta i thread nativi del sistema operativo.

Per quanto detto, se si vuole veramente eseguire thread in parallelo, è necessario tenere presente quale versione di Ruby si sta utilizzando e su quale sistema. Altrimenti, una possibile soluzione è di utilizzare il **parallelismo tra processi** anziché quello tra thread. La *Ruby Core Library* contiene il modulo *Process* con il metodo *Process.fork* che permette di creare un nuovo processo. Inoltre, la libreria standard contiene la libreria *Distributed Ruby (dRuby/dRb)*, che permette di eseguire codice Ruby distribuendolo tra diversi processi, non necessariamente in esecuzione sulla stessa macchina, ma anche attraverso una rete.

4.3 Global Interpreter Lock

Il multithreading è un modo per ottenere un'esecuzione concorrente. **In Ruby un'esecuzione concorrente non significa necessariamente più thread che operano in parallelo**. Infatti, in diverse versioni di Ruby, solo processi possono essere eseguiti in parallelo, non i thread.

Per capire perché ciò accade, è necessario tenere presente il funzionamento di Ruby al momento dell'esecuzione. Quando un'applicazione Ruby viene lanciata, un'istanza di un interprete Ruby viene a sua volta lanciata per tradurre il codice, costruisce un albero AST e poi esegue l'applicazione richiesta (tutto questo avviene fortunatamente in modo trasparente all'utente). Comunque, come parte di questo runtime, l'interprete crea anche un'istanza di un Global Interpreter Lock (GIL), che è la chiave della mancanza di concorrenza: **il GIL è un lock di mutua esclusione** creato dal thread dell'interprete del linguaggio di programmazione per permettere la condivisione sicura di codice tra thread. **Viene creato un unico GIL per ogni processo interprete**.

L'utilizzo di un Global Interpreter Lock in un linguaggio effettivamente limita la concorrenza all'interno di un singolo processo interprete composto da più thread (non c'è nessun aumento, o quasi, nella velocità di esecuzione del processo su una macchina con un multiprocessore).

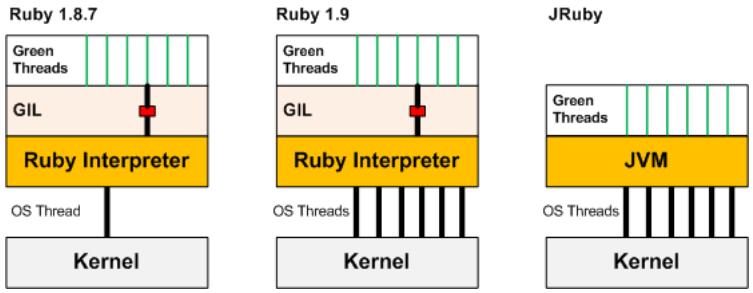

Figura 4.2: Confronto tra Ruby 1.8, Ruby 1.9 e JRuby

Per rendere il discorso più concreto, nel seguito verranno messi a confronto Ruby 1.8, 1.9 e JRuby (vedi fig. 4.2), di cui si è già accennato nella sezione precedente.

In **Ruby 1.8**, un singolo thread del sistema operativo viene allocato per l'interprete Ruby, viene istanziato un GIL e i thread Ruby sono green thread che vengono eseguiti uno alla volta. Non c'è modo per questo processo Ruby di ottenere dei vantaggi dalla presenza di processori multipli.

Ruby 1.9 è più 'promettente'. Mette a disposizione più thread nativi per l'interprete Ruby, ma comunque viene istanziato un GIL, o Giant VM Lock (citando i commenti del codice sorgente *thread.c*). L'interprete si protegge da solo contro codice non thread-safe, permettendo ad un unico thread per volta di venire eseguito. Questo modello di gestione dei thread è simile a quello di Python. Un singolo processo non può ottenere dei vantaggi da un processore multicore, poiché per avere un'esecuzione in parallelo è necessario avere più processi.

JRuby sfrutta la presenza di processori multipli. In seguito alla compilazione, il bytecode viene eseguito sulla JVM, e i thread Ruby vengono mappati su thread del sistema operativo, senza istanziare nessun GIL.

4.3.1 Inside Ruby's Interpreter: Threading in Ruby 1.9.1

In questa sezione si vuole esaminare più nel dettaglio come è stato implementato l'interprete di Ruby 1.9.1 (implementazione in C). In particolare si vuole mettere in luce come i thread in C/C++ (o in altri linguaggi) interagiscono con il codice Ruby e come vengono 'schedulati' i thread in modo che appaiano in esecuzione contemporaneamente.

Il codice che implementa i thread si trova nel file sorgente *thread.c*. Come già detto in precedenza, l'interprete Ruby utilizza un oggetto mutex globale (chiamato GIL o Global VM Lock) per limitare l'esecuzione simultanea dei thread. Si ha accesso al GVL attraverso l'espressione `th->vm->global_vm_lock`, in cui `th` è un puntatore verso qualsiasi thread valido all'interno dell'interprete, contenuto nella struttura `rb_thread_t`.

Considerando il livello dell'implementazione in C di Ruby è possibile distinguere due tipi di thread:

- i normali thread di Ruby, che ad alto livello appaiono come istanze della classe *Thread*. A livello dell'interprete sono associati alla struttura `rb_thread_t`.
- thread a basso livello, che nel seguito chiameremo C-thread. Tali thread non sono mai visibili a livello Ruby.

L'architettura di Ruby impone diverse limitazioni al comportamento di tali thread. **La limitazione più importante, ossia l'esecuzione esclusiva, viene applicata solo ai thread Ruby (quelli di alto livello)**.

Tale restrizione non è infatti imposta ai C-thread. Quindi, **quando un estensione per Ruby è scritta in C, essa può utilizzare più di un thread per volta**.

Bisogna però tener presente che **solo i thread Ruby possono interagire con gli oggetti Ruby**. Nessun C-thread può modificare, creare, cancellare e nemmeno accedere ad oggetti Ruby contenuti nel sistema. In realtà, ciò non è vero in assoluto, ma in questo testo non si vuole indagare in che modo sia possibile fare in modo che un thread C possa modificare un oggetto Ruby.

L'uso tipico dei C-thread nelle estensioni Ruby richiede che tutti i dati su cui un C-thread opera siano convertiti dal formato Ruby in strutture C prima che essi vengano passati al C-thread per l'elaborazione. Un discorso analogo vale per i risultati, che vengono convertiti in modo da renderli elaborabili unicamente da thread Ruby.

La variabile GVL è del tipo `rb_thread_lock_t`, e tale tipo di dato viene definito tramite `typedef` a partire dal tipo mutex. Il lock viene inizializzato e acquisito dal main thread dell'interprete Ruby. Se sono presenti nel codice Ruby da eseguire più di un Ruby thread, uno di questi ha il GVL, e gli altri thread possono essere in uno dei seguenti stati:

- in attesa del GVL per terminare una chiamata di acquisizione;
- in attesa di terminare un operazione che ha messo il thread in 'pausa': un'-operazione di IO, l'acquisizione del lock su un semaforo (diverso dal GVL) ecc.
- Un nuovo thread è stato inizializzato - prima che il GVL possa esseer acquisito qualche struttura interna potrebbe essere creata, allocando memoria, ecc. Tutto viene fatto a livello C, senza che nessun oggetto Ruby venga modificato.

– Qualche chiamata a livello di sistema viene eseguita dopo la fine di un'operazione di attesa e prima che il GVL sia acquisito. Questo è un breve momento in cui teoricamente più di un Ruby thread potrebbe essere eseguito contemporaneamente, ma nessun oggetto Ruby viene modificato durante questo veloce periodo.

Quando un thread Ruby sta facendo una chiamata ad una libreria che potenzialmente può bloccare il sistema, il thread rilascia sempre il GVL e dopo che la chiamata ha terminato la propria esecuzione, deve controllare i flag di interruzione (attraverso la macro RUBY_VM_CHECK_INT).

L'esecuzione dei thread Ruby viene gestita dallo scheduler del sistema operativo. **A livello dell'interprete lo scheduling è cooperativo, poiché i thread volontariamente rilasciano il GVL. Questa cooperazione non è però visibile a livello del codice Ruby. Per il programma lo scheduling sembra di tipo preemptive**; al programmatore non è richiesto di preoccuparsi di come vengono schedulati i thread: anche i thread privi di lunghe operazioni di IO vengono regolarmente 'schedulati'.

Lo scheduling dei thread implica il passaggio regolare tra un thread e l'altro. Nell'interprete Ruby questo viene ottenuto chiamando la funzione `rb_thread_schedule` periodicamente.

```
0  void
1  rb_thread_schedule(void)
2  {
3  thread_debug("rb_thread_schedule\n");
4  if (!rb_thread_alone()) {
5  rb_thread_t *th = GET_THREAD();
6
7  thread_debug("rb_thread_schedule/switch start\n");
8
9  rb_gc_save_machine_context(th);
10 native_mutex_unlock(&th->vm->global_vm_lock);
11 {
12 native_thread_yield();
13 }
14 native_mutex_lock(&th->vm->global_vm_lock);
15
16 rb_thread_set_current(th);
17 thread_debug("rb_thread_schedule/switch done\n");
18
19 RUBY_VM_CHECK_INTS();
20 }
21 }
```

La funzione `rb_thread_schedule` viene spesso utilizzata nel codice che implementa Ruby. Ad esempio, la classe *Thread.pass* si limita a chiamare tale funzione. Per raggiungere l'illusione di "preemptiveness" nei programmi Ruby, l'interprete Ruby assicura che la funzione `rb_thread_schedule` sia chiamata in modo regolare. Questo viene implementato con l'aiuto di un thread che fa da timer e utilizzando una sorta di meccanismo di interruzione. Ogni 10 microsecondi il timer-thread imposta un interrupt flag. Alla fine di ogni chiamata a un metodo, prima che il risultato della chiamata sia restituito al chiamante, l'interprete Ruby controlla quali flag di interruzione sono impostati (attraverso la macro `RUBY_VM_CHECK_INTS`). Se vengono trovati bit impostati, viene chiamata la funzione `rb_thread_execute_interrupts`, che ricorda lo stato dei flag e li può cancellare. Se è stata impostata l'interruzione del timer, allora viene eseguito il thread per lo scheduling con la chiamata a *rb_thread_schedule*.

```
0
1  #define RUBY_VM_CHECK_INTS_TH(th) do { \
2      if (UNLIKELY(th->interrupt_flag)) { \
3  rb_thread_execute_interrupts(th); \
4  } \
5  } while (0)
6
```

```
7 #define RUBY_VM_CHECK_INTS() \
8 RUBY_VM_CHECK_INTS_TH(GET_THREAD())
```

Il timer-thread nominato sopra è un C-thread che viene eseguito in background. Esso esegue un loop infinito: dorme per un tempo prestabilito e dopo setta il flag d'interruzione timer per il thread Ruby che viene considerato quello corrente in quel momento. **C'è sempre un thread Ruby corrente, formalmente quello che è in possesso del GVL**.

4.3.2 Parallelismo tra processi

Le implicazioni generate dal GIL sono sorprendenti all'inizio, ma esiste una soluzione al problema: invece di pensare ai thread, è necessario pensare a come sia possibile suddividere l'applicazione in diversi processi. Per fare questo, che comunque non sempre è sufficiente per risolvere il problema, è necessario:

- Partizionare il lavoro, o suddividere l'applicazione;
- Aggiungere una *communications/work queue*;
- Avviare, o eseguire più istanze dell'applicazione.

Non dovrebbe sorprendere, che molte delle applicazioni scritte in Ruby hanno adottato questa strategia.

Creazione di Processi

Per ottenere un'esecuzione concorrente utilizzando processi multipli è necessario utilizzare la funzione *fork* o il suo sinonimo *Process.fork*. Il modo più semplice di utilizzare tale funzione è con un blocco:

```
0 fork {
1   puts "Hello from the child process: #$$"
2 }
3 puts "Hello from the parent process: #$$"
```

Quando si utilizza un codice come questo, il processo originale continua l'esecuzione del codice che appare dopo il blocco e il nuovo processo esegue il codice nel blocco.

Quando si invoca *fork* senza un blocco, si comporta diversamente. Nel processo padre, la chiamata a *fork* restituisce un intero che è l'identificatore di processo (process ID) del processo figlio appena creato. Nel processo figlio, la stessa chiamata a *fork* restituisce *nil*. Quindi, il codice precedente può essere riscritto come:

```
0 pid = fork
1 if (pid)
2   puts "Hello from parent process: #$$"
3   puts "Created child process #{pid}"
4 else
5   puts "Hello from child process: #$$"
6 end
```

È necessario tenere presente la **differenza fondamentale tra processi e thread: i processi non condividono tra loro la memoria**. Quando si invoca *fork*, il nuovo processo viene creato come copia esatta del processo padre. Qualsiasi modifica allo stato del processo (attraverso la modifica o la creazione di oggetti) vengono fatti all'interno dello spazio di memoria del processo. **Il processo figlio non può**

modificare le strutture dati del processo padre, come non può avvenire il viceversa.

Se è necessario che il processo padre e quello figlio comunichino tra loro, bisogna usare *open*, e passare "| −" come primo argomento. Ciò apre una **pipe** verso il processo appena creato con *fork*. La chiamata a *open* restituisce al blocco del processo figlio *nil*, mentre al blocco del processo padre un oggetto per l'IO. Leggere da questo oggetto restituisce i dati scritti dal figlio, mentre modificare l'oggetto attraverso la scrittura di dati rende tali dati disponibili in lettura per lo standard input del processo figlio. Ad esempio:

```
0  open("|-", "r+") do |child|
1    if child
2      # This is the parent process
3      child.puts("Hello child")        # Invia al figlio
4      response = child.gets            # Leggi dal figlio
5      puts "Child said: #{response}"
6    else
7      # This is the child process
8      from_parent = gets               # Leggi dal padre
9      STDERR.puts "Parent said: #{from_parent}"
10     puts("Hi Mom!")                  # Invia al padre
11   end
12 end
```

La funzione *Kernel.exec* è utile in combinazione con la funzione *fork* o il metodo *open*. È possibile inviare un comando arbitrario alla shell del sistema operativo attraverso il metodo ' oppure attraverso funzioni di sistema. Entrambi i metodi sono però sincroni: non si ritorna dal metodo finché esso non ha completato la propria esecuzione. Se si vuole eseguire un comando del sistema operativo in un processo separato, è necessario prima utilizzare *fork* per creare un processo figlio, e poi chiamare *exec* nel figlio per eseguire il comando. Una chiamata a *exec* non ritorna mai: sostituisce il processo concorrente con un nuovo processo. Gli argomenti per *exec* sono gli stessi che vengono usati per il sistema. Se ne è presente solo uno, viene gestito come un comando per shell. Se ne sono presenti diversi, allora il primo identifica l'eseguibile da invocare, e gli altri diventano gli "ARGV" per l'eseguibile:

```
0  open("|-", "r") do |child|
1    if child
2      # This is the parent process
3      files = child.readlines   # Leggi l'output del figlio
4      child.close
5    else
6      # Questo è il processo figlio
7      exec("/bin/ls", "-l")     # Esegui un altro eseguibile
8    end
9  end
```

Il modulo *Process* mette a disposizione diversi metodi per lavorare coi processi.

In questo approccio alla concorrenza, basato su processi anziché thread, è necessario tenere presente che **la creazione di molti processi può portare ad un esaurimento della memoria disponibile**. È pertanto necessario considerare a seconda dell'hardware a disposizione e delle prestazioni che si vogliono ottenere quale strategia tra quelle che il linguaggio mette a disposizione sia la più adatta. In certi casi è conveniente l'utilizzo dei fiber.

4.4 Thread e attività CPU o IO bound

Il codice Ruby e i risultati dei test riportati in questa sezione sono stati tratti dall'articolo "Concurrency, Real and Imagined, in MRI; Threads" di Kirk Haines, reperibile sul suo blog sul sito della Engine Yard.

Come detto in precedenza, i green thread di Ruby 1.8 non portano a dei vantaggi per task CPU bound.

```ruby
#cpu_bound_threads.rb

require 'benchmark'
threads = []
thread_count = ARGV[0].to_i
iterations = ARGV[1].to_i
increment = iterations / thread_count.to_f
sum = 0

Benchmark.bm do |bm|
  bm.report do
    thread_count.times do |counter|
      threads << Thread.new do
        my_sum = 0
        queue = (1 + (increment * counter).to_i)..(0 + (increment * (counter + 1)).to_i)
        queue.each do |x|
          my_sum += x
        end
        Thread.current[:sum] = my_sum
      end
    end

    threads.each {|thread| thread.join; sum += thread[:sum]}

    puts "The sum of #{iterations} is #{sum}"

  end
end
```

Questo semplice programma considera un grande intervallo di numeri, li divide in intervalli più piccoli, e passa ciascun intervallo più piccolo a un thread che calcola la somma dei numeri che gli sono stati assegnati. I risultati ottenuti da ciascun thread vengono poi sommati insieme per arrivare al risultato finale.

Tutti gli esempi riportati in questa sezione sono stati eseguiti su una macchina Linux a 8 core. I risultati dei test che vengono riportati sono una media di 100 esecuzioni per ciascun input.

Tabella 4.1: Risultato dei test effettuati utilizzando thread per un'attività CPU-bound

# thread	50000 iterazioni	500000 iterazioni	5000000 iterazioni
1	0.01730298	0.17149276	1.70610744
2	0.01724724	0.17179465	1.70557474
4	0.01729293	0.17181384	1.70570264
8	0.01741591	0.17210276	1.71201153

Come si può notare dai numeri riportati, **i thread di MRI 1.8 non aiutato affatto le performance di un applicazione CPU bound. Infatti, dai numeri è possibile notare un piccolo ma misurabile overhead dovuto alla gestione dei thread**.

Si considera ora cosa accade utilizzando processi per svolgere un compito analogo al precedente. Riportiamo nel seguito il codice modificato:

```ruby
#cpu_bound_processes.rb

require 'benchmark'
processes = []
process_count = ARGV[0].to_i
iterations = ARGV[1].to_i
increment = iterations / process_count.to_f
sum = 0

def in_subprocess
  from_subprocess, to_parent = IO.pipe

  pid = fork do
    from_subprocess.close
```

```
14      r = yield
15      to_parent.puts [Marshal.dump(r)].pack("m")
16      exit!
17    end
18
19    to_parent.close
20    [pid,from_subprocess]
21  end
22
23  def get_result_from_subprocess(pid, from_subprocess)
24    r = from_subprocess.read
25    from_subprocess.close
26    Process.waitpid(pid)
27    Marshal.load(r.unpack("m")[0])
28  end
29
30  Benchmark.bm do |bm|
31    bm.report do
32      process_count.times do |counter|
33        processes << in_subprocess do
34          my_sum = 0
35          queue = (1 + (increment * counter).to_i)..(0 + (increment * (counter + 1)).to_i)
36          queue.each do |x|
37            my_sum += x
38          end
39          my_sum
40        end
41      end
42
43      processes.each {|process| sum += get_result_from_subprocess(*process)}
44
45      puts "The sum of #{iterations} is #{sum}"
46
47    end
48  end
```

In questo esempio vengono utilizzate pipe di IO per mandare i dati dal processo padre a quelli figli, e per ricevere indietro dati dai figli.

Come prima, i test sono stati fatti su una macchina Linux a 8 core, con 100 esecuzioni per ciascun test. Il programma è equivalente alla versione multithread presentata in precedenza, cambia solo quel tanto che basta per permettere di utilizzare il modello a processi multipli.

Tabella 4.2: Risultati dei test effettuati utilizzando processi per un'attività CPU-bound

# processi	50000 iterazioni	500000 iterazioni	5000000 iterazioni
1	0.01805432	0.17199047	1.70812685
2	0.0098329	0.08675517	0.85509328
4	0.00609409	0.0446612	0.43100698
8	0.00847991	0.05346145	0.25621009

Osservando i risultati, si può notare una stranezza: il tempo aumeta sia per le 50000 e le 500000 iterazioni effettuate su 8 processi rispetto a quelle effettuate su 4 processi.

È possibile dare una spiegazione a questo fatto: i processi sono, in molti casi, un buon modo per gestire la concorrenza, ma sono strutture pesanti. Linux utilizza una **semantica copy-on-write** quando crea un processo. Questo significa che non duplica a tutti gli effetti lo spazio degli indirizzi del nuovo processo finché esso non viene modificato. Quindi, duplica quello che cambia. Questo comporta la **possibilità, in Linux, di creare nuovi processi velocemente**.

MRI 1.8 non è molto efficiente con la sematica copy-on-write, a causa del funzionamento del garbage collector. Quando il garbage collector scorre lo spazio dedicato agli oggetti per ricercare oggetti da raccogliere, esamina ogni oggetto nello spazio degli indirizzi. Se un processo è stato creato con la semantica copy-on-write,

questo porta il kernel a copiare tutte le pagine dedicate ad esso. Ciò richiede tempo, e quindi si perdono i benefici derivati dalla creazione veloce di processi.

In tutte le forme di concorrenza, bisogna sempre fare un **bilancio tra i benefici portati dalla suddivisione del lavoro e gli svantaggi dovuti all'overhead causato da tale suddivisione**.

Questi primi due esempi rappresentavano entrambi problemi CPU bound. Comunque, molti problemi reali non sono CPU bound, ma IO bound. Poiché i problemi IO bound hanno latenze imposte dall'esterno del programma stesso, i problemi IO bound possono essere un esempio eccellente di come l'utilizzo dei green thread possa portare ad un aumento delle prestazioni.

Si consideri il programma seguente basato su thread:

```
0  #io_bound_threads.rb
1
2  require 'net/http'
3  require 'thread'
4  require 'benchmark'
5
6  def get_data(url)
7    tries = 0
8    response = nil
9    if /^http/.match(url)
10     m = /^http:\/\/([^\/]*)(.*)/.match(url)
11     site = m[1]
12     path = m[2]
13     begin
14       http = Net::HTTP.new(site)
15       http.open_timeout = 30
16       http.start {|h| response = h.get(path)}
17     rescue Exception
18       tries += 1
19       retry if tries < 5
20     end
21   end
22   response.kind_of?(Array) ? response[1] : response.respond_to?(:body) ? response.body : ''
23 end
24
25 mutex = Mutex.new
26 signal = ConditionVariable.new
27 thread_count = ARGV[0].to_i
28 fetches = ARGV[1].to_i
29 url = ARGV[2]
30 threads = []
31 count = 0
32 active_threads = 0
33
34 Benchmark.bm do |bm|
35   bm.report do
36     while count < fetches
37       while count < fetches && active_threads < thread_count
38         mutex.synchronize do
39           active_threads += 1
40           count += 1
41         end
42         Thread.new do
43           get_data(url)
44           mutex.synchronize do
45             active_threads -= 1
46             threads << Thread.current
47             signal.signal
48           end
49         end
50       end
51
52       mutex.synchronize do
53         signal.wait(mutex)
54       end
55       while th = threads.shift
56         th.join
57       end
58     end
59   end
60 end
```

Questo script fa diverse richieste HTTP. Per semplicità, possiamo dire che continua a fare la stessa richiesta in continuazione, ma può facilmente venir modificato per far si che consideri una lista di URL e per far qualcosa di utile con i dati restituiti. Utilizza i thread in un modo leggermente più sofisticato rispetto al programma precedente:

tiene traccia del lavoro che viene assegnato ai thread generati e aspetta che tutti i thread abbiano completato la loro esecuzione.

La tabella seguente mostra i tempi necessari per l'esecuzione. L'URL utilizzato non era locale alla macchina di test. Ciascuna esecuzione utilizza il numero di thread indicato e vengono confrontati i tempi d'esecuzione per un URL con una risposta della rete di 35 richieste al secondo e per un URL con una velocità di risposta di 3 richieste al secondo, 400 volte. Il tutto è stato ripetuto 100 volte. I numeri riportati sono una media dei valori ottenuti di volta in volta.

Tabella 4.3: Risultati dei test effettuati utlizzando thread per attività IO-bound

# thread	35 risposte/secondo	3 risposte/secondo
1	6.53462668	61.1016239
2	3.34861606	30.4514539
5	1.38942396	12.1620945
10	0.72804622	6.0968646
20	0.47964698	3.0411382

Basta un'occhiata a questi numeri per capire che **i thread in Ruby portano ad un netto miglioramento delle performance per un'attività di tipo IO bound**. La relazione tra numero di thread e riduzione di tempo per completare l'attività non è lineare. Il miglioramento nel tempo d'esecuzione è più lineare ed evidente per richieste lente, perché viene speso più tempo in attesa dei dati di IO.

Sebbene nelle applicazioni IO bound l'utilizzo dei thread possa portare a miglioramenti evidenti, **bisogna tenere presente che è possibile che un estensione scritta in C di Ruby possa prendere il controllo del processo e impedire l'esecuzione di altri thread**. È possibile scrivere estensioni in C che non si comportino in questo modo, ma molte sono scritte così. Un esempio utile è quello dell'interazione con un database. Si potrebbe ragionevolmente pensare a una query a un database come ad un'attività di IO - quello che il processo Ruby fa è inviare una richiesta al DB e attendere la risposta. Comunque, molte delle librerie che interagiscono con i DB sono implementate come estensioni in C, e molte non lavorano bene in presenza di thread. Una delle più utilizzate è *Mysql-Ruby*. Tale libreria blocca Ruby in attesa del risultato di una query che richiede lunghi tempi per la risposta. Questo significa che l'intero processo in esecuzione rimane in attesa finché il codice C non termina la propria esecuzione. Al contrario, *Ruby-PG*, il driver per Postgres, permette il cambiamento del contesto all'interno di *pgconn_block()*, la funzione che fa le chiamate al database, consentendo ad altri thread di MRI 1.8 di essere eseguiti durante una query che richiede tempi lunghi.

4.5 Oltre ai thread

Come già accennato in precedenza, Ruby mette a disposizione altri meccanismi per la sincronizzazione, oltre l'utilizzo di thread e processi. In particolare, in questa sezione si vuole accennare brevemente alle potenzialità dei fiber, dell'utilizzo degli actor e dei nuovi sviluppi riguardanti la Eventmachine basata su Reactor.

4.5.1 Fiber

Con la versione 1.9 di Ruby i Fiber sono diventati parte integrante del linguaggio. A differenza di una funzione, che ha un punto d'ingresso ed uno d'uscita ben definiti, l'esecuzione di un Fiber può essere arbitrariamente sospesa e ripresa dallo stesso punto. Anche se a volte causa di errori di difficile individuazione, i fiber hanno grandi potenzialità: il loro utilizzo permette di preservare lo stato senza che il programmatore debba preoccuparsene. Per loro natura, l'utilizzo di Fiber implica uno scheduling di tipo cooperativo. Ciò può aumentare la complessità del programma in quanto è il programmatore che impone a chi spetta l'utilizzo della CPU.

4.5.2 Actors

Il modello basato su Actor è un modello matematico per computazioni concorrenti che utilizza gli "actor" come la primitiva universale per la concorrenza: in risposta a un messaggio che viene ricevuto, un actor può attuare delle decisioni locali, creare nuovi actor, mandare più messaggi, e determinare come rispondere a messaggi futuri.

Per capire meglio il funzionamento degli actor, può essere utile la spiegazione seguente tratta dalla pagina di "Revactor's Philsophy":

''The basic operation of an Actor is easy to understand: like a thread, it runs concurrently with other Actors. However, unlike threads it is not pre-emptable. Instead, each Actor has a mailbox and can call a routine named 'receive' to check its mailbox for new messages. The 'receive' routine takes a filter, and if no messages in an Actor's mailbox matches the filter, the Actor sleeps until it receives new messages, at which time it's rescheduled for execution.

Well, that's a bit of a naive description. In reality the important part about Actors is that they cannot mutate shared state simultaneously. That means there are no race conditions or deadlocks because there are no mutexes, conditions, and semaphores, only messages and mailboxes.''

4.5.3 Eventmachine

La Eventmachine di Ruby si basa su Reactor, un modello per la programmazione concorrente per la gestione di richieste di servizio spedite in modo concorrente a un server da uno o più client. Il server deve gestire le richieste in entrata distribuendole poi al gestore associato alla determinata richiesta.

Tutti i sistemi basati su Reactor sono single-threaded per definizione, ma potrebbe esistere un sistema in ambiente multi-threaded.

Può essere utile citare la somiglianza tra la Eventmachine di Ruby e Node.js di JavaScript.

Esula dagli scopi di questo testo approfondire oltre tali argomenti.

Capitolo 5

Costrutti per la sincronizzazione

Alla luce di quanto detto nei capitoli precedenti, verrà qui introdotto il concetto di ***semaforo*** *e successivamente, si considererà la possibilità di implementare in Ruby il costrutto di* ***regione critica****. Ci si soffermerà sul costrutto di* ***monitor****, che rende più facile la leggibilità dei programmi concorrenti e più semplice la verifica della loro correttezza. Infine, si farà un breve accenno al* ***rendez-vous*** *in Ruby.*

Nel capitolo successivo tali argomenti verranno approfonditi mediante esempi.

5.1 Semaforo

Il **Semaforo** è una soluzione proposta da Dijkstra al problema dell'accesso ad una zona critica. Ad un semaforo (binario, a due valori) sono associati una variabile logica s, che memorizza lo stato del semaforo, e due operazioni **wait** e **signal** (**P** e **V**: iniziali degli equivalenti termini olandesi in onore di Dijkstra). Le due operazioni, logicamente indivisibili, stanno ad indicare rispettivamente l'operazione di attesa ad un semaforo e quella di indicazione di via libera e formalizzano il comportamento descritto in fig. 5.2.

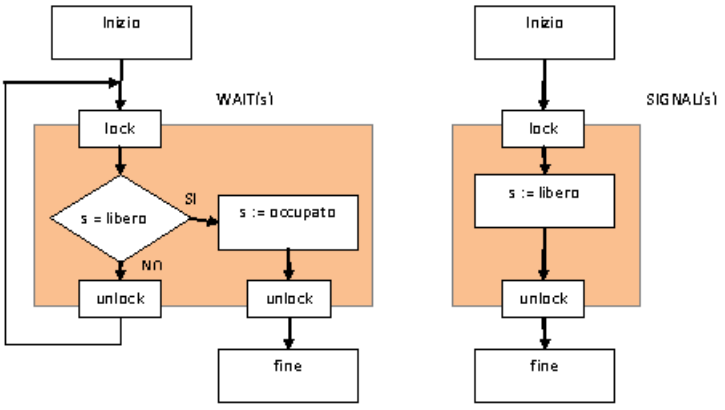

Figura 5.1: Schema per la realizzazione delle operazioni di wait e signal su un semaforo

I processi possono agire sul semaforo solo con le due operazioni wait e signal prima indicate. Quando un processo ha bisogno di entrare in una sezione critica, per prima cosa interroga il semaforo appositamente inserito come regolatore del traffico

di tale sezione critica. In base allo stato del semaforo, il processo può attendere (se il semaforo è 'rosso') o 'oltrepassarlo' lasciando traccia della sua presenza (se il semaforo è 'verde'). Modificando il semaforo e rendendolo occupato impedisce a qualsiasi altro processo di oltrepassarlo: questo comportamento è ottenuto dalla procedura wait. Quando il processo abbandona la zona critica dovrà provvedere ad abilitare il semaforo corrispondente, utilizzando la procedura signal che ha il compito di riportare a libero il valore del semaforo.

L'indivisibilità delle operazioni è la garanzia del corretto funzionamento del meccanismo.

5.1.1 Lock e Unlock

In una macchina a singolo processore, i processi concorrenti non possono sovrapportsi: possono solamente alternarsi; pertanto un processo continua l'esecuzione fino a quando non chiama un servizio del sistema operativo o viene interrotto. Quindi, per garantire la mutua esclusione, è sufficiente evitare che un processo venga interrotto: questo può essere realizzato con apposite primitive del kernel di sistema per abilitare e disabilitare gli interrupt. Se la sezione critica non può essere interrotta, la mutua esclusione è garantita. Nei processori che prevedono più livelli di interruzione, per l'operazione di lock può essere sufficiente porre il processore ad un livello L sufficientemente elevato e per unlock ad un livello l<L sufficientemente basso.

In generale, questo approccio ha un prezzo molto alto: l'efficienza può peggiorare perché il processore non può alternare i programmi liberamente. Un altro problema è che tale approccio non funziona in un sistema multiprocessore: quando il sistema è composta da vari processori, due o più processi possono essere eseguiti contemporaneamente, e in tal caso disattivare gli interrupt non garantisce la mutua esclusione.

In un sistema a più processori con memoria comune le due operazioni lock e unlock, oltre a quelle che si manifestano all'interno del medesimo processore, devono evitare possibili interferenze tra porcessori distinti. Il meccanismo legato alla disabilitazione delle interruzioni non è in questo caso sufficiente per garantire la indivisibilità delle istruzioni, dato il funzionamento in parallelo dei processori, e quindi la possibilità di accedere contemporaneamente a celle di memoria comuni, a meno che la disabilitazione stessa, 'comandata' da una CPU, non si potesse propagare anche alle altre CPU. L'inefficienza che si introdurrebbe nell'ingresso ad una sezione critica a causa della trasmissione di questo comando alle altre CPU rende questa soluzione impraticabile.

A livello hardware, l'accesso ad una locazione di memoria esclude qualunque altro accesso alla stessa locazione; basandosi su questo fatto, i progettisti hanno porposto varie istruzioni macchina che effettuano due azioni in modo atomico, come lettura e scrittura, o lettura e test, di una locazione di memoria in un solo ciclo di istruzione. Poiché queste azioni vengono effettuate in un solo ciclo, non sono soggette ad interferenze con altre istruzioni. Pertanto, in un sistema multiprocessore, il supporto da parte dell'hardware delle CPU è fornito da istruzioni simili a quelle indicate di seguito:

```
0 EXCHANGE register , memory
1 TEST AND SET memory
```

L'istruzione *exchange* scambia il contenuto di un registro e di una cella in memoria in un solo ciclo. L'istruzione *test and set*, che verrà indicata con TAS(memory), provvede in un unico ciclo a copiare il contenuto di una locazione di memoria in un registro e ad assegnare il valore logico vero alla stessa locazione.

Per la realizzazione di lock e unlock in un ambiente multiprocessore si analizzerà la soluzione che adotta l'istruzione TAS (l'estensione all'altro caso è banale). Si supponga che TAS, ad alto livello, si comporti come una vera e propria funzione che restituisce il valore di una locazione di memoria e la pone comunque al valore vero; si può osservare che, se in precedenza la variabile è vera, la *test and set* non ha alcun effetto. In un linguaggio pseudo Pascal la primitiva lock è realizzata da una macro di *busy waiting* centrata sulla TAS. L'operazione termina solo se la variabile è falsa. La realizzazione di lock e unlock è la seguente:

```
0 macro lock(x): while TAS(x) do end;
1    {esegui ciclo nullo se x è occupata}
2
3 macro unlock(x): reset(x);
4    {rendi x libera}
```

Nella lock l'istruzione TAS(x) viene ripetuta finchè la variabile x non viene trovata **libera** (cioè al valore falso), ma nello stesso istante in cui è valutata libera, con un unico ciclo di memoria, viene posta occupata (cioè al valore vero); ed ecco realizzata l'indivisibilità.

L'uso di un'istruzione macchina speciale come TAS per garantire la mutua esclusione ha vari vantaggi:

- Si può applicare a qualunque numero di processi, su un singolo processore o su un multiprocessore con memoria condivisa.
- È semplice e quindi facile da verificare.
- Si può usare per fornire più di una sezione critica: ciascuna sezione avrà una propria variabile.

Ci sono però anche degli svantaggi importanti:

- Bisogna usare la tecnica dell'attesa attiva, quindi mentre un processo aspetta di avere accesso alla sezione critica, usa il tempo di esecuzione del processore.
- È possibile che si verifichi starvation: quando un processo abbandona la sezione critica e ci sono vari processi in attesa, la scelta del processo da attivare è arbitraria, quindi è possibile che un processo debba aspettare per un tempo illimitato.
- È possibile che si verifichi uno stallo; si consideri il caso di un singolo processore, in cui un processo P1 esegue l'istruzione speciale TAS (o exchange) ed entra nella sezione critica; dopo di che viene interrotto per concedere il processore a P2, che ha una priorità più alta. Se P2 tenta di accedere alla stessa risorsa di P1, riceverà un rifiuto a causa del meccanismo di mutua esclusione, così entrerà in un ciclo di attesa attiva. Comunque P1 non verrà mai riattivato perchè la sua priorità è più bassa di quella di P2, che è attivo.

5.1.2 Semaforo binario

La classe *Mutex* implementa un semplice semaforo binario. Mette a disposizione i metodi d'istanza:

- *lock*: fa il lock su un mutex. Se il mutex è già occupato da un altro thread sospende il thread corrente. Se il mutex è occupato dal thread corrente si verifica un *ThreadError*.
- *locked?*: restituisce lo stato corrente del mutex.
- *sleep(time — nil*: Rilascia il corrente lock sul mutex, attende per *time* secondi (o per sempre, se viene passato *nil* come argomento), e poi fa nuovamente il lock sul mutex. Restituisce il numero di secondi di attesa totali.
- *synchronize*: acquisisce il lock del mutex, esegue il blocco associato al metodo, e dopo rilascia il lock. Restituisce l'ultimo valore assunto dal blocco.
- *try_lock*: Se il mutex non è occupato, lo occupa e restituisce *true*. Altrimenti, restituisce *false*. (In pratica, *try_lock* è uguale a *lock*, solo che non aspetta che il mutex si liberi).
- *unlock*: fa l'unlock del mutex, che deve essere stato occupato precedentemente dal thread corrente.

La libreria *mutex_m* propone una variante della classe *Mutex* che permette ai metodi di tale classe di venir 'mixati' in un altro oggetto. Il modulo *mutex_m* definisce metodi che corrispondono a quelli della classe *Mutex*, ma col prefisso *mu_*. Questi metodi sono *alias* degli originali appartenenti alla classe *Mutex*.

5.1.3 Semaforo numerico

Come semplice estensione dei semafori binari è possibile realizzare i cosiddetti **semafori numerici** (o **a più valori**), che risultano uno strumento più completo per risolvere i problemi di cooperazione tra processi. Per definire un semaforo ad N valori si utilizza, al posto di una variabile logica, una variabile s intera non negativa che abbia un opportuno valore iniziale. L'operazione di wait sul semaforo decrementa il valore di s di una unità se s>0, altrimenti sospende il processo. L'operazione di signal produce invece il risveglio di un processo se ne è presente almeno uno nella coda del semaforo (in questo caso il valore di s è necessariamente 0), o altrimenti l'incremento di una unità della variabile s.

I semafori numerici corrispondono in Ruby agli oggetti *Queue* e *Sized-Queue*. L'operazione di *push* corrisponde infatti al signal sul semaforo, mentre *pop* corrisponde al wait.

5.1.4 Semaforo privato

In problemi di sincronizzazione complessi, le condizioni per cui un processo può evolvere (**condizione di sincronizzazione**) possono dipendere da molteplici variabili. In tali situazioni, non è possibile l'utilizzo di un singolo semaforo. La condizione di sincronizzazione dipende generalmente dal valore assunto da un insieme di variabili comuni a più processi.

Un **semaforo** è detto **privato** ad un particolare processo (o ad un insieme di processi), quando quel processo (i processi dell'insieme) è l'unico (sono gli unici) ad essere abilitato(i) ad eseguire l'operazione di wait su di esso. Il processo (i processi) *proprietario(i)* del semaforo lo utilizza(no) come meccanismo di **autosospensione** se necessaria in base alla verifica sulla condizione di sincronizzazione. L'oper-

azione di signal, invece, può essere eseguita da chiunque, compreso il(i) processo(i) proprietario(i).

È importante tenere presente che l'analisi della condizione di sincronizzazione debba essere fatta in una sezione critica e l'attesa avvenga sempre esternamente alla stessa sezione per evitare lo stallo. Ciò comporta che al risveglio di un processo che ha dovuto attendere, quest'ultimo non debba rientrare in mutua esclusione per l'aggiornamento delle variabili, cui ha provveduto il processo che l'ha risvegliato. In questo modo si evita che un altro processo in procinto di entrare in mutua esclusione possa *inserirsi* dopo che il primo processo sia stato risvegliato, ma prima che rientri in mutua esclusione.

5.2 Regione critica

Anche se i semafori possono essere utilizzati per risolvere qualsiasi problema di sincronizzazione, l'introduzione di molti semafori, oltre che accrescere la difficoltà di controllo sulla correttezza, rende di difficile lettura il codice.

Un meccanismo generale per risolvere problemi di sincronizzazione prende il nome di **region** o **regione critica**. Per la regione è definito un protocollo d'accesso di mutua esclusione. Tale costrutto permette direttamente al compilatore di generare in modo corretto il codice di controllo della sezione critica, cioè la coppia di primitive wait e signal all'inizio e alla fine della sezione, rendendo meno gravoso il lavoro del programmatore.

È possibile ottenere una regione critica semplicemente usando il metodo *synchronize* della classe *Mutex*, che in pratica esegue un'operazione di *lock* e al termine una di *unlock*. Nel caso si voglia ottenere una regione critica condizionale è utile l'utilizzo di una *ConditionVariable*, in modo che la sezione critica venga eseguita solo quando la condizione di sincronizzazione è soddisfatta.

5.3 Monitor

Un buon modo per evitare **race condition** e rendere il codice *thread-safe* è utilizzare la classe *Monitor*:

```
0 @monitor = Monitor.new
1
2 @monitor.synchornize do
3     # qui viene eseguito solo un thread alla volta
4 end
```

Un monitor è un costrutto sintattico che permette di associatare ad un insieme di variabili condivise da più processi un insieme di procedure, le uniche che possano operare su tali variabili. Il monitor garantisce che le procedure siano eseguite in mutua esclusione. A differenza delle regioni critiche il monitor, non solo garantisce la mutua esclusione nell'accesso ad una sezione critica condizionale, ma fornisce anche un livello di protezione di tipo procedurale. Un'altra caratteristica, non fornita nelle regioni critiche, è la possibilità fornita al processo di attendere una sincronizzazione all'interno di una procedura del monitor, potendo disporre di un insieme di code distinte, una per ogni causa diversa di sospensione.

È possibile utilizzare la libreria *monitor* come classe padre, come mixin, e come estensione di un particolare oggetto. A differenza di altri linguaggi (come C++), che supportanto l'ereditarietà multipla, in cui una classe può avere più di un genitore, ereditando funzionalità da ognuno, **Ruby, come Java e C#, supporta l'ereditarietà singola**. L'ereditarietà multipla, anche se potente, può essere pericolosa, in quanto la gerarchia di ereditarietà può diventare ambigua. A differenza di altri linguaggi ad ereditarietà singola, Ruby offre però il vantaggio di includere le funzionalità di più classi, grazie ai mixin (v. 2.4.2). Si hanno quindi i vantaggi dell'ereditarietà multipla, senza gli svantaggi che la caratterizzano. Il metodo *include* aggiunge a tutti gli effetti un modulo come superclasse di *self*. È usato all'interno della definizione della classe per rendere i metodi d'istanza disponibili all'interno di un modulo utilizzabili dalle istanze della classe. Comunque, a volte è utile aggiungere i metodi d'istanza direttamente a un oggetto particolare, estendendo l'oggetto. La classe *Monitor* contiene il modulo *MonitorMixin*, che come detto può essere 'mixato' (incluso) ad una classe o può essere utilizzato per estendere un oggetto. Tale modulo implementa i metodi:

- *mon_initialize()* : inizializza il monitor, creando un nuovo oggetto *Mutex*, impostando a 0 la variabile *mon_count* e a *nil* la variabile *mon_owner*;
- *mon_enter()*: entra nella senzione critica;
- *mon_exit()*: esce dalla sezione critica;
- *mon_synchronize*: gestisce in modo automatico l'ingresso e l'uscita dalla sezione critica;
- *mon_try_enter()*: cerca di entrare nella sezione critica. Se è occupata restituisce false;
- *new_cond()*: crea una nuova condizione;
- *mon_check_owner()*: se il thread corrente non è in possesso del monitor viene lanciata un'eccezione;
- *mon_enter_for_cond(count)*: il thread corrente diventa il occupa il monitor e *@mon_count* viene impostato a *count*;
- *mon_exit_for_cond()*: il monitor viene liberato e viene restituito il valore di *@mon_count* prima di impostarlo nuovamente a 0.

5.3.1 Confronto con il Monitor di Java

In Java, viene creato un *lock* per ogni oggetto nel sistema. Quando un metodo viene dichiarato *synchronized*, il thread in esecuzione deve entrare in possesso del *lock* associato all'oggetto prima di poter proseguire con le proprie operazioni. Al termine del metodo, il *lock* viene automaticamente rilasciato. Definire un metodo *synchronized* ha l'effetto di renderlo atomico. Ciò significa che non è possibile eseguire lo stesso metodo in un altro thread mentre è già in esecuzione. Se due metodi dichiarati *synchronized* diversi, ma appartenenti allo stesso oggetto vengono chiamati in esecuzione, si otterrà un risultato analago al caso in cui due thread cerchino di eseguire lo stesso metodo *synchronized*. Questo perché entrambi richiederanno il *lock* associato allo stesso oggetto, e non è possibile acquisire per entrambi i metodi lo stesso *lock* nello stesso momento. In altre parole, anche se due o più metodi dello stesso oggetto vengono invocati, non verranno mai eseguiti in parallelo da thread diversi. Questo fatto viene illustrato in fig. 5.2.

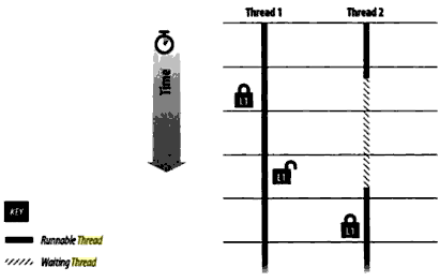

Figura 5.2: Acquisizione e rilascio del *lock* associato ad un oggetto in Java.

In Ruby, è possibile emulare la parola chiave *synchronized* presente in Java, attraverso un metodo globale chiamato per l'appunto *synchronized*. Tale metodo richiede come argomento un oggetto singolo e deve essere associato ad un blocco. Tale metodo ottiene il *mutex* associato all'oggetto, e utilizza *Mutex.synchronize* per invocare il blocco. È necessario prestare attenzione al fatto che, **a differenza di Java, Ruby non associa un *Mutex* a ciascun oggetto**. Nell'esempio seguente, viene definito un metodo d'istanza chiamato *mutex* nella classe *Object*. È interessante notare come questa implementazione del metodo *mutex* utilizza *synchronized* come fosse una parola chiave del linguaggio:

```ruby
require 'thread'  # Ruby 1.8, Mutex è contenuto in questa libreria

# Ottiene il Mutex associato all'oggetto o, e poi valuta
# il blocco sotto la protezione di tale Mutex.
# Lavora come la parola chiave di Java synchronized.
def synchronized(o)
  o.mutex.synchronize { yield }
end

# Object.mutex non esiste. Dobbiamo definirlo.
# Questo metodo restituisce un unico Mutex per ogni oggetto, e
# permette di restituire lo stesso Mutex per un particolare oggetto.
class Object
  # Restituisce il Mutex per questo oggetto, creando se necessario.
  # Bisogna essere certi che due thread non chiamino questo metodo
  # nello stesso momento, terminando con due mutex differenti.
  def mutex
    # Se questo oggetto ha già un mutex, restituiscilo
    return @__mutex if @__mutex

    # Altrimenti, bisogna crearlo.
    # Per evitare problemi di concorrenza 'sincronizziamo' la classe object.
    synchronized(self.class) {
      # Ricontrolliamo: magari mentre noi entravamo in questo blocco
      # sincronizzato, qualche altro thread ha già creato il mutex.
      @__mutex = @__mutex || Mutex.new
    }
    # Il valore restituito è @__mutex
  end
end

# Il metodo Object.mutex richiede di fare il lock sulla classe,
# se l'oggetto non ha ancora un Mutex. Se la classe non ha
# già il suo Mutex, allora la classe della classe (la classe object)
# verrà 'bloccata' dal lock. Al fine di prevenire una ricorsione infinita,
# dobbiamo assicurarci che la classe object abbia un mutex.
Class.instance_eval { @__mutex = Mutex.new }
```

Questo esempio mette in luce la potenza dei blocchi in Ruby: attraverso i blocchi è possibile sostituire costrutti o parole chiave del linguaggio. Un programmatore Ruby può facilmente modificare codice 'insicuro' per quanto riguarda la concorrenza, aggiungendo un blocco di sincronizzazione, e tutto continua a funzionare.

5.3.2 Monitor di Hoare

Il monitor di Hoare permette di eseguire più signal all'interno della stessa procedura, al contrario di quanto avviene nel monitor di Brinch Hansen in cui la chiamata a

signal è unica e alla fine della procedura stessa.

Grazie ai moduli messi a disposizione dal modulo MonitorMixin è semplice l'implementazione anche del Monitor di Hoare. In particolare sarà utile l'utilizzo dei metodi:

- mon_enter();
- mon_exit();
- broadcast().

5.4 Rendez-vous

È possibile ottenere costrutti simili al Rendez vous di Ada utilizzando una particolare libreria di Ruby 1.9, chiamata *DRB* (*Distributed Ruby*). Utilizzando *DRB*, un particolare processo può comportarsi da server, da client o da entrambi. Al client è nascosto il fatto che i metodi vengono eseguiti in remoto. Internamente al server vengono eseguiti dei thread per gestire le diverse chiamate da parte dei client.

Capitolo 6

Esempi

In questo capitolo verranno presentati alcuni problemi classici di sincronizzazione, come il problema del Produttore-Consumatore e della cena dei cinque filosofi. Nell'ultimo paragrafo verranno riportati degli esempi interessanti di applicazioni reali che richiedono l'uso della concorrenza (tratti da ***Ruby Cookbook*** *di L. Carlson e L. Richardson).*

6.1 Problema del Produttore-Consumatore...

Nel seguito verranno proposte diverse soluzioni al problema del produttore-consumatore utilizzando i diversi costrutti presentati nel capitolo precedente. In questo modo sarà possibile fare un confronto tra le diverse implementazioni.

6.1.1 ... utilizzando un semaforo numerico

Di seguito è riportato l'esempio del Produttore/Consumatore implementato in Ruby mediante semafori numerici:

```
0  require 'Thread'
1
2  messaggio_disponibile = SizedQueue.new(5) #Semaforo numerico
3  mess = 0                  #messaggio prodotto/consumato
4
5  p = Thread.new do
6      loop {
7          puts "Produco il messaggio\n"
8          sleep(2)
9          puts "Inserisco il messaggio\n"
10         messaggio_disponibile.push(mess)
11     }
12 end
13
14 c = Thread.new do
15     loop {
16         puts "***\tAttendo messaggio\n"
17         messaggio_disponibile.pop(non_block = false)
18         puts "***\tConsumo il messaggio\n"
19         puts "MESSAGGI DISPONIBILI: ", messaggio_disponibile.length()
20         sleep(10)
21     }
22 end
23
24 p.join
25 c.join
```

6.1.2 ... utilizzando una regione critica

Nel seguito è riportato un semplice esempio del problema del Produttore-Consumatore:

```
0  require 'Thread'
1
2  mutex = Mutex.new
3  risorse_disponibili = ConditionVariable.new
4
5
6  c = Thread.new do
7      loop {
8          mutex.synchronize{
9              puts "\tAttendo risorsa\n"
10             risorse_disponibili.wait(mutex)
11             puts "\tConsumo risorsa\n"
12         }
13     }
14 end
15
16 p = Thread.new do
17     loop {
18         mutex.synchronize {
19             puts "produco risorsa\n"
20             risorse_disponibili.signal
21         }
22         sleep(6)
23     }
24 end
25
26 c.join
27 p.join
```

6.1.3 ... utilizzando un monitor

Di seguito è riportata un'altra variante dell'esempio del produttore/consumatore, in cui viene esteso l'oggetto buffer:

```
0  require 'monitor'
1
2  buf = []
3  buf.extend(MonitorMixin)
4  empty_cond = buf.new_cond
5
6  # consumatore
7  c = Thread.new do
8      loop do
9          buf.synchronize do
10             puts "aspetto che vengano digitati dei caratteri...\n"
11             empty_cond.wait_while { buf.empty? }
12             puts "Ho letto questi caratteri:\n"
13               print buf.shift
14         end
15     end
16 end
17
18 #produttore
19 while line = ARGF.gets
20     buf.synchronize do
21         buf.push(line)
22         empty_cond.signal
23         end
24 end
25
26 c.join
```

6.2 5 filosofi a cena: Ruby vs Java

Il problema dei filosofi a cena è un problema classico di controllo della concorrenza.

L'esempio fu descritto nel 1965 da Dijkstra, che se ne servì per esporre un problema di sincronizzazione. Cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti, una forchetta a destra e una forchetta a sinistra (bastoncini cinesi secondo un'altra versione). Ci sono dunque cinque filosofi, cinque piatti di spaghetti e cinque forchette.

La vita di un filosofo consiste nell'alternanza di periodi in cui il filosofo mangia e quelli in cui pensa, e ciascun filosofo necessita di due forchette per mangiare, ma

quest'ultime vengono prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare. Il problema consiste nello sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation).

Riportiamo di seguito il codice Ruby che risolve il problema dei 5 filosofi:

```ruby
require 'mutex_m'

class Philosopher

def initialize(name, left_fork, right_fork)
    @name = name
    @left_fork = left_fork
    @right_fork = right_fork
    @meals = 0
end

def go
    while @meals < 5
        think
        dine
    end
    puts "#@name is full!"
end

def think
    puts "#@name is thinking..."
    sleep(rand())
    puts "#@name is hungry..."
end

def dine
    fork1, fork2 = @left_fork, @right_fork
    while true
        pickup(fork1, :wait => true)
        puts "#@name has fork #{fork1.fork_id}..."
        if pickup(fork2, :wait => false)
            break
        end
        puts "#@name cannot pickup second fork #{fork2.fork_id}..."
        release(fork1)
        fork1, fork2 = fork2, fork1
    end
    puts "#@name has the second fork #{fork2.fork_id}..."
    puts "#@name eats..."
    sleep(rand())
    puts "#@name belches"
    @meals += 1
    release(@left_fork)
    release(@right_fork)
end

def pickup(fork, opt)
    puts ''#@name attempts to pickup fork #{fork.fork_id}...''
    opt[:wait] ? fork.mutex.mu_lock : fork.mutex.mu_try_lock
end

def release(fork)
    puts ''#@name releases fork #{fork.fork_id}...''
    fork.mutex.unlock
end

end

n = 5
Fork = Struct.new(:fork_id, :mutex)
forks = Array.new(n) {|i| Fork.new(i, Object.new.extend(Mutex_m))}
philosophers = Array.new(n) do |i|
    Thread.new(i, forks[i], forks[(i+1)%n]) do |id, f1, f2|
        ph = Philosopher.new(id, f1, f2).go
    end
end

philosophers.each {|thread| thread.join}
```

Il codice Java che esegue operazioni analoghe è:

```java
java.util.Random;

class Monitor {
    int phil_States[] = new int[5];    // 0=not_waiting, 1=waiting // 2=eating
    boolean fork_States[] = new boolean[5];  // false = in use, true = free
    Monitor() {         // constructor
        for(int i=0;i<5;i++) {
            phil_States[i]=0;
            fork_States[i]=true;
        }
    }

```

```
12      synchronized void print_State(){
13          System.out.println();          // newline
14          for(int i=0;i<5;i++)
15              System.out.print('' '' + phil_States[i]);
16      }
17
18      synchronized void ask_to_eat(int pId){
19          while(!fork_States[pId] || !fork_States[(pId+1)%4])    {
20                  // while it can't have both forks, wait
21              phil_States[pId] = 1;
22              try    {wait();}
23              catch(InterruptedException e){} // it gets released
24          }     // by a process doing a call to notify()
25          phil_States[pId] = 2;  // eating
26          fork_States[pId] = false;  // in use
27          fork_States[(pId+1)%4] = false;
28      }
29
30      synchronized void ask_to_leave(int pId){
31          fork_States[pId] = true; // available
32          fork_States[(pId+1)%4] = true;
33          phil_States[pId] = 0;  // thinking
34          notify(); // free the Phil that has waited the longest
35      }
36  }
37
38  class Diners {
39      public static void main(String args[]) { // execution of the whole
40          Monitor m = new Monitor(); // thing begins here
41          Timer t = new Timer(m);  // make a new timer
42          Phil p[] = new Phil[5];  // make an array of 5 refs to Phils
43          for(int i=0; i<5; i++)
44              p[i] = new Phil(i,m,t); // create the phils and start them
45      }
46  }
47
48  class Phil implements Runnable {
49      Monitor m;
50      Timer t;
51      Random r = new Random();  // Random number generator
52      objectint pId;
53      float time;
54      Phil(int pId, Monitor m, Timer t) { // constructor
55          System.out.println(pId + ''is started: '');
56          this.pId = pId;
57          this.m = m;
58          this.t = t;
59          new Thread(this, ''Phil'').start(); // make a new thread and start it
60      }
61
62      public void run() {   // must override run, this is what
63          for(int i=0; i<20; i++) {  // is executed when the thread starts
64              m.ask_to_eat(pId);  // running
65              time = 1000*r.nextFloat();
66              try {Thread.sleep((int)time);} catch(Exception e){}
67              m.ask_to_leave(pId);
68              time = 1000*r.nextFloat();
69              try {Thread.sleep((int)time);} catch(Exception e){}
70          }
71          t.report_Stop();   // tell the timer this one is done
72      }
73  }
74
75  class Timer implements Runnable {
76      Monitor m;
77      int completed;
78      Timer(Monitor m) {   // constructor
79          this.m = m;
80          new Thread(this, ''Tim'').start(); // make a new thread and start it
81          completed=0;
82      }
83
84      public void report_Stop() {
85          completed++;
86      }
87
88      public void run() {   // must override run(), this is
89          while(completed!=5) {  // what happens when the thread
90              m.print_State();  // begins
91              try {Thread.sleep(500);} catch(Exception e){}
92          }
93      }
94  }
```

Facendo il confronto tra le due soluzioni presentate, il codice Ruby risulta di più facile lettura.

6.3 Un server Internet in 30 righe di codice

Supponiamo di avere la necessità di scrivere del codice Ruby per un server per un protocollo a livello d'applicazione TCP/IP.

La libreria *gserver* della libreria standard, implementa un generico server TCP/IP utilizzabile per task di piccole-medie dimensioni.

Si presenta ora il codice per un chat server molto semplice scritto con *gserver*. Gli utenti si connettono al server sfruttando una connessione telnet, e si identificano tra loro attraverso il nome dell'host. Nella sua semplicità, è comunque un server perfettamente funzionante, multithreaded, scritto in circa 30 righe di codice.

```
0   #!/usr/bin/ruby -w
1   # chat.rb
2   require 'gserver'
3
4   class ChatServer < GServer
5
6     def initialize(port=20606, host=GServer::DEFAULT_HOST)
7       @clients = []
8       super(port, host, Float::MAX, $stderr, true)
9     end
10
11    def serve(sock)
12      begin
13        @clients << sock
14        hostname = sock.peeraddr[2] || sock.peeraddr[3]
15        @clients.each do |c|
16          c.puts "#{hostname} has joined the chat."
17          unless c == sock
18        end
19        until sock.eof? do
20          message = sock.gets.chomp
21          break if message == "/quit"
22          @clients.each { |c| c.puts "#{hostname}: #{message}"
23                          unless c == sock }
24        end
25      ensure
26        @clients.delete(sock)
27        @clients.each { |c| c.puts "#{hostname} has left the chat." }
28      end
29    end
30  end
31
32  server = ChatServer.new(*ARGV[0..2] || 20606)
33  server.start(-1)
34  server.join
```

Per utilizzare il server, è sufficiente avviarlo in una sessione Ruby, e poi utilizzare diverse istanze di Telnet connettendosi alla porta 20606 (da diversi host, se possibile...). La sessione telnet permetterà la comunicazione con gli altri host attraverso il server. La sessione Ruby visualizzerà un log con le connessioni e le disconnessioni.

Si analizza ora come ciò sia reso possibile da queste poche linee di codice.

La classe *GServer* contiene la classe *TCPServer* in un ciclo che continuamente riceve le connessioni TCP e crea nuovi thread per gestirle. Ogni nuovo thread passa la propria connessione TCP (che è un oggetto *TCPSocket*) al metodo *GServer#serve*.

TCPSocket lavora come un file bidirezionale. Scrivere su di esso equivale ad eseguire un'operazione di push dei dati per il client, e leggere da esso permette di ricevere i dati dal client. Un server come il semplice chat server presentato legge una riga per volta dal client; un web server leggerebbe l'intera richiesta prima di inviare i dati in risposta.

Nell'esempio presentato, il server invia l'input ricevuto da un client a tutti gli altri. Nella maggior parte delle applicazioni però, il client non vuole avere nessuna informazione proveniente da altri client (basti pensare ad applicazioni web o FTP).

Il costruttore di *GServer* merita uno sguardo da vicino:

```
0     def initialize(port, host = DEFAULT_HOST, maxConnections = 4,
1         stdlog = $stderr, audit = false, debug = false)
```

La porta e l'host dovrebbero essere familiari conoscendo altri tipi di server. *maxConnections* controlla il massimo numero di client che possono essere contemporaneamente connessi al server. Tale numero per la nostra applicazione deve essere posto sufficientemente alto. *stdlog* è un oggetto per l'IO da usare come log. È possibile scrivere una voce nel log chiamando il metodo *GServer#log*. Se *audit* è impostato a true alcuni messaggi predefiniti vengono mostrati ogni qual volta un client si connette al server o si disconnette da esso. Infine, impostare *debug* a true significa che se il codice incontra un'eccezione, l'oggetto *exception* verrà passato a *GServer#error*. È possibile sovrascrivere tale metodo per gestire le eccezioni come si preferisce.

GServer è facile da usare, ma non è il modo più efficiente per ottenere un server Internet in Ruby. Per server ad alte prestazioni, è consigliato utilizzare *IO.select* e oggetti *TCPServer*, facendo riferimento anche alla *C sockets API*.

6.4 Implementare una coda distribuita

Si supponga di avere la necessita di utilizzare un server centrale per gestire le richieste di client remoti, elaborandole una per volta.

Per fare ciò esiste un metodo che condivide un oggetto *Queue* tra diversi client. I client mettono l'oggetto job all'interno della coda, e il server gestisce tali oggetti passandoli ad un blocco di codice.

```
0     #!/usr/bin/ruby
1     # queue_server.rb
2
3     require 'thread'            # For Ruby's thread-safe Queue
4     require 'drb'
5
6     $SAFE = 1                   # Minimum acceptable paranoia level when sharing code!
7
8     def run_queue(url='druby://127.0.0.1:61676')
9       queue = Queue.new         # Containing the jobs to be processed
10
11      # Start up DRb with URI and object to share
12      DRb.start_service(url, queue)
13      puts 'Listening for connection...'
14      while job = queue.deq
15        yield job
16      end
17    end
```

Ogni volta che un client immette un job all'interno della coda del server, il server passa tale oggetto come paramentro per il blocco di codice che gestisce tali richieste. Di seguito viene presentato un semplice blocco che può gestire un job che richiede poco tempo ("Report") o uno che richiede molto tempo ("Process"):

```
0     run_queue do |job|
1       case job['request']
2       when 'Report'
3         puts "Reporting for #{job['from']}... Done."
4       when 'Process'
5         puts "Processing for #{job['from']}..."
6         sleep 3           # Simulate real work
7         puts 'Processing complete.'
8       end
9     end
```

Se si hanno due client che inviano le loro richieste, l'output dovrebbe essere del tipo:

```
0     $ ruby queue_server.rb
1     Listening for connection...
2
```

```
3       Processing for Client 1...
4       Processing complete.
5       Processing for Client 2...
6       Processing complete.
7       Reporting for Client 1... Done.
8       Reporting for Client 2... Done.
9       Processing for Client 1...
10      Processing complete.
11      Reporting for Client 2... Done.
12      ...
```

Un Client per il queue server definito in questa soluzione deve semplicemente connettersi al DRB server e aggiungere un job del tipo "Report" o "Process" alla coda. Di seguito viene presentato il codice per un client che si connette al Drb Server e aggiunge 20 job alla coda in modo casuale:

```
0       #!/usr/bin/ruby
1       # queue_client.rb
2
3       require 'thread'
4       require 'drb'
5
6       # Get a unique name for this client
7       NAME = ARGV.shift or raise "Usage: #{File.basename($0)} CLIENT_NAME"
8
9       DRb.start_service
10      queue = DRbObject.new_with_uri("druby://127.0.0.1:61676")
11
12      20.times do
13        queue.enq('request' => ['Report', 'Process'][rand(2)], 'from' => NAME)
14        sleep 1 # simulating network delays
15      end
```

Bisogna notare come la classe *Queue*, essendo thread-safe, ci evita di dover costruire un'apposita coda per le richieste.

Appendice: Shoooes!

Figura 6.1: I dieci punti essenziali di Shoes. Tratto da "Nobodyknowsshoes"

Concetti fondamentali

Shoes è un ambiente di sviluppo per applicazioni grafiche basato su Ruby, che va installato successivamente a quest'ultimo. Non è una semplice libreria che si può utilizzare all'interno di applicazioni Ruby, ma un'interprete Ruby aggiuntivo esteso con funzionalità grafiche. É stato implementato da _whytheluckystiff, programmatore ed artista.

Grazie a Shoes, è possibile trasformare in un singolo click applicazioni contenute in file *.rb* in file eseguibili pronti per la distribuzione su Linux, Windows o Mac OS X, grazie ad un apposito Packager. La sintassi semplice permette di sviluppare interfacce grafiche molto velocemente. Come aspetto negativo bisogna però nominare il fatto che Shoes è stato interamente creato per Ruby: non è basato su uno standard per lo sviluppo di GUI, anche se alcune delle tecniche che vengono utilizzate possono essere applicate anche altrove.

Per creare una semplice applicazione dotata di GUI è sufficiente scrivere del codice del tipo:

```
0 Shoes.app do
1     # codice
2 end
3 # come al solito, al posto di do-end avremmo potuto scrivere { }
```

Per eseguirla è necessario lanciare il file .rb in cui è salvata attraverso Shoes.

para

Per inserire un paragrafo all'interno della finestra corrente, è sufficiente scrivere il codice:

```
0 para("Testo che si vuole visualizzare")
```

Generalmente tale metodo viene inserito all'interno di un blocco.

Stacks & Flows

I flow e gli stack permettono di impostare facilmente il layout per gli elementi che compongono la nostra GUI.

Figura 6.2: Gli stack assomigliano al domino

Gli stack permettono di impaginare gli oggetti come se fossero contenuti in dei riquadri uno sovrapposto all'altro verticalmente.

Figura 6.3: I flow assomigliano a una scatola di fiammifieri

I flow invece, sono come dei riquadri giustapposti uno all'altro orizzontalmente.

La combinazione di stack e flow permette di ottenere ogni sorta di impaginazione. Nella figura seguente riportiamo alcuni esempi:

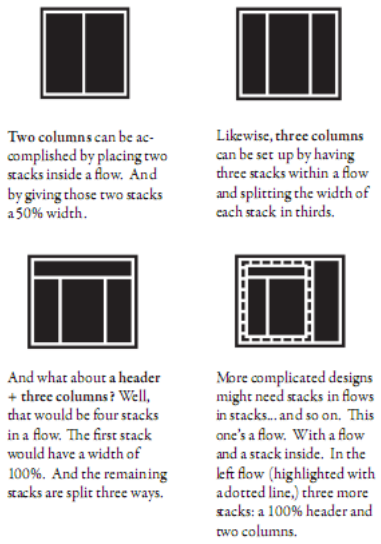

Figura 6.4: Stack e flow permettono di ottenere diverse impaginazioni.

Button

Per inserire un pulsante, è sufficiente utilizzare il metodo *button("Testo sul pulsante").*

Ad esempio, il semplice codice:

```
0 Shoes.app {button("Click me!") {alert("Good job.")}}
```

produce un risultato simile a quello riportato in fig. 6.5.

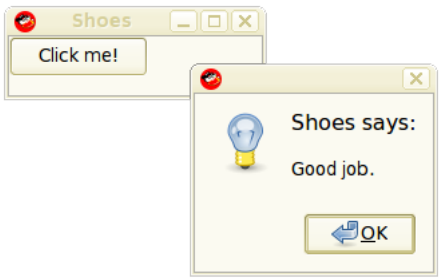

Figura 6.5: Esempio di semplice applicazione.

Image

Per inserire un'immagine, si utilizza il metodo *image("percorso immagine").* Il percorso può essere anche un URL web. Shoes carica le immagine remote utilizzando i thread del sistema, evitando in questo modo che i tempi di attesa per l'IO blocchino l'esecuzione. Le immagini possono anche venire ridimensionate.

edit_line

edit_line("Riga di testo") permette di aggiungere una casella per l'immissione di testo composta da un'unica riga all'interno della finestra.

Figura 6.6: Esempio di come appare in una finestra una casella per l'immissione di testo.

Link

È possibile inserire dei link a pagine web o ad altre finestre. È sufficiente utilizzare il metodo *link(Testo, :click => proc o stringa)*.

Background

Uno sfondo è un colore, sfumato o tinta unita, o un'immagine che può essere posto in secondo piano su tutta una finestra o un'area di essa. Ad esempio, per definire nero lo sfondo per un'applicazione basta scrivere il codice:

```
0 Shoes.app do
1     background black
2 end
```

Shoes.url

Non è necessario passare un blocco a Shoes.app. È possibile invece creare una classe figlia da Shoes e collocare ciascuna pagina che dovrà comporre il programma in un metodo. Si riporta di seguito un esempio:

```
0  class BookList < Shoes
1      url '/',          : index
2      url '/twain',     : twain
3      url '/kv',        : vonnegut
4
5      def index
6          para "Books: ",
7              link("by Mark Twain", :click => "/twain"),
8              link("by Kurt Vonnegut", :click => "/kv")
9      end
10
11     def twain
12       para "Just Huck Finn.\n",
13         link("Go back.", :click => "/")
14     end
15
16     def vonnegut
17       para "Cats Cradle.  Sirens of Titan. ",
18         "Breakfast of Champions.\n",
19             link("Go back.", :click => "/")
20     end
21 end
22
23 Shoes.app :width => 400, :height => 500
```

Poiché la classe BookList discende dalla classe Shoes, ne eredita il metodo url. Nell'esempio presentato si utilizza il metodo url tre volte.

clear

Riportiamo in fig. 6.7 la spiegazione del metodo clear scritta direttamente da whytheluckystiff.

Figura 6.7: Clear spiegato da whytheluckystiff.

Semplice applicazione per visualizzare .txt

Per realizzare una semplice applicazione per visualizzare .txt, eseguire il codice Ruby presente nel file selezionato e ricercare parole nel testo è stato sufficiente modificare i file d'esecuzione di Shoes, in particolare i file *shoes.rb* e *help.rb*.

Nel seguito riportiamo le parti di tali file che sono state modificate.

```
0  require 'Shoes'
1
2  #Lancia l'applicazione e imposta il titolo che comparirà sulla barra
3  #della finestra ("Guida a Ruby")
4  Shoes.app :title => "Guida a Ruby",
5            :width => 400,
6            :height => 300,
7            :resizable => false do
8          #imposta i parametri di layout per il testo
9        style(Para, :align => "center", :weight => "bold", :font => "Lacuna Regular")
10       style(Link, :stroke => yellow, :underline => nil)
11       style(LinkHover, :stroke => yellow, :fill => nil)
12
13 #Questo è uno dei metodi principali
14 #rimanda a help.rb che permette di selezionare il
15 #file da visualizzare
16 def guida_come format, *args
17     require 'shoes/search'
18     require 'help'
19
20     case format
21     when :shoes
22             #apre una nuova finestra
23             Shoes.app(:width => 720, :height => 640, &Shoes::Help)
24     else
25         #[...]
26         #(questa parte del codice non è fondamentale per capire
27         #in che modo vengano aperti e visualizzati i file .txt
28 end
29
30
31 #imposta il parametro di visualizzazione
32 def self.mostra_guida
33       guida_come :shoes
34 end
35
36 #Questo stack corrisponde alla finestra principale del programma
37 #che viene visualizzata all'avvio
38       stack :margin => 22 do
39         para "Guida a", :stroke => "#DFA", :margin => 0
40         para "RUBY", :size => 48, :stroke => "#DFA", :margin_top => 0
41         stack do
42           background black(0.2), :curve => 8
43           #cliccando su ''Leggi la guida'' viene aperta una nuova finestra
44           #vedi il metodo mostra_guida e guida_come
45           para link("Leggi la guida.") { self.mostra_guida and close}, :margin => 12
46         end
47
48
49   end
50 end
```

Alla riga di codice numero 23 viene aperta una nuova finestra, facendo riferimento al codice presente nel file help.rb a partire da Shoes::Help (ultima riga di codice). Riportiamo di seguito il codice di interesse:

```
0  module Shoes::Manual
1    #[...]
2    #(in questo modulo sono implementati i metodi che permettono di
3    # formattare un file .txt. Vengono qui presentati però solo i
4    # metodi che sono stati aggiunti/modificati rispetto alla versione
5    # originale del file help.rb di Shoes. )
6
7  #utilizzando l'esempio presente in Shoes ''expert-irb.rb''
8  #questo metodo apre un interprete interattivo ruby
9  def show_irb
10     fname = "expert-irb.rb"
11     Shoes.visit(fname)
12 end
13
14 #QUESTO è IL METODO PRINCIPALE
15 #qui si sceglie il file .txt da aprire e vengono chiamati i metodi per la sua
16 #formattazione per la visualizzazione
17 def Shoes.make_help_page
18   font "#{DIR}/fonts/Coolvetica.ttf" unless Shoes::FONTS.include? "Coolvetica"
19   proc do
20     extend Shoes::Manual
21     #il metodo ask_open_file apre la tipica finestra
22     #per selezionare un file da aprire
```

```
23      docs = load_docs ask_open_file #QUI SI SCEGLIE IL DOCUMENTO DA APRIRE!
24      #se il file non è valido non viene gestito l'errore
25
26      #layout
27      style(Shoes::Image, :margin => 8, :margin_left => 100)
28      style(Shoes::Code, :stroke => "#C30")
29      style(Shoes::LinkHover, :stroke => green, :fill => nil)
30      style(Shoes::Para, :size => 12, :stroke => "#332")
31      style(Shoes::Tagline, :size => 12, :weight => "bold", :stroke => "#eee",
32          :margin => 6)
33      style(Shoes::Caption, :size => 24)
34      background "#ddd"..."#fff", :angle => 90
35
36      #riquadro in alto
37      stack do
38        background black
39        stack :margin_left => 118 do
40                para "Guida a Ruby", :stroke => "#eee", :margin_top => 8,
41          :margin_left => 17,  :margin_bottom => 0
42          @title = title docs[0][0], :stroke => white, :margin => 4, :margin_left => 14,
43            :margin_top => 0, :font => "Coolvetica"
44        end
45
46        background "rgb(66, 66, 66, 180)"..."rgb(0, 0, 0, 0)", :height => 0.7
47        background "rgb(66, 66, 66, 100)"..."rgb(255, 255, 255, 0)",
48              :height => 20, :bottom => 0
49      end
50
51      #nel seguito vengono chiamati metodi che non vengono qui presentati
52      # ma che scompongono il testo contenuto nel file .txt selezionato
53      # per individuare titoli, sottotitoli, parti di codice,
54      # immagini ecc. e visualizzarli di conseguenza.
55      @doc =
56        stack :margin_left => 130, :margin_top => 20, :margin_bottom => 50,
57              :margin_right => 50 + gutter,
58          &dewikify(docs[0][-1]['description'], true)
59      add_next_link(0, -1)
60      stack :top => 80, :left => 0, :attach => Window do
61        @toc = {}
62        stack :margin => 12, :width => 130, :margin_top => 20 do
63          docs.each do |sect_s, sect_h|
64            sect_cls = sect_h['class']
65            para strong(link(sect_s, :stroke => black) { open_section(sect_s) }),
66              :size => 11, :margin => 4, :margin_top => 0
67            @toc[sect_cls] =
68              stack :hidden => @toc.empty? ? false : true do
69                links = sect_h['sections'].map do |meth_s, meth_h|
70                  [link(meth_s) { open_methods(meth_s) }, "\n"]
71                end.flatten
72                links[-1] = {:size => 9, :margin => 4, :margin_left => 10}
73                para *links
74              end
75          end
76        end
77
78        #in questo riquadro si trova il link per la ricerca nel testo
79        stack :margin => 12, :width => 118, :margin_top => 6 do
80          background "#330", :curve => 4
81          para "Chi cerca...", strong(link("Trova", :stroke => white) { show_search }),
82          "!", :stroke => "#ddd", :size => 9, :align => "center", :margin => 6
83        end
84
85        #in questo riquadro cliccando sul link è possibile aprire un nuovo file
86        stack :margin => 12, :width => 118 do
87                background "#330", :curve => 4
88                para link("Leggi un altro capitolo."){nuova_pagina()}, :stroke => "#ddd",
89                      :size => 9, :align => "center", :margin => 6
90        end
91
92        #in questo riquadro è presente un collegamento a irb
93        stack :margin => 12, :width => 118 do
94                background "#330", :curve => 4
95                para "Mettiti alla prova! ", link("IRB"){show_irb()}, :stroke => "#ddd",
96                      :size => 9, :align => "center", :margin => 6
97        end
98      end
99      image :width => 120, :height => 120, :top => -18, :left => 6 do
100       image "static/logo.png", :width => 100, :height => 100, :top => 10, :left => 10
101       glow 2
102     end
103   end
104 rescue => e
105   p e.message
106   p e.class
107 end
108
109 #questo metodo apre un nuovo file. Semplicemente richiama Shoes::Help
110 def nuova_pagina()
111     Shoes.app(:width => 720, :height => 640, &Shoes::Help)
112 end
113
114 #viene chiamato il metodo make_help_page che crea la pagina della guida
115 Shoes::Help = Shoes.make_help_page
```

Bibliografia

[1] **D. Thomas, Programmin Ruby 1.9, 2009, The Pragmatic Programmers'Guide**
Questa guida, di circa 1000 pagine, è adatta per chi vuole approcciarsi alla programmazione in Ruby, in quanto espone in modo abbastanza esaustivo i concetti fondamentali su cui si basa il linguaggio. Per la stesura dei capitoli iniziali di questa tesina si è fatto riferimento agli argomenti esposti nei primi 10 capitoli di tale guida e in modo particolare ai capitoli 23 ("Duck Typing") e 24 ("Metaprogramming"). Il cap. 12 è dedicato ai Fibers, thread e processi, ma non spiega nel dettaglio come viene gestita la sincronizzazione.

[2] **D.Glanaga, U. Matsumoto, The Ruby Programming Language, 2008, O'Reilly**
Il libro incomincia con una rapida guida a Ruby, e poi espone nel dettaglio le particolarità del linguaggio seguendo una strategia bottom-up. Anche in questo testo viene dedicato un intero capitolo alla metaprogrammazione (cap. 8). Di interesse per quanto riguarda la concorrenza sono la sezione 5.8 "Threads, Fibers and Continuations" e la sezione 9.9 "Threads and Concurrency", a cui si è fatto riferimento per la stesura del capitolo sulla multiprogrammazione.

[3] **G. Clemente, F. Filira, M. Moro, Sistemi Operativi, 2006, Edizioni Libreria Progetto**
Per capire gli argomenti esposti è opportuno aver seguito un corso di Sistemi Operativi. In particolare per la stesura di questo testo si è fatto riferimento ai cap. 2, 3 e 4 del volume sopra citato (rispettivamente "La concorrenza nei sistemi", "La sincronizzazione", "Programmazione Concorrente"), e in piccola parte al cap. 12 ("La multiprogrammazione in Java").

[4] **S. Oaks, H. Wong, Java Threads, 1997, O'Reilly**
Come dice il titolo, il testo tratta dei thread in Java. Vengono esposti i metodi per la gestione dei thread offerti dalla libreria standard di Java. Successivamente vengono presentate le tecniche di sincronizzazione, quindi viene introdotto il concetto di Mutex Lock e vengono presentati i metodi synchronized. Nel cap. 4 viene esposto nel dettaglio il funzionamento delle primitive wait e notify. Di particolare interesse il cap. 7 in cui vengono trattati argomenti avanzati riguardanti la sincronizzazione. Negli altri capitoli vengono riportati alcuni esempi e la descrizione del funzionamento dello scheduling dei thread in Java o di come vengono gestite le eccezioni.

[5] **L. Carlson, L. Richardson, Ruby Cookbook, 2006, O'Reilly**
Questo testo è pensato per risolvere problemi reali che si incontrano frequentemente nella vita di tutti i giorni. I problemi vengono presentati in modo chiaro e discussi approfonditamente. Per la stesura della tesina si è fatto riferimento a

due dei problemi presentati, in particolare alla sezione 16.11 ("Implementing a Distributed Queue") e 14.14("Writing a Internet Server").

[6] **M. Gheda, Supporta alla catalogazione per Keywords nel Filesystem, Relazione finale di Elaborato, Relatore Prof. G. Clemente, A. A. 2006-2007**
L'unico capitolo di interesse per gli argomenti qui trattati è il cap.2 "Breve introduzione al Ruby" che espone, anche se molto velocemente, le caratteristiche di Ruby.

[7] **D. Costa, Multithreading e costrutti concorrenti in Ruby 1.9, Elaborato finale, Relatore Prof. G. Clemente, A. A. 2008-2009**
Dopo aver introdotto i cambiamenti in Ruby 1.9 rispetto alla versione 1.8, viene introdotto il concetto di Thread e la classe Thread della libreria standard di Ruby. Nei capitoli successivi vengono presentati i concetti di semaforo, monitor e regione critica fornendo degli esempi dell'implementazione di tali costrutti in Ruby. Nella stesura della tesina non si è preso però particolare spunto da questo elaborato.

[8] **A. Martinelli, I-Time: Progetto di una GUI integrata, Relatore Prof. G. Clemente, A.A. 2008-2009**
Di interesse per gli argomenti qui trattati è il primo capitolo su Shoes.

Altre fonti

- http://mislav.uniqpath.com/poignant-guide/book/chapter-1.html
why_theluckystiff, "why's (poignant) guide to Ruby"
- **why_theluckystiff, "nobodyknowsshoes"**
- http://www.engineyard.com/blog/2010/concurrency-real-and-imagined-in-mri-threads/
Articolo scritto da **Kirk Haines, "Concurrency, Real and Imagined, in MRI; Threads", 11.08.2010**
- http://www.ibmsystemsmag.com/ibmi/ruby_web/33468p1.aspx
Articolo scritto da **Andrea Ribuoli, "Getting Ruby Up and Running", Luglio 2010**
- http://www.viddler.com/explore/GreggPollack/videos/40/
OSCON (Open Source Convention) 2010 e RailsConf 2010: "No Callbacks, No Threads: Async webservers in Ruby 1.9", presentata da Ilya Grigorik.