Complementi di Programmazione

Gestione della Memoria

CdL Informatica - Università degli studi di Modena e Reggio Emilia
AA 2023/2024

Filippo Muzzini

Gestione della Memoria

I dati di un programma vengono salvati in memoria.

A seconda del punto del programma essi vengono salvati in aree differenti.

- stack
- heap

Gestione della Memoria

stack

variabili dichiarate nel codice

```
int main() {
        int a = 10;
}
```

a viene salvata nello stack

Gestione della Memoria

heap

memoria esplicitamente allocata

```
int main() {
        int *a = (int*)malloc(sizeof(int));
}
```

a viene salvata nello heap

Gestione della Memoria

Problema: fino a quando questa memoria deve essere occupata?

ovvero fino a quando la variabile deve esistere in memoria?

Gestione della Memoria

Diversi approcci:

- manuale -> il programmatore decide
- automatica -> compilatore/interprete decidono
  - Reference counting
  - Garbage collector
    - Generational
    - Tracing

Manuale

Il programmatore è responsabile delle allocazione e deallocazioni della memoria.

Es. C

malloc()

free()

Manuale

Vantaggi

- gestione esplicita
- molto veloce

Svantaggi

- frequenti errori
- Memory leaks
- puntatori a zone non allocate
- segmentation faults

Reference Counting

Algoritmo che libera memoria automaticamente quando un oggetto non è più referenziato.

Utilizzato in linguaggi come Perl e PHP

Reference Counting

Molto semplice:

- Ogni oggetto ha un campo aggiuntivo che indica il numero di riferimenti ad esso
- Ogni volta che si aggiunge un riferimento, questo contatore viene incrementato
  - a = 1; b=a;
- Ogni volta che un riferimento viene cancellato, il contatore viene decrementato
  - a = 2
- Quando il contatore è a 0, l’oggetto viene distrutto

Reference Counting

Vantaggi

- molto veloce
- reattivo (non appena si raggiunge lo zero l’oggetto viene eliminato)

Svantaggi

- Mantenere aggiornato il contatore implica un overhead
- **Problema dei riferimenti circolari!**

Reference Counting

Problema dei riferimenti circolari!

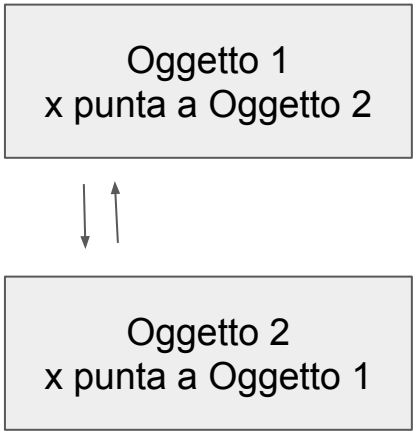
Due oggetti possono referenziarsi a vicenda.

Il contatore di ogni oggetto è almeno 1

Anche quando i due oggetti non sono più

referenziati altrove essi avranno comunque 1

Non verranno mai distrutti



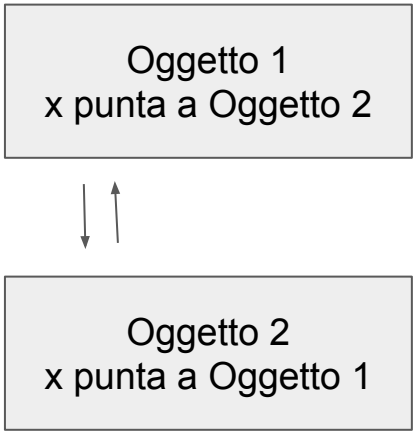
Reference Counting

Problema dei riferimenti circolari!

Possibile soluzione: marcare come deboli i riferimenti ricorsivi.

Considerarli come tali per la rimozione:

- li uso solo se almeno un oggetto è referenziato altrove
- maggiore overhead



Garbage Collector

Il garbage collector è un entità che ciclicamente controlla lo stato degli oggetti in memoria e eventualmente li elimina (liberando memoria)

Garbage Collector - Tracing

Un algoritmo utilizzato dai garbage collectors è il Tracing.

- Partendo da degli oggetti radice segue tutti gli oggetti referenziati
- Prosegue a cascata
- Gli oggetti non referenziati vengono considerati garbage ed eliminati

In questo modo vengono mantenuti in memoria tutti gli oggetti raggiungibili (direttamente o tramite altri oggetti) dagli oggetti radice

Garbage Collector - Tracing

Gli oggetti radici sono:

- variabili globali
- variabili locali
- argomenti della funzione

Garbage Collector - Tracing

Un oggetto può essere non più raggiungibile per due motivi:

- Ragione sintattica -> la sintassi implica che quell'oggetto non sia più raggiungibile
  - a = 1; a = 2 -> l'oggetto 1 non è più raggiungibile
- Ragione semantica -> oggetti non più raggiungibile per via del flusso di codice
  - branch di codice mai utilizzati

Garbage Collector - Tracing

Un oggetto può essere non più raggiungibile per due motivi:

- Ragione sintattica -> la sintassi implica che quell'oggetto non sia più raggiungibile
  - Facile da trovare: **i garbage collectors si basano sopratutto su questa**
- Ragione semantica -> oggetti non più raggiungibile per via del flusso di codice
  - difficile da trovare: euristiche

Garbage Collector

Questo tipo di garbage collectors implicano un overhead per trovare gli oggetti non più raggiungibili. -> bisogna stabilire quando chiamarlo

- ogni tot tempo (Java)
- quando la memoria occupata raggiunge un limite (Python)

Garbage Collector - Mark and Sweep

Ogni oggetto ha associato un flag

- 0: irraggiungibile
- 1: raggiungibile

Fase di Scansione -> oggetti marchiati come 0 o 1

Fase di Deallocazione -> oggetti 0 rilasciati

Garbage Collector - Mark and Sweep

Limiti:

- ogni volta si riscansiona tutto
- ogni volta si ricicla sugli oggetti 0
- ogni volta il programma si interrompe per eseguire il Garbage Collector

Garbage Collector - Tri-color Marking

Algoritmo organizzato in tre insiemi:

- White -> candidati alla rimozione
- Grey -> oggetti raggiungibili ma i cui oggetti referenziati da essi non sono stati ancora analizzati
- Black -> oggetti raggiungibili che non referenziano nessuno oggetto nel white set

Gli oggetti possono avere solo il seguente flusso:

White Set -> Grey Set -> Black Set

Garbage Collector - Tri-color Marking

Prima fase (inizializzazione degli insiemi):

Tutti gli oggetti radice messi nel grey set.

Tutti gli altri oggetti nel white set.

Black set vuoto.

Garbage Collector - Tri-color Marking

Seconda fase (si fa fino a quando il grey set non si vuota):

Si seglie un oggetto O dal grey set.

Si identificano tutti gli oggetti referenziati da O (R1, R2, ….).

Tra questi si identificano quelli appartenenti al White set (W1, W2, ...)

Essi sono referenziati (da O) quindi vengono messi nel Grey set)

Ora O può essere messo nel black (sono stati analizzati gli oggetti referenziati e essi non sono più nel white set).

Garbage Collector - Tri-color Marking

Terza fase:

Viene liberata la memoria degli oggetti nel White set

- non raggiungibili direttamente (prima fase)
- non raggiungibili indirettamente (seconda fase)

Garbage Collector - Tri-color Marking

Vantaggi:

- fase 1 e 2 possono essere eseguite mentre il programma gira (senza interromperlo)

Garbage Collector - Rilascio memoria

Vi sono due strategie per liberare la memoria:

- in movimento: copio tutti gli oggetti raggiungibili in una nuova area di memoria
- non in movimento: rilascio le zone di memoria degli oggetti non raggiungibili

La strategia in movimento sembra meno efficiente ma:

- meno frammentazione
- possibilità di avere più oggetti che si referenziano nella cache

Garbage Collector - Generational

Ipotesi (avvallata da studi empirici):

gli oggetti creati più di recente hanno la maggiore probabilità di diventare irraggiungibili nell'immediato futuro.

Garbage Collector - Generational

I garbage collectors generazionali suddividono gli oggetti in insiemi di “vecchiaia”

- Nei vari cicli di esecuzione, fa controlli frequenti solo sugli oggetti delle generazioni più giovani
- Contemporaneamente tiene traccia della creazione di riferimenti tra generazioni

Più veloce ma meno preciso (alcuni oggetti non raggiungibili potrebbero rimanere)

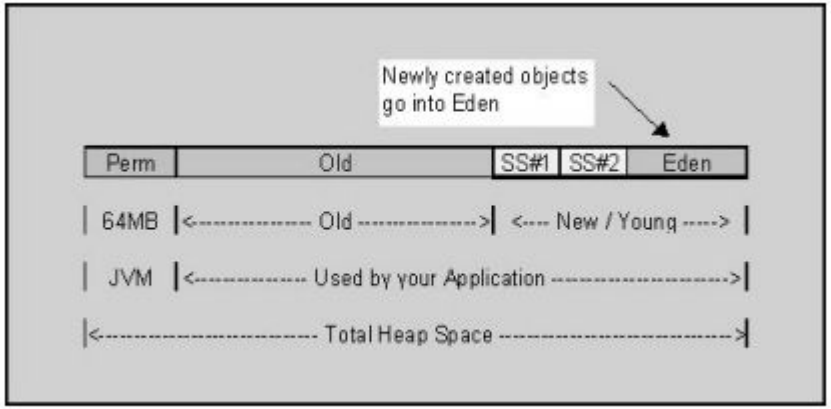
Garbage Collector - Generational

Generazioni:

- Eden: oggetti appena creati
- Surviror 2: oggetti sopravvissuti ad un certo numero di cicli
- Survivor 1: oggetti sopravvissuti ad un certo numero di cicli (più che S2)
- Old: oggetti più vecchi

Gli oggetti permanenti usati dall’interprete non rientrano in questi insiemi

Garbage Collector - Generational



Garbage Collector - Generational

Ogni volta che un gruppo supera una soglia viene invocato il Garbage collector solo su quel gruppo.

Gli oggetti raggiungibili vengono copiati nell’insieme immediatamente più vecchio

La regione viene svuotata

Garbage Collector - Generational

Vantaggi:

- agisce su set ridotti di oggetti

Svantaggi:

- minor precisione

Garbage Collector - Approcci ibridi

Minor cycle:

- fatto frequentemente (es. Generational)
  - molto performante
  - minor precisione

Major cycle:

- fatto ogni tanto (es. Mark and sweep):
  - su tutti gli oggetti

Gestione Memoria in Python

Dipende dalla versione di Python (e dall’interprete).

Ma ora è un approccio ibrido:

- Reference counting
- Generational Garbage collector

Gestione Memoria in Python

API

il modulo **gc** consente di impostare il garbage collector.

gc.enable()/disable() -> abilita/disabilita il garbage collector

gc.collect(generation=2) -> esegue il garbage collector su una generazione

e molto altro

https://docs.python.org/3/library/gc.html#module-gc

Gestione Memoria in Python

A dirla tutta dipende dall'interprete:

vi sono diversi interpreti python

- CPython (più utilizzato)
- Jython
- PyPy
- …

Gestione Memoria in Python

CPython è il più usato.

Scritto in C.

Principale meccanismo di gestione memoria -> reference counting

Poi gc generazionale a soglia

Gestione Memoria in Python

Principale meccanismo di gestione memoria -> reference counting

```
import sys

a = ‘aaa’

sys.getrefcount(a)
```

Gestione Memoria in Python

Principale meccanismo di gestione memoria -> reference counting

import sys

a = ‘aaa’

sys.getrefcount(a) -> 2

Gestione Memoria in Python

Principale meccanismo di gestione memoria -> reference counting

import sys

a = ‘aaa’

sys.getrefcount(a) -> 2

Abbiamo un ulteriore riferimento dato dal passaggio di parametro alla funzione

Gestione Memoria in Python

a = ‘aaa’

sys.getrefcount(a)

x = [a]

d = {‘a’:a}

sys.getrefcount(a)

Gestione Memoria in Python

a = 1

sys.getrefcount(a) -> 2

x = [a]

d = {‘a’:a}

sys.getrefcount(a) -> 4

Quando si esce dalla prima funzione il contatore viene decrementato (il riferimento dell’argomento non esiste più)

Gestione Memoria in Python

Come profilare il consumo di memoria?

Il SO mette a disposizione diversi strumenti ma non sono granulari

Gestione Memoria in Python

Python ha diverse librerie per fare ciò:

- memory-profiler (apt install python3-memory-profiler)
- tracemalloc

Gestione Memoria in Python

```
import tracemalloc

def foo():
        f = [ x for x in range(0, 100000) ]

tracemalloc.start()

foo()

current,peak = tracemalloc.get_traced_memory()

tracemalloc.stop()

print( "Istananea ",current, " Picco ", peak )
```

Gestione Memoria in Python

```
import tracemalloc

def foo():
        f = [ x for x in range(0, 100000) ]

tracemalloc.start()

foo()

current,peak = tracemalloc.get_traced_memory()

tracemalloc.stop()

print( "Istananea ",current, " Picco ", peak )
```

Gestione Memoria in Python

Come viene gestita la memoria a basso livello.

Ovviamente dipende dall’interprete e in che linguaggio è stato scritto.

Gestione Memoria in Python

in CPython esiste un allocatore di oggetti che si occupa di allocare la memoria per essi.

- Quando creao un oggetto l’interprete chiama questo allocatore
- Esso si occupa di gestire lo spazio ed effettivamente allocare la memoria tramite l’SO

Gestione Memoria in Python

Siccome in Python tutto è un oggetto, ogni volta bisognerebbe scomodare il SO per allocare piccole zone di memoria.

L’allocatore Python fa da filtro e alloca zone grandi meno volte e poi le gestisce lui.

In pratica gestisce un suo heap privato.

Gestione Memoria in Python

Siccome in Python tutto è un oggetto, ogni volta bisognerebbe scomodare il SO per allocare piccole zone di memoria.

L’allocatore Python fa da filtro e alloca zone grandi meno volte e poi le gestisce lui.

a = 10

b = a

a = 100 -> nuovo oggetto

Gestione Memoria in Python

Siccome in Python tutto è un oggetto, ogni volta bisognerebbe scomodare il SO per allocare piccole zone di memoria.

L’allocatore Python fa da filtro e alloca zone grandi meno volte e poi le gestisce lui.

a = 10

b = a

a = 100 -> nuovo oggetto

Immaginate nei cicli!

Gestione Memoria in Python

- Quando si crea un oggetto l’interprete lo alloca nell’heap privato (usando una funzione di allocazione propria)
- L’allocatore si occupa di crearlo in questa zona e gli assegna una zona di memoria
- Se serve più memoria viene chiamata una malloc()

Viceversa quando si rimuove un oggetto

Gestione Memoria in Python

Se un oggetto è molto grande (> di 512 byte) viene chiamata direttamente la malloc()

Se l’oggetto ha un allocatore specifico viene chiamato esso.

Gestione Memoria in Python

Le zone di memoria gestite dall’allocatore sono divise in Aree e Pool

Ogni Area è divisa in pool

Questo per gestire più efficientemente la memoria

Gestione Memoria in Python

Le zone di memoria gestite dall’allocatore sono divise in Arene e Pool

Ogni Arena è divisa in pool

Questo per gestire più efficientemente la memoria

Gestione Memoria in Python

Quando si alloca un oggetto viene messo nell’Arena con più pool pieni

In questo modo è più probabile che le arene si svuotino potendole liberare dalla memoria (free())

Gestione Memoria in Python

Un pool ha la dimensione di una pagina stabilita dal SO.

- memoria contigua
- allocazione determinata dal SO

Al suo interno ha dei blocchi di una dimensione prefissata ed uguale

Gestione Memoria in Python

Un pool ha la dimensione di una pagina stabilita dal SO.

- memoria contigua
- allocazione determinata dal SO

Al suo interno ha dei blocchi di una dimensione prefissata ed uguale

Quando si alloca un oggetto si cerca (nell’arena) il pool con i blocchi della dimensione richiesta.

Gestione Memoria in Python

Un pool ha la dimensione di una pagina stabilita dal SO.

- memoria contigua
- allocazione determinata dal SO

Al suo interno ha dei blocchi di una dimensione prefissata ed uguale

Quando si alloca un oggetto si cerca (nell'arena) il pool con i blocchi della dimensione richiesta.

Se non esistono viene creato un nuovo pool (con blocchi dimensione richiesta)

Gestione Memoria in Python

Quando si libera un oggetto il blocco viene segnato come libero.

Ogni blocco contiene un solo oggetto!