Parte 5

Gestione della memoria

Due diverse filosofie

Gestione della memoria (heap):

- Allocazione e rilascio di risorse

Filosofia C

- Gestione lasciata al *programmatore* attraverso le primitive di sistema (malloc, free, ...)
  - *Velocissima, ma…*
  - *…straordinariamente soggetta ad errori*

Filosofia Java

- Gestione della memoria va lasciata al *sistema*
  - *Gestore automatico della memoria reso disponibile dall'ambiente di esecuzione*

Gestione manuale della memoria

- *Vantaggi*:
  - La velocità delle operazioni di gestione è elevata (è essenzialmente quella del kernel sottostante)
  - I pattern di acquisizione/rilascio sono espliciti
- *Svantaggi*:
  - Elevato onere di programmazione → rischi
    - aritmetica dei puntatori → segmentation fault
    - puntatore a memoria liberata → dangling reference
    - memoria non liberata → memory leak
  - Gestione manuale difficilmente scala con le dimensioni del programma (molto facile dimenticarsi qualche free() → memory leak)

Gestione della memoria nei linguaggi dinamici

→ESEMPI
esegui_perl_c/*

- I *linguaggi dinamici seguono la filosofia Java*: il sistema (run time environment) mette a disposizione un gestore automatico della memoria
  - Alloca e dealloca a tempo di esecuzione

- Esempio (Perl):

```
my @array1 = ();
my @array2 = ();
push @array1, 10;
push @array2, 20.0;
push @array1, “ciao”;
push @array2, “dieci”;
```

La gestione automatica è particolarmente complessa nei linguaggi a tipizzazione dinamica

Garbage Collection

- La garbage collection è la forma più usata di gestione automatica della memoria
  - John McCarthy, LISP, 1958
- Garbage collector (run time environment):
  - Effettua le seguenti operazioni:
    - Individua oggetti (variabili, strutture dati, istanze di classe, ...) che non potranno mai più essere referenziati dall'applicazione *(garbage)*
    - Rilascia la memoria relativa a tali entità

Uso

- Linguaggi dotati di garbage collector
  - Java, C#, Smalltalk, Lisp, Haskell
  - Python (da versione 2.0), Ruby, Lua, Tcl
- Linguaggi nati con una gestione manuale, estendibili con un garbage collector:
  - C, C++ (Boehm Garbage Collector)
- Linguaggi che fanno uso di tecniche dinamiche alternative alla garbage collection:
  - Perl e PHP (*reference counting*)

Tracing Garbage Collector

Tracing garbage collector

- La maggioranza dei garbage collector segue la strategia detta tracing garbage collection:
  - Traccia gli oggetti ancora referenziabili attraverso una catena di riferimenti che parte da oggetti 'radice'
- Un oggetto si definisce raggiungibile in maniera ricorsiva: *se raggiungibile per se stesso o se esiste un riferimento ad esso tramite un oggetto esso stesso raggiungibile*

Raggiungibilità di un oggetto

La raggiungibilità può essere di due tipi:

1) Diretta: una variabile contiene l'oggetto

- **Gli oggetti radice sono direttamente raggiungibili: es. variabili globali, oggetti dello stack frame corrente (variabili locali, parametri della funzione correntemente in esecuzione)**

2) Indiretta: una variabile contiene un "puntatore" all'oggetto

- **Ogni oggetto riferito da un oggetto raggiungibile direttamente è, a sua volta, un oggetto raggiungibile (*proprietà transitiva*)**

Cause della irraggiungibilità

- La non raggiungibilità di un oggetto può avere due cause: sintattica o semantica
- Causa sintattica (syntactic garbage):
  - Oggetti non più raggiungibili per vincoli sintattici: es. *riassegno un puntatore o un riferimento*
  - Semplice da verificare: la non raggiungibilità è intesa solo in senso sintattico da quasi tutti i garbage collector
- Causa semantica (semantic garbage)
  - Oggetti non più raggiunti per via del flusso del codice eseguito: es. branch di codice mai utilizzato a run time
  - Non esiste algoritmo in grado di identificare i semantic garbage per ciascun possibile input → *euristiche nei garbage collector*

Algoritmi per garbage collection

- Gli algoritmi di garbage collection possono essere richiamati periodicamente (es. Java) o essere attivati sulla base di una soglia (es. Python)
- Nei sistemi a soglia un ciclo è attivato quando viene notificato al collector che il sistema ha bisogno di memoria
- Esempi di algoritmi più usati
  - Algoritmo naive Mark and Sweep
    - Inefficiente, set dati scandito più volte
  - Algoritmo Tri-color Marking
    - Evoluzione del precedente

Algoritmo naive: Mark and Sweep

- Uno dei primi algoritmi usati
- Ad ogni oggetto in memoria è associato un flag (un bit) utilizzato dall'algoritmo di garbage collection
- Il valore dei flag viene interpretato nel modo seguente:
  - 0/False: oggetto non raggiungibile (default)
  - 1/True: oggetto riconosciuto come raggiungibile
- Funzionamento:
  - Ogni oggetto viene scandito una volta: se è raggiungibile come radice o da un oggetto radice si imposta il suo flag a 1
  - Il set viene scandito una seconda volta: si libera la memoria per gli oggetti con flag a 0 (e si re-imposta a 0 il flag degli oggetti nei quali esso vale 1)

Limiti

- L’algoritmo è semplice ma poco performante
- L'interprete si deve interrompere durante l'intera esecuzione dell'algoritmo di garbage collection (*stop the world effect*)
  - L'intero insieme degli oggetti deve essere scandito più volte linearmente (analisi e rilascio memoria)
  - Problemi in sistemi con memoria paginata
  - Problemi in ambienti che necessitano di basse latenze di risposta o in sistemi real-time
- Evoluzione che ne supera i limiti: *algoritmo Tri-color Marking*

Algoritmo Tri-color Marking

- L'algoritmo usa tre sottoinsiemi che servono a mantenere lo stato degli oggett a run-time
- White set:
  - Oggetti candidati alla rimozione (che alla fine dell'algoritmo saranno tutti distrutti)
- Grey set:
  - Oggetti che sono raggiungibili, ma i cui oggetti referenziati non sono ancora stati analizzati
- Black set:
  - Oggetti raggiungibili che non referenziano alcun oggetto nel white set

Algoritmo Tri-color Marking

- **Prima fase dell'algoritmo (inizializzazione): l'insieme degli oggetti presenti in memoria viene partizionato nei tre sottoinsiemi**
  - **Tutti gli oggetti che sono referenziati a livello radice vengono inizialmente messi nel Grey set**
  - **Tutti gli altri oggetti vengono posizionati inizialmente nel White set**
  - **Il Black set è inizialmente vuoto (conterrà alla fine gli oggetti da non rimuovere)**
- **Nota: gli oggetti possono solo passare da *White Set* a *Grey Set* e da *Grey Set* a *Black Set* :**

White Set → Grey Set → Black Set

Algoritmo Tri-color Marking

Seconda fase dell'algoritmo

- **Finché ci sono oggetti nel Grey set:**
  - **scegli un oggetto *O* nel Grey set e spostalo nel Black set**
  - **Identifica tutti gli oggetti R_1, ..., R_k referenziati direttamente da *O***
  - **Considera l'insieme di oggetti**

    $\{W_1, ..., W_j\} = \{R_1, ..., R_k\} \cap$ **White set**

    e spostali nel Grey set

Algoritmo Tri-color Marking

- **Terza fase dell'algoritmo**

  - **La logica dell'algoritmo mostra che gli oggetti nel White set sono:**
    - **non raggiungibili direttamente (prima fase)**
    - **non raggiungibili neppure indirettamente (seconda fase)**

    → *candidati alla rimozione*
  - **La memoria relativa agli oggetti nel White set viene quindi rilasciata**

Vantaggi dell'algoritmo

- L'algoritmo Tri-color Marking può essere eseguito senza richiedere l'interruzione completa dell'interprete
  - Le prime due fasi dell'algoritmo di garbage collection vengono eseguite contestualmente all'allocazione ed alla modifica di oggetti
  - Solo durante la terza fase viene interrotto il programma
- Non ci sono molteplici scansioni dell'intero set di oggetti
  - Nella seconda fase viene scandito solo il Grey set
    - Accesso diretto agli oggetti riferiti nel white set
  - La terza fase accede solo agli oggetti rimasti nel White set

Rilascio della memoria: movimento vs. non movimento

- Due modalità possibili per il rilascio di memoria:
  - rilasciare gli oggetti irraggiungibili senza spostarli (GC in *non movimento)*
  - copiare tutti gli oggetti raggiungibili in una nuova area di memoria (GC in *movimento)*
- All'apparenza costosa e inefficiente, la strategia *in movimento* porta vantaggi:

1) Grandi regioni contigue di memoria disponibili
   - GC in non movimento portano dopo qualche iterazione a heap molto frammentati

2) Ottimizzazioni possibili: oggetti che si riferiscono l'uno all'altro sono messi vicini aumentando la probabilità che si trovino sulla stessa linea della cache o della pagina di memoria virtuale

Generational Garbage Collector

Ipotesi generazionale

- Uno studio empirico dimostra che, per diverse tipologie di programmi, gli oggetti creati più di recente hanno la maggiore probabilità di diventare irraggiungibili nell'immediato futuro
- Gli oggetti in memoria hanno una mortalità infantile elevata
- Tale osservazione è stata presa come ipotesi di lavoro per la realizzazione di strategie di gestione della memoria: ipotesi generazionale

Generazioni

Idea di base

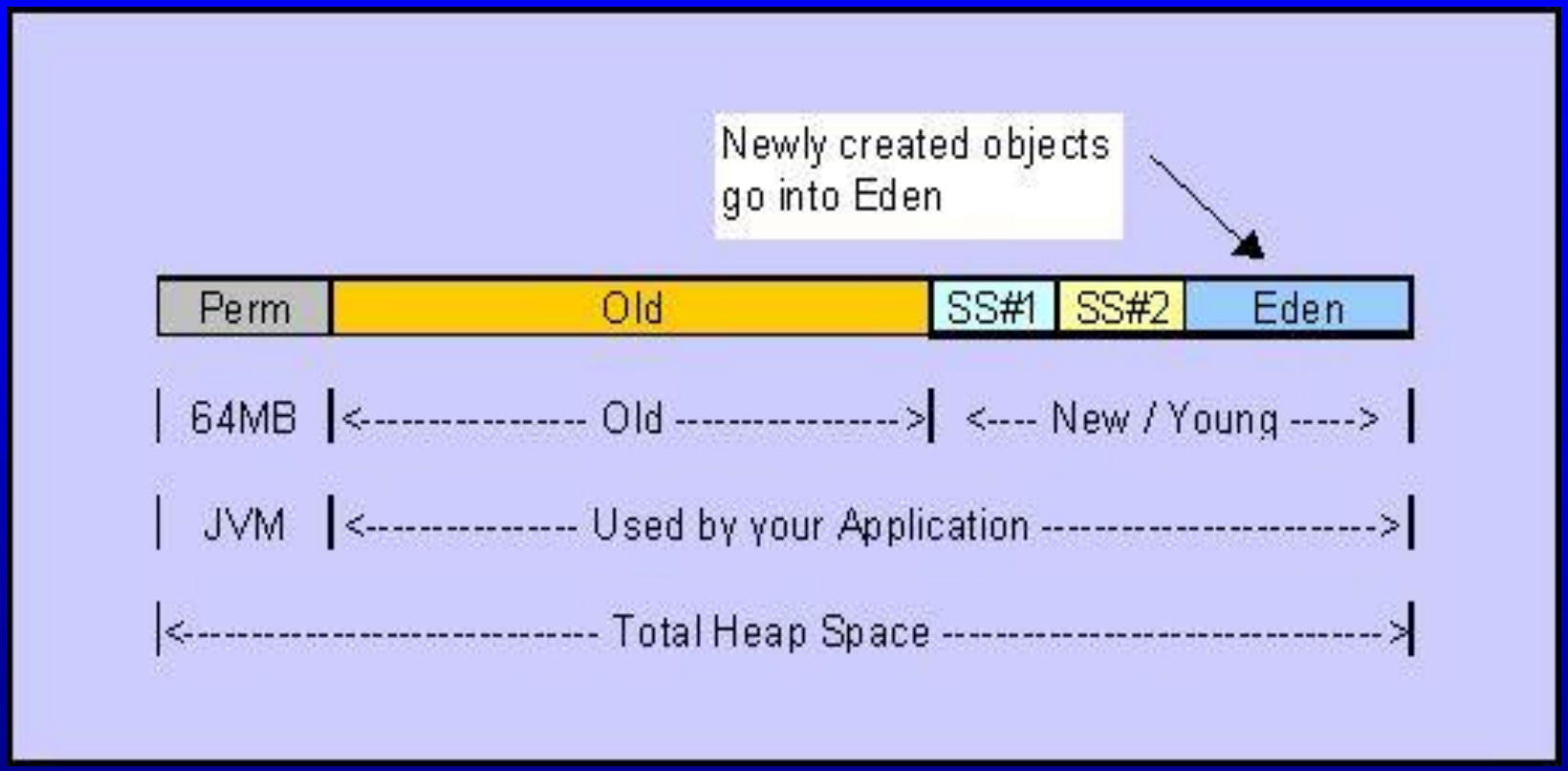
- Un GC generazionale divide gli oggetti per generazioni (diversa "vecchiaia")
- Nei vari cicli di esecuzione, fa controlli frequenti solo sugli oggetti delle generazioni più giovani
- Contemporaneamente tiene traccia della creazione di riferimenti (intra e inter generazioni)

Approccio euristico: se l'ipotesi generazionale è vera, riesce ad essere molto più veloce anche se più impreciso (possono sfuggire temporaneamente oggetti più vecchi diventati irraggiungibili)

Generazioni e regioni heap

- L'algoritmo associa “regioni” dell'heap a diverse generazioni via via più vecchie
- New/Young Generation: oggetti recenti
  - *Eden*: oggetti appena creati
  - *Survivor Space 1 e 2*: oggetti sopravvissuti a precedenti cicli di garbage collection
- Old generation: oggetti più vecchi
- Perm: oggetti permanenti usati dalla VM (definizioni di classi e metodi)

Generazioni



Algoritmo

- Ogni volta che una regione tende a riempirsi (soglia) viene invocato un ciclo di garbage collection sugli oggetti in essa contenuti
  - Gli oggetti raggiungibili della regione vengono copiati nella regione successiva (più “anziana”)
  - Gli oggetti irraggiungibili sono eliminati
  - La regione viene svuotata e può essere utilizzata per l'allocazione di nuovi oggetti
- Eccezione per il Survivor Space 1, dove gli oggetti rimangono per un certo numero di cicli prima di essere spostati nella generazione Old (*contatore*)
- *NOTA: Lo spazio assegnato alle regioni influenza la frequenza di invocazione (tradeoff → prestazioni vs Precisione)*

Osservazioni

- Le generazioni più giovani (es. Eden) tendono a riempirsi più in fretta
  - Opera di frequente ma su un set ridotto
- Realizza una garbage collection incrementale e veloce (agisce su ridotte zone dell'heap)
- Svantaggio: approccio euristico non ottimale, alcuni oggetti irraggiungibili potrebbero non esser scoperti
- Spesso usati approcci ibridi: cicli di algoritmo generazionale uniti a cicli occasionali di "major garbage collection" (su tutto l'heap)

→ESEMPI
esegui_java.sh

Major garbage collection

I termini “minor cycle” e “major cycle” sono spesso usati per riferirsi ai diversi cicli di garbage collection negli approcci ibridi

Per i cicli di major collection solitamente viene usato un algoritmo Mark-and-Sweep

Nei minor cycle usato approccio in movimento

Spesso usato anche nel major cycle per combattere la frammentazione dello spazio di memoria

- Java e .NET framework

Alternative al GC: Reference Counting

Algoritmo Reference Counting

- Alcuni linguaggi dinamici adottano alternative più snelle di un garbage collector classico
- La più interessante di tali alternative è il reference counting
  - Perl, PHP
- A ciascun oggetto viene associato un contatore degli oggetti che lo stanno referenziando
- Quando il contatore scende a 0 significa che nessuno sta più referenziando l'oggetto, che può essere eliminato

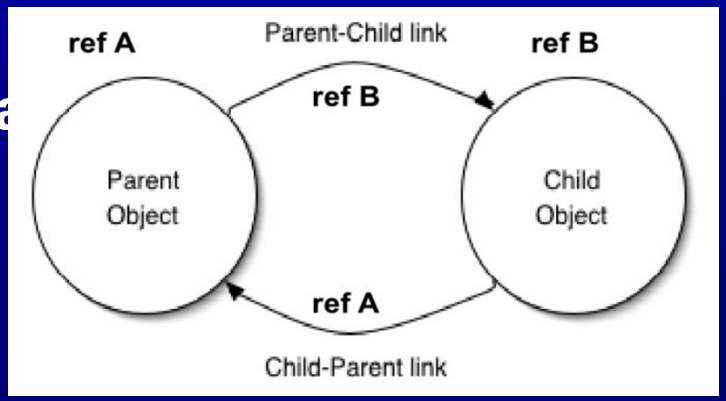
Vantaggi

- Algoritmo di reference counting: molto più veloce di un qualunque algoritmo di garbage collection
- *Ci si può letteralmente "dimenticare" di chiudere i descrittori dei file*
  - Quando si esce da un blocco di codice, il reference count della variabile descrittore va a 0
  - La variabile descrittore viene immediatamente distrutta
  - Con un garbage collector, l'operazione non è necessariamente immediata

→ Migliore reattività (promptness)

Svantaggi

- È necessario mantenere e gestire un contatore in ogni oggetto che è continuamente aggiornato
  - Overhead di memoria e risorse di calcolo
- L'algoritmo di reference counting presenta il problema dei riferimenti circolari
  - Si considerino 2 oggetti A e B che puntano l'uno all'altro
  - Sia A che B hanno due riferimenti:
    - Quello della variabile che li contiene (diretto)
    - Quello dell'oggetto che punta a



Riferimenti circolari

- Quando uno dei due oggetti (ad esempio, A) esce dal suo scope
  - Non è più accessibile tramite la sua variabile
  - La sua memoria non può essere liberata, dal momento che un altro oggetto la sta referenziando (rischio di dangling reference)
- L'effetto esterno è quello di un memory leak (consumo memoria, ma non ho la variabile)
- Soluzioni: algoritmi cycle-detecting con adozione dei riferimenti deboli *(weak reference)*
  - Riferimenti circolari esplicitamente marcati come deboli e non considerati dal reference counting