Estudi comparatiu de la gestió de memòria de dues màquines virtuals

Pere Sánchez Miranda
Enginyeria en Informàtica

Jordi Ferrer Duran

Divendres, 15 de Juny de 2007

Resum

En aquest projecte s'han comparat els gestors de memòria de les màquines virtuals de Java i del projecte Mono.

Per començar a introduir el tema s'ha fet una breu descripció del què s'entén per màquina virtual i els diferent tipus que hi ha, i seguidament s'han explicat les funcions dels gestors de memòria, els avantatges i inconvenients generals, i s'han exposat diferents mètodes de gestió de memòria amb el funcionament de cada un d'ells i els avantatges i inconvenients específics, així com les possibles situacions en què pot ser més convenient utilitzar un o un altre.

A continuació s'ha estudiat la màquina virtual de Java (arquitectura i funcionament bàsic) per continuar amb una anàlisi més detallada del seu gestor de memòria. Aquí s'han analitzat els modes en què pot funcionar el recol·lector d'escombraries, les decisions de disseny que s'han de prendre per triar el mètode més adequat, i el funcionament del gestor de memòria.

Després s'ha estudiat la màquina virtual del projecte Mono (característiques i components principals) passant a continuació a analitzar detalladament el seu gestor de memòria. S'ha començat pel gestor actual basat en el GC de Boehm-Demers-Weiser, explicant el funcionament bàsic i els passos que segueix des de la reserva de memòria quan es crea un objecte fins a la finalització dels objectes que s'han eliminat en la recollida d'escombraries.

Un tema important és el de l'SGen, el futur gestor de memòria de Mono, que actualment ja es pot provar si bé no es recomana utilitzar-lo encara en aplicacions de producció. Es preveu que aquest nou gestor millorarà substancialment l'eficiència del gestor actual. S'ha parat especial atenció tant en l'estructura com en el funcionament detallat d'aquest nou gestor.

Els últims capítols s'han dedicat a resumir les principals diferències entre aquests gestors de memòria, a la realització de les proves, a analitzar els resultats i a les conclusions finals del projecte.

Índex de continguts

Índex de figures

1 Introducció

Un dels principals problemes que sorgeixen durant el desenvolupament d'un programa i fins i tot durant la seva vida útil, és el que se'n deriva de la incorrecta gestió de la memòria per part dels programadors: oblidar-se d'alliberar la memòria reservada que ja no s'utilitza o alliberar la memòria reservada quan encara s'està utilitzant.

El primer cas pot portar a un desaprofitament de la memòria i pot arribar a bloquejar el sistema per falta de memòria.

El segon cas pot provocar errades en els resultats del programa que poden ser difícils de detectar o avortar-lo inesperadament amb la pèrdua de les dades que no s'hagin guardat.

Aquest tipus de problemes no només solen ser difícils de detectar sinó que, fins i tot quan es detecten, pot costar molt de temps trobar la part del codi responsable i resoldre-ho.

Una forma de resoldre aquestes situacions és crear un sistema que s'encarregui de la gestió automàtica de la memòria, tant de la reserva com de l'alliberament, de forma que el programador es pugui despreocupar d'aquests aspectes.

Malauradament, com sol passar en informàtica, les grans idees acostumen a portar un cost associat i rarament tenen una única solució ideal que funcioni meravellosament bé en tots els casos.

El què cal és estudiar les diverses opcions de què disposem per conèixer les característiques, els avantatges i els inconvenients de cada sistema, i així poder triar el més adequat a les nostres necessitats en funció de cada circumstància.

1.1 Justificació

Deixant de banda que l'àrea que jo hauria volgut fer no té res a veure amb aquesta i que els projectes que s'oferien no s'assemblen gens als que s'indiquen com a possibles en el pla d'estudis de PFC – Compiladors que es troba a Secretaria, el que m'ha motivat a triar aquest projecte entre les diferents possibilitats que s'oferien en aquesta àrea ha estat la possibilitat de comparar dues màquines virtuals diferents i, especialment, conèixer el projecte Mono del qual havia sentit parlar molt bé però que fins ara mai havia trobat el temps o el moment per poder-ho fer.

Una altra cosa molt important a banda de l'arquitectura i el funcionament intern de les màquines virtuals i els seus gestors de memòria és el rendiment de cada un d'ells. Aquest ha estat un apartat totalment pràctic que m'ha permès comprovar com es comporten a la realitat aquests dos gestors.

1.2 Objectius

Com en tot projecte de final de carrera, es poden distingir tres tipus o nivells d'objectius: generals, específics i desitjables.

1.2.1 Objectius generals

Els objectius generals de qualsevol projecte de final de carrera són estimular en l'estudiant la capacitat de:

- Analitzar un problema o una àrea de coneixement complexos.
- Aportar-hi solucions de síntesi extretes del bagatge de coneixements adquirits.
- Aportar-hi, dins del possible, solucions innovadores fruit de la seva reflexió i d'informacions recopilades d'altres fonts.
- Planificar adequadament el treball a fer.

- Elaborar els documents que es demanin, necessàriament una memòria que inclourà tot el desenvolupament del projecte, segons la metodologia que s'exigeixi.

1.2.2 Objectius específics d'aquest projecte

Concretament, amb aquest projecte es pretén:

- Conèixer millor la màquina virtual de Java i la màquina virtual del projecte Mono.
- Conèixer els diferents mètodes de gestió de memòria que es poden utilitzar en una màquina virtual.
- Entendre com fa una màquina virtual la gestió de memòria, en concret les màquines virtuals de Java i del projecte Mono.
- Aprendre a analitzar dades i treure'n conclusions.
- Comparar la gestió de memòria d'aquestes dues màquines virtuals, tant els mètodes utilitzats com el rendiment.
- Descobrir les millores que es produiran en les futures versions dels gestors de memòria.

1.2.3 Altres objectius desitjables

Si el temps ho hagués permès, també hauria estat desitjable:

- Analitzar el codi dels gestors de memòria estudiats en aquest projecte.
- Cercar possibles millores per a algun d'ells.

1.3 Enfocament i mètode seguit

El projecte es pot dividir en dues parts ben diferenciades:

- **Part teòrica:** en què es realitza l'estudi de les màquines virtuals i els seus respectius gestors de memòria.

- **Part pràctica:** en què es realitzen les proves necessàries per comprovar el funcionament i l'eficiència de cada un dels gestors.

La part teòrica s'ha realitzat cercant informació principalment a Internet, contrastant i complementant les diverses fonts.

En la part pràctica s'han realitzat i executat els programes amb les opcions que permeten obtenir dades sobre la gestió de la memòria i, a partir de les dades obtingudes, s'han realitzat les comparacions entre els dos gestors.

1.4 Planificació

En els següents apartats es detallen les tasques, les fites principals i la planificació que s'havien previst inicialment, així com els riscos que es podien preveure, possibles solucions i les desviacions que finalment s'han produït.

1.4.1 Tasques

1. Estudi de la JVM: cercar informació sobre la JVM i redactar el capítol 2
2. Estudi de la gestió de memòria: cercar informació sobre la gestió de memòria de la JVM i redactar el capítol 3
3. Estudi de la *garbage collection*: cercar informació sobre la *garbage collection* i redactar el capítol 4
4. Estudi sobre la implementació de la *garbage collection* per l'OpenJDK: cercar informació, estudiar el codi font i redactar el capítol 5
5. Estudi sobre la implementació de la *garbage collection* pel Projecte Mono: cercar informació, estudiar el codi font i redactar el capítol 6
6. Hipòtesis sobre el rendiment de la *garbage collection* de les dues màquines virtuals: redactar el capítol 7
7. Desenvolupament dels programes per verificar el rendiment de OpenJDK: anàlisi, disseny, programació i redacció del capítol 8
8. Desenvolupament dels programes per verificar el rendiment del Projecte Mono: anàlisi, disseny, programació i redacció del capítol 9

9. Obtenció i processament de les dades obtingudes pels programes de prova: realització de les proves, processament de les dades, realització de gràfics i redacció del capítol 10
10. Conclusions: redacció del capítol 11
11. Revisió de la memòria: completar (portada, resum, índex, introducció, glossari, bibliografia i annexos) i revisar la memòria
12. Presentació virtual: realització de la presentació virtual

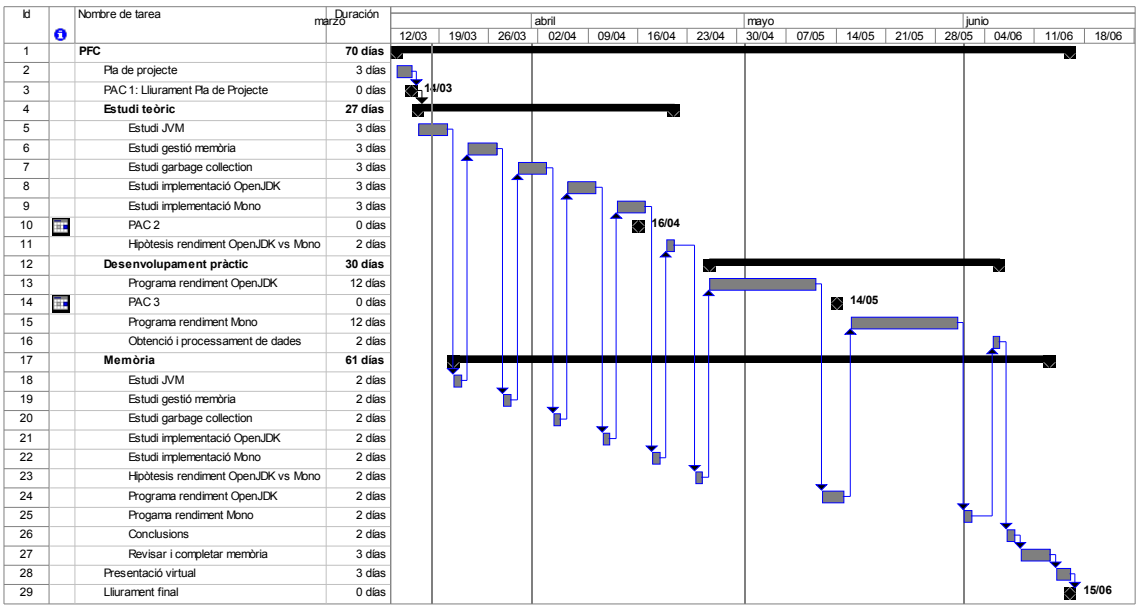
1.4.2 Temporització prevista

Tasca	Durada (dies)	Inici	Final
1	5	jue 15/03/07	mié 21/03/07
2	5	jue 22/03/07	mié 28/03/07
3	5	jue 29/03/07	mié 04/04/07
4	5	jue 05/04/07	mié 11/04/07
5	5	jue 12/04/07	mié 18/04/07
6	4	jue 19/04/07	mar 24/04/07
7	14	mié 25/04/07	lun 14/05/07
8	14	mar 15/05/07	vie 01/06/07
9	2	lun 04/06/07	mar 05/06/07
10	2	mié 06/06/07	jue 07/06/07
11	3	vie 08/06/07	mar 12/06/07
12	3	mié 13/06/07	vie 15/06/07

1.4.3 Principals fites i lliurables

- 18 – 04 – 07: Lliurament segona PAC
- 24 – 04 – 07: Finalització de l'estudi teòric
- 14 – 05 – 07: Lliurament tercera PAC
- 5 – 06 – 07: Finalització de la part pràctica
- 15 – 06 – 07: Lliurament final del projecte

1.4.4 Diagrama de Gantt



Id	Nombre de tarea	Duración
1	**PFC**	**70 días**
2	Pla de projecte	3 días
3	PAC 1: Lliurament Pla de Projecte	0 días
4	**Estudi teòric**	**27 días**
5	Estudi JVM	3 días
6	Estudi gestió memòria	3 días
7	Estudi garbage collection	3 días
8	Estudi implementació OpenJDK	3 días
9	Estudi implementació Mono	3 días
10	PAC 2	0 días
11	Hipòtesis rendiment OpenJDK vs Mono	2 días
12	**Desenvolupament pràctic**	**30 días**
13	Programa rendiment OpenJDK	12 días
14	PAC 3	0 días
15	Programa rendiment Mono	12 días
16	Obtenció i processament de dades	2 días
17	**Memòria**	**61 días**
18	Estudi JVM	2 días
19	Estudi gestió memòria	2 días
20	Estudi garbage collection	2 días
21	Estudi implementació OpenJDK	2 días
22	Estudi implementació Mono	2 días
23	Hipòtesis rendiment OpenJDK vs Mono	2 días
24	Programa rendiment OpenJDK	2 días
25	Progama rendiment Mono	2 días
26	Conclusions	2 días
27	Revisar i completar memòria	3 días
28	Presentació virtual	3 días
29	Lliurament final	0 días

1.4.5 Anàlisi de riscos

El principal risc associat a aquest projecte està una possible falta de temps de dedicació degut a problemes personals imprevistos. En cas que el retard produït sigui excessiu o es produeixi massa a prop del final de projecte, això podria impedir la culminació del projecte amb èxit.

L'única solució possible seria dedicar més hores setmanals de les previstes inicialment, traien-les de les hores de son, esbarjo, vida familiar...

Altres riscos que poden determinar la qualitat del projecte o retards imprevistos:

- **Dificultat de trobar informació sobre algun dels temes:** és poc previsible que passi donat que no són temes excessivament nous.
- **Que la informació es trobi en anglès:** és molt probable que passi però ja és massa tard per posar-se a estudiar anglès a fons, o sigui que ens haurem d'espavilar amb el que sabem, i utilitzar diccionaris i traductors on-line.
- **Dificultats en la programació:** és difícil que hi hagi algun problema en aquest aspecte ja que els programes a realitzar no tenen cap complicació a part d'aprendre algunes instruccions del llenguatge C#.

1.4.6 Desviacions

El principal problema que s'ha produït durant la realització del projecte ha estat que, degut a problemes personals de tipus familiar, he estat més de dues setmanes pràcticament sense poder avançar en el projecte.

Això ha portat a un retard en el lliurament de la tercera PAC i a haver de dedicar un munt d'hores extres en les últimes tres setmanes.

Tot i això s'ha pogut completar tot el que s'havia previst si bé no amb la profunditat que alguns temes es mereixien, especialment el últims apartats.

També s'han realitzat alguns canvis sense massa importància en l'estructura de la memòria (canviar l'ordre d'alguns capítols o posar-los en un tema apart).

1.5 Descripció dels següents capítols

- **Capítol 2:** breu descripció del que és una màquina virtual i dels diferents tipus de màquines virtuals.
- **Capítol 3:** explicació dels mètodes que es poden utilitzar per gestionar automàticament la memòria.
- **Capítol 4:** descripció de l'arquitectura, el funcionament i la implementació d'una màquina virtual Java.
- **Capítol 5:** com gestiona la memòria la màquina virtual Java. Quines són les decisions de disseny que s'han pres per dissenyar el gestor de memòria i quines decisions s'han de prendre a l'hora de dissenyar una aplicació per seleccionar el millor mètode de gestió.
- **Capítol 6:** descripció de les característiques i els components de la màquina virtual del projecte Mono.
- **Capítol 7:** com gestiona la memòria la màquina virtual del projecte Mono. Com funciona l'actual gestor de memòria i el funcionament detallat i les millores del futur gestor.

- **Capítol 8:** comparació entre els gestors de memòria de les dues màquines. Semblances, diferències, avantatges i inconvenients de cadascun, així com algunes hipòtesis sobre el rendiment.
- **Capítol 9:** proves realitzades i resultats obtinguts
- **Capítol 10:** conclusions

2 Què és una màquina virtual?

De forma molt general es pot dir que, en l'entorn informàtic, una màquina virtual és un programari que crea un entorn virtual que permet l'execució de programari independentment de la plataforma que s'utilitzi.

El cor del sistema es coneix com a monitor de màquina virtual, i s'executa sobre la plataforma de què es disposa proporcionant diverses màquines virtuals al següent nivell de programari.

Depenent de la forma i el nivell en què funcionen podem tenir diferents tipus de màquines virtuals. Les principals són les següents:

2.1 Màquines virtuals "hardware"

Aquestes màquines virtuals proporcionen la il·lusió de múltiples maquinaris en un únic ordinador o d'un ordinador amb un maquinari diferent del que hi ha en realitat. D'aquesta forma es poden executar diversos sistemes operatius simultàniament sobre un únic ordinador com si estiguessin en ordinadors independents, o executar un sistema operatiu sobre un maquinari diferent del qual havia estat programat.

Com es pot veure en la figura següent, aquestes màquines virtuals se situen entre el maquinari i els sistemes operatius.

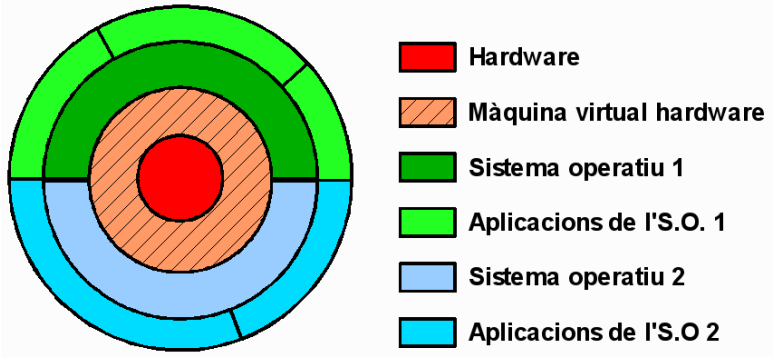


Figura 2.1: Capa on se situa una màquina virtual "hardware"

2.2 Màquines virtuals "software"

En aquest cas la finalitat és proporcionar a les aplicacions un entorn d'execució independent i aïllat del sistema operatiu i del maquinari subjacent. D'aquesta forma, fent màquines virtuals per diverses plataformes no serà necessari crear diferents versions d'una mateixa aplicació per a cada ordinador i sistema operatiu.

Actualment el cas més popular és el de Java, del qual existeixen màquines virtuals gairebé per a qualsevol plataforma, incloent telèfons mòbils i electrodomèstics.

Els principals avantatges d'aquestes màquines virtuals són:

- Independència del sistema: una aplicació feta per a una màquina virtual determinada funcionarà de la mateixa forma independentment del maquinari i programari subjacents al sistema.
- Seguretat: no hi ha la possibilitat de perjudicar el funcionament del sistema operatiu o d'altres aplicacions que s'executen fora de la màquina virtual donat que l'aplicació no interactua directament amb el sistema.

Aquest segon avantatge però, també presenta un inconvenient: al executar les aplicacions de forma totalment separada del sistema operatiu, aquestes no poden fer ús de les característiques especials de cada sistema operatiu.

En la figura següent es pot veure la situació de la màquina virtual de Java respecte del maquinari, el sistema operatiu i les aplicacions.

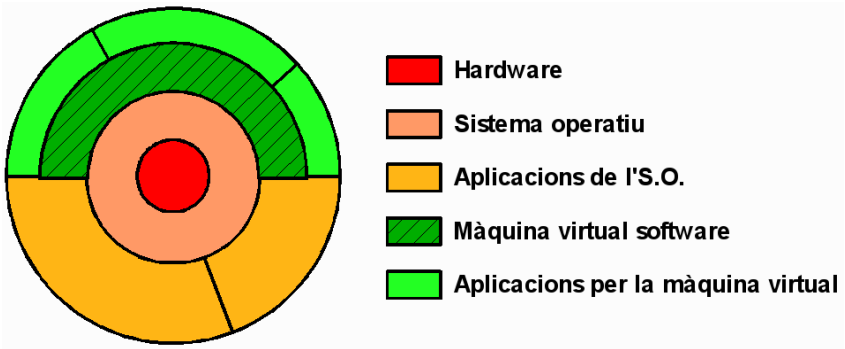


Figura 2.2: Capa on se situa una màquina virtual "software"

2.3 Agregació de diverses màquines

Tot i que no és tan freqüent, també ens podem referir amb el terme *màquina virtual* al programari capaç de fer veure un conjunt d'ordinadors com si fossin una única màquina molt més potent.

3 Sistemes gestors de memòria

Els sistemes gestors de memòria faciliten molt la feina dels programadors fent que no s'hagin de preocupar d'aquest tema, evitant errades freqüents com ara mantenir reservada la memòria que ja no s'utilitza o alliberar la memòria que encara s'està utilitzant, coses que poden provocar efectes desastrosos en els programes.

Però també cal considerar els efectes negatius d'aquests gestors sobre el funcionament dels programes:

- El temps i la memòria extra que han de dedicar per gestionar la memòria de forma eficient.
- El fet que puguin trigar un temps excessiu durant la recollida d'escombraries, cosa que els podria fer inacceptables per a aplicacions de temps real.

Per tot això, trobar el millor mètode per gestionar la memòria d'una màquina virtual és una feina complicada. D'una banda, s'ha de tenir en compte que els objectes poden ser referenciats des de diferents llocs de la memòria (les variables d'una classe, la pila, el conjunt de constants, l'àrea de mètodes, els mètodes nadius...). D'altra banda, a més d'alliberar l'espai que ocupen, convé evitar o reduir la fragmentació de l'espai lliure, i s'ha de fer de la forma més eficient possible. A més, en alguns casos s'ha evitar que el programa que s'està executant s'aturi durant llargs períodes de temps mentre es fa la recollida d'escombraries. Altres decisions a tenir en compte poden ser utilitzar algorismes de recollida en sèrie o en paral·lel, o aturar del tot el programa o permetre que segueixi treballant.

A continuació s'estudiaran alguns dels mètodes utilitzats en els GC, indicant els avantatges i inconvenients que tenen, i en quines situacions pot ser més convenient utilitzar un o altre.

3.1 Comptadors de referències

Aquest mètode serveix per determinar quins objectes ja no són referenciats enlloc i per tant es poden eliminar. El sistema utilitzat és mantenir per cada objecte un comptador de referències.

Quan objecte és creat i la seva referència s'assigna a una variable, el comptador d'aquest objecte s'inicialitza a 1. Cada cop que s'assigna la seva referència a una variable, el comptador s'incrementa en 1. Cada cop que s'elimina una variable (perquè s'ha sortit de l'àmbit del mètode on s'havia creat o perquè s'ha esborrat l'objecte que la contenia) o se li assigna una altra referència, es redueix en 1 el comptador corresponent a la referència que contenia.

En el moment en què el comptador arriba a 0, vol dir que l'objecte ja no és referenciat enlloc i per tant es pot eliminar.

Aquest mètode és força eficient, no suposa gaire sobrecàrrega pel sistema. A més, els objectes es poden eliminar en el mateix moment en què deixen de ser utilitzats, amb la qual cosa la interrupció del programa és força breu i per tant pot ser utilitzat en entorns de temps real.

El principal inconvenient d'aquest mètode és que no pot tractar correctament les referències cícliques. Un exemple seria quan un objecte pare té una referència cap al seu fill i aquest en té una cap al seu pare: en aquest cas, el comptador de referències d'aquests objectes mai arribaran a 0.

Aquest és el motiu de què aquest mètode no sigui gaire utilitzat actualment.

3.2 Traçadors

Aquests mètodes creen un graf (una mena d'arbre) amb les referències als diferents objectes. El sistema per detectar quins objectes estan referenciats consisteix en anar seguint el graf començant pels nodes arrel i marcar d'alguna manera els objectes de les referències que es van trobant.

Aquest mètode s'ha d'executar cada cert temps o quan la memòria està a punt d'exhaurir-se. Primer cal desmarcar tots els objectes, després es recorre el graf marcant els objectes que hi apareixen referenciats i finalment s'eliminen els que no estan marcats.

El principal inconvenient és que pot trigar força temps en realitzar tot el procés, temps durant el qual el programa ha d'estar aturat, i per tant no és recomanable per entorns de temps real. També en entorns interactius pot ser molest per a l'usuari haver d'esperar que s'acabi el procés si estava fent una altre cosa.

3.3 Compactadors de memòria

Els compactadors de memòria, igual que els sistemes copiadors que veurem a continuació, s'acostumen a utilitzar conjuntament amb els traçadors, i serveixen per evitar la fragmentació del *heap*. En aquest cas, el que es fa després d'eliminar els objectes no referenciats, és moure cap a un dels extrems del *heap* els objectes que queden, deixant lliure tot un bloc contigu de memòria.

Naturalment, quan es mou un objecte cal canviar totes les seves referències per tal d'actualitzar la nova posició de l'objecte. Això pot ser complicat i llarg. Una forma de fer-ho més fàcil i ràpid és afegir un nivell més d'indirecció: les referències en realitat no apunten a l'objecte sinó a una altra referència que en aquest cas sí que apunta a l'objecte. D'aquesta forma, quan es canvia la posició d'un objecte només cal canviar aquesta segona referència. Evidentment això suposa un cost extra cada cop que s'ha d'accedir a un objecte.

3.4 Copiadors

El que fan els copiadors per desfragmentar el *heap* és en realitat mantenir dues àrees de la mateixa mida, i cada cop que s'executa el GC copia els objectes que estan referenciats de una de les àrees a l 'altre. D'aquesta forma, en una de les àrees queden tots els objectes útils sense espais buits al mig, i l'altra àrea es considera que queda buida (els objectes útils s'han copiat a l'altra banda i s'han canviat les seves referències, i els objectes no referenciats es poden esborrar).

L'inconvenient d'aquest mètode és que es necessita el doble de memòria que amb el sistema de compactació.

3.5 GC Generacionals

Aquests GC representen una millora respecte dels mètodes de anteriors i es pot aplicar tant als traçadors com als de compactació i còpia d'objectes.

Una de les coses que redueix el rendiment del mètode anterior és que sovint es perd molt de temps copiant una vegada i una altra els mateixos objectes. Tenint en compte que la majoria de programes tenen un conjunt gran d'objectes de curta durada i uns quants objectes de llarga vida, es pot dividir el *heap* en diferents zones, cada una per objectes de diferent durada. En les zones que contenen objectes de vida curta, el GC actuarà més sovint; en les zones amb objectes de llarga vida, el GC actuarà més de tant en tant.

Per detectar quins objectes han d'estar en una zona o en una altra, el que es fa és posar els objectes creats en la zona d'objectes més “joves”. Cada cop que l'objecte sobreviu a un procés de recollida d'escombraries augmenta la seva edat, fins que arriba a un llindar en què es passa l'objecte a la següent zona on hi ha objectes més “vells”.

3.6 GC Incrementals

Fins ara s'ha dit que un dels principals inconvenients de la majoria dels recol·lectors d'escombraries és que poden aturar els programes durant un període de temps indeterminat i prou llarg com per fer que no siguin aconsellables en aplicacions de temps real o fins i tot per aplicacions interactives si els retards que provoquen poden arribar a ser percebuts pels usuaris.

Una possibilitat per intentar evitar aquestes situacions és utilitzar algorismes que recullin les escombraries de forma incremental en comptes d'intentar alliberar de cop tots els objectes que han deixat d'estar referenciats.

Es tracta d'intentar limitar el temps de cada recol·lecció a una durada acceptable per aquells casos en què no es desitja que els programes quedin interromputs durant un període llarg de temps.

3.7 GC Adaptatius

Com s'ha comentat al principi d'aquest capítol, alguns algorismes funcionaran millor en algunes situacions i altres ho faran millor en altres casos. Els algorismes adaptatius recullen informació sobre la situació del *heap* i utilitzen una tècnica o una altra depenent de la que considerin més adequada.

4 La màquina virtual de Java (JVM)

La màquina virtual de Java, d'ara endavant JVM (de l'anglès Java Virtual Machine), pertany al segon grup dels vistos a l'apartat anterior.

Consisteix en un programa nadiu (executable en una plataforma concreta) que s'encarrega d'interpretar i executar les instruccions d'un programa expressades en un codi binari especial (el Java bytecode) generat per un compilador a partir del codi font en llenguatge Java.

Igual que qualsevol màquina física, la JVM disposa d'uns components que defineixen la seva arquitectura: instruccions, tipus de dades, àrees de memòria i subsistemes.

4.1 Arquitectura de la JVM

4.1.1 Instruccions

El codi binari de la JVM és semblant al codi màquina de qualsevol CPU però en comptes de ser executat per una CPU física és executat per la JVM. Per tant, aquest codi ha de disposar d'instruccions pels diferents tipus de tasques que pot realitzar una CPU i que estan enumerades a continuació:

- Càrrega i emmagatzematge de dades
- Aritmètiques
- Conversió de tipus
- Creació i manipulació d'objectes
- Gestió de piles
- Transferència de control
- Invocació i retorn de/a mètodes
- Llençar excepcions

4.1.2 Tipus de dades

Els tipus de dades de la JVM es detallen a la següent taula:

Tipus	Rang
`byte`	8 bits amb signe
`short`	16 bits amb signe
`int`	32 bits amb signe
`long`	64 bits amb signe
`char`	16 bits sense signe (caràcters Unicode)
`float`	32 bits decimal simple precisió (IEEE 754)
`double`	64 bits decimal doble precisió (IEEE 754)
`reference`	32 bits (referència a un objecte en memòria, o `null`)

La unitat bàsica per a emmagatzemar les dades en memòria és el *word*, que com a mínim ha de poder contenir els tipus de fins a 32 bits. Els tipus *long* i *double* poden ocupar 2 *words*.

A la següent figura queden representats els diferents tipus i la relació que hi ha entre ells:

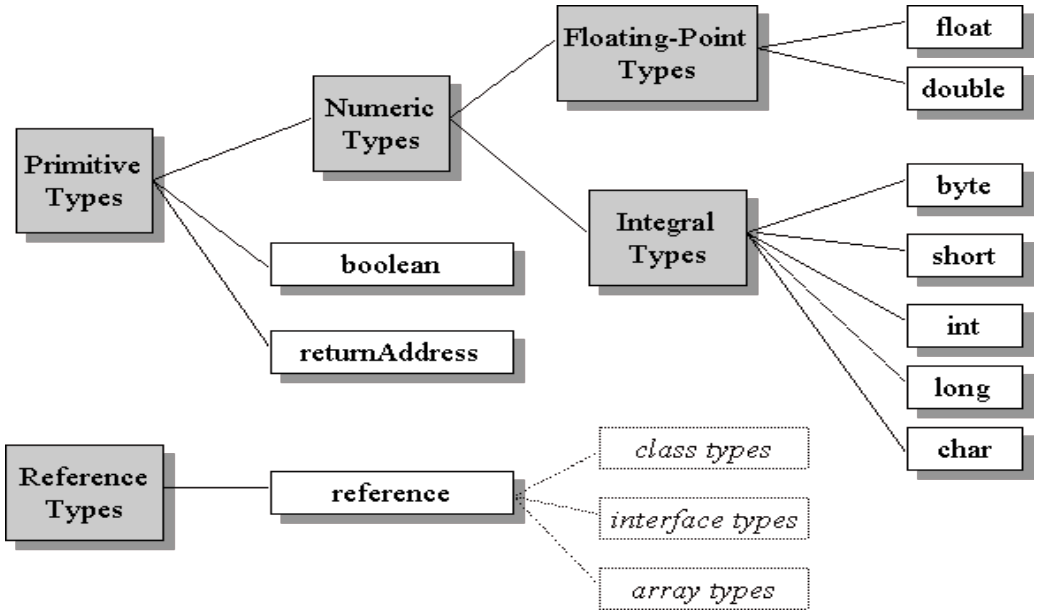


Figura 4.1: Tipus de dades de la JVM

4.1.3 Àrees de memòria

Per poder executar un programa, la JVM necessita memòria per guardar el codi i les dades. Aquesta memòria s'ha organitzat en diferents àrees, algunes de les quals són comunes a tots els fils de l'aplicació (Method area i Heap) i altres que són creades per cada fil que s'executa (Program counter registres i Java stacks).

- Method area: serveix per a emmagatzemar informació sobre cada classe que carrega la JVM, incloent les variables estàtiques, les constants i tota la informació necessària sobre els diferents mètodes de cada classe: el nom, el nombre i els tipus dels paràmetres, el tipus de retorn, així com el codi i la mida que cal reservar a la pila per les variables locals.

- Heap: emmagatzema les dades de cada instància dels objectes que va creant l'aplicació. La JVM té instruccions per reservar memòria quan es crea un objecte o una taula però no en té cap per alliberar-la. Per gestionar la memòria que ocupen els objectes que ja no són referenciats en lloc, la JVM necessita un recol·lector d'escombraries (garbage collection).

- Program Counter Registers: cada fil de l'aplicació que s'està executant té el seu propi comptador de programa que conté l'adreça de la instrucció que s'està executant actualment.

- Java Stacks: serveix per emmagatzemar els paràmetres, les variables locals, resultats intermedis i altres dades del mètode al que pertany, com ara les dades necessàries per gestionar les excepcions. En el cas dels resultats intermedis funciona com una pila introduint les dades amb instruccions tipus *push* i extraient-les amb instruccions tipus *pop*.

- Native Method Stacks: quan la JVM crida mètodes nadius de la plataforma on s'està executant, aquests mètodes no poden utilitzar la pila de dades de Java i per tant la JVM ha de reservar un espai de memòria que serveixi de pila als mètodes nadius.

Per altra banda, si considerem la memòria des del punt de vista del tipus de gestor (gestor generacional), podem trobar les següents zones:

- **Generació jove**
  - **Eden:** és on es posen inicialment els objectes creats.
  - **From / To:** contenen els objectes que han sobreviscut a la primera recollida.
- **Generació vella**
  - **Tenured:** conté els objectes que han sobreviscut a diverses recollides
- **Generació permanent:** pels objectes que s'utilitzen durant tot el programa.

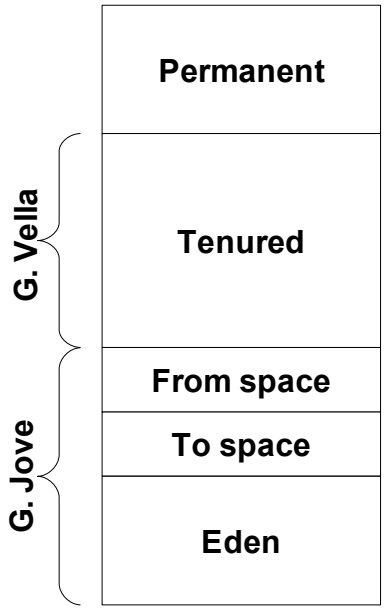


Figura 4.2: Àrees de memòria del gestor generacional

4.1.4 Subsistemes

- Execution Engine: és el nucli de qualsevol JVM. S'encarrega d'executar les instruccions (byecodes) de l'aplicació Java. Les especificacions de la JVM descriuen què s'ha de fer per cada instrucció però no diu com s'ha de fer. Per tant cada implementació pot decidir com executar els bytecodes: es podria fer en forma d'intèrpret, compilant prèviament amb un compilador en temps d'execució, executant el codi directament en una CPU que accepti codi Java de forma nadiua, utilitzant alguna altre nova tecnologia o amb qualsevol combinació de les anteriors.
- Class Loader: s'encarrega de cercar i carregar els tipus de dades que necessita l'aplicació. La JVM conté dos tipus de carregadors de classes: el *bootstrap class loader* que forma part de la implementació de la JVM i s'encarrega de les classes i interfícies de confiança (per exemple les classes de l'API de Java), i els *user-defined class loaders* que formen part de l'aplicació Java i s'encarreguen de carregar els altres tipus de classes.

- Bytecode verifier: aquest subsistema en realitat està inclòs dins el *class loader* però és una part important de la JVM doncs s'encarrega de verificar que les classes que es carreguen compleixen la semàntica de la JVM i, el més important, que no poden violar la integritat de la JVM.
- Just In Time compiler: encara que no és obligatori incloure un compilador en temps d'execució, aquest subsistema permet accelerar notablement l'execució de les aplicacions Java traduint els *bytecodes* a codi nadiu abans d'executar-lo. Això també té un inconvenient que és el temps que es triga abans de començar a executar l'aplicació.
- Native Method Interface: no és imprescindible que una JVM proporcioni una interfície per executar codi nadiu, però si ho fa ha de proporcionar mètodes per interactuar amb la JVM. A més, cal tenir en compte que els objectes que s'estan utilitzant en els mètodes nadius poden ser moguts pel recol·lector d'escombraries i s'ha d'evitar que siguin esborrats encara que hagin desaparegut totes les altres referències.
- Garbage collection: el recol·lector d'escombraries no és en realitat un subsistema en sí mateix sinó que forma part de la gestió de memòria del *Heap*, però s'ha inclòs aquí ja que és una part important del funcionament de la JVM i el principal objectiu d'aquest projecte.

A la figura següent es mostren els principals subsistemes i àrees de memòria de la JVM.

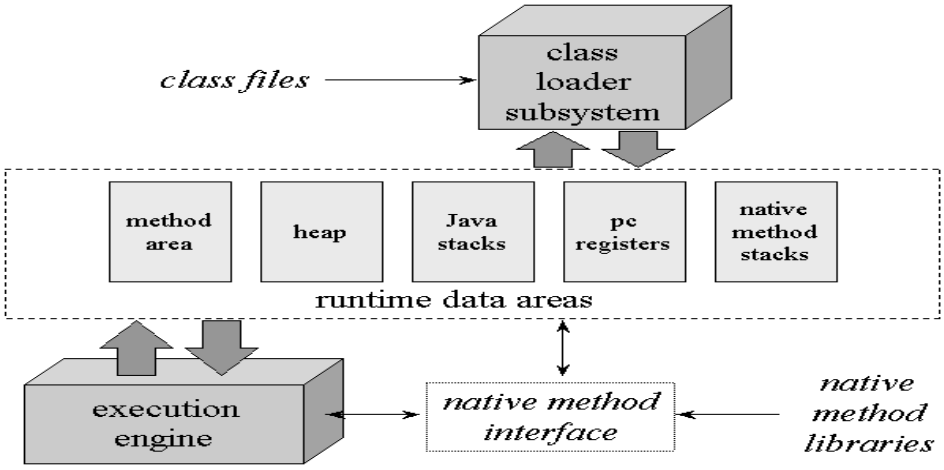


Figura 4.3: Subsistemes i àrees de memòria de la JVM

4.2 Funcionament de la JVM

La missió d'una JVM és executar una aplicació Java, per tant es crea una instància de la JVM quan una aplicació s'ha d'executar, i es destrueix quan l'aplicació acaba.

Si s'executen tres aplicacions Java, hi haurà tres instàncies de la JVM en execució, i cada aplicació s'executarà dins la seva JVM.

La JVM engega l'aplicació cridant el mètode `main()` de l'aplicació. Aquest mètode engega el fil inicial de l'aplicació que al seu torn pot engegar altres fils.

El fil principal és un fil dels anomenats *no-dimoni*. Els fils *dimoni* poden ser engegats pel fil principal o per la mateixa JVM (per exemple, el recol·lector d'escombraries).

La JVM finalitza quan no queden fils *no-dimoni* en execució o bé, si ho permet el gestor de seguretat, invocant el mètode `exit()`.

4.3 Implementació d'una JVM

Les especificacions de la JVM no indiquen com s'han de dissenyar els components interns de la JVM sinó com han de ser les interfícies de la JVM. Per tant, mentre sigui capaç de reconèixer els arxius de classe Java i reconegui i executi correctament la semàntica del codi contingut en aquests arxius, no importa com estiguin organitzats o implementats els subsistemes interns o les diferents àrees de memòria.

5 Gestió de la memòria en una JVM

Tal com s'ha comentat anteriorment, l'especificació de la JVM només indica quines instruccions ha de tenir per crear objectes i guardar-los al *heap*, però no diu res sobre l'alliberament i reaprofitament de la memòria que ocupen aquests objectes un cop han deixat de ser referenciats. Això implica que una JVM no té perquè incloure un mecanisme per aprofitar la memòria que ocupen aquest objectes que ja no s'utilitzen però, a menys que es disposi d'un sistema amb memòria il·limitada o que només s'utilitzin programes amb un nombre determinat d'objectes que hi càpiguen, es fa necessari algun mètode per gestionar adequadament la memòria de què es disposa.

Aquests mètodes que s'encarreguen de “reciclar” la memòria se'n diuen recol·lectors d'escombraries, d'ara endavant GC (de l'anglès Garbage Collector).

La seva funció no és només detectar i eliminar els objectes que ja no són referenciats enlloc, sinó que també han de cridar el mètode de finalització d'aquests objectes. A més, és molt convenient que redueixin la fragmentació del *heap* per poder aprofitar millor l'espai que es va alliberant ja que, si no es fa, es podria arribar a la situació de que no hi hagi cap espai lliure prou gran per guardar un nou objecte tot i que l'espai lliure total sigui suficient.

El fet de disposar d'un sistema que gestioni la memòria de forma automàtica té alguns avantatges:

- Els programadors no s'han de preocupar d'alliberar la memòria reservada prèviament quan ja no la necessiten i evita els problemes de “fugues” de memòria (*memory leaks*) o accessos a objectes que ja s'han alliberat.
- Ajuda a garantir la integritat dels programes i per tant la seguretat dins la JVM ja que no es pot espatllar degut a un alliberament incorrecte de memòria per part del programador, sigui intencionat o no.

Però també hi ha alguns inconvenients:

- Els GC afecten negativament al rendiment dels programes degut a la sobrecàrrega que suposa haver de controlar quins objectes estan referenciats o no.
- No es pot conèixer amb exactitud el temps que pot trigar una rutina en executar-se donat que el GC es pot posar en marxa en qualsevol moment i no se sap el temps que pot necessitar, per això no és gaire recomanable per sistemes de temps real.

5.1 Gestió de la memòria en OpenJDK

Sun Microsystems ha alliberat sota llicència GPL v2 la seva tecnologia Java creant el projecte OpenJDK. De moment aquest projecte inclou la màquina virtual de Java, anomenada HotSpot, el compilador de Java, javac, i el sistema d'ajuda JavaHelp, i està previst que en breu alliberi també la major part de la resta de components.

En aquest apartat es detallaran els aspectes més rellevants de la gestió de memòria de la JVM HotSpot: quins mètodes de recol·lecció d'escombraries utilitza i quines decisions de disseny s'han pres per millorar el rendiment i adaptar-lo tant com sigui possible a les diferents situacions.

Finalment s'explica el funcionament del gestor de memòria de la JVM.

5.1.1 Tipus de recol·lectors d'escombraries en la JVM

Els tipus de recol·lectors de què disposa la JVM HotSpot són els quatre que veurem a continuació i, tal com s'ha dit anteriorment, es pot deixar que la JVM triï el que consideri més convenient en funció de les característiques del maquinari, o triar-ne un manualment segons les característiques de l'aplicació que s'ha d'executar.

Recol·lector sèrie

Amb aquest mètode, el programa s'atura mentre es realitza el procés de recol·lecció. Primer es duu a terme amb el nivell jove i després amb el nivell vell i el permanent. Amb el nivell jove s'utilitza el sistema de còpia mentre que en els altres nivells s'utilitza el sistema de compactació.

Aquest mètode pot ser la millor opció quan es treballa amb maquinari que només disposa d'una CPU i amb entorns o programes en què no importa la durada de les pauses degudes a la recol·lecció.

Recol·lector paral·lel

Aquest mètode és similar a l'anterior excepte que el procés de recol·lecció en el nivell jove es realitza en paral·lel.

Es recomana utilitzar-lo quan es disposa de maquinari amb més d'una CPU o CPUs multi-nucli. En aquests casos es redueix la durada de les pauses degut a la reducció del temps que es triga en fer la recol·lecció en el nivell jove. De totes formes, encara es poden produir llargues pauses en els altres nivells.

Recol·lector compactador paral·lel

Similar al cas anterior però també realitza en paral·lel el procés de recol·lecció del nivell vell i permanent.

Pot ser útil en els mateixos casos que l'anterior i redueix encara més la durada de les pauses, per la qual cosa pot ser útil fins i tot quan és important que les pauses no siguin gaire llargues.

Recol·lector concurrent

La recollida del nivell jove la realitza en paral·lel i aturant el programa, igual que en els dos casos anteriors. La recollida dels altres nivells la realitza en paral·lel i sense aturar el programa sempre que és possible.

Aquest mètode és l'únic que no compacta la memòria. Això fa que la durada de les pauses sigui inferior però allarga el procés total de la recollida, complica i alenteix l'assignació de memòria i necessita més espai.

També es pot utilitzar en un mode en què la fase concurrent es realitza de forma incremental, reduint encara més la durada de les pauses.

Per tot això, aquest mètode només es recomana en casos en què la durada de les pauses és crítica, per exemple en sistemes de temps real, però no és tan important la durada total del procés.

5.1.2 Decisions de disseny

Per tal de permetre la màxima flexibilitat i escalabilitat, la JVM pot adaptar automàticament els seus paràmetres en funció del nombre de processadors de l'ordinador i la quantitat de memòria, però també permet especificar manualment alguns d'aquests paràmetres.

Recol·lecció d'escombraries amb processat en sèrie o en paral·lel?

Quan només es disposa d'una CPU no cal plantejar-se aquesta qüestió, però si es disposa de múltiples CPUs, la complexitat afegida del procés en paral·lel queda compensada pel major aprofitament del sistema que permet reduir el temps dedicat a la recollida d'escombraries.

La decisió d'utilitzar procés en paral·lel o no es pot deixar en mans de la JVM però també es pot fixar manualment, indicant fins i tot el nombre de fils en paral·lel que pot utilitzar el GC.

Recol·lecció concurrent o aturar el programa?

La forma més senzilla de fer la recollida d'escombraries és aturar el programa ja que d'aquesta manera no hi ha cap perill d'interaccions perilloses entre el programa i el GC, però això pot suposar pauses llargues que en alguns casos poden ser inacceptables.

Els GC que realitzen la recollida sense aturar el programa són més complicats de programar i necessiten més espai de memòria, però a canvi el temps que el programa està aturat és inferior.

També en aquest cas la JVM permet triar manualment si es desitja que es realitzi el procés concurrentment o no. Igualment es pot indicar el temps màxim que pot estar aturat el programa mentre es fa la recollida d'escombraries.

Compactar, no compactar o copiar?

Tot i que pot semblar més ràpid i senzill no realitzar cap mena de reorganització de la memòria alliberada, el cas és que la gestió dels espais buits es complica i es fa més lent degut a que s'han de cercar llocs adients per situar els nous objectes, i a més es desaprofita més la memòria, fent que s'hagi d'incrementar més sovint l'espai per al *heap*.

Ara bé, entre utilitzar la compactació de memòria o la còpia d'objectes, la còpia és més ràpida però necessita més espai que la compactació.

Recollida incremental o tot de cop?

Pel cas d'aplicacions que no poden estar llargs períodes de temps aturades és recomanable activar el mode de recollida incremental.

Recollida generacional

La JVM utilitza el sistema de recollida generacional. D'aquesta forma es pot utilitzar el mètode més adequat segons la vida de cada tipus d'objecte.

5.1.3 Funcionament de la recollida d'escombraries en la JVM

En la JVM HotSpot, tots els tipus de recol·lectors són del tipus generacional organitzat en 3 nivells o generacions: la generació jove, la generació vella i la permanent.

La generació jove conté una àrea anomenada Edèn, que és on es posen inicialment els objectes creats, i dues àrees més petites dedicades als objectes que han sobreviscut al menys a la primera recollida d'escombraries. En aquest nivell, s'utilitza el mètode de còpia d'objectes per mantenir el *heap* desfragmentat.

Cada cop que es fa una recollida d'escombraries, els objectes vius (que encara són referenciats en algun lloc) són copiats a la zona inactiva dels supervivents,

excepte els que són prou vells per passar a la generació vella. La resta s'esborren, per tant l'Edèn queda buit i les zones dels supervivents intercanvien les seves funcions: la zona activa queda neta i passa a zona inactiva i la zona inactiva passa a zona activa.

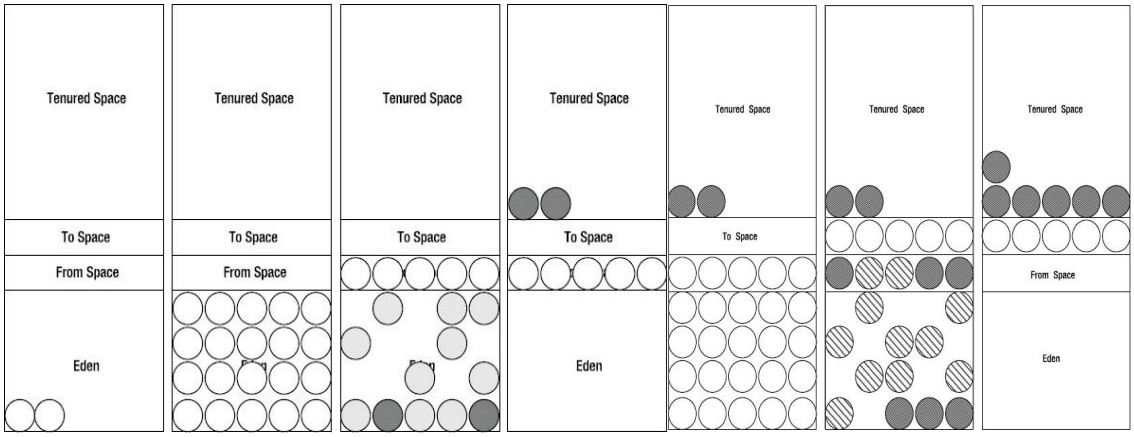


Figura 5.1: Recollida d'escombraries a la zona jove

La generació vella serveix per emmagatzemar els objectes que han sobreviscut a unes quantes recollides d'escombraries. En aquest nivell s'utilitza la compactació per mantenir el *heap* desfragmentat un cop s'han esborrat els objectes que ja no estan referenciats enlloc.

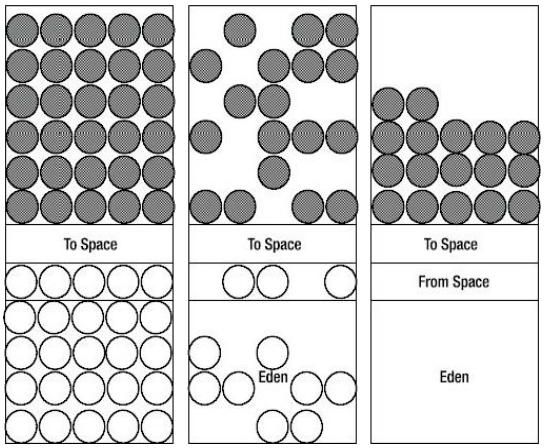


Figura 5.2: Recollida d'escombraries en la zona vella

La generació permanent és on la JVM emmagatzema els objectes que considera que tenen una durada més llarga i que gairebé mai caldrà eliminar, com ara les classes i el mètodes.

Recollida d'escombraries en els diferents nivells

Hi ha dos tipus de recollida: la recollida menor que es produeix quan s'omple l'Edèn del nivell jove, i la recollida major que es produeix quan s'omple el nivell permanent o el nivell vell. En aquest últim cas, la recollida es produeix en tots els nivells, primer en el nivell jove i després en el nivell vell i en el permanent. La compactació de memòria es realitza separadament en cada nivell.

Assignació de memòria pels nous objectes

Exceptuant el GC de tipus concurrent que es veurà més endavant, la resta de recol·lectors compacten la memòria, per tant la reserva i posicionament dels nous objectes dins el *heap* és senzill i ràpid doncs només cal situar-los a continuació de l'últim objecte creat.

5.2 Eines per avaluar el rendiment del GC

Per avaluar les prestacions del GC i poder decidir les millors estratègies, es poden utilitzar algunes dades i utilitats que proporciona la mateixa JVM:

- Percentatge de temps utilitzat pel GC
- Percentatge de temps utilitzat per l'aplicació
- Durada de les pauses mentre es realitza la recollida d'escombraries
- Freqüència de les recol·leccions
- Mides del diferents espais de memòria utilitzats
- Temps que es triga a recuperar la memòria d'un objecte sense referències
- **jmap:** permet obtenir informació de la configuració del *heap* i del mètode utilitzat pel GC

jstat: permet obtenir informació sobre el consum de recursos de l'aplicació

6 El projecte Mono

Mono és una implementació lliure de la plataforma de desenvolupament .NET Framework que pretén proporcionar el programari necessari per desenvolupar i executar aplicacions client i servidor en múltiples llenguatges i sobre múltiples plataformes.

6.1 Característiques principals

- És multiplataforma: actualment hi ha implementacions de Mono per a les arquitectures x86, x86-64, PowerPC, SPARC, Alpha, MIPS amb sistemes operatius Linux, Solaris, Mac OSX, Windows i altres.
- Multillenguatge: permet utilitzar múltiples llenguatges, sempre que hi hagi el corresponent compilador que el tradueixi al llenguatge intermedi CIL.
- Ampli suport per a múltiples tipus de bases de dades
- Basat en estàndards: segueix els estàndards ECMA/ISO.
- És programari lliure: és gratuït i de codi obert.
- Pot executar aplicacions ASP.NET i Winforms.
- Es pot contractar suport tècnic

6.2 Components

- La màquina virtual: consisteix en un intèrpret de llenguatge CIL (Common Intermediate Language). Conté un carregador de classes, un compilador JIT i un gestor de memòria.
- Compilador JIT (Just in Time compiler): tradueix el llenguatge CIL a codi màquina depenent de la plataforma mentre s'executa l'aplicació.

- Compilador AOT (Ahead of Time compiler): tradueix el llenguatge CIL a codi màquina depenent de la plataforma i crea un arxiu executable de forma que no calgui tornar-lo a traduir en successives execucions.

També compta amb els següents elements:

- Un compilador de llenguatge C#, MonoBasic (versió de VisualBasic per a Mono), Java i Python.
- Una biblioteca de classes que pot ser utilitzada per qualsevol llenguatge.
- Un entorn de desenvolupament anomenat MonoDevelop.

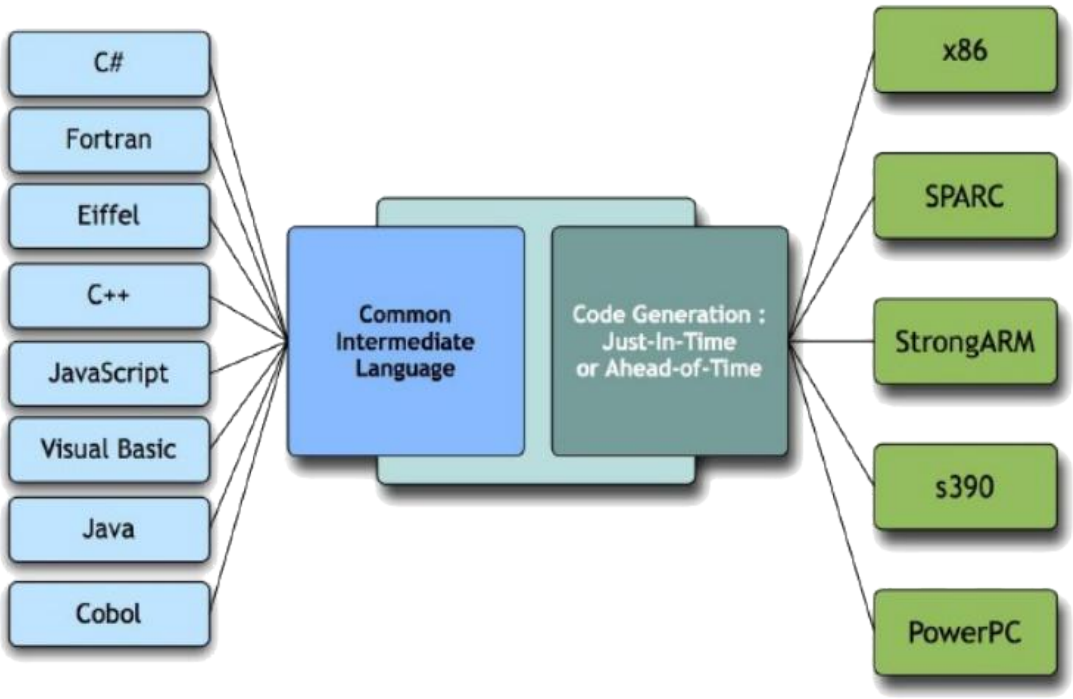


Figura 6.1: Traducció de codi font a codi màquina en Mono

7 Gestió de la memòria en el projecte Mono

Actualment, el mètode utilitzat pel projecte Mono per realitzar la gestió de memòria és el recol·lector de deixalles Bohem-Demers-Weiser, que és del tipus conservatiu, però la interfície del GC s'està aïllant per tal de permetre que s'utilitzin diferents recol·lectors de forma que es pugui triar i afinar de la forma més adequada a cada aplicació.

En aquests moments, aquest GC permet la recollida incremental i en paral·lel però no la compactació de memòria.

Tanmateix, s'està treballant en un nou recol·lector del tipus compactador generacional anomenat SGen que implementarà aquestes opcions i del qual es parlarà més endavant.

Les zones de memòria en què treballa el recol·lector de deixalles són:

- El *heap*
- Els registres i les piles de dades dels i fils
- L'àrea de dades estàtiques
- Estructures de dades assignades per la màquina virtual

7.1 El GC Boehm-Demers-Weiser

Aquest recol·lector és del tipus conservatiu (poden quedar deixalles sense recollir) però en algunes zones s'ha optimitzat de forma que detecti pràcticament tots els objectes que no són utilitzats.

Per exemple, les dades estàtiques es poden tractar amb total precisió.

També la detecció de deixalles en el heap és força precís però encara es poden donar algunes situacions en què es marqui incorrectament un punter a un objecte que en realitat no està referenciat enlloc.

En les estructures de dades assignades per la màquina virtual, algunes estructures de dades es poden tractar amb precisió però altres no, per exemple els objectes “clavats” (els que estan sent utilitzats per rutines nadiues).

Per últim, els objectes que es troben en les piles dels registres i els fils encara es tracten en forma totalment conservativa, per tant és la zona on poden quedar més deixalles sense recollir.

7.1.1 Funcionament de la recollida d'escombraries en Mono

La recol·lecció es realitza bàsicament amb un algoritme del tipus “marcar i escombrar” que, en essència, consisteix en marcar tots els objectes que poden ser localitzats seguint la cadena de punters i després esborrar aquells que no estan marcats.

A continuació s'explica més detalladament cada una de les operacions que realitza el recol·lector i les fases per les quals passa.

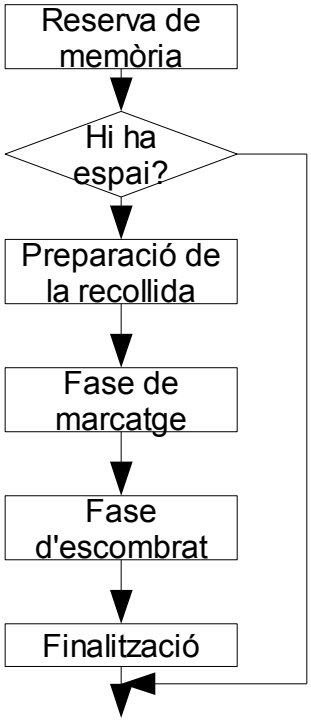


Figura 7.1: Recollida d'escombraries en Mono

Reserva de memòria per situar objectes nous

Aquest GC té el seu propi mètode de reserva de memòria, però primer obté la memòria del sistema segons la plataforma en què s'estigui executant la màquina virtual. Per exemple, en el cas de sistemes basats en UNIX es pot utilitzar `malloc`.

La majoria de dades utilitzades pel GC s'emmagatzemen en zones específiques, a part dels objectes del programa, per tal de poder-les ignorar fàcilment quan es realitza la recollida d'escombraries.

El GC tracta els diferents tipus d'objectes de forma diferent, tenint en compte que hi ha objectes que no contenen punters i per tant no cal analitzar-los en la fase de marcatge o diferenciant la forma d'assignar memòria a objectes grans o petits: en el primer cas es reserva memòria en una quantitat múltiple de la mida d'una pàgina de memòria que es gestiona amb una llista d'objectes grans,

mentre que en el segon cas es reserven trossos més petits dins d'un bloc de memòria i es mantenen en llistes separades per cada tipus i mida d'objectes.

Preparació de la fase de recollida

Cada objecte té associat un bit que permet marcar-lo i desmarcar-lo per indicar si l'objecte és accessible o no.

El primer que cal fer és marcar tots els objectes com si no fossin accessibles.

Fase de marcatge

En aquesta fase es marquen tots els objectes que són accessibles seguint la cadena de punters que comença en les variables arrels. Aquestes variables es poden trobar en la zona de registres, en la pila i en la zona de dades estàtiques.

Aquesta fase es pot realitzar de manera incremental i s'ha optimitzat al màxim donat que habitualment consumeix gran part del temps dedicat a la recollida d'escombraries. De fet, això permet utilitzar el mateix mètode quan es treballa en el mode en paral·lel.

Cada pila és processada examinant tots els punters candidats, que normalment seran aquells que apunten a una adreça alineada amb 4 bytes en un processador de 32 bits o amb 8 bytes en un de 64 bits.

Per determinar si el punter candidat conté realment l'adreça d'un bloc dins del *heap*, al final de la fase de marcatge s'esborren els bits de les llistes lliures restants per evitar el cas que una llista lliure hagi estat marcada accidentalment per una dada identificada errròniament com un punter a un objecte.

Fase d'escombrat

Al final de la fase de marcatge s'examinen tots els blocs de memòria (pàgines) que hi ha al *heap*.

Els blocs que contenen objectes grans que ja no s'utilitzen es posen a la llista de blocs lliures.

Els blocs que contenen objectes petits s'analitzen comprovant si tots els objectes que contenen es poden esborrar. Si és així, el bloc es posa a la llista de blocs lliures. Si encara conté algun objecte “viu”, es posa en una llista per fer un escombrat posterior excepte si l'espai lliure que queda en el bloc és molt petit.

Fase de finalització

Un cop s'ha realitzat la fase de marcatge, tots els objectes identificats com a prescindibles són afegits a una llista per tal d'executar seguidament les seves rutines de finalització.

De moment, la fase de finalització no es realitza de forma incremental.

7.2 SGen: el futur GC de Mono

Aquest recol·lector encara està en fase de desenvolupament i la documentació no està completa però ja està disponible pels programadors que vulguin començar a fer proves.

S'ha decidit implementar aquest nou recol·lector per superar algunes de les limitacions dels recol·lectors purament conservatius, com ara el consum excessiu de memòria degut a la fragmentació del *heap*.

Pertany al tipus de recol·lectors compactadors generacionals i tindrà les següents característiques:

- Recol·lecció precisa
- Compactació de la memòria
- Zones independents per cada fil del programa
- Utilització de barreres d'escriptura per reduir la feina en recollides petites

Tot i que la compactació de memòria pot ajudar a reduir el consum de memòria evitant ampliacions innecessàries del *heap*, cal tenir en compte alguns inconvenients:

- Alguns objectes no es poden moure
- Moure objectes grans resulta molt costós en temps

Per resoldre el primer inconvenient, s'ha creat un mètode específic que es pot cridar per reservar memòria per objectes que no es poden moure.

També es pot reservar memòria de forma específica per les objectes arrel.

Per minimitzar el segon punt, SGen situa els objectes grans en pàgines de memòria gestionades directament pel sistema operatiu, de forma que quan s'alliberen, la memòria retorna directament al sistema operatiu.

De totes formes, s'està considerant tenir en compte l'"edat" dels objectes per situar-los en una zona especial.

Tal com s'ha comentat anteriorment, s'està realitzant una interfície pública separada i amb uns pocs mètodes que serà comú a tots els mètodes de recollida d'escombraries, i una interfície interna més extensa per a ús intern.

7.2.1 Funcionament general de SGen

SGen gestiona la memòria separant-la en tres zones:

- La "guarderia" que és on se situen els objectes de mida petita que s'acaben de crear.
- El magatzem de grans objectes.
- El magatzem d'objectes antics que és on se situen els objectes petits que es troben en la guarderia quan es realitza una recol·lecció de deixalles.

A més, la reserva de memòria es fa de diferent forma depenent de la mida i el tipus de l'objecte.

La recollida d'escombraries es realitza quan s'intenta reservar memòria però ja no en queda de lliure. Es distingeixen dos tipus de recollides: la recollida general i la recollida a la guarderia. Això es fa així perquè s'ha comprovat que la majoria de programes creen molts objectes que tindran una curta durada, per

tant és molt probable que la majoria d'objectes creats recentment siguin esborrats en la següent recollida d'escombraries. Per tant, com que només s'hauran de passar una petita part dels objectes de la guarderia al magatzem d'objectes antics, tot i que aquest procés es realitzi freqüentment, la durada serà curta.

Quan tampoc queda prou espai al magatzem d'objectes antics, llavors es posa en marxa la recollida general que consisteix en l'alliberament de la memòria dels objectes no referenciats d'aquesta zona i la compactació.

Només s'haurà d'augmentar la memòria dedicada a aquesta àrea en cas que la recollida general no deixi prou espai pels objectes que s'han de portar de la guarderia.

La complicació d'aquest sistema és deguda a diverses qüestions:

- Ha de poder treballar amb múltiples fils i la recollida l'ha de poder engegar qualsevol fil.
- Cal tenir en compte els objectes que no es poden moure (objectes “clavats”).
- Costa massa moure els objectes grans, per tant cal gestionar-los de forma diferent, utilitzant la tècnica de marcar i escombrar quan es realitza una recollida general.
- També hi ha altres objectes que no poden ser moguts, com ara les cadenes internes i gestors de tipus. Aquests també es gestionen amb el mètode de marcar i escombrar.

Per poder realitzar la recollida, tant si és una recollida a la guarderia com si és una recollida general, SGen ha d'aturar tots els fils que s'estiguin executant per tal de poder moure els objectes sense perill de que el programa pugui utilitzar els punters dels objectes que s'han de moure.

Veiem més detalladament tots aquests aspectes.

7.2.2 Gestió de la memòria a baix nivell

La reserva de memòria pels objectes nous es fa de dues formes diferents depenent de la mida de l'objecte.

Quan l'objecte és gran, es demana directament al sistema operatiu. Els bloc obtinguts tenen la mida d'una pàgina i estan alineats i gestionats amb el sistema de pàgines del sistema operatiu. Quan l'objecte emmagatzemat ja no s'utilitza, s'allibera aquest bloc de memòria retornant-lo al sistema operatiu.

Aquests objectes no participen en el procés de còpia/compactació donat que degut a la seva mida resultaria massa costós en temps. En canvi, es gestionen amb una llista global a part i amb el sistema de marcar i escombrar quan es realitza una recollida general.

Quan l'objecte és petit, primer s'intenta localitzar un espai lliure prou gran a la guarderia. Si hi ha prou espai, se situa a continuació de l'últim objecte. Si no hi ha prou espai es realitza una recollida d'escombraries a la guarderia. Si no hi ha espai a la zona d'objectes antics per copiar els objectes vius de la guarderi, es realitza una recollida general i es compacta la zona d'objectes antics. Si un cop fet això encara falta espai a la guarderia o a la zona d'objectes antics, el GC sol·licita al sistema operatiu un bloc de memòria de la mida d'una pàgina per afegir-la a la zona que la necessiti.

Igual que passa amb els blocs dels objectes grans, els nous bloc reservats es poden tornar al sistema operatiu si queden buits.

7.2.3 Aturar el programa

Quan s'ha de realitzar una recollida d'escombraries cal aturar tots els fils del programa que s'està executant. Això garanteix que no hi haurà canvis en els punters dels objectes mentre el GC està copiant o movent els objectes i actualitzant els punters a aquests objectes.

Per aturar els fils s'utilitzen un parell de senyals: una per indicar que cal aturar els fils i l'altra per indicar que ja es poden tornar a posar en marxa.

Per poder-ho fer, la màquina virtual del projecte Mono porta un control de tots els fils que es posen en marxa per tal de poder-los enviar aquests senyals. Però també és necessari tenir constància de qualsevol altre fil que s'hagi de comunicar amb la màquina virtual. Per això cal que els desenvolupadors que programin aquest tipus de rutines registrin també els seus fils cridant la funció `mono_gc_register_thread`.

El control dels fils que s'estan executant es realitza amb una taula de tipus hash. Els elements d'aquesta taula contenen entre d'altres coses informació sobre el final de la pila que pertany al fil i també els objectes sobre els quals s'han realitzat modificacions en els punters a altres objectes de la guarderia.

7.2.4 Fase de marcatge d'objectes

Aquesta fase consisteix en marcar els objectes per tal que el GC sàpiga quins objectes no es poden moure, quins s'han de moure i quins es poden eliminar.

Primer es marquen tots els objectes com si no fossin accessibles. Després es marquen els objectes que no es poden moure, i finalment, començant en els objectes arrel, se segueixen el punters a altres objectes que contenen per marcar-los com a objectes vius.

Quan s'han de moure els objectes de la guarderia a la zona d'objectes antics o per compactar la memòria, cal detectar de la forma més ràpida possible quins objectes estan clavats i quins objectes ja han estat copiats.

Per tal d'evitar les llistes o taules per guardar aquests objectes, ja que pot resultar força lent haver-les de recórrer per saber si un objecte es troba en alguna d'elles, el què s'ha fet és aprofitar un parell de bits d'un punter que contenen tots els objectes de Mono. Donat que els punters sempre apunten a una posició de memòria alineada a 4 bits, els dos bits de menys pes no s'utilitzen per calcular l'adreça ja que sempre valen zero, per tant es poden aprofitar per marcar els objectes: un per indicar si és un objecte clavat i l'altre per indicar si l'objecte ja s'ha copiat.

Un cop s'ha realitzat la recollida, aquests bits es tornaran a posar a zero.

Gestió dels objectes arrel

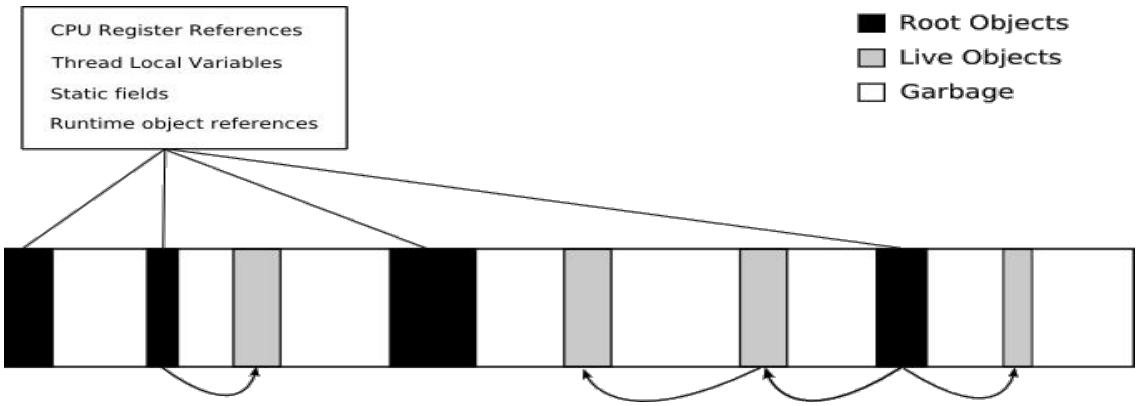
Els objectes arrel són el conjunt inicial d'objectes a partir dels quals, seguint els seus punters a altres objectes, podem trobar tots els objectes vius de l'aplicació.

Aquests objectes són objectes estàtics creats a l'inici del programa, registres de la CPU que contenen adreces d'altres objectes, variables emmagatzemades en les piles dels diferents fils i qualsevol altre objecte intern referenciat per la màquina virtual.

Tots els objectes als què no es pot accedir a través d'aquests, són el que en diem escombraries.

Internament, el GC gestiona dos tipus de registres: els registres arrel conservatius i els registres arrel precisos. La informació que es guarda de cada registre és l'inici i la mida del bloc de memòria d'aquest registre, i si és de tipus conservatiu o precís. Aquesta última dada servirà per indicar al GC si la recerca per marcar objectes vius s'ha de fer de forma conservativa o precisa.

Actualment, les piles i els registres de la CPU s'analitzen de forma conservativa i la resta de forma precisa.



Descriptors d'objectes arrel

Per gestionar els objectes arrel (objectes estàtics creats normalment al principi del programa) a partir dels quals es poden localitzar els objectes vius, el GC

manté un mapa de bits en el que cada bit posat a 1 es correspon amb una adreça on es troba l'adreça real d'un objecte arrel.

El mapa de bits serveix per saber ràpidament quines de les adreces reservades per objectes arrel estan lliures per situar nous objectes arrel, i quines adreces estan ocupades per començar la recerca d'objectes vius.

Quan es crea un objecte arrel, es retorna un descriptor que permet accedir a l'objecte a través d'aquest mapa de bits.

Gestió d'objectes clavats

Abans de començar la còpia d'objectes de la guarderia a la zona d'objectes antics cal identificar els objectes que no es poden canviar de lloc. Això s'aconsegueix rastrejant tant els objectes que s'han marcat explícitament en el moment de la seva creació com les piles i els registres de la màquina virtual.

La rutina encarregada de fer-ho construeix una llista amb totes les adreces aquests objectes, les ordena i elimina les que estan repetides. Un cop creada la llista es marquen tots els objectes referenciats per indicar que són objectes clavats.

El problema és que aquestes adreces no apunten necessàriament al principi de l'objecte, per tant cal trobar una forma ràpida de localitzar el principi d'un objecte a partir d'una adreça que pot estar apuntant a qualsevol posició dins d'ell.

Donat que els objectes estan situats seqüencialment dins la guarderia i es coneix la mida de cada objecte, una forma senzilla de fer-ho seria rastrejant tota la memòria des del principi i anar localitzant cada objecte fins que es trobi un, tal que l'adreça cercada estigui entre la posició inicial i la posició final d'aquest objecte. El problema d'aquest mètode és que seria massa lent.

El sistema que utilitza SGen consisteix en mantenir una taula on s'emmagatzemen les adreces d'inici d'un grup d'objectes que ocupen entre 4KB i 8KB. Llavors, per trobar un objecte, primer es busca en la taula d'adreces d'inici i un cop s'ha trobat el bloc on es troba l'objecte, s'escaneja tot el bloc fins

a trobar l'objecte concret. D'aquesta forma, un cop trobat el bloc, només cal rastrejar una zona de la guarderia relativament petita.

Un cop identificats i marcats tots els objectes que no es poden moure, ja es podrà començar el procés de còpia dels objectes vius.

Exploració conservativa

Abans de començar una recollida d'escombraries és important marcar tots els objectes que no es poden moure (objectes clavats).

Hi ha dos tipus d'objectes clavats: els que han estat marcats explícitament així per la màquina virtual o pel programador, i els que han estat marcats en ser localitzats per una exploració conservativa de les piles i dels registres.

Aquest últim cas és necessari perquè actualment Mono no segueix l'ús que es fa dels objectes que es troben a les piles ni als registres, per tant no pot saber quins valors dels que es guarden en les piles són punters a objectes ni quins valors dels registres apuntaven a objectes en el moment en que es van aturar els fils.

Donat que el GC no pot distingir entre les variables que contenen valors i les que sembla que contenen punters a objectes, el que fa és “clavar” tots els objectes que sembla que són referenciats per alguna variable tant de les piles com dels registres per tal d'evitar que es moguin aquests objectes i canviar algun valor que pugui no ser un punter a un objecte.

Una versió futura de Mono hauria de poder conèixer quines referències a objectes estan emmagatzemades en les piles i quines estan en els registres. D'aquesta forma es podrien examinar les piles i els registres de forma precisa en lloc d'haver-ho de fer de forma conservativa. Això suposaria un gran avantatge doncs s'evitaria el temps que es triga en clavar aquests objectes i conseqüentment quedarien menys objectes clavats en la guarderia, cosa que reduiria la fragmentació i per tant milloraria el temps que es triga en reservar espai pels nous objectes.

7.2.5 Fase de recollida

En aquesta fase mouen els objectes vius, sigui de la guarderia a la zona d'objectes antics o dins de la zona d'objectes antics per compactar la memòria.

Es poden produir dos tipus de recollida:

- Recollida a la guarderia: es produeix quan s'intenta reservar espai per a un nou objecte però a la guarderia no en queda prou. Aquest tipus de recollida s'acostuma a produir sovint i es realitza força ràpidament.
- Recollida general: es produeix quan, en produir-se una recollida a la guarderia, no hi ha prou espai a la zona d'objectes antics per traslladar els objectes vius de la guarderia. Aquesta recollida es produeix menys sovint però acostuma a ser més lenta que l'anterior.

Recollida a la guarderia

En un moment donat, a la guarderia es poden trobar objectes “clavats”, objectes “vius” i objectes “morts” o escombraries. Els objectes “clavats” són els que no es poden moure, els objectes “vius” són els que són referenciats per altres objectes, i les escombraries són aquells objectes que ja no estan referenciats enlloc.

Quan s'intenta reservar espai a la guarderia per a un objecte nou, si no es troba prou espai s'inicia una recollida d'escombraries a la guarderia. Els objectes “clavats” no es toquen d'allà on estan, els objectes “vius” es mouen a la zona d'objectes antics, i els objectes morts s'eliminen, deixant lliure l'espai de memòria que ocupaven.

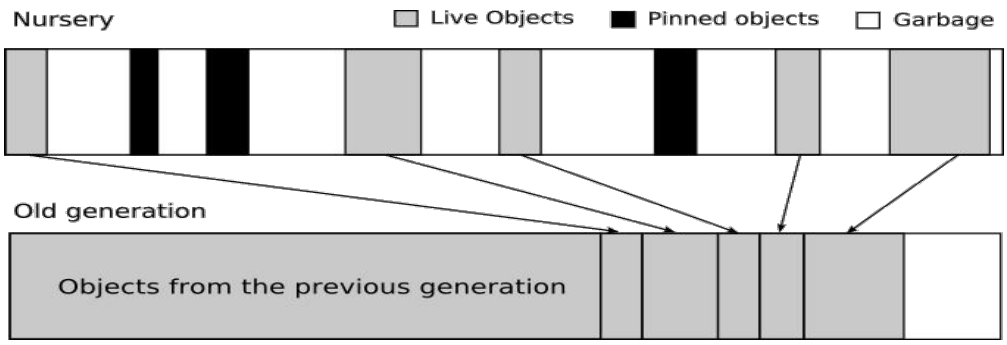


Figura 7.2: Recollida d'escombraries a la guarderia del projecte Mono

En aquest moment només quedaran a la guarderia els objectes “clavats”; la resta de la memòria de la guarderia queda lliure.

Degut als objectes “clavats”, l'espai de memòria de la guarderia pot quedar fragmentat. Per millorar l'eficiència de l'assignació de memòria en un espai fragmentat és convenient portar un control dels diferents fragments indicant on comença i on acaba cadascun d'ells.

Funcionament detallat de la guarderia

En un principi, els nous objectes se situen dins la guarderia a continuació de l'últim objecte situat. Això es pot fer ràpidament gràcies a un punter que s'actualitza cada cop que s'afegeix un objecte indicant el primer lloc lliure. També hi ha dos punters que indiquen el principi el final de la guarderia per evitar situar objectes més enllà de límit.

Si hi ha objectes “clavats”, quan es fa una recollida d'escombraries la memòria de la guarderia pot quedar fragmentada. En aquest cas, a més del punter que indica el següent lloc on es pot situar un objecte, també es necessita un altre punter per indicar el final del fragment i una llista per poder saber en quina posició de la memòria comença i acaba cada fragment.

De moment aquestes dades són globals, però es preveu que hi hagi un joc per cada fil de forma que es puguin situar nous objectes des de qualsevol fil sense haver de bloquejar el GC.

Millorant l'eficiència de la recollida de la guarderia

Una forma de reduir el temps que es triga en realitzar la recollida de la guarderia és guardant en una llista aquells objectes antics en què ha canviat algun dels seus camps fent que apunti a un objecte de la guarderia. D'aquesta forma no serà necessari recórrer tots els objectes antics per marcar els objectes vius de la guarderia: hi haurà prou seguint les referències dels objectes d'aquesta llista.

Aquests canvis en els punters d'un objecte es poden produir quan s'actualitza el valor d'algun camp d'un objecte o bé quan es mou un objecte de la guarderia a la zona d'objectes antics.

En el primer cas és el mateix compilador el que genera el codi necessari per posar l'objecte modificat en una llista pertanyent al mateix fil.

En el segon cas és el GC qui s'encarrega de posar els objectes afectats en una llista global.

D'aquesta forma, quan es fa la recollida de la guarderia, només cal mirar en aquestes llistes per saber quins objectes cal seguir per trobar els objectes vius que es troben a la guarderia en lloc d'haver de seguir tots els objectes.

Recollida general

Quan s'ha de realitzar una recollida general d'escombraries, es comproven tots els objectes, tant els objectes antics (els que han sobreviscut a una recollida a la guarderia i per tant es troben a la zona d'objectes antics), com els objectes joves (els que encara es troben a la guarderia).

En quest cas, a més de copiar els objectes vius de la guarderia a la zona d'objectes antics i compactar aquesta zona, també es realitza el procés de marcar i escombrar els objectes clavats i els objectes grans que no estan referenciats.

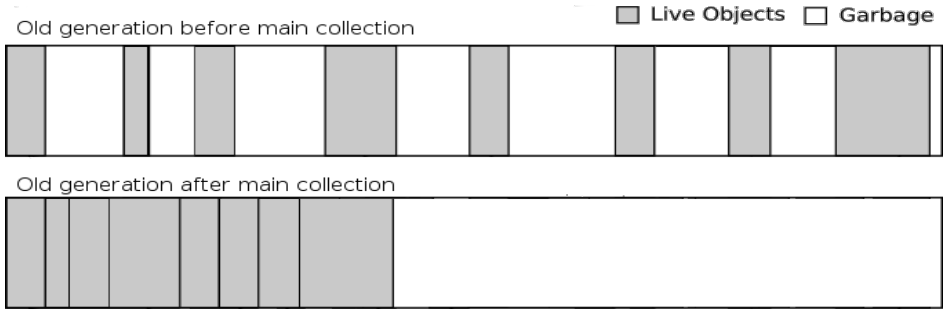


Figura 7.3: Recollida d'escombraries a la zona vella de Mono

Còpia d'objectes

Quan es realitza la recollida d'escombraries i la compactació de la memòria de la zona d'objectes antics, els objectes vius s'han de copiar o moure d'un lloc a

un altre, però cal fer-ho de forma que després es puguin actualitzar les adreces dels objectes que hi fan referència.

La còpia d'objectes vius es realitza bit a bit, però un cop s'ha copiat l'objecte cal realitzar dos processos més.

D'una banda s'ha de marcar la còpia antiga per tal d'indicar que l'objecte ja ha estat copiat. Això es fa amb un dels dos bits d'una variable de tipus adreça que conté tot objecte (recordem que l'altre bit s'utilitzava per indicar que l'objecte no es pot moure).

D'altra banda, en la còpia antiga de l'objecte s'actualitza un punter que també contenen tots els objectes i que serveix per indicar la nova adreça de l'objecte. D'aquesta forma, un cop copiats tots els objectes, es podran actualitzar correctament els punters dels objectes que hi feien referència. Només caldrà realitzar el mateix procés seguit per trobar els objectes vius però ara, en lloc de marcar-los, s'actualitzarà la seva adreça a tot arreu on es fa referència.

8 Comparació entre JVM GC i Mono GC

8.1 Semblances

8.1.1 Àrees de memòria

Tots dos gestors de memòria tenen en comú les següents àrees:

- El heap, on s'emmagatzemen les dades de cada instància dels objectes que va creant l'aplicació.
- L'àrea de registres, on es guarden els registres propis de la màquina virtual.
- L'àrea per les piles dels diferents fils, on es guarden els paràmetres, les variables locals, els resultats intermedis i altres dades dels mètodes que s'executen en cada fil.

Si ens fixem en l'estructura interna d'aquestes àrees podem dir que, en el cas del nou gestor de Mono, la semblança amb el de la JVM és molt gran donat que tots dos utilitzen un recol·lector generacional. Tenen en comú una àrea per objectes joves i una per objectes vells.

8.1.2 Mètodes de gestió

Aquí cal distingir entre el gestor actual de Mono i el futur SGen.

En el primer cas, les semblances són mínimes: només el mètode utilitzat per localitzar i eliminar els objectes és en tots dos casos el mètode de marcar i esborrar.

En el segon cas hi ha moltes més semblances donat que tots dos utilitzen un sistema generacional amb compactació a la zona vella.

8.2 Diferències

8.2.1 Àrees de memòria

El GC de la JVM té una àrea específica per guardar informació sobre cada classe que carrega la màquina virtual, incloent variables estàtiques, constants i tota la informació necessària sobre els mètodes de cada classe.

També conté una àrea per les piles dels mètodes nadius.

Per la seva part, el GC de Mono té una àrea específica dedicada a guardar les estructures de dades assignades per la màquina virtual i una per les dades estàtiques.

També té una àrea específica pels objectes grans.

Si ens fixem en l'estructura interna d'aquestes àrees podem dir que, en el cas del gestor actual de Mono, són totalment diferents donat que un utilitza un sistema generacional i l'altre no.

Si comparem el nou gestor de Mono amb la JVM, tot i semblar-se molt, hi ha una diferència bàsica: el gestor de la JVM té dues zones que s'utilitzen amb el mètode de còpia abans de passar els objectes a la zona d'objectes vells.

8.2.2 Mètodes de gestió

Aquí també hem de distingir entre el gestor actual de Mono i el futur SGen.

En el primer cas les diferències són notables ja que JVM utilitza un sistema generacional amb compactació de memòria i unes zones intermèdies entre la zona jova i la zona vella on s'utilitza el mètode de còpia. En canvi, l'actual gestor de Mono no distingeix entre objectes joves i vells, i no realitza compactació. En canvi sí que diferencia els objectes grans, cosa que no fa la JVM.

En el cas de l'SGen, la principal diferència amb JVM està en les zones intermèdies de la JVM (From space i To space).

9 Realització i resultats de les proves

Per comprovar el funcionament i l'eficiència dels gestors de memòria s'han realitzat dos petits programes, un per cada màquina virtual però tan semblants com ha estat possible, que intenten simular el funcionament un programa normal en els aspectes relacionats amb la gestió de memòria.

9.1 Característiques de l'entorn de les proves

Els programes s'han realitzat amb l'IDE NetBeans 5.5 per la JVM i amb l'IDE MonoDevelop per Mono.

Per obtenir els resultats sobre el funcionament dels gestors s'han utilitzat les següents opcions de les màquines virtuals:

- Per JVM s'ha utilitzat `-XX:+PrintGCTimeStamps` i `-XX:+PrintGCDetails` que permeten obtenir uns resultats molt detallats, tant de l'evolució de la mida i ocupació de les diferents àrees de memòria com del temps que s'ha trigat en cada una de les operacions.
- Per Mono, l'opció més semblant és `--profile=default:alloc`.

També s'ha comprovat que no s'utilitzés memòria virtual ja que l'intercanvi de dades amb el disc dur podria desvirtuar els resultats.

En cap cas s'han canviat les opcions per defecte referents als modes de recollida en sèrie, paral·lel o concurrent ja que les proves s'han realitzat en un ordinador amb una única CPU d'un sol nucli i per tant no s'hauria notat cap millora.

Tampoc s'ha aprofitat l'opció de compilació amb el AOT (Ahead of Time Compiler) que genera un programa directament executable per evitar donar avantatge a Mono.

9.2 Característiques del programa de prova

S'ha comprovat que, en la majoria de programes, només el 10% dels objectes resten a la memòria durant un període llarg de temps mentre que el 90% són objectes de curta durada.

El programa intenta simular aquesta característica creant un 10% d'objectes que no s'esborraran durant l'execució del programa, mentre que el 90% restant dels objectes creats s'aniran eliminant aleatòriament essent substituïts per altres objectes.

Respecte de la mida dels objectes, s'han creat tres tipus d'objectes: un de mida petita (uns 250 bytes), un de mida mitjana (aproximadament 1,2 KB) i un de mida molt gran (250 KB). Els de mida petita seran els més utilitzats i els de mida mitjana els que menys en una proporció de 80% i 20% respectivament. Encara que no sigui habitual treballar amb objectes molt grans, també s'ha fet una prova específica amb aquest tipus d'objectes tenint en compte que el gestor de Mono tracta de forma diferent aquests objectes i per tant ha semblat interessant comprovar si hi havia diferències notables en el rendiment respecte de la prova normal.

Per últim, s'ha decidit crear un gran nombre d'objectes de forma que també es treballi amb el redimensionament de les àrees de memòria.

9.3 Resultats de les proves

Per minimitzar l'efecte de les opcions que mostren l'evolució dels gestors de memòria sobre el temps que han trigat les diverses proves, primer s'ha calculat el temps sense posar les opcions que permeten obtenir els resultats sobre la memòria.

Per tal d'obtenir resultats fiables, s'han realitzat diverses proves i s'ha fet la mitjana dels resultats obtinguts.

Durada de cada prova (en mil·lisegons)		
	JVM	Mono
Prova 1	12.624	18.635
Prova 2	6.492	5.446

Consum de memòria (en KB)		
	JVM	Mono
Prova 1	59.788	344.404
Prova 2	50.048	117.204

10 Conclusions

La principal conclusió que es pot treure d'aquest estudi és que, tot i que actualment hi ha força diferències entre els gestors de memòria de JVM i de Mono, la tendència amb el nou gestor SGen és a assemblar-se cada cop més, cosa que demostra la millor eficiència del mètode generacional combinat amb la còpia i compactació.

En l'apartat de consum de memòria es veu clarament que JVM és millor ja que consumeix només un 17% i un 43% respecte el que consumeix Mono. Això era d'esperar tenint en compte que Mono no compacta la memòria. Aquí es veu clarament que els mètodes de compactació són millors tant amb objectes petits o mitjans com amb objectes grans

En l'apartat de rendiment, quan es treballa amb objectes de mida no massa gran, el gestor de la JVM és més eficient: Mono triga un 50% més. Però quan es tracta de treballar amb objectes grans, Mono és millor: JVM triga un 20% més.

11 Bibliografia

Documentació escrita

“Memory Management in the Java HotSpot Virtual Machine”
http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

“Performance Through Parallelism: Java HotSpot Virtual machine Garbage Collection Improvements”
http://developers.sun.com/learning/javaoneonline/2006/coreplatform/TS-1168.pdf

Documentació electrònica

http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.htm

http://www.artima.com/insidejvm/ed2/index.html

http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html

https://openjdk.dev.java.net/hotspot/docs/StorageManagement.html

http://java.sun.com/docs/hotspot/gc/

http://www.mono-project.com/Compacting_GC

http://www.mono-project.com/Performance_Tips

http://www.hpl.hp.com/personal/Hans_Boehm/gc/

http://www.iecc.com/gclist/GC-algorithms.html

http://en.wikipedia.org/