Εικονικές Μηχανές, Διερμηνείς και Δυναμική Διαχείριση Μνήμης

Wassily Kandinsky, *Composition VII*, 1913

Κωστής Σαγώνας <kostis@cs.ntua.gr>

Περιεχόμενα

- Εικονικές μηχανές (Virtual Machines)
- Διερμηνείς (Interpreters)
- Δυναμική διαχείριση μνήμης
  - Στοίβες (Stacks)
  - Σωροί (Heaps)
- Συλλογή σκουπιδιών (Garbage Collection)
  - Μαρκάρισμα και σκούπισμα (mark and sweep)
  - Αντιγραφή (copying)
  - Μέτρημα αναφορών (reference counting)

Εικονικές Μηχανές (Virtual Machines)

Οι εικονικές μηχανές (VMs) αποτελούν ένα ενδιάμεσο στάδιο στη μεταγλώττιση των γλωσσών προγραμματισμού

- Οι VMs είναι μηχανές διότι επιτρέπουν τη βήμα προς βήμα εκτέλεση των προγραμμάτων
- Οι VMs είναι εικονικές (αφηρημένες) διότι συνήθως
  - δεν υλοποιούνται σε hardware
  - παραλείπουν πολλές από τις λεπτομέρειες των πραγματικών υπολογιστικών μηχανών (αυτών σε hardware)
- Οι VMs are tailored to the particular operations required to implement a particular (class of) source language(s)

Εικονικές Μηχανές: Πλεονεκτήματα

- Γεφυρώνουν το χάσμα μεταξύ του υψηλού επιπέδου των γλωσσών προγραμματισμού και του χαμηλού επιπέδου των πραγματικών υπολογιστών μηχανών
- Απαιτούν λιγότερη προσπάθεια για την υλοποίησή τους και για τη μεταγλώττιση των προγραμμάτων
- Ο πειραματισμός και η μετατροπή τους είναι ευκολότερη
  - Σημαντικές ιδιότητες, ειδικά για πρωτοεμφανιζόμενες γλώσσες

Εικονικές Μηχανές: Πλεονεκτήματα

- Προσφέρουν καλύτερη φορητότητα (portability)
  - Οι διερμηνείς VM συνήθως υλοποιούνται σε C
  - Ο VM κώδικας μπορεί να μεταφερθεί μέσω διαδικτύου και να τρέξει στις περισσότερες υπολογιστικές πλατφόρμες
  - Συνήθως, ο VM κώδικας είναι (σημαντικά) μικρότερος σε μέγεθος από τον αντίστοιχο κώδικα μηχανής
- Αρκετές ιδιότητες ασφάλειας του VM κώδικα μπορούν να επαληθευθούν (verified)
- Είναι ευκολότερο να αποδείξουμε τυπικά την ορθότητα του σχεδιασμού και της υλοποίησής τους
- Το profiling και η αποσφαλμάτωση (debugging) προγραμμάτων είναι ευκολότερα

Εικονικές Μηχανές: Μειοκτήματα

- Χειρότερη επίδοση (σε χρόνο) των διερμηνέων εικονικών μηχανών σε σχέση με την εκτέλεση εντολών σε γλώσσα μηχανής
  - Επιπλέον κόστους διερμηνίας (overhead of interpretation)
  - Είναι σημαντικά πιο δύσκολο να επωφεληθούμε από κάποια χαρακτηριστικά του μοντέρνου hardware των υπολογιστών (π.χ. hardware-based branch prediction)

Ιστορία των Εικονικών Μηχανών

- VMs have been built and studied since the late 1950's
- The first Lisp implementations (1958) used VMs with garbage collection, sandboxing, reflection, and an interactive shell
- Forth (early 70's) used a very small and easy to implement VM with high level of reflection
- Smalltalk (late 70's) allowed changing code on the fly (first truly interactive OO system)
- USCD Pascal (late 70's) popularized the idea of using pseudocode to improve portability
- Self (late 80's), a language with a Smalltalk flavor, had an implementation that pushed the limits of VM performance
- Java (early 90s) made VMs popular and well known

Κάποιες επιλογές σχεδιασμού

- Some design choices are similar to the choices when designing the intermediate code format of a compiler:
  - Should the machine be used on several different physical architectures and operating systems? (JVM)
  - Should the machine be used for several different source languages? (CLI/CLR (.NET))
- Some other design choices are similar to those of the compiler backend:
  - Is performance more important than portability?
  - Is reliability more important than performance?
  - Is (smaller) code size more important than performance?
- And some design choices are similar to those in an OS:
  - How to implement memory management, concurrency, exceptions, I/O, …
  - Is low memory consumption, scalability, or security more important than performance?

Συστατικά των Εικονικών Μηχανών

- The components of a VM vary depending on several factors:
  - Is the language (environment) interactive?
  - Does the language support reflection and/or dynamic loading?
  - Is performance paramount?
  - Is concurrency support required?
  - Is sandboxing required?

Υλοποίηση των Εικονικών Μηχανών

- Οι εικονικές μηχανές συνήθως γράφονται σε “φορητές” γλώσσες προγραμματισμού όπως η C ή η C++
- Για μέρη που είναι σημαντικά για την επίδοση της εικονικής μηχανής, συνήθως χρησιμοποιούμε assembly
- Οι VMs για κάποιες γλώσσες (π.χ. Lisp, Forth, Smalltalk) γράφονται με χρήση της ίδιας τη γλώσσας
- Πολλοί διερμηνείς για VMs γράφονται σε GNU C, για λόγους που θα γίνουν προφανείς στις επόμενες διαφάνειες

Μορφές Διερμηνέων

- Αρκετές υλοποιήσεις γλωσσών προγραμματισμού χρησιμοποιούν διερμηνείς δύο ειδών:
  - Command-line interpreter
    - Διαβάζει και αναλύει συντακτικά κομμάτια πηγαίου κώδικα της γλώσσας τα οποία και εκτελεί
    - Χρησιμοποιείται σε συστήματα που αλληλεπιδρούν με το χρήστη
  - Virtual machine instruction interpreter
    - Διαβάζει και εκτελεί εντολές μιας ενδιάμεσης μορφής εκτελέσιμου κώδικα όπως bytecode εντολών μιας εικονικής μηχανής

Υλοποίηση των Διερμηνέων

Υπάρχουν πολλοί τρόποι υλοποίησης διερμηνέων:

1. Direct string interpretation

   Source level interpreters are very slow because they spend much of their time in doing lexical analysis

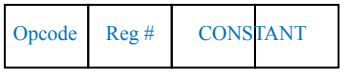
2. Compilation into a (typically abstract syntax) tree and interpretation of that tree

   Such interpreters avoid lexical analysis costs, but they still have to do much list scanning (e.g. when implementing a ‘goto’ or ‘call’)

3. Compilation into a virtual machine and interpretation of the VM code

Διερμηνείς Εντολών Εικονικών Μηχανών

- By compiling the program to the instruction set of a virtual machine and adding a table that maps names and labels to addresses in this program, some of the interpretation overhead can be reduced
- For convenience, most VM instruction sets use integral numbers of bytes to represent everything
  - opcodes, register numbers, stack slot numbers, indices into the function or constant table, etc.



Example: The `GET_CONST2` instruction

Components of Virtual Machine Implementations

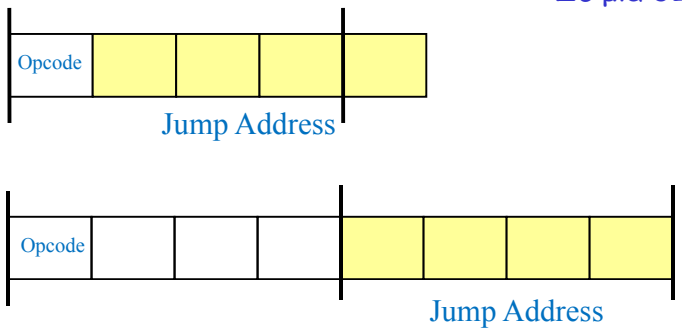
- Program store (code area)
  - Program is a sequence of instructions
  - Loader
- State (of execution)
  - Stack
  - Heap
  - Registers
    - Special register (program counter) pointing to the next instruction to be executed
- Runtime system component
  - Memory allocator
  - Garbage collector

Η βασική δομή ενός Bytecode Interpreter

```
byte *pc = &byte_program[0];
while(TRUE) {
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
        source_reg_num = pc[1];
        const_num_to_match = get_2_bytes(&pc[2]);
        … // get_const2 code
        pc += 4;
        break;
     …
     case JUMP:
        jump_addr = get_4_bytes(&pc[1]);
        pc = &byte_program[jump_addr];
        break;
     …
   }
}
```


To align or to not align VM instructions?

Σε μια 32-bit μηχανή



NOTE: Padding of instructions can be done by the loader.
The size of the bytecode files need not be affected.

Bytecode Interpreter with Aligned Instructions

```
byte *pc = &byte_program[0];
while(TRUE) {
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
        source_reg_num = pc[1];
        const_num_to_match = get_2_bytes(&pc[2]);
        … // get_const2 code
        pc += 4;
        break;
     …
     case JUMP: // aligned version
        jump_addr = get_4_bytes(&pc[4]);
        pc = &byte_program[jump_addr];
        break;
     …
   }
}
```


Διερμηνέας με αφηρημένη κωδικοποίηση εντολών

```
byte *pc = &byte_program[0];
while(TRUE) {
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
        source_reg_num = pc[GET_CONST2_ARG1];
        const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
        … // get_const2 code
        pc += GET_CONST2_SIZEOF;
        break;
     …
     case JUMP: // aligned version
        jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
        pc = &byte_program[jump_addr];
        break;
     …
   }
}
```

```
#define GET_CONST2_SIZEOF 4
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 1
#define GET_CONST2_ARG2 2
#define JUMP_ARG1 4
```


Διερμηνέας με αφηρημένο έλεγχο

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
        source_reg_num = pc[GET_CONST2_ARG1];
        const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
        … // get_const2 code
        pc += GET_CONST2_SIZEOF;
        NEXT_INSTRUCTION;
     …
     case JUMP: // aligned version
        jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
        pc = &byte_program[jump_addr];
        NEXT_INSTRUCTION;
     …
   }
}
```

```
#define NEXT_INSTRUCTION \
    goto next_instruction
```


Έμμεσα Νηματικοί Διερμηνείς

- In an indirectly threaded interpreter we do not switch on the opcode encoding; instead we use the bytecodes as indices into a table containing the addresses of the VM instructions
- The term *threaded code* refers to a code representation where every instruction is implicitly a function call to the next instruction
- A threaded interpreter can be very efficiently implemented in assembly
- In GNU CC, we can use the labels as values C language extension and take the address of a label with &&labelname
- We can actually write the interpreter in such a way that it uses indirectly threaded code if compiled with GNU CC and a switch for compatibility

Δομή ενός Έμμεσα Νηματικού Διερμηνέα

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
     get_const2_label:
        source_reg_num = pc[GET_CONST2_ARG1];
        const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
        … // get_const2 code
        pc += GET_CONST2_SIZEOF;
        NEXT_INSTRUCTION;
     …
     case JUMP: // aligned version
     jump_label:
        jump_addr = get_4_bytes(&pc[JUMP_ARG1]);
        pc = &byte_program[jump_addr];
        NEXT_INSTRUCTION;
     …
   }
}
```

```
static void *label_tab[] {
   &&get_const2_label;
   &&jump_label;
}
#define NEXT_INSTRUCTION \
   goto **(void **)(label_tab[*pc])
```


Άμεσα Νηματικοί Διερμηνείς

- In a directly threaded interpreter, we do not use the bytecode instruction encoding at all during runtime
- Instead, the loader replaces each bytecode instruction encoding (opcode) with the address of the implementation of the instruction
- This means that we need one word for the opcode, which increases the VM code size

Δομή ενός Άμεσα Νηματικού Διερμηνέα

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
     get_const2_label:
        source_reg_num = pc[GET_CONST2_ARG1];
        const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
        … // get_const2 code
        pc += GET_CONST2_SIZEOF;
        NEXT_INSTRUCTION;
     …
     case JUMP: // aligned version
     jump_label:
        pc = get_4_bytes(&pc[JUMP_ARG1]);
        NEXT_INSTRUCTION;
     …
   }
}
```

```
static void *label_tab[] {
   &&get_const2_label;
   &&jump_label;
}
#define NEXT_INSTRUCTION \
   goto **(void **)(pc)
```

```
#define GET_CONST2_SIZEOF 8
#define JUMP_SIZEOF 8
#define GET_CONST2_ARG1 5
#define GET_CONST2_ARG2 6
#define JUMP_ARG1 4
```


Νηματικός Διερμηνέας με Χρήση Prefetching

```
byte *pc = &byte_program[0];
while(TRUE) {
next_instruction:
   opcode = pc[0];
   switch (opcode) {
     …
     case GET_CONST2:
     get_const2_label:
        source_reg_num = pc[GET_CONST2_ARG1];
        const_num_to_match = get_2_bytes(&pc[GET_CONST2_ARG2]);
        pc += GET_CONST2_SIZEOF; // prefetching
        … // get_const2 code
        NEXT_INSTRUCTION;
     …
     case JUMP: // aligned version
     jump_label:
        pc = get_4_bytes(&pc[JUMP_ARG1]);
        NEXT_INSTRUCTION;
     …
   }
}
```

Stack-based vs. Register-based VMs

- A VM can either be *stack-based* or *register-based*
  - In a stack-based VM most operands are (passed) on the stack
    - The stack can grow as needed
  - In a register-based VM most operands are passed in (virtual) registers
    - These registers are often implemented using an array rather than physical machine registers
    - The number of registers is limited
- Most VMs are stack-based
  - Stack machines are simpler to implement
  - Stack machines are easier to compile to
  - Less encoding/decoding to find the right register
  - Unless virtual registers are mapped to physical registers, virtual registers are not faster than stack slots

Virtual Machine Interpreter Tuning

Common VM interpreter optimizations include:

- Writing the interpreter loop and key instructions in assembly
- Keeping important VM registers (pc, stack top, heap top) in hardware registers
  - GNU C allows global register variables
- Top of stack caching
- Splitting the most used set of instruction into a separate interpreter loop

Instruction Merging and Specialization

Instruction Merging: A sequence of VM instructions is replaced by a single (mega-)instruction

- Reduces interpretation overhead
- Code locality is enhanced
- Results in more compact bytecode
- C compiler has bigger basic blocks to perform optimizations on

Instruction Specialization: A special case of a VM instruction is created, typically one where some arguments have a known value which is hard-coded

- Eliminates the cost of argument decoding
- Results in more compact bytecode representation
- Reduces the register pressure from some basic blocks

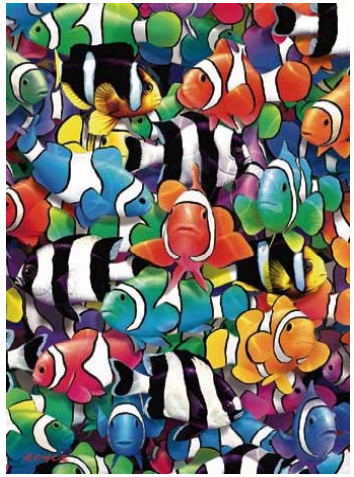
Δυναμική δέσμευση μνήμης

- Κατά την εκτέλεση του προγράμματος υπάρχει ανάγκη για δέσμευση μνήμης:
  - Εγγραφών δραστηριοποίησης
  - Αντικειμένων
  - Άμεσων κλήσεων δέσμευσης μνήμης: **`new`**, **`malloc`**, κ.λπ.
  - Εμμέσων κλήσεων δέσμευσης μνήμης: δημιουργία συμβολοσειρών, buffers για αρχεία, πινάκων με δυναμικά καθοριζόμενο μέγεθος, κ.λπ.
- Οι υλοποιήσεις των γλωσσών παρέχουν τη δυνατότητα διαχείρισης μνήμης κατά το χρόνο εκτέλεσης του προγράμματος

Στοίβες από εγγραφές δραστηριοποίησης

- Στις περισσότερες γλώσσες, οι εγγραφές δραστηριοποίησης δεσμεύονται δυναμικά
- Σε πολλές γλώσσες, επαρκεί να δεσμεύσουμε μια εγγραφή κατά την κλήση μιας συνάρτησης η οποία αποδεσμεύεται κατά την επιστροφή της συνάρτησης
- Με αυτόν τον τρόπο παράγεται μια στοίβα από εγγραφές δραστηριοποίησης
- Μια στοίβα χρειάζεται ένα σχετικά απλό αλγόριθμο διαχείρισης μνήμης

Σωροί (heaps)



Διάταξη σε σωρό

- Η διάταξη σε στοίβα έχει σχετικά εύκολη υλοποίηση
- Αλλά δεν επαρκεί πάντα: τι για παράδειγμα συμβαίνει εάν οι δεσμεύσεις και οι αποδεσμεύσεις κομματιών μνήμης ακολουθούν τυχαία σειρά;
- Ένας **σωρός (heap)** είναι μια ακολουθία από μπλοκ μνήμης τα οποία μας δίνουν τη δυνατότητα να έχουμε μη ταξινομημένη δέσμευση και αποδέσμευση μνήμης
- Υπάρχουν πολλοί μηχανισμοί υλοποίησης μνήμης σωρού

Υλοποίηση μέσω του αλγόριθμου First Fit

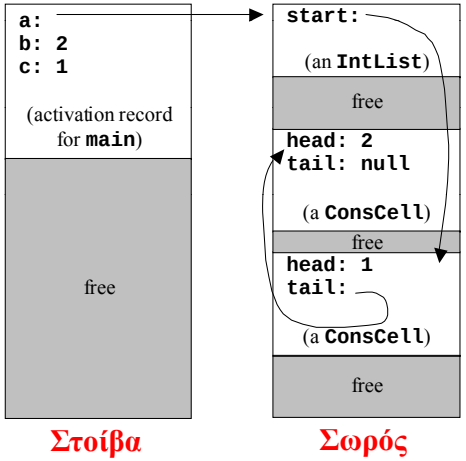
- Ο σωρός υλοποιείται ως μια συνδεδεμένη λίστα από ελεύθερα μπλοκ (**free list**)
- Αρχικά αποτελείται από μόνο ένα μεγάλο μπλοκ
- Για δέσμευση κάποιου κομματιού μνήμης:
  - Ψάχνουμε στη free list για το πρώτο μπλοκ με μέγεθος τέτοιο ώστε να ικανοποιούνται οι απαιτήσεις της ζήτησης
  - Εάν το μπλοκ είναι μεγαλύτερο από τη ζήτηση, τότε επιστρέφουμε το αχρησιμοποίητο κομμάτι στη free list
  - Ικανοποιούμε την απαίτηση δέσμευσης μνήμης και επιστρέφουμε την αρχική διεύθυνση του δεσμευμένου μπλοκ
- Για αποδέσμευση, απλώς προσθέτουμε το ελευθερωθέν μπλοκ στην αρχή της free list

Τρέχοντες σύνδεσμοι στο σωρό

- Το πρόγραμμα, κατά την εκτέλεσή του, απαιτεί δεσμεύσεις μνήμης
- Συνήθως, το πρόγραμμα αποθηκεύει τις διευθύνσεις μνήμης που του παραχωρούνται σε κάποια ονόματα (μεταβλητές)

- Ένας **τρέχων σύνδεσμος στο σωρό (current heap link)** είναι μια θέση μνήμης που περιέχει μια τιμή την οποία θα χρησιμοποιήσει το πρόγραμμα ως δείκτη στο σωρό

Τρέχοντες σύνδεσμοι στο σωρό




```
IntList a =
  new IntList(null);
int b = 2;
int c = 1;
a = a.cons(b);
a = a.cons(c);
```


Εύρεση των τρεχόντων συνδέσμων στο σωρό

- Αρχίζουμε με ένα **σύνολο ριζών (root set)**: θέσεις μνήμης εκτός του σωρού που περιέχουν συνδέσμους σε θέσεις μνήμης στο σωρό, π.χ. σε
  - ενεργές εγγραφές δραστηριοποίησης (στη στοίβα)
  - καθολικές ή στατικές μεταβλητές
- Για κάθε στοιχείο του συνόλου, κοιτάμε το δεσμευμένο μπλοκ στο οποίο δείχνει ο σύνδεσμος και προσθέτουμε όλες τις θέσεις μνήμης του συγκεκριμένου μπλοκ στο σύνολό μας
- Επαναλαμβάνουμε την παραπάνω διαδικασία μέχρις του σημείου όπου δε βρίσκουμε άλλες νέες θέσεις μνήμης

Λάθη κατά την εύρεση των τρεχόντων συνδέσμων

- **Παραλείψεις**: ξεχνάμε να περιλάβουμε μια θέση μνήμης που έχει έναν τρέχοντα σύνδεσμο στο σωρό
- **Άχρηστες τιμές**: συμπεριλαμβάνουμε θέσεις μνήμης για τις οποίες το πρόγραμμα ποτέ δε θα χρησιμοποιήσει τις τιμές που αποθηκεύουν
- **Διευθύνσεις που δεν είναι θέσεις μνήμης**: συμπεριλαμβάνουμε κάποιες θέσεις μνήμης στο σωρό μόνο και μόνο επειδή δε μπορούμε να ξεχωρίσουμε τιμές που χρησιμοποιούνται π.χ. ως δείκτες στο σωρό ή ως ακέραιοι (με τιμή μια διεύθυνση μνήμης στο σωρό)

Δε μπορούμε πάντα να αποφύγουμε τα λάθη

- Για τη σωστή διαχείριση του σωρού, τα λάθη παράλειψης είναι μη αποδεκτά και δε συγχωρούνται
- Άρα είμαστε υποχρεωμένοι να συμπεριλάβουμε όλες τις θέσεις μνήμης για τις οποίες υπάρχει πιθανότητα η τιμή που αποθηκεύεται στη συγκεκριμένη θέση μνήμης να χρησιμοποιηθεί από το πρόγραμμα
- Κατά συνέπεια, το να συμπεριλάβουμε άχρηστες θέσεις μνήμης στον υπολογισμό είναι αναπόφευκτο
- Ανάλογα με τη γλώσσα, μπορεί να είναι αδύνατο να αποφύγουμε να συμπεριλάβουμε και τιμές που απλώς δείχνουν σα θέσεις μνήμης στο σωρό

Τιμές που δείχνουν σα διευθύνσεις μνήμης

- Για κάποιες μεταβλητές μπορεί να μην είμαστε σε θέση να καταλάβουμε πως χρησιμοποιούνται οι τιμές τους
- Για παράδειγμα, η παρακάτω μεταβλητή **x** μπορεί να χρησιμοποιηθεί είτε ως δείκτης είτε ως πίνακας από τέσσερις χαρακτήρες

```
union {
  char *p;
  char tag[4];
} x;
```

Σημείωση: το παραπάνω πρόβλημα είναι ακόμα χειρότερο στη C, διότι οι C compilers δεν κρατούν πληροφορία για τους τύπους των μεταβλητών κατά τη διάρκεια εκτέλεσης του προγράμματος

Συμπίεση του σωρού (heap compaction)

- Μια λειτουργία που χρειάζεται πληροφορία για το σύνολο των τρεχόντων συνδέσμων στο σωρό
- Ο διαχειριστής της μνήμης μετακινεί δεσμευμένα μπλοκ:
  - Αντιγράφει το μπλοκ σε μια νέα θέση, και
  - Επικαιροποιεί όλους τους συνδέσμους στο (ή κάπου μέσα στο) μπλοκ
- Κατά συνέπεια συμπιέζει το σωρό, μετακινώντας όλα τα δεσμευμένα μπλοκ στο ένα άκρο του και αφήνοντας ένα μεγάλο ελεύθερο μπλοκ χωρίς κατακερματισμό

Συλλογή Σκουπιδιών

Κάποια συνηθισμένα προβλήματα με δείκτες

```
type
  p: ^Integer;
begin
  new(p);
  p^ := 21;
  dispose(p);
  p^ := p^ + 1
end
```

Ξεκρέμαστος δείκτης: το διπλανό πρόγραμμα Pascal χρησιμοποιεί ένα δείκτη σε μπλοκ μνήμης μετά την αποδέσμευση του συγκεκριμένου μπλοκ

```
procedure Leak;
  type
    p: ^Integer;
  begin
    new(p)
  end;
```

Διαρροή μνήμης: το διπλανό πρόγραμμα Pascal δεσμεύει ένα μπλοκ μνήμης αλλά ξεχνάει να το αποδεσμεύσει

Συλλογή σκουπιδιών (garbage collection)

- Αφού τόσα σφάλματα λογισμικού συμβαίνουν λόγω λαθών αποδέσμευσης μνήμης…
- …και επειδή δεν είναι βολικό για τον προγραμματιστή να σκέφτεται για το πώς θα γίνει σωστά η αποδέσμευση μνήμης…
- …για ποιο λόγο να μην είναι ευθύνη της υλοποίησης της γλώσσας η αυτόματη ανακύκλωση μνήμης;

Τρεις βασικοί μηχανισμοί ανακύκλωσης μνήμης

1. Μαρκάρισμα και σκούπισμα (mark and sweep)
2. Αντιγραφή (copying)
3. Μέτρημα αναφορών (reference counting)

Memory Management Terminology

- Dynamically allocated objects in the heap forms a directed graph:
  - nodes are the heap objects and
  - edges are the pointers between objects
- Program variables, registers, and stack variables are the roots of this graph
- Data which is reachable by a chain of pointers from program variables is called live data
- All other data is dead (garbage) and
  - can be reclaimed by the garbage collector
  - in order to be used by the memory allocator

Μαρκάρισμα και σκούπισμα

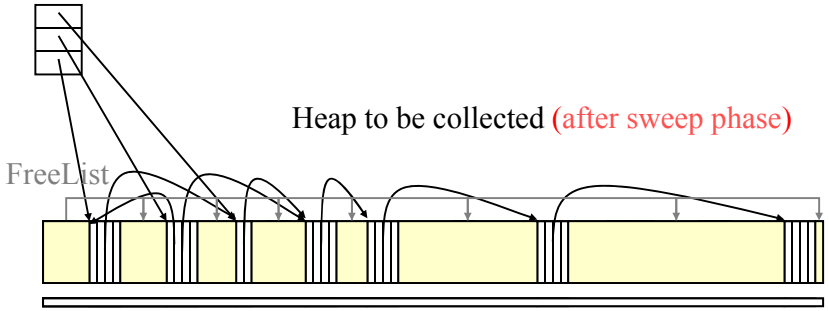
- Ένας συλλέκτης μαρκαρίσματος και σκουπίσματος (mark-and-sweep collector) χρησιμοποιεί τους τρέχοντες συνδέσμους στο σωρό σε μια διαδικασία που έχει δύο φάσεις:
  - **Μαρκάρισμα**: βρίσκουμε όλους τους τρέχοντες συνδέσμους στο σωρό και μαρκάρουμε όλα τα μπλοκ του σωρού τα οποία δείχνονται από κάποιον από τους συνδέσμους
  - **Σκούπισμα**: κάνουμε ένα πέρασμα στο σωρό και επιστρέφουμε τα αμαρκάριστα μπλοκ στη λίστα με τα ελεύθερα μπλοκ
- Τα μπλοκ που μαρκάρονται στην πρώτη φάση του αλγόριθμου δε μετακινούνται (non-moving collector)

Ο Αλγόριθμος Mark-Sweep

```
proc mark_sweep_gc() ≡
   foreach root ∈ rootset do mark(*root)
   sweep()

proc mark(object) ≡
   if marked(object) = false →
    marked(object) := true
    foreach pointer in object do
       mark(*pointer)
```


Mark-Sweep Collection [McCarthy 1960]



Mark-Sweep Collection: Ιδιότητες

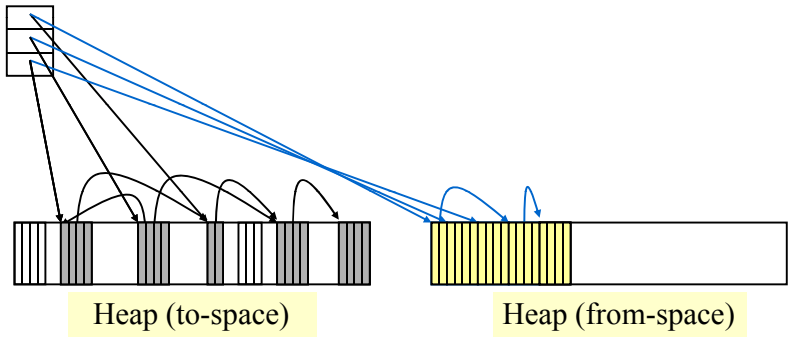
+ Δε μετακινεί αντικείμενα
+ Requires relatively little additional memory for the collector

- Allocation is slower
  - (and may need to deal with fragmentation)
- The collection time is proportional to the size of the heap that is collected, not the size of the live data

Συλλογή σκουπιδιών μέσω αντιγραφής

- Ένας συλλέκτης αντιγραφής (copying collector) διαιρεί τη μνήμη στη μέση και χρησιμοποιεί τη μισή μόνο μνήμη για να ικανοποιήσει τις αιτήσεις δέσμευσης
- Όταν το μισό αυτό μέρος γεμίσει, βρίσκουμε συνδέσμους που δείχνουν σε μπλοκ στο πρώτο μισό και αντιγράφουμε αυτά τα μπλοκ στο δεύτερο μισό
- Η διαδικασία αυτή συμπιέζει τα χρησιμοποιούμενα μπλοκ, με αποτέλεσμα να εξαφανίζει τον κατακερματισμό
- Μετακινεί μπλοκ (moving collector)

Συλλογή μέσω Αντιγραφής [Cheney 1970]



Copying Collection: Ιδιότητες

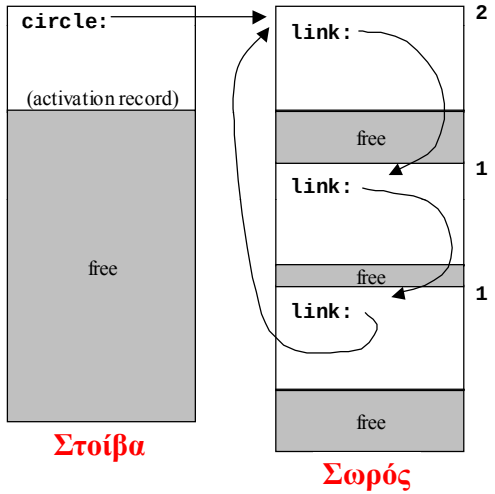
+ After a copying collection, allocation is very fast
+ Time complexity is proportional to the size of the live data set, not the size of the heap that is collected!

- Half of the available memory is reserved for servicing the collector, not for the needs of the application
- Objects are moved

Μέτρημα αναφορών

- Κάθε μπλοκ έχει ένα μετρητή που κρατάει τον αριθμό των συνδέσμων σωρού που δείχνουν στο μπλοκ
- Ο μετρητής αυτός αυξάνεται όταν κάποιος σύνδεσμος προς το μπλοκ αντιγράφεται (μέσω ανάθεσης), και μειώνεται όταν κάποιος σύνδεσμος is discarded
- Όταν η τιμή του μετρητή γίνει μηδέν, το μπλοκ είναι άχρηστο και μπορεί να ελευθερωθεί
- Δηλαδή στη διαχείριση μνήμης με μέτρημα αναφορών δε χρειάζεται να βρούμε δυναμικά τους τρέχοντες συνδέσμους στο σωρό

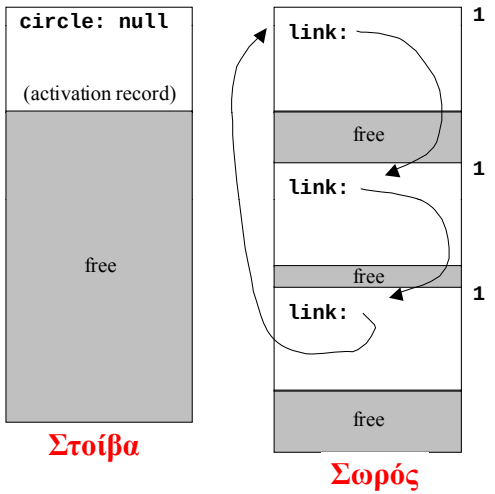
Πρόβλημα μετρήματος αναφορών



Ένα πρόβλημα μετρήματος αναφορών: δε μπορούμε να ανακαλύψουμε κύκλους από άχρηστα μπλοκ.

Στην εικόνα, μια κυκλικά συνδεδεμένη λίστα που δείχνεται από τη μεταβλητή **`circle`**.

Πρόβλημα μετρήματος αναφορών



Αν αναθέσουμε π.χ. την τιμή null στη μεταβλητή **`circle`**, η τιμή του μετρητή θα μειωθεί.

Κανένας μετρητής αναφορών δε γίνεται μηδέν, παρόλο που όλα τα μπλοκ είναι άχρηστα (σκουπίδια).

Μέτρημα αναφορών

- Δε μπορεί να συλλέξει κύκλους σκουπιδιών
- Επιβάλλει κάποιο μη αμελητέο κόστος στην εκτέλεση του προγράμματος, λόγω της ανάγκης να ενημερώσουμε τους μετρητές αναφορών σε κάθε ανάθεση
- Ένα πλεονέκτημα: ο αλγόριθμος εκτελείται από τη φύση του σε μικρά βήματα (incremental), και δεν επιβάλλει μεγάλες παύσεις κατά την εκτέλεση του προγράμματος

Εκλεπτυσμένοι συλλέκτες σκουπιδιών

- **Συλλέκτες γενεών (Generational collectors)**
  - Η προς συλλογή μνήμη χωρίζεται σε *γενιές (generations)* με βάση την ηλικία των αντικειμένων στο μπλοκ
  - Συλλέγουμε τα σκουπίδια στις νέες γενιές πιο συχνά (χρησιμοποιώντας κάποια από τις προηγούμενες μεθόδους)
- **Incremental collectors**
  - Συλλέγουν σκουπίδια σε μικρά χρονικά διαστήματα τα οποία παρεμβάλλονται με την εκτέλεση του προγράμματος
  - Κατά συνέπεια αποφεύγουν το σταμάτημα της εφαρμογής για μεγάλο χρονικό διάστημα και οι παύσεις λόγω αυτόματης ανακύκλωσης μνήμης έχουν πολύ μικρή διάρκεια

Γλώσσες με αυτόματη διαχείριση μνήμης

- Σε κάποιες γλώσσες είναι απαίτηση: Lisp, Scheme, ML, Haskell, Erlang, Clean, Prolog, Java, C#
- Κάποιες άλλες όπως η Ada απλώς την ενθαρρύνουν
- Τέλος, κάποιες άλλες γλώσσες όπως η C και η C++ την καθιστούν πολύ δύσκολη
  - Όμως ακόμα και για αυτές είναι δυνατή
  - Υπάρχουν βιβλιοθήκες που αντικαθιστούν τη συνηθισμένη υλοποίηση των **`malloc`**/**`free`** με ένα διαχειριστή μνήμης που χρησιμοποιεί **συντηρητική συλλογή σκουπιδιών (conservative garbage collection)** για την αυτόματη ανακύκλωση μνήμης

Τάσεις

- Μια ιδέα από τις αρχές του 1960, η δημοτικότητα της οποίας έχει αυξηθεί σημαντικά τα τελευταία χρόνια
- Οι καλές υλοποιήσεις των συλλεκτών σκουπιδιών έχουν επίδοση παρόμοια ή μόνο κατά λίγο χειρότερη από αυτή την οποία είναι δυνατή με διαχείριση μνήμης υπό τον έλεγχο του προγραμματιστή
- Όλο και περισσότεροι προγραμματιστές συνειδητοποιούν ότι η αυτόματη ανακύκλωση μνήμης αυξάνει την παραγωγικότητά τους, οδηγεί σε λιγότερα σφάλματα, και κατά συνέπεια αξίζει τον έξτρα χρόνο εκτέλεσης

Συμπερασματικά

- Η διαχείριση μνήμης είναι ένα σημαντικό συστατικό του σχεδιασμού και της υλοποίησης των γλωσσών προγραμματισμού
- Τόσο η επίδοση όσο και η αξιοπιστία των προγραμμάτων είναι σημαντικοί παράγοντες της ανάπτυξης λογισμικού
- Η αυτόματη διαχείριση μνήμης είναι μια από τις πιο ενεργές περιοχές ερευνητικής δραστηριότητας και πειραματισμού στην περιοχή της υλοποίησης γλωσσών προγραμματισμού