Imperial College London
Department of Computing

Compilers (221)

Exercises – Runtime Organisation

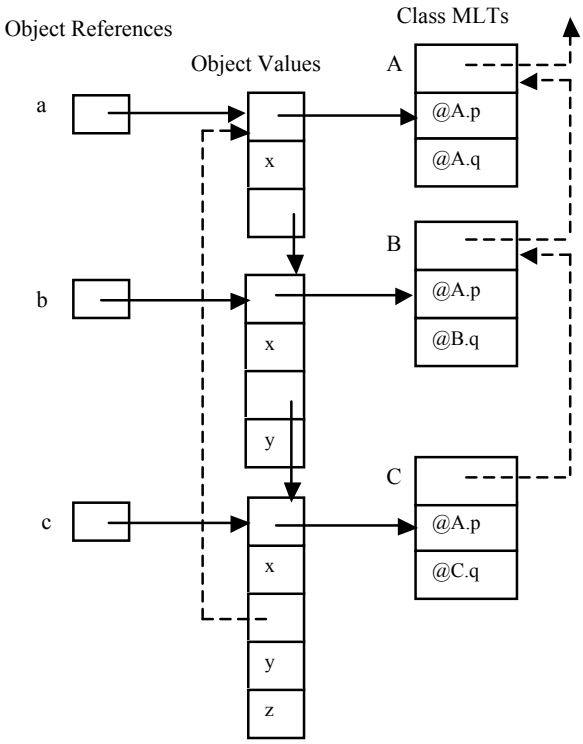
Check your answers with the tutorial helpers during tutorials and with each other on Piazza.

1. Given classes A, B and C below draw the memory layout after execution of the following sequence of assignments: L3

```
class A {
  int x
  A next
  method p(A t) { print 1; t.q(this); }
  method q(A t) { print 2; t.q(this); }
}

class B extends A {
  int y
  method q(A t) { print 3, t.q(this); }
}

class C extends B {
  int z
  method q(A t) { print 4; t.q(this); }
}

A a = new A()
B b = new B()
C c = new C()
a.next = b
b.next = c
c.next = a
```

After execution of the sequence of assignments what would `c.next.next.q(c)` print?

++



The call `c.next.next.q(c)` will call B.q which prints 3 and calls C.q which prints 4 and calls B.q which prints 3 and calls C.q which prints 4 and calls B.q which prints etc... forever

2. Consider a compiler and runtime system that implements immutable strings as memory blocks managed by a garbage collector. Discuss the time and memory issues that the following program might give rise to: L3

```
string line = randomChar;              // initialise line to a random character
for(int k=1; k<10000; k++) {
      line = line ++ randomChar();     // append random character to line
      process(line);                   // do something with line.
}
```

++

Appending a single character to a string's memory block will cause: (1) a new memory block to be created that can hold an extra character, (2) copying the contents of the old block and the new character to it (3) freeing the old block, e.g. marking it as free. This will be very inefficient for large strings, since the time taken to do the copy will be greater each time around the loop - ever larger memory blocks will be allocated and freed leading to slower garbage-collection. In the code fragment function `process` might keep references to string *`line`*. Heap memory may also get fragmented. For mutable strings one solution would be to allocate a bigger memory block for strings to allow for future expansion of strings. A linked list implementation could also be contemplated in some cases c.f. cost to index and slice string. char might be implemented as a string.

3. In some programming languages variables are not initialised by default and any attempt to read an un-initialised variable is considered to be an error. For example: L3

```
int x, y[100];     // not initialised
print x;           // error
print y[expr];     // error
```

Describe how such errors can be detected. Consider un-initialised object attributes, array elements, statically-allocated variables, stack-allocated variables and heap-allocated variables. Comment on the costs of your detection mechanism(s).

++

In simple cases compile-time checks are sufficient, e.g. using symbol table tracking and dataflow analysis. More generally, initialisation bits need to be allocated, initialised, updated and checked at runtime for each variable, object attribute, array element etc.

For statically-allocated and stack-allocated variables, the compiler can allocate a per-scope bit array in which the *k*th bit indicates the initialisation status of the *k*th variable. Zero should probably indicate "un-initialised", because an all-zero array for global/static data can be placed in zero-filled memory which need not occupy space in the object file.

For dynamically allocated variables, an initialisation bit can be included in the header of the variable's heap block and set to "un-initialised" when the heap block is created.

The compiler would need to generate code to check the initialisation bit for any variable where the initialisation status of the variable couldn't be determined statically. Such checks could be costly e.g. bit-array indexing, bitwise testing, lots of memory, initialisation and testing for array elements, fields of structures, handling reference parameters.

4. In programming languages that support nested procedures, procedure parameters can be represented by a pair of values: the address of the procedure and a pointer to the appropriate stack frame. Consider the following program which has a nested procedure beta and a procedure parameter `proc`: L4

```
def alpha(num, proc):

    def beta(): print(num)

    if num==1:
        alpha(2,beta)
    else:
        proc()

def gamma(): skip

alpha(1, gamma)
```

For this program, draw and explain the state of the stack when procedure `beta` is called. Explain why the output of the program is 1. Assume parameters are passed via the stack.

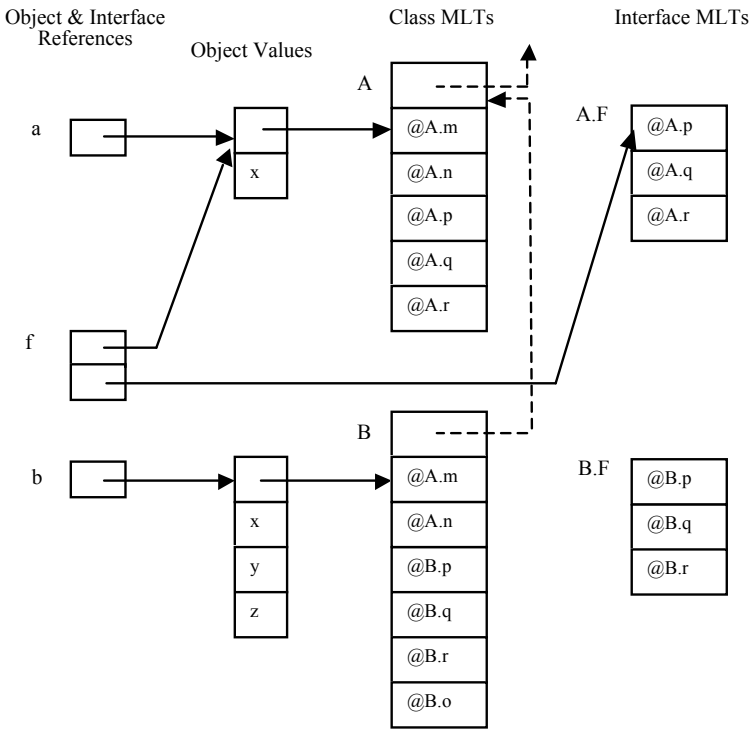
++

```
mainFP:    mainCallerFP (oldFP)              main stackframe
           returnaddr
-------------------------------------------------------------------------
           1  (num)                          alpha stackframe outercall
           gamma (proc.addr) + mainFP (proc.FP)
alphaFP:   mainFP   (oldFP)
           returnaddr
           beta object referencing closure allowed here - for alt solution
-------------------------------------------------------------------------
           2 (num)                            alpha stackframe innercall
           beta (proc.addr) + alphaFP  (proc.FP)
alpha'FP:  alphaFP (oldFP)
           returnaddr
           beta object allowed here - for alt solution
-------------------------------------------------------------------------
procFP:    proc.FP=alphaFP                   proc stackframe
           returnaddr
```

When **`beta`** is called via a parameter **`proc`** in **`alpha`**, two instances of **`alpha`** exist. Because **`proc`** was created in the original call to **`alpha`**, **`beta`**'s link is to the earlier invocation where **`num`** is 1 i.e. print num will use offset to num frm proc.FP=alphaFP to get 1. Also accept

	solutions that build a closure object for **beta** copying in value of num at time of call.	

5.	Some OO languages have the concept of interface types and interface variables. Devise a scheme for implementing such variables and show the memory layout for the following:
	
	`interface F {`
	`  method p()`
	`  method q()`
	`  method r()`
	`}`
	
	`class A implements F {`
	`  int x`
	`  method m() { ... }`
	`  method n() { ... }`
	`  method p() { ... }`
	`  method q() { ... }`
	`  method r() { ... }`
	`}`
	
	`class B extends A implements F {`
	`  int y`
	`  int z`
	`  method o() { ... }`
	`  method p() { ... }`
	`  method q() { ... }`
	`  method r() { ... }`
	`}`
	`A a = new A()`
	`B b = new B()`
	`F f = a`
	++
	
	Here's a simple scheme (others are possible too). In addition to a Method Lookup Table for each class we also have a Method Lookup Table for each interface that a class implements, i.e. if a class C implements interfaces X, Y and Z, then the compiler will produce (1) a MLT for C which has entries for all methods of C including those that implement methods defined in X, Y and Z, and (2) MLTs for C.X, C.Y, C.Z which have entries only for the methods of the respective interface type.
	
	With this scheme an interface variable can be represented by two references, one to the object that is assigned to it, and another to the appropriate Interface Method Lookup Table for that object's class. Conversion (casting) from an interface variable to an object reference requires a run-time check that the class of the object held in the interface variable corresponds to the required class or a subclass.
		L5



6. Continuing the previous question now translate the following statements into Intel IA-32 assembly language. Assume that the addresses of a, b and f are already in registers EAX, EBX, ECX: L3

```
f.r()
f = b
a = (A)f
```

++

```
; ========================================================================
; f.r ( )
    push    [ECX]           ; Push 1st field of f (the object reference) onto the stack as 1st
                            ; parameter
    lea     EDX, [ECX+4]    ; Load address of 2nd field of f (address of MLT) into EDX
    call    [EDX+8]         ; Call 3rd method of  f  (A.r)
    add     ESP, 4          ; Pop parameter off stack
; ========================================================================
; f = b                     ; Note: a static check suffices to ensure that b implements interface F
    mov     [ECX], [EBX]    ; Copy b's object reference into the 1st field of f
    mov     [ECX+4], @B.F   ; Copy the address of B's F Interface MLT into the 2nd field of f
; ========================================================================
; a = (A) f                 ; We need to check that f's object is an A class
    mov     EDX, [ECX]      ;  EDX will traverse the Class hierarchy, i.e. the MLT hierarchy
CheckClass:
    mov     EDX, [EDX]      ; Get address of MLT
    cmp     EDX, @A         ; Is address of MLT = class A's MLT?
    je      Okay
    cmp     EDX, NULL
    je      ClassError      ; Null. f does not refer to an A object, raise exception
    jne     CheckClass      ; If not null, check superclass
Okay:
    mov     [EAX], [ECX]    ; f refers to an A object. Copy 1st field of f into 'a'
```

7. Some languages allow the programmer to omit variable declarations entirely. Other languages require the programmer to declare all variables, but not to declare their types. Still other languages require the programmer to declare both variables and their types. Give a short L3

argument in favour of each approach. Which argument do you find most convincing? Why?

++

The 1st approach (e.g Python) leads to shorter programs that are arguably easier to write and read. It provides polymorphism "for free," and relieves the programmer of the need to prove to the compiler that the program is type safe.

The 2nd approach (e.g. Scheme) is naturally polymorphic, but avoids errors due to misspellings and to accidental use of the same variable name for multiple purposes.

The 3rd approach (e.g. Java) is self-documenting, and arguably forces the programmer to think through things more carefully, reducing the incidence of bugs. Bugs that do occur are likely to be found at compile time. Given detailed knowledge of types, the compiler can produce more efficient code than it can for Python or Scheme.

Which approach is better is largely a matter of personal preference. Experience suggests, however, that very large systems are easier to maintain (and run much faster) when written in statically typed languages.

8. Consider a programming language that creates stack frames (activation records) on the heap rather than on the system stack.

   i) Elaborate on how this approach might be implemented.

   ii) Give 2 advantages and 2 disadvantages of this approach.

++

For each method call, allocate a block of memory on the heap, sufficient to hold the parameters, return address, local variables, saved registers and temporaries. Use a register, e.g. EBP to point to this block and access the stack frame data via it. On method return, free the heapframe and return it to the free list or underlying garbage collector.

Advantages: Potentially good for languages with closures/continuations, threads/coroutines, generators etc since we have a frame tree instead of frame stack. Potentially we never need to return from a method if there is no reference to the method, we just let the garbage collector do its work. Useful protection against stack buffer overflow attacks.

Disadvantages: Stack organisation is very efficient and architectures have been optimised for stacks (e.g. in imperative/oo languages) over the years. Heap-based allocation can be slow, unless heapframe creation, access and removal are carefully optimised. It may not be possible to implement low-level languages with explicit pointer arithmetic and aliasing using this approach.

9. **OPTIONAL** L5

Outline an implementation of (i) Two-Space Copying, (ii) Point-reversal marking, i.e. implement Mark (Block). For two-space copying you'll need some housekeeping data, e.g. a forwarding pointer which records the address where a node has already been copied to.
For pointer-reversal marking, as a simplification, assume that each block has just two child pointers, left and right.

++

(i) There are a number of ways of doing this. The following algorithm is due to Cheney:

```
FromSpaceStart, ToSpaceStart = HeapStart, HeapStart + (HeapSize/2)-1
FromSpaceEnd = FromSpaceStart + HeapSize/2
FreePtr = FromSpaceStart

def New(Size):
   if FreePtr + Size > FromSpaceEnd:   # copy and swap roles
         FreePtr = ToSpaceStart
         for RootReference in NonHeapArea:   # do root blocks
            RootReference = CopyBlock(RootReference)
         ScanBlock = ToSpaceStart
```

```
         while ScanBlock < FreePtr:    # now do children
            for ChildBlock in Children(ScanBlock)
               ChildBlock = Copy(ChildBlock)
            ScanBlock = ScanBlock + ScanBlock.Size
         FromSpaceStart, ToSpaceStart = ToSpaceStart, FromSpaceStart
         FromSpaceEnd = FromSpaceStart + HeapSize/2
   if FreePtr + Size > FromSpaceEnd:
         error (“Out of memory”)
   BlockPtr = FreePtr
   FreePtr = FreePtr + Size
   return BlockPtr

def CopyBlock(Block):
   if Block.ForwardingAddr in ToSpaceHalf:  # i.e. already copied/forwarded
         return Block.ForwardingAddr
   else:
         NewBlock, FreePtr = FreePtr, FreePtr + Block.size   # allocate space
         MemCopy Block.Size bytes from Block to NewBlock
         Block.ForwardingAddr = NewBlock
         return NewBlock
```

(ii) Here’s an outline solution for blocks with two child pointers, left & right. To handle n-children each Block needs a count/index of the number of currently marked children which it increments when it advances to a Block. We also need type information that gives us the offset of each child pointer in a Block.

```
def Mark(Root):
   current = Root
   previous = None
   while True: # advance to next child
         while current != None and current.markbit == 0:
               current.markbit = 1
               if ! current.isLeafBlock:
                     # circular rotation of pointers
                     next = current.left
                     current.left = previous
                     previous = current
               current = next
         # retreat to parent - similar to advance
         while previous != None and previous.flagbit = 1:
               previous.flagbit = 0
               next = previous.right
               previous.right = current
               current = previous
               previous = next
         if previous == None:
               return
         # move back to advance
         previous.flagbit = 1
         next = previous.left
         previous.left = current
         current = previous.right
         previous.right = next
```