Administration

- **Minijava project**
  - Phase 6 (Register allocation)
    - due Wednesday April 4th
    - There is a contest 😊
    - Link is on the course homepage
  - Overall
    - Due Wednesday April 11th 11:59pm
    - A few more days to work out any issues
The Plan

- Dataflow analysis
- Optimization
- A Few final words on optimization
  - Garbage collection
    - A few words on techniques
    - Integration with the compiler
- Other compiler-like things
- JIT
Mark and Sweep GC

1. Mark Phase
   - Traverse object graph from roots
   - Mark all reachable objects

2. Sweep Phase
   - Scan memory for unmarked objects
   - Collect all unmarked objects onto a free list
Cost of Mark and Sweep

- Let $H$ be the size of the heap
- Let $R$ be the amount of reachable memory in the heap
- Let $c_1$ be the number of instructions (or memory references) for each Mark phase iteration (say 10)
- Let $c_2$ be the number of instructions (or memory references) for each Sweep phase iteration (say 5)

\[
\frac{c_1 \times R + c_2 \times H}{H - R}
\]
Copying GC

1. Divide memory into two equal sized pieces
   - from space and to space
2. Allocate in from space
3. When from space fills up
   - Copy every live object into to space, bringing them all together at one end
   - You have to adjust every pointer since every object moves
4. Swap from space and to space and go to step 2
Copy Phase

Uses depth-first search

function dfs(p)
  if p is a valid pointer into from space
    if the object at *p has not been moved
      copy the object at *p to to space
      leave a pointer to remember where it went
    for each pointer field f of the object at *p
      f = dfs(f)
  return the new location of the object
Cost of Copying GC

- Let $H$ be the size of the heap
- Let $R$ be the amount of reachable memory in the heap
- Let $c_3$ be the number of instructions (or memory references) for each copy phase iteration (say 10)
Cost of Copying GC

- Let H be the size of the heap
- Let R be the amount of reachable memory in the heap
- Let c3 be the number of instructions (or memory references) for each copy phase iteration (say 10)

\[
c3 \times R \quad \frac{H}{2} - R
\]
The problem with both M&S and Copying is ...

- Object lifetimes
- Empirically, objects either:
  - Die young
  - Live forever
Java: Typical Distribution for Lifetimes of Objects
Next choice – generational collection

- Keep G generations
- New objects are allocated in the newest generation
- Objects get promoted to older generations when they survive multiple collections
Generational Collection

Possible Implementation: Two Generations

G0: Young space
    Copying

G1: Old space
    Mark and Sweep
Complication

How to handle pointers between generations?

G0 (younger)

G1 (older)

roots

Q: What you expect more frequently?

old ➔ young
young ➔ old
Inter-generational Pointers

• Pointers in the older generations are treated as roots when collecting the newer generations

• How do we find them?
  – Use a Remembered Set of pointers in the old generations that point into a newer generation
  – We must monitor writes into objects to update this set
  – This is expensive – a few instructions per write
Cost of Generational GC

• Let H be the size of the heap
• Let R be the amount of reachable memory in the heap
• It depends on way too many things
  – Cost of writes
  – Detailed information about object lifetimes
  – How many generations
  – When objects get promoted
• Empirically, it does very well with only 2 generations
Parallel (concurrent) collectors

- The collectors we’ve seen so far are all “Stop The World”, a.k.a., STW
- Can we do collection work in parallel with the application?
- Yes, but it is tricky because:
  - Locking of key data structures
  - The truth changes as you are trying to find it
    - “Floating garbage”
Parallel collectors

- Java has a bunch of garbage collectors
  - Serial collector (generational – copying)
  - Parallel collector (throughput)
  - Concurrent Mark and Sweep
  - Garbage-First (G1) (generational – copying – compacting)

- https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html
Integration of GC with the compiler

- How do we know the set of roots?
- How do we know that all live objects are reachable by following pointers from the roots?
- How does the GC know how big an allocated object is?
- How does the GC know where the pointers are inside of allocated objects?
Integration of GC with the compiler

- How do we know the set of roots?
  - Some of them are in global variables in the implementation
    - These are pretty easy
  - Some of them are in run-time data structures
    - Stack
    - Registers
    - These are a bit trickier
Integration of GC with the compiler

• Pointer maps
  – The compiler must tell the GC about all live temporaries, whether in the stack or in registers, for every point in the program where GC might happen
  – What are these points where GC might happen?
  – What kinds of information must the pointer map contain?
  – What about callee-save registers?
Integration of GC with the compiler

- How do we know that all live objects are reachable by following pointers from the roots?
  - The language must not allow pointer arithmetic
  - Otherwise the programmer can "hide" pointers
    \[
    \text{int } x = (\text{int}) p ^ (-1);
    \]
    \[
    \text{whatever } *p = (\text{whatever } *) (x ^ (-1));
    \]
  - But even if we tie the hands of the programmer, the compiler writer can spoil the fun!

- The compiler compiles user code to assembly code
- Assembly code has arbitrary pointer arithmetic!
Derived pointers

- The compiler can generate code that creates intermediate pointer values

```c
int[] arr;
len = arr.length - 10;
while (i < len) {
    sum = sum + arr[10 + i];
    i = i + 1;
}
```

```assembly
movq %rdi/*i*/, %rcx
addq $10, %rcx
movq (%rsi/*arr*/, %rcx, 8), %rcx
addq %rcx, %rax/*sum*/
incq %rdi/*i*/
```
Derived pointers

- **arr + 8 * 10 is a constant**

  movq  %rdi/*i*/, %rcx
  addq  $10, %rcx
  movq  (%rsi/*arr*/, %rcx, 8), %rcx
  addq  %rcx, %rax/*sum*/
  incq  %rdi/*i*/

- **Before the loop**

  movq  %rsi/*arr*/, %rdx
  addq  $80, %rdx

- **In the loop**

  movq  (%rdx/*&arr[10]*, %rdi/*i*/, 8), %rcx
  addq  %rcx, %rax/*sum*/
  incq  %rdi/*i*/
Derived pointers

- use a pointer rather than subscripting

movq  %rdi/*i*/, %rcx
addq  $10, %rcx
movq  (%rsi/*arr*/, %rcx, 8), %rcx
addq  %rcx, %rax/*sum*/
incq  %rdi/*i*/

- Before the loop

movq  %rsi/*arr*/, %rdx
addq  $80, %rdx

- In the loop

movq  (%rdx), %rcx
addq  %rcx, %rax/*sum*/
addq  $8, %rdx
Integration of GC with the compiler

• How do we know that all live objects are reachable by following pointers from the roots?
  – The compiler writer has to handle derived pointers
  – In the previous examples, %rdx is derived from %rsi (the pointer to the beginning of the array)
  – If the GC runs when %rdx is live but %rsi is not, then we have a problem
  – What if we have a copying collector?
• How do we update %rdx when the collector runs?
Integration of GC with the compiler

• How does the GC know how big an allocated object is?

    A a;
    a = new A();
    class B extends A { ... }
    a = new B();
    int[] b;
    b = new int[37];
    b = new int[3];
Integration of GC with the compiler

• How does the GC know where the pointers are inside of allocated objects?
  
  ```java
  A a;
  class B extends A { ... }
  a = new B();
  ```

• Based on the runtime type of `a`, the compiler needs to know what fields contain pointers

• Every different memory layout that an object might have needs a descriptor

• From an object, the GC must be able to find the descriptor
Integration of GC with the compiler

- What about stacks?
- What is on the stack?
  - Parameters
  - Return address
  - Saved frame pointer
  - Local variables
  - Saved registers
- Who knows whether a register contains a pointer?
- Who puts that register onto the stack?
My Favourite GC’d Language – Emerald

- Invented in 1984-6
- Distributed object-oriented programming language
- Java-like objects with Pascal-like syntax
Objects move around

const Kilroy <- object Kilroy

process

    const home <- locate Kilroy

    for there in allhosts
        move Kilroy to there
            there.getstdout.PutString["Kilroy was here\n"]
        end for

    move Kilroy to home

end process

end Kilroy
Pointer maps are called templates

One kind of pointer map describes each object

0x2ca664 is a concrete type 0x100000 "kilroy" "kilroy.m"
  name: "kilroy"
  file name: "kilroy.m"
  instance size: 4
  template: %d
  variables
    counter@O4
Pointer maps are called templates

Another kind of pointer map describes each operation

0x2ca6f0 is an op vector element
  name: "process" id: nil
  code is 324 - 0 bytes long
...
  template: %6x%f%5x%2d%10x%3d%5x

variables
  home@L0
  there@L4
  starttime@L8
  diff@L12
  all@L16