Basic Memory Management

Pablo Tesone - Guille Polito - Fundación Uqbar

Memory in Managed Languages

- Generally use automatic memory management
- Objects are explicitly allocated by special operations like ***new***
  - Implemented e.g., as special AST nodes, or native methods
- When objects are not referenced anymore, their memory is *eventually* reclaimed by the GC
- There are many different GC algorithms
  - e.g., mark-sweep, mark-compact, reference count, copy collection…

Strategy 1.1: Use the implementation language

The case of managed languages

- Managed languages already have a good memory management
  - => reuse it!
- For example, use objects in the implementation language to represent objects in the interpreted language
- Could lead to a very efficient memory implementation
- Good integration with the implementation language could benefit from its optimisations too!

Strategy 1.2: Use the implementation language

The case of non-managed languages (e.g., C)

- Non-managed languages require lots of manual intervention
- For example, use a *malloc* to allocate each object
- Main cons:
  - manual tracking of objects, references and manual deallocation
  - system calls are slow

Strategy 2: Do your memory manager

- Allocate a big chunk of memory and manage it yourself
- Probably a bad idea in a managed language (speed)
- Probably the best solution in a non-managed language (speed)
  - And still: manual tracking of objects, references and manual deallocation
- Good solution when the memory model of the host do not fit:
  - Size mismatch: e.g., A lot of small objects
  - Missing Optimizations: e.g., quick dying objects, variable objects
  - Missing Features: weak objects / ephemerons

Garbage Collector Kinds

The easy alternatives

- Reference counting
- Mark-sweep
- Semi-space

Reference counting

- On every assignment
  - Decrement reference count of previous object
  - Increment reference count of new object
  - Deallocate object when count reaches 0

- Needs space to manage the count
- Performance is homogeneous
- Special care needs to be taken to handle cycles

Mark-sweep

- When no more space is available
  - iterate all objects and mark the reachable ones
  - iterate all objects and deallocate non-marked ones

- Needs space to manage a mark
- Generates pauses (generally linear to the size of the memory)
- Fragments memory

Semi Space

- Allocate a big chunk and split in two (present and future)
- When no more space is available
  - iterate reachable objects in present, copy them to future
  - swap(present, future)

- Wastes half of the memory
- Generates pauses (generally linear to the size of the semi-space)
- Memory is compacted automatically

Conclusion

- Strategies for memory management depend on tradeoffs and the implementation technology
- If you need to implement your own memory manager, several options appear, and three main families:
  - reference count
  - mark-sweep
  - semi-space