Major Areas of Memory

Static area

* Fixed size, fixed content, allocated at compile time

Run-time stack

* Variable size, variable content (activation records)
* Used for managing function calls and returns

Heap

* Fixed size, variable content
* Dynamically allocated objects and data structures
  + Examples: ML reference cells, malloc in C, new in Java

Cells and Liveness

Cell = data item in the heap

* Cells are “pointed to” by pointers held in registers, stack, global/static memory, or in other heap cells

Roots: registers, stack locations, global/static variables

A cell is live if its address is held in a root or held by another live cell in the heap

Garbage

Garbage is a block of heap memory that cannot be accessed by the program (Cells that are not live)

* An allocated block of heap memory does not have a reference to it (cell is no longer “live”)
* Another kind of memory error: a reference exists to a block of memory that is no longer allocated

Garbage collection (GC) - automatic management of dynamically allocated storage

* Reclaim unused heap blocks for later use by program

Example of Garbage

Diagram

Description automatically generated

This is not live bc there is nothing that can access that particular cell, there is no reference to it.

Garbage collection should handle these things.

Why Garbage Collection?

Today’s programs consume storage freely

* 8-16 GB laptops, 16-64GB desktops, 64-512GB servers
* 64-bit address spaces (SPARC, Itanium, Opteron)

… and mismanage it

* Memory leaks, dangling references, double free, misaligned addresses, null pointer dereference, heap fragmentation
* Poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging

Explicit memory management breaks high-level programming abstraction

GC and Programming Languages

GC is not a language feature, it is secondary program

GC is a pragmatic concern for automatic and efficient heap management

* Cooperative langs: Lisp, Scheme, Prolog, Smalltalk …
* Uncooperative languages: C and C++
  + But garbage collection libraries have been built for C/C++

Recent GC revival

* Object-oriented languages: Modula-3, Java
  + In Java, runs as a low-priority thread; System.gc may be called by the program
* Functional languages: ML and Haskell

The Perfect Garbage Collector

No visible impact on program execution

Works with any program and its data structures

* For example, handles cyclic data structures

Collects garbage (and only garbage) cells quickly

* Incremental; can meet real-time constraints

Has excellent spatial locality of reference

* No excessive paging, no negative cache effects

Manages the heap efficiently

* Always satisfies an allocation request and does not fragment

Summary of GC Techniques

Reference counting

* Directly keeps track of live cells
* GC takes place whenever heap block is allocated
* Doesn’t detect all garbage

Tracing

* GC takes place and identifies live cells when a request for memory fails
* Mark-sweep
* Copy collection

Modern techniques: generational GC

Reference Counting

Simply count the number of references to a cell

Requires space and time overhead to store the count and increment (decrement) each time a reference is added (removed)

* Reference counts are maintained in real-time, so no “stop-and-gag” effect
* Incremental garbage collection

Unix file system uses a reference count for files

C++ “smart pointer” (e.g., auto\_ptr) use reference counts

*Example*:

Diagram

Description automatically generated

From time to time, I will go and say “now I need to collect some garbages (everything with 0 reference)”.

*Strengths*:

Incremental overhead

* Cell management interleaved with program execution
* Good for interactive or real-time computation

Relatively easy to implement

Can coexist with manual memory management

Spatial locality of reference is good

* Access pattern to virtual memory pages no worse than the program, so no excessive paging

Can re-use freed cells immediately

* If RC == 0, put back onto the free list

*Weaknesses*:

Space overhead

* 1 word for the count, 1 for an indirect pointer

Time overhead

* Updating a pointer to point to a new cell requires:
  + Check to ensure that it is not a self-reference
  + Decrement the count on the old cell, possibly deleting it
  + Update the pointer with the address of the new cell
  + Increment the count on the new cell

One missed increment/decrement results in a dangling pointer / memory leak

Cyclic data structures may cause leaks

*Cycles*:

Diagram

Description automatically generated

You can catch these cycles by going after them. This is gonna be much more expensive than just counting these numbers.

These type of garbage collection mechanisms are implemented in some of the libraries like smart pointers in C++.

“Smart Pointer” in C++

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Another algorithm is Mark-Sweep Garbage Collection.

Mark-Sweep Garbage Collection

Each cell has a mark bit

Garbage remains unreachable and undetected until heap is used up; then GC goes to work, while program execution is suspended

Marking phase

* Starting from the roots, set the mark bit on all live cells

Sweep phase

* Return all unmarked cells to the free list
* Reset the mark bit on all marked cells

Starts from the root, mark everything that is touched.

Example 1:

Diagram, engineering drawing

Description automatically generated

Example 2:

Diagram, engineering drawing

Description automatically generated

Example 3:

Diagram

Description automatically generated

Example 4:

Diagram

Description automatically generated

These 2 are not touched.

Expensive because you need to go through the entire thing.

Mark-Sweep Costs and Benefits

Good: handles cycles correctly

Good: no space overhead

* 1 bit used for marking cells may limit max values that can be stored in a cell (e.g., for integer cells)

Bad: normal execution must be suspended

Bad: may touch all virtual memory pages

* May lead to excessive paging if the working-set size is small and the heap is not all in physical memory

Bad: heap may fragment

* Cache misses, page thrashing; more complex allocation

Another algorithm which is more expensive but useful: Copying Collector

Copying Collector

Divide the heap into “from-space” and “to-space”

Cells in from-space are traced and live cells are copied (“scavenged”) into to-space

* To keep data structures linked, must update pointers for roots and cells that point into from-space
  + This is why references in Java and other languages are not pointers, but indirect abstractions for pointers
* Only garbage is left in from-space

When to-space fills up, the roles flip

* Old to-space becomes from-space, and vice versa

*Copying a Linked List (Cheney’s algorithm):*

Diagram, engineering drawing

Description automatically generated

Takes your thing and copies it and during this copying, it gets rid of the unreachable ones.

Advantage of this one is also it can enforce some sort of defragmentation of your memory.

*Flipping Spaces:*

Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Diagram, engineering drawing

Description automatically generated