# Automatic Memory Management

- Concepts
- Approaches

# Automatic Memory Management

- Automatic memory management distrusted by systems programmers
  - Perceived high processor and memory overheads, unpredictable timing
  - But, memory management problems are common:
    - Unpredictable performance
      - Calls to **`malloc()`**/**`free()`** can vary in execution time by several orders of magnitude
    - Memory leaks
    - Memory corruption and buffer overflows
    - Use-after-free
    - Iterator invalidation
- New automatic memory management schemes solve many problems
  - Garbage collectors → lower overhead, more predictable
    - Also system performance improvements made overhead more acceptable
  - Region-based memory management → predictability, compile time guarantees

# Memory Management in Systems Programs

- Systems programs traditionally used a mix of manual and automatic memory management:
  - Stack memory managed automatically:
    - In the example, memory for **`di`** is automatically allocated when the function executes; freed on completion
    - Simple and efficient for languages like C/C++ that have complex value types
    - Less effective for Java-like languages, where objects always allocated on the heap
  - Heap memory managed manually:
    - Allocation using **`malloc()`**, deallocation using explicit **`free()`**

```
int saveDataForKey(char *key, FILE *outf)
{
   struct DataItem di;

   if (findData(&di, key)) {
       saveData(&di, outf);
       return 1;
   }
   return 0;
}
```

# Managing the Heap

- Aim: to find objects that are no longer used, make their space available for reuse
  - An object is no longer used (ready for reclamation) if it is not reachable by the running program via any path of pointer traversals
  - Any object that is *potentially* reachable is preserved – better to waste memory than deallocate an object that's in use

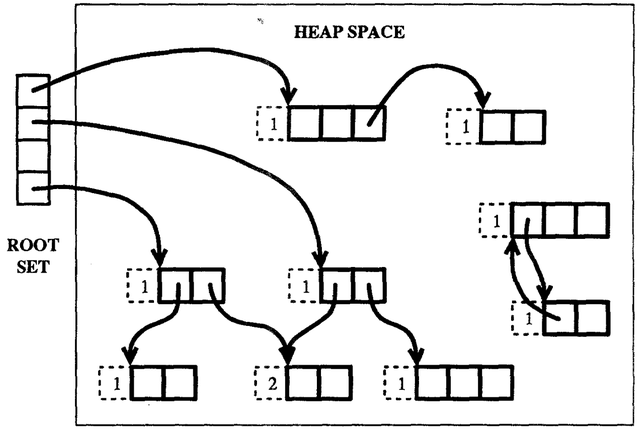
- Approaches to automatic heap management:
  - Reference counting
  - Region-based lifetime tracking
  - Garbage collection → lecture 6

# Reference Counting

- Reference counting
- Costs and benefits

# Reference Counting

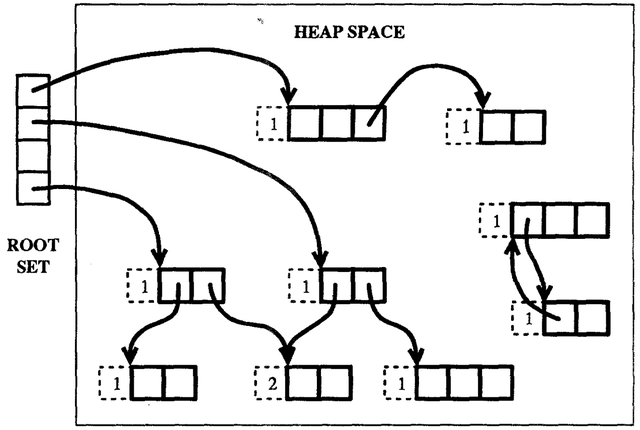
- Simplest automatic heap management
- Allocations contain space for an additional **reference count**
  - An extra **`int`** is allocated along with every object
  - Counts number of references to the object
    - Increased when new reference to the object is created
    - Decremented when a reference is removed
  - When reference count reaches zero, there are no references to the object, and it may be reclaimed
    - Reclaiming object removes references to other objects
    - May reduce their reference count to zero, so triggering further reclamation




Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI:10.1007/BFb0017182

# Reference Counting: Benefits

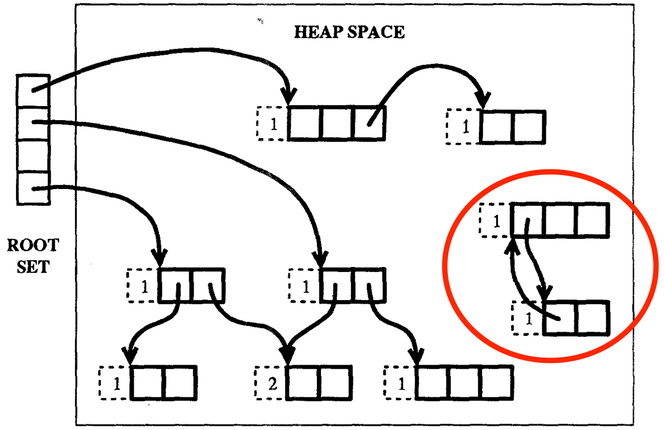
- Incremental operation – memory reclaimed in small bursts
- Predictable and understandable
  - Easy to explain
  - Easy to understand when memory is reclaimed
  - Easy to understand overheads and costs
  - Follows programmer intuition




Source: P. Wilson, “Uniprocessor garbage collection techniques”, Proc IWMM’92, DOI:10.1007/BFb0017182

# Reference Counting: Costs

- Cyclic data structures contain mutual references
  - Objects that reference each other aren't reclaimed, as reference count doesn't go to zero
  - Memory leaks unless cycle explicitly broken; needs programmer action
- Stores reference count alongside each object
  - Maybe also a mutex if concurrent access possible
  - Per-object overhead significant for small objects; wastes memory
- Processor cost of updating references can be significant for short-lived objects




Source: P. Wilson, "Uniprocessor garbage collection techniques", Proc IWMM'92, DOI:10.1007/BFb0017182

# Reference Counting

- Widely used in scripting languages
  - Python, Ruby, etc.
  - Memory and processor overhead not significant in interpreted runtime

- Used on small scale for systems programming
  - e.g., Objective C runtime on iOS
  - Ease of understanding is important
  - Tends to be for large, long-lived, data – reduces overheads
  - Not typically used in kernel code, high-performance systems

# Automatic Memory Management

- Concepts
- Reference counting