

Runtime Environments II

Basilio B. Fraguera

Heap

- Storage for data that is
 - neither declared statically
 - nor declared as a procedure variable
 - typically data created in a procedure that outlives it
- Objects are explicitly allocated in the heap
 - at runtime
 - here “object” is not necessarily in the sense of Object Oriented languages
- Deallocation can be manual or automatic

Memory Manager

- Responsible for *allocation/deallocation* of space in the heap
 - Can interact with OS to request more memory
- Desirable properties:
 - Space efficiency: minimize fragmentation
 - Program efficiency: seek locality
 - Low overhead

Memory Hierarchy

- Computer and embedded systems memory is organized as a hierarchy
 - No single level can be large + fast + cheap
 - Small, expensive, fast memories near the processor
 - Large, cheap, slow memories far from the processor
 - Data moves in blocks between levels of the hierarchy under demand

Locality

- Memory hierarchy is successful because of locality
 - Ensures data in small/fast memories near the processor get used much more than data in large/slow memories
- Two kinds of locality:
 - Temporal locality: same data reused
 - Spatial locality: nearby data used

Allocation

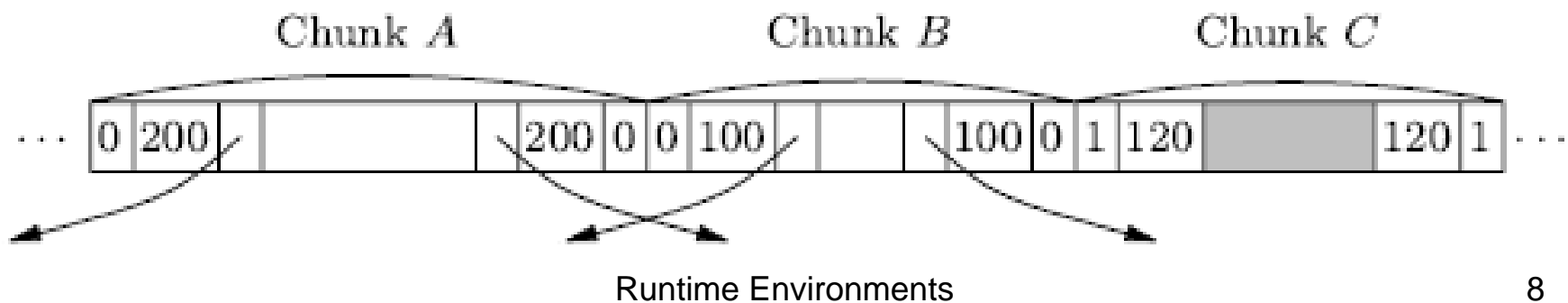
- Find a *hole* of appropriate size
 - If its size is $>$ the one sought, *split*
- Several algorithms to choose the hole:
 - First Fit: First chunk found where object can fit
 - Best Fit: Smallest chunk so that the object can fit
 - Worst Fit: Always in the largest chunk
 - Next Fit: Try first in the chunk used in the last allocation
- Usage of *bins* helps
 - Extensible *wilderness chunk*

Deallocation

- *Coalescing* helps reduce fragmentation
- If bins are used, it may be better not to coalesce
 - instead, keeps same-size chunks in pages
 - and use a bitmap to know their status
- Efficient coalescing requires data structures:
 - Boundary tags
 - Double-linked free list

Illustration

- In this figure:
 - Chunk A is 200 bytes long, was free
 - Chunk B is 100 bytes long, just deallocated
 - Chunk C is 120 bytes long, not free



Manual Deallocation Dangers

- *Memory leak*: Forgetting to deallocate data that is no longer used
 - Does not break program, but may slowdown
- Dereferencing a *dangling pointer*: Trying to access deallocated data
 - May break the program (if space still free)
 - Or yield random results (if space reallocated)

Solutions

- Automatic garbage collection
- Programming strategies:
 - Object ownership (by only one pointer)
 - When object's lifetime can be determined statically
 - Reference counting
 - When lifetime needs to be determined at runtime
 - Does not catch circular data structures
 - Region-based allocation
 - For set of objects related to one stage of the program
 - Can be seldom applied

Garbage Collectors

- Requirements
 - Language must be *type safe*
 - Allows collector to identify pointers and sizes
 - Pointers point to the beginning of objects
 - No arbitrary manipulation of pointers allowed
- Performance metrics
 - Overall Execution Time: collection is expensive
 - Space Usage: avoid fragmentation
 - Pause Time: importance depends on application
 - Program Locality: both temporal and spatial

Reachability

- *Root set*: data available without following pointers
- Reachable object:
 - Object in the root set
 - Object reachable from reachable object
- Beware of compiler interaction
 - Compiler/garbage collector cooperation

Changes to Reachable Set

- Allocations
 - Parameter passing and return
 - Reference assignment
 - Procedure returns
-
- When an object becomes unreachable, it may turn other objects unreachable too

Reference Counting: Algorithm

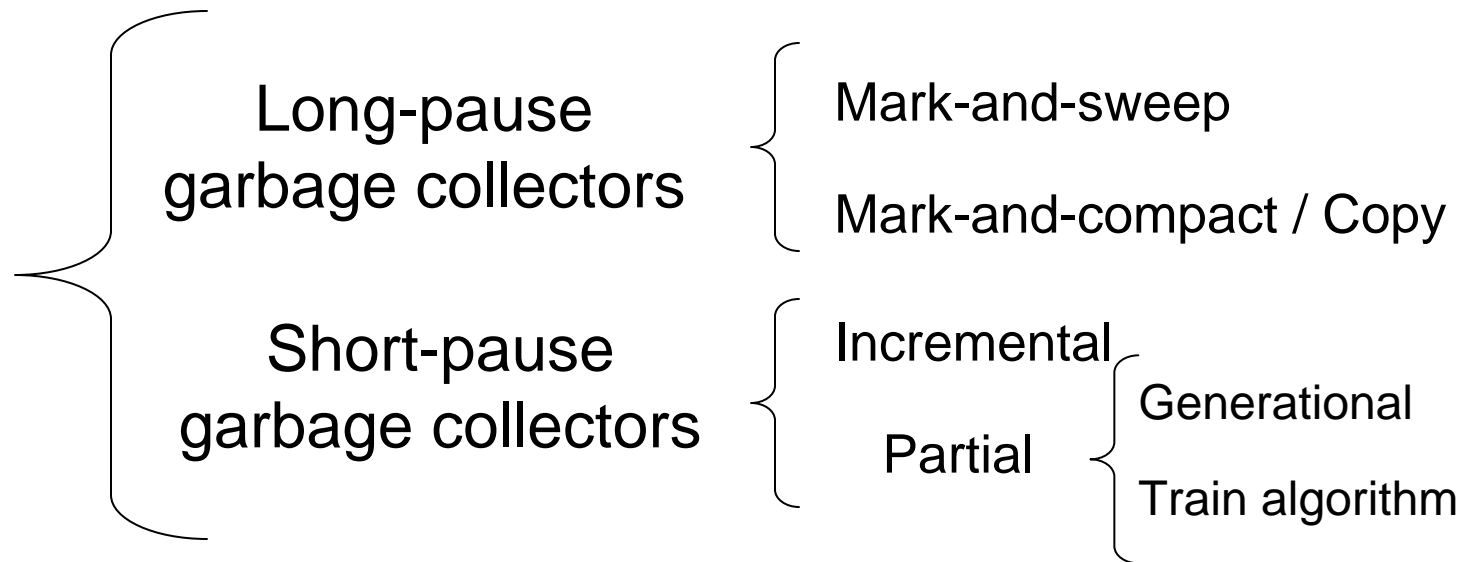
- Operations:
 - Allocation of A: $c(A)=1$
 - Passing A as parameter: $c(A)++$
 - Reference assignment $u=v$:
 - $c(*u)--$ (here $*u$ is whatever u pointed to before)
 - $c(*v)++$
 - Procedure return
 - For every reference to A to from local frame, $c(A)--$
 - If $c(A)==0 \Rightarrow c(B)--$ for every B it points to

Reference Counting : Properties

- Pros:
 - Allows incremental garbage collection
 - Garbage is collected immediately
 - Potential for improving locality
- Cons:
 - Expensive
 - Does not detect unreachable cyclic structures

Trace-Based Collection

- Check reachability of objects periodically
 - Available space < threshold, for instance
- Classification



Concepts in Trace-Based Collectors

- A *Free* list holds the unallocated chunks
- Initially all objects are *unreached*
 - Some algorithms have an initial *Unreached* list
- As reachable objects are discovered
 - they are put in an *Unscanned* list to check their pointers later
 - A reached-bit may be activated
- Checked objects may go to a *Scanned* list

Basic Mark-and-sweep Collector

```
    /* marking phase */
1)  set the reached-bit to 1 and add to list Unscanned each object
    referenced by the root set;
2)  while (Unscanned  $\neq$   $\emptyset$ ) {
3)      remove some object o from Unscanned;
4)      for (each object o' referenced in o) {
5)          if (o' is unreachable; i.e., its reached-bit is 0) {
6)              set the reached-bit of o' to 1;
7)              put o' in Unscanned;
            }
        }
    }
    /* sweeping phase */
8)  Free =  $\emptyset$ ;
9)  for (each chunk of memory o in the heap) {
10)     if (o is unreachable, i.e., its reached-bit is 0) add o to Free;
11)     else set the reached-bit of o to 0;
    }
```

Optimized (Baker's) Mark-and-sweep Collector

- 1) $Scanned = \emptyset$;
- 2) $Unscanned =$ set of objects referenced in the root set;
- 3) **while** ($Unscanned \neq \emptyset$) {
- 4) move object o from $Unscanned$ to $Scanned$;
- 5) **for** (each object o' referenced in o) {
- 6) **if** (o' is in $Unreached$)
- 7) move o' from $Unreached$ to $Unscanned$;
- }
- }
- 8) $Free = Free \cup Unreached$;
- 9) $Unreached = Scanned$;

Fragmentation Reduction

- Makes more effective usage of memory
 - Easier allocation of large objects
- +Program locality \Rightarrow +Performance
- Simpler free storage management data structures

Mark-and-compact Collector

```
/* mark */
1) Unscanned = set of objects referenced by the root set;
2) while (Unscanned ≠ ∅) {
3)     remove object o from Unscanned;
4)     for (each object o' referenced in o) {
5)         if (o' is unreached) {
6)             mark o' as reached;
7)             put o' on list Unscanned;
            }
        }
    }
/* compute new locations */
8) free = starting location of heap storage;
9) for (each chunk of memory o in the heap, from the low end) {
10)    if (o is reached) {
11)        NewLocation(o) = free;
12)        free = free + sizeof(o);
    }
}
/* retarget references and move reached objects */
13) for (each chunk of memory o in the heap, from the low end) {
14)    if (o is reached) {
15)        for (each reference o.r in o)
16)            o.r = NewLocation(o.r);
17)        copy o to NewLocation(o);
    }
}
18) for (each reference r in the root set)
19)    r = NewLocation(r);
```

Cheney's Copying Collector

```
1) CopyingCollector () {
2)     for (all objects o in From space) NewLocation(o) = NULL;
3)     unscanned = free = starting address of To space;
4)     for (each reference r in the root set)
5)         replace r with LookupNewLocation(r);
6)     while (unscanned ≠ free) {
7)         o = object at location unscanned;
8)         for (each reference o.r within o)
9)             o.r = LookupNewLocation(o.r);
10)        unscanned = unscanned + sizeof(o);
        }
    }

    /* Look up the new location for object if it has been moved. */
    /* Place object in Unscanned state otherwise. */
11) LookupNewLocation(o) {
12)     if (NewLocation(o) = NULL) {
13)         NewLocation(o) = free;
14)         free = free + sizeof(o);
15)         copy o to NewLocation(o);
        }
16)     return NewLocation(o);
    }
```

Cost of the Different Algorithms

- Basic Mark-and-Sweep: proportional to the number of chunks
- Baker's Mark-and-Sweep: proportional to the number of reachable objects
- Basic Mark-and-Compact: proportional to the number of chunks + total size of the reachable objects
- Cheney's Copying Collector: proportional to the total size of the reachable objects

Incremental Garbage Collection

- Operates interleaved with the execution of the program (*mutator*)
- Complex: mutator can modify already analyzed objects
 - Meaningful mutator actions are recorded
- Simplification allowing some *floating garbage* not to be collected till next round
 - It can only consist of objects that were reachable at the beginning of the round

Mutator – Incremental Garbage Collector Conflict

- Objects that are actually reachable can be erroneously classified as unreachable
- Scenario:
 - Mutator copies reference from object not yet analyzed o_1 to object not yet analyzed o_2 to an already scanned object X
 - The reference to o_2 in o_1 is overwritten o_1 before is analyzed (or o_1 became unreachable)
 - o_2 is reachable through X , but we misclassify it

Mutator Actions to Monitor

- Writes in already scanned objects of referenced to unreachd objects: Write Barriers
 - Most efficient approach
- Reads of references in unreachd or unscanned objects: Read Barriers
- Loss of reference in an unreachd/ unscanned object: Transfer Barriers

Partial Garbage Collection

- Key observations
 - Most objects (80-98%) die quickly
 - Objects that survive a collection are more likely to survive future collections
- A classification of objects according to their age would make the garbage collection more efficient: Generational garbage collection
 - Garbage-collect in younger sets more often