Runtime Environments

Sudakshina Dutta

IIT Goa

27^{th} April, 2022

Run-Time Environments

- Address binding is the process of mapping from one address space to another address space
- Logical address is an address from the perspective of an executing application program
- Physical Address refers to location in memory unit
- If we know where process will reside in memory, then physical address is embedded to the executable of the program during compilation
- If it is not known at the compile time, then relocatable address will be generated. Loader translates the relocatable address to absolute address
- Sometimes programs/compiled units may need to be relocated during execution. The logical address needs to be translated

—Need for memory management unit

Heap Memory Management

- Portion of the virtual store that is used for data that lives indefinitely, or until the program explicitly deletes it
- Local variables become inaccessible when their procedures end
- Memory manager allocates and deallocates space within the heap

Heap memory management

- Dynamic memory allocation and deallocation based on the requirements of the program
  - malloc()and free()in C programs
  - new() and delete() in C++ programs
  - new() and garbage collection in Java programs

Allocation and deallocation may be completely manual (C/C++), semi-automatic (Java), or fully automatic (Lisp)

The memory manager

- Handles heap memory allocation and deallocation
- Allocation : A chunk of contiguous memory chunk is allocated of requested size. If possible, allocation request is satisfied. If such chunk is not available, it requests more virtual memory from operating system
- Deallocation : Deallocated space is returned to the pool
- We require space efficiency, program efficiency and low overhead
  - Spatial and temporal locality
- Task of a memory manager is to reduce fragmentation, managing and coalescing free space
  - First-fit, best-fit, etc

The memory manager

- As allocation and deallocation requests are made, the memory space is broken up into free and used chunks of memory
- The free chunks need not reside in contiguous area of memory
- With each allocation request, the memory manager finds the large enough free memory
  - It may need to split
- With deallocation request, the freed chunks are returned back to the pool

–Coalescing smaller holes into larger holes

Need for garbage collection

- Memory leaks

  —Memory which is no longer needed is not released
- Failing to delete data that cannot be referenced - not returned to heap
- Important in long program
- Referencing dangling pointers - returned to heap
- Solution: automatic garbage collection

Introduction to garbage collection

- Data that cannot be referenced are known as garbage

  Node *p = $\cdots$ malloc $\cdots$;
  Node *q = $\cdots$ malloc $\cdots$;

  p = q;

- Many high-level language removes the burden of manual memory management from the programmer by offering automatic garbage collection, which deallocates unreachable data
- Languages that offer garbage collection : Java, Perl
- Periodically the garbage collector runs and frees unreachable objects

Process of garbage collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- Objects have type which can be determined by garbage collector at run-time
- A language for which type of any data component can be determined is said to be type safe e.g., Java
- Unsafe languages - C, C++
  - Arbitrary arithmetic operations can be applied to generate new pointers
- A user program, called mutator, is used by acquiring space from memory manager and drop references to existing objects
- Garbage collector finds these unreachable objects and hand them over to memory manager if mutator program cannot find them

Evaluation of performances

- Garbage collector can be very expensive
- Overall execution time can be very slow
- It is important that garbage collector avoids fragmentation and make best use of the available memory
- Mutators can pause suddenly for extremely long time and garbage collector kicks in without warning

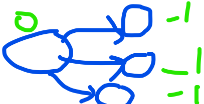
Reference counting garbage collector

- A simple garbage collector
- It identifies objects from being reachable to being unreachable
- Each object has a reference count field
- The object is deleted when the count becomes zero

Reference counting garbage collector

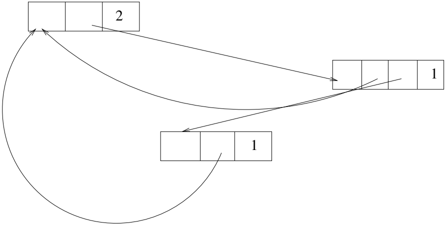
Reference counts can be maintained as follows:

- **Object allocation** : The reference count is set to 1
- **Parameter passing** : Reference count of each object passed into procedure gets incremented
- **Reference assignment** : If a statement u = v is present, the reference count of the object pointed to by v is incremented and the reference count of the object pointed to by u gets decremented
- **Procedure return** : If a procedure returns, reference counts of all the local variables get decremented
- **Reachability** : If reference count becomes zero for an object, the count of each object pointed to by a reference gets decremented



Decrement 1

Disadvantage



Cannot be reclaimed by reference counting garbage collector

Disadvantage

- Cannot collect unreachable cyclic data structure

```
struct Q
{
    struct R *r;
};
struct P
{
    struct Q *q;
};
struct R
{
    struct P *p;
};
```

struct P *$p_1 = \cdots$ malloc $\cdots$;
struct P *$p_2 = \cdots$ malloc $\cdots$;

$p_1 = p_2$;

- Overhead of keeping reference field

Mark and sweep garbage collector

- Can collect unreachable cyclic data structure
- Instead of collecting garbage as it is created, trace-based collector runs periodically
  - Based on demand of free space or if the amount of free space goes below threshold
- Two phases
  - Marking reachable objects
  - Sweeping to reclaim storage
- It keeps a set root set of variables that can be referenced by program without dereferencing any pointer

Mark and sweep garbage collector

If TIME, Understand this again!

- Input : A root set of objects, a heap and a free list, *Free*, with all unallocated data from the heap
- Output : A modified free list *Free* after removing all the garbage

Mark

- Start scanning from root set, mark all reachable objects (set reached_bit $= 1$), place them on the list *Unscanned*
- while (*Unscanned* $\neq \phi$) do
  $\{$
  object o $=$ delete(*Unscanned*);
  for (each object o_1 referenced in o)do$\{$
  if (reached_bit(o_1) $==$ 0)
  $\{$reached_bit(o_1) $=$ 1; place o_1 on *Unscanned*;$\}$
  $\}$
  $\}$

Sweep

- Free $= \phi$;
- for (each object o in the heap) do

```
{ if (reached-bit(o) == 0)
    add(Free, o);
  else
    reached-bit(o) = 0;
}
```