Runtime Environments(Group 16)

Dipesh Kafle- 106119029
Hanan Abdul Jaleel - 106119045
Marmik Upadhyay - 106119073
Udith Kumar V - 106119139

National Institute of Technology, Tiruchirappalli

Section 1

Introduction

Introduction

- A compiler must accurately implement the abstractions embodied in the source language definition. These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs. The compiler must co- operate with the operating system and other systems software to support these abstractions on the target machine
- To do so, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed
- Two most important things to address is
  - Storage Locations
  - Access to variables and data
- These two things involve concepts such as memory management, stack allocation, heap management and garbage collection

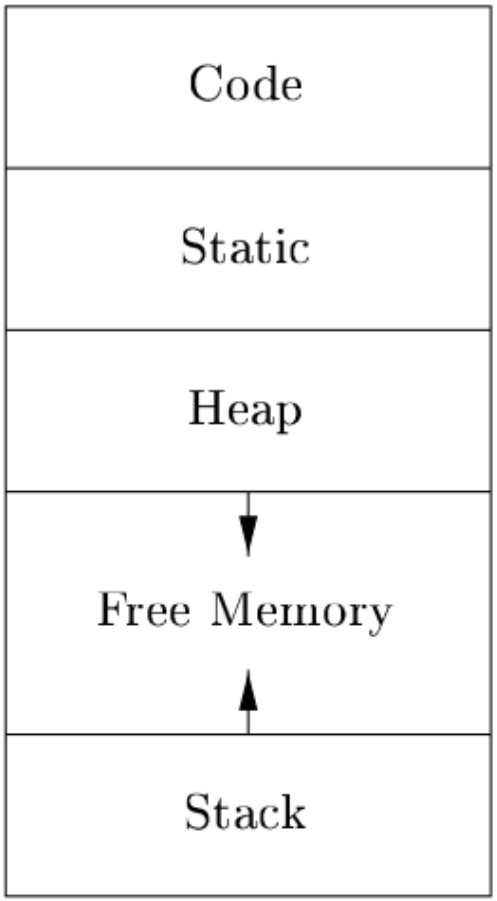
Section 2

Storage Organization

Data Types at runtime

- During the runtime of a program, we won't know the exact type of data.
- The amount of storage needed for a name is determined by it's type.
- An elementary data type, such as a character, integer, or float, can be stored in an integral number of bytes. Storage for an aggregate type, such as an array or structure, must be large enough to hold all its components.
- In modern CPUs, when a memory access is aligned, it's significantly faster(and in some CPUs some instructions expect accesses to be aligned). Compiler may decide to add padding to structs as to improve alignment in such cases.

Memory subdivision



Memory Subdivision(contd.)

Code

The size of the generated target code is fixed at compile time, so the com- piler can place the executable target code in a statically determined area Code, usually in the low end of memory.

Static

The size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called Static.

Memory Subdivision(contd.)

Stack

- The stack is used to store data structures called activation records that get generated during procedure calls
- In practice, the stack grows towards lower addresses, the heap towards higher(the diagram has it the other way around, but that doesn't matter).
- The stack will store things such as return address from a function call, local variables, etc.

Heap

- Many programming languages allow the programmer to allocate and deallocate data under program control.
- The heap is used to manage this kind of long-lived data.
- C/C++ has malloc/realloc/free functions for doing heap memory management.

Memory Subdivision(contd.)

Static vs Dynamic Storage Allocation

- We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes
- A storage-allocation decision is dynamic if it can be decided only while the program is running.
- Compilers use combination of Stack storage and Heap storage for dynamic storage allocation.

Section 3

Stack Allocation

Stack Allocation

- Storage for local variables can be allocated on a run- time stack for languages that allow or require local variables to become inaccessible when their procedures end.
- Each live activation has an activation record (or frame) on the control stack
- The root of the activation tree is at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides.

Section 4

Heap Management

Heap Management

- The heap is the portion of the store that is used for data that can live indefinitely, or until the program deletes it explicitly
- There are usually two approaches with heap management.
  1. Manual Memory Management
  2. Automatic Memory Management with Garbage Collection

Manual Memory Management

Languages such as C/C++ and Rust offer this mechanism for heap memory management. Two major issues with this are :

- Memory leak error
- Dangling pointer dereference error

This is unsafe and requires a lot of book keeping from programmer's side. Although, newer langauges are increasing safety and reducing programming burden with advanced type system.

```
void leak(){
    int *arr = malloc(100*sizeof(int));
    throw std::runtime_error("Something went wrong");
    free(arr);
    return arr;
}
int* dangling(int n){
    int *arr = malloc(100*sizeof(int));
    int *nth = arr[n];
    free(arr);
    return nth;
}
```

Automatic Memory Management with Garbage Collection

- Languages such as Java, Go, Ruby, Python offer this.
- This is much safer as the programmer doesn't have the mental overhead of managing memory.

```
def f():
    arr = [i for i in range(1,1000)]
    return arr

def main():
    a = f()
    print(a)

# No explicit calls made to malloc/free made by the programmer
# Programmer never needs to know the gory details of memory management
# Memory is managed automatically by the python runtime, using reference counting GC
main()
```

Section 5

Garbage Collection

Garbage Collection

- Data that cannot be referenced is generally known as garbage.
- Garbage Collection dates back to the initial implementation of Lisp in 1958.

Goals of a Garbage Collector

- **Type Safety** : C++ is inherently unsafe in this due to it providing mechanism to create pointers arbitrarily.So,it's difficult to have GC for C++.
- **Performance** : Performance metrics that are generally considered for GC are:
  - Overall Execution Time
  - Space Usage
  - Pause Time(GC pauses)
  - Program Locality: Since GC handles all the heap memory, it might decide to make accesses cache friendly by putting them close by(some garbage collectors perform memory compaction, which improves locality and fragmentation)
- **Reachability** : The data that can be accessed directly by a program, without having to dereference any pointer, as the root set. Program registers, static field members, stack variables,etc should be considered as roots. A program obviously can reach any member of its root set at any time and any object with a reference that is stored in the field members or array elements of any reachable object is itself reachable.

Section 6

Operations of a GC mutator

Operations of a GC mutator

- **Object Allocations** : Adds members to the reachable objects set
- **Parameter Passing and Return Values**: Passed parameters are still reachable and doesn't affect reachable set.
- **Reference Assignments**: Of the form u=v. u now refers to v. The old u's reference is gone and if it was the last reference, reachable set decreases.
- **Procedure Returns**: As a procedure exits, it's frame is also popped off. So the reachable set will either stay the same or decrease.

In summary, new objects are introduced through object allocations. Parameter passing and assignments can propagate reachability; assignments and ends of procedures can terminate reachability. As an object becomes unreachable, it can cause more objects to become unreachable.

Section 7

Two Approaches To Track Reachability

Two Approaches To Track Reachability

- We catch the transitions as reachable objects turn unreachable. Example: Reference counting GC in Python, Swift,etc., C++ shared_ptr, Rust's Rc and Arc types
- We periodically locate all the reachable objects and then infer that all the other objects are unreachable. Example: Trace based GC such as mark and sweep garbage collector

Trace Based Collection

- Instead of collecting garbage as it is created, trace-based collectors run periodically (typically when the free space is exhausted or it's amount drops below some threshold) to find unreachable objects and reclaim their space

All trace-based algorithms compute the set of reachable objects and then take the complement of this set. Memory is therefore recycled as follows:

1. The program or mutator runs and makes allocation requests.
2. The garbage collector discovers reachability by tracing.
3. The garbage collector reclaims the storage for unreachable objects.

- An object may be in one of the 4 states: Free(Blue), Unreached(White), Unscanned(Gray), Scanned(Black)

Tracing Based Collection(contd.)

Mark-and-sweep garbage-collection algorithms are straightforward, stop-the-world algorithms that find all the unreachable objects, and put them on the list of free space.

Basic Mark-and-Sweep Collector

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

Tracing Based Collection(contd.)

Baker's Mark-and-Sweep Collector

```
1)  Scanned = Unscanned = ∅;
2)  move objects referenced by the root set from Unreached to Unscanned;
3)  while (Unscanned ≠ ∅) {
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 ∪ Unreached;
9)  Unreached = Scanned;
```

Mark-and-Compact Garbage Collectors

- Relocating collectors move reachable objects around in the heap to eliminate memory fragmentation.
- Relocating collectors vary in whether they relocate in place or reserve space ahead of time for the relocation
- They are similar to normal mark and sweep but they need to map one address to another address for relocation.

(Algorithm too big to fit in the slide)

Tracing Based Collection(contd.)

Copying Collectors

The memory space is partitioned into two semispaces, A and B . The mutator allocates memory in one semispace, say A, until it fills up, at which point the mutator is stopped and the garbage collector copies the reachable objects to the other space, say B . When garbage collection completes, the roles of the semispaces are reversed.

```
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);
    }
```

Figure 1: Cheney's copying collector

Section 8

Not Sure if it's in the syllabus

Not Sure if it's in the syllabus

Advanced Topics in Garbage Collection

- Incremental GC
- Parallel and Concurrent GC
- Precise and Conservative Garbage Collectors
- Reducing GC pauses

Advanced Topics in Runtimes

- Language runtimes for parallelism/concurrency (such as Golang) to support m:n parallelism
- Fault tolerant virtual machines such as BEAM virtual machines, etc

Section 9

References

References

- Compilers Principles, Techniques and Tools by Aho, Lam, Sethi and Ullman