Recitation - 08

Jahnavi - jp5867@nyu.edu

Heap allocation:

- Heap allocation is necessary when we need to dynamically allocate memory during runtime for data structures that are too large to fit on the stack or that need to be allocated and deallocated at runtime.
- When we allocate memory on the heap, we can use it for as long as we need, even after the function in which it was allocated has returned. We can also allocate and deallocate memory on the heap during runtime, which gives us more flexibility in managing memory.

Allocation Methods:

Free list:

- A free list is a data structure used in heap allocation to keep track of blocks of memory that are not currently being used.
- When memory is allocated on the heap, it's often not possible to allocate exactly the amount of memory that's needed for a particular data structure.
- As a result, we may end up with unused blocks of memory on the heap that are too small to be useful for other data structures.
- In a linked list free list, each block of memory is represented by a node in the list. When a block of memory is freed, a new node is created for the block and added to the head of the list.
- When memory is allocated, the allocator searches the list from the head to find a block of memory that's large enough to satisfy the request. If a suitable block is found, it's removed from the list and returned to the caller.

Free list allocation methods:

First fit: select the first block large enough to satisfy the request.

Best fit: select the smallest block large enough to satisfy the request.

Worst fit: always select the largest available block.

Disadvantages:

1. Overhead
2. Limited scalability: Free list allocation can become less efficient as the size of the heap increases, since the overhead of managing the free list also increases.
3. Memory leaks: For example, if a memory block is not properly removed from the free list when it's in use, the block will be lost and cannot be used again.
4. Fragmentation: If the memory blocks in the free list are of different sizes, making it difficult to find a block that matches the size of the requested allocation.

Heap pointer:

Initially, the heap pointer is set to bottom of heap

Allocation: the heap pointer is incremented an appropriate amount

Deallocation: defragmentation eventually required

Disadvantages:

Moving of live objects in the memory

Automatic garbage collection algorithms:

Mark and sweep algorithm:

The basic idea of mark and sweep is to periodically identify objects that are no longer being used by the program, and then reclaim the memory used by those objects for reuse.

An object x is live (i.e., can be referenced) if:

1. x is pointed to by some variable located on the stack (e.g., in an activation record) or in static memory
2. there is a register (containing a temporary or intermediate value) that points to x
3. there is another object on the heap (e.g., y) that is live and points to x

Automatic garbage collection algorithms:

Mark phase:

1. The algorithm traverses the memory heap and identifies all the objects that are still being used by the program.
2. This is done by starting at a set of root objects (which are typically global variables or objects on the stack), and then recursively tracing all the references to other objects that are still in use.
3. During the marking phase, each object that is found to be in use is marked as "live".

Sweep phase:

4. The algorithm traverses the memory heap again and frees all the memory that is not marked as "live". This memory can then be reused for future memory allocations.

Copying garbage collection:

1. Copying garbage collection is a type of garbage collection algorithm that works by copying all live objects to a new memory space, and then discarding the old memory space that contains the dead objects.
2. This is done by dividing the heap into two regions, called the "from" space and the "to" space.
3. During the garbage collection process, the garbage collector starts by scanning the root objects to identify all live objects in the "from" space.
4. It then copies each live object to the "to" space, updating any references to other objects as it goes. When an object is copied, all of its referenced objects are also copied, recursively. This process continues until all live objects have been copied to the "to" space.
5. Once the copying phase is complete, the roles of the "from" and "to" spaces are swapped, so that the "to" space becomes the new "from" space, and the old "from" space becomes the new "to" space. This means that all live objects are now in the new "from" space, and the old "to" space can be discarded.

Advantages and disadvantages of copying GC:

1. It eliminates fragmentation, since all live objects are moved to a contiguous memory region during the copying process.
2. It can be faster than mark and sweep for small to medium-sized heaps, since it avoids the need to traverse the entire heap during the garbage collection process.
3. However, copying garbage collection can be less efficient than mark and sweep for large heaps, since it requires twice as much memory as the heap size.
4. Additionally, the copying process can be costly in terms of CPU time, especially for large objects or objects with complex reference graphs.

Generational GC:

1. It takes advantage of the observation that most objects die young, meaning they are allocated and then quickly become garbage.
2. The heap is divided into multiple generations, with each generation containing objects of a specific age.
3. Objects are initially allocated in the youngest generation, and as they survive garbage collection cycles, they are promoted to older generations.
4. The oldest generation typically contains objects that are long-lived, such as those representing global data or program state.
5. The garbage collector runs most frequently on the youngest generation, since that is where most objects are allocated and die.
6. This is called the "young generation" or "eden space". During each garbage collection cycle, the garbage collector identifies and frees all dead objects in the young generation.

Generational GC:

1. Objects that survive a certain number of garbage collection cycles are promoted to the next older generation, called the "survivor space".
2. The survivor space is further divided into two regions, called "from" and "to", and objects that survive a garbage collection cycle are copied from the "from" space to the "to" space.
3. The survivor space is typically smaller than the young generation, since fewer objects are expected to survive.
4. The oldest generation, called the "tenured space" or "old generation", contains objects that are long-lived, such as objects representing global data or program state. The garbage collector runs less frequently on the tenured space, since fewer objects are expected to become garbage.

Generational GC:

Advantages and Disadvantages:

1. It focuses garbage collection efforts on the areas of the heap where most garbage is expected to be found. This can result in faster and more efficient garbage collection than other algorithms.
2. It requires more complex bookkeeping than other algorithms, since it needs to track objects as they move between generations.

Garbage First G1 in Java:

1. The G1 collector divides the heap into a set of equal-sized regions and uses a concurrent, multi-threaded marking algorithm to identify regions that are garbage.
2. When G1 determines that a region contains a lot of garbage, it will prioritize cleaning that region first, hence the name "Garbage First". This approach helps reduce the time needed for garbage collection, making the program run more efficiently and with lower pause times.
3. It then compacts the live objects within those regions and copies them to another part of the heap.
4. This process is repeated for multiple cycles until the garbage has been cleared from the heap.
5. By using a concurrent, incremental approach, and by prioritizing the cleaning of regions with the most garbage, G1 is able to reduce pause times and make them more predictable, improving the overall performance of Java applications
6. Additionally, G1 uses a technique called "adaptive sizing", where it adjusts the size of the heap based on the current workload of the application.

Reference Counting:

1. Reference counting garbage collection is a type of garbage collection algorithm that works by keeping track of the number of references to each object in the heap.
2. Every time a reference to an object is created or destroyed, the reference count for that object is updated. When the reference count for an object reaches zero, it means that there are no more references to that object, and it can be safely freed.

Each object contains a field called the reference count

a. During execution, the reference count for an object contains count of all the pointers pointing to the object
b. When a pointer to an object is copied, the object's reference count is incremented
c. When a pointer to an object is destroyed, the object's reference count is decremented
d. When an object's reference count becomes zero, the object can be reclaimed
e. CAF is a C++ library that provides support for reference counting

```cpp
#include <iostream>
#include <memory>
class MyClass {
public:
    MyClass() {
        std::cout << "MyClass constructor called" << std::endl;
    }
    ~MyClass() {
        std::cout << "MyClass destructor called" << std::endl;
    }
};
int main() {
    std::shared_ptr<MyClass> ptr1(new MyClass());
    std::shared_ptr<MyClass> ptr2 = ptr1;
    std::shared_ptr<MyClass> ptr3 = ptr1;
    std::cout << "Reference count: " << ptr1.use_count() << std::endl;
    ptr1.reset();
    std::cout << "Reference count: " << ptr2.use_count() << std::endl;
    ptr2.reset();
    std::cout << "Reference count: " << ptr3.use_count() << std::endl;
    ptr3.reset();
    std::cout << "End of program " << std::endl;
    return 0;
}
```

```
MyClass constructor called
Reference count: 3
Reference count: 2
Reference count: 1
MyClass destructor called
End of program
```

Advantages and disadvantages of Reference Counting GC:

1. It can immediately reclaim memory for objects that are no longer in use, without needing to wait for a garbage collection cycle.
2. It cannot handle cycles of objects that reference each other, also known as circular references or cyclic references. In this case, the reference count for each object in the cycle will never reach zero, even if there are no external references to the cycle.
3. It can be less efficient than other garbage collection algorithms for large heaps, since it requires each object to maintain a reference count, which can be expensive in terms of memory and CPU time.