9b. Stack/CPU Registers vs Heap

Martin Alfaro
PhD in Economics

INTRODUCTION

Memory allocations occur every time a new object is created. It involves setting aside a portion of the computer's Random Access Memory (RAM) to store the object's data. Alternatively, the compiler could optimize storage for extremely simple objects and directly opt to store them in CPU registers.

Conceptually, RAM is divided into two main areas: the stack and the heap. These areas aren't physical locations, but rather logical models that govern how memory is managed. When an object is created, its storage location is implicitly determined by its type. For example, collections defined as vectors are stored on the heap, whereas those defined as tuples can be allocated either on the stack or on registers, but never on the heap.

The choice between heap and non-heap allocations has significant implications for performance. Allocating memory on the heap is a comparatively expensive operation. It requires a systematic search for an available block of memory, bookkeeping to track its status, and an eventual deallocation process to reclaim the space once it's no longer needed. [1] Stack, by contrast, is simpler and therefore faster.

The performance gap between stack/registers and heap allocations can easily become a critical bottleneck when an operation is performed repeatedly. This disparity in performance explains the common convention in programming, including Julia, where **memory allocations exclusively refer to heap allocations**. In the following, we provide a brief overview of how each operates.

STACK AND CPU REGISTERS

The stack is typically reserved for holding objects with primitive types (e.g., integers, floating-point numbers, and characters) and fixed-size collections like tuples. These objects are characterized by their fixed size, precluding the possibility of dynamically growing or shrinking. Simultaneously, such constraint also makes the allocation and deallocation of memory extremely efficient. In some cases, the compiler can optimize the storage of these objects and directly store them in CPU registers.

The primary downside of the stack is its limited capacity, with CPU registers being even smaller than the stack. This makes them suitable only for objects with a few elements. Indeed, attempting to allocate more memory than the stack can accommodate will result in a "stack overflow" error, causing program termination. And, even if an object fits on the stack, allocating too many elements will significantly degrade performance. [2]

HEAP ALLOCATIONS

Common objects stored on the heap include arrays (such as vectors and matrices) and strings. Unlike registers and the stack, the heap is designed to support more complex data structures. In particular, the heap enables us to handle objects as large as the available RAM permits, with the ability to grow or shrink dynamically.

While the heap offers greater flexibility, its more complex memory management comes at the cost of slower performance. [3] Due to its overhead, the following sections will discuss strategies for reducing heap allocations. These include computational techniques that translate vectorized operations into scalar operations, as well as programming practices that favor mutation of existing objects over the creation of new ones.

FOOTNOTES

1. This deallocation process is often automated by what's known as a garbage collector.

2. There's no hard and fast rule about how many elements are "too many". Benchmarking is the only reliable way to determine the performance consequences for each particular case. As a rough guideline, objects with more than 100 elements will certainly suffer from poor performance, while those with fewer than 15 elements are likely to benefit from stack allocation.

3. Heap memory is managed by what's known as the *garbage collector*, which automatically identifies and reclaims memory no longer in use. This process, while convenient, can be computationally expensive.