Csc344 Problem Set: Memory Management / Perspectives on Rust

Task 1 - The Runtime Stack and the Heap

Memory management is a key component in computer programming. It is the allocating and deallocating of memory during program execution. For efficient memory management, it is essential to have a thorough understanding of the distinctions between two fundamental elements, the runtime stack and the heap memory,

A stack is a linear data structure used to store local variables, function calls, and return addresses. It has a last in, first out data structure that is managed by the runtime environment. Whenever a function is called, the machine reserves a portion of the stack memory for it. After the function returns, the allocated stack memory is removed, and the program continues executing from the previous stack memory. Allocating and deallocating stack memory is performed automatically.

On the other hand, heap memory is a hierarchical data structure. The heap is used for dynamic memory allocation and is not managed by the runtime environment of the program. Heap memory is managed manually by the programmer through the use of functions. The programmer can explicitly allocate and deallocate heap memory at runtime which makes it more versatile than the runtime stack. However, since the heap memory is not managed by the program automatically, this can result in memory leaks and memory related bugs.

Resources: https://www.guru99.com/stack-vs-heap.html#:~:text=Stack%20is%20a%20linear%20data,you%20to%20access%20variables%20globally

https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/

Task 2 - Explicit Memory Allocation/Deallocation vs Garbage Collection

Memory management plays a critical role in any computer system, as it can significantly impact the efficiency and performance of programs. There are two primary methods of managing memory that should be well comprehended: explicit memory allocation/deallocation and garbage collection.

Explicit memory allocation/deallocation involves the manual allocation and deallocation of memory in a programming language, such as C++ and C. This means that programmers must actively allocate memory for specific data structures using functions and release it when it is no longer needed. Although being able to manually allocate/deallocate gives the programmer precise control over how much memory is used, this process can lead to issues such as memory leaks.

Garbage collection involves automatically managing the allocation of memory and the deallocation of memory that is no longer being used. For this approach, the system checks the program's memory for objects that are no longer in use and deallocates the memory. Garbage collection decreases the risk of errors such as memory leaks and segmentation faults. Programming languages Python and Java utilize garbage collection which makes it easier to use since the programmer does not have to worry about manually allocating and deallocating memory. Not having control over how the memory is used can also be a challenge in certain situations.

Resources: https://rabingaire.medium.com/memory-management-rust-cf65c8465570

Task 3 - Rust: Memory Management

“The suggestion was that Rust allows more control over memory usage, like C++.”

“In this part, we'll discuss the notion of **ownership**. This is the main concept governing Rust's memory model. Heap memory always has **one owner**, and once that owner goes out of scope, the memory gets de-allocated. We'll see how this works; if anything, it's a bit easier than C++!”

“If you code in Java, C, or C++, this should be familiar. We declare variables within a certain scope, like a for-loop or a function definition. When that block of code ends, the variable is **out of scope**. We can no longer access it.”

“Rust works the same way. When we declare a variable within a block, we cannot access it after the block ends. (In a language like Python, this is actually not the case!)”

“What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it!”

“Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.”

“When s1 and s2 go out of scope, Rust will call drop on both of them. And they will free the same memory! This kind of "double delete" is a big problem that can crash your program and cause security problems.”

“Memory can only have one owner.”

“Like in C++, we can pass a variable by reference. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership.”

“You can only have a single mutable reference to a variable at a time”

Resources: https://mmhaskell.com/rust/memory

Task 4 - Paper Review: Secure PL Adoption and Rust

With different programming languages, such C and C++, there are numerous memory safety-related vulnerabilities that can occur. Highly dangerous violations of memory safety, such as use-after-frees, buffer overflows, and out-of-bounds reads/writes may give rise to these vulnerabilities. To combat these vulnerabilities, Rust and Go were developed to solve common and potentially devastating memory safety-related vulnerabilities.

Rust is a multi-paradigm language, with elements drawn from functional, imperative, and object-oriented languages. A survey conducted on Rust development community forums showed that participants largely perceived Rust to succeed at its goals of security and performance. Instead of garbage collection, Rust enforces a programming discipline involving ownership, borrowing, and lifetimes to avoid dangerous errors. Ownership refers to the idea that each value has a variable that serves as an owner. There can only be one owner at a time and when the owner goes out of scope, the memory can be deallocated. Borrowing refers to allowing Rust to transfer the ownership between other owners. Rust's lifetimes help prevent the scenario where a resource is deallocated while a borrowed memory is still being used. Lifetime annotations help tell the compiler that the reference is valid only within that scope.

Rust's secure memory management and the simplicity in which bug-free code can be implemented makes it a great choice for low-level systems. Rust's firm ownership, borrowing, and lifetime concepts ensure that it avoids any dangerous errors involving references, making it a strong and secure language for memory management.

Resource: https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf