Writing My Own Operating System In Rust

An Expolration Into Computer Archetecture

Teddy Maloney

June 5, 2023

1 What is an Operating System Kernel?

An operating system kernel is the central component of an operating system that serves as a core bridge between software applications and the underlying hardware of a computer system. It manages system resources, provides essential services, and facilitates communication between software and hardware.

The kernel manages various aspects of the system, including process management, memory management, and input/output operations. It schedules and coordinates the execution of processes, ensuring fair allocation of CPU time and efficient multitasking. Additionally, it manages memory by allocating and deallocating memory to processes, optimizing memory usage, and providing memory protection.

The kernel also handles input/output operations, allowing software applications to interact with peripheral devices such as disks, printers, and networks. It provides device drivers as translators between the hardware and the software, enabling communication and data transfer.

2 What Can My Kernel Do?

2.1 Memory Paging

My OS kernel uses page tables to convert virtual memory addresses to physical memory addresses. The page tables serve as a maps between the two address spaces. When a program accesses virtual memory, the kernel consults the page tables for the corresponding physical address. If a valid mapping exists, the operation is directed to the correct physical location. If not, a process called page fault handling is triggered. This involves retrieving data from secondary storage and allocating new physical memory. The page table is updated with the new mapping, enabling future accesses to be translated directly to the physical memory location. This mechanism allows efficient memory management and supports multitasking across different processes.

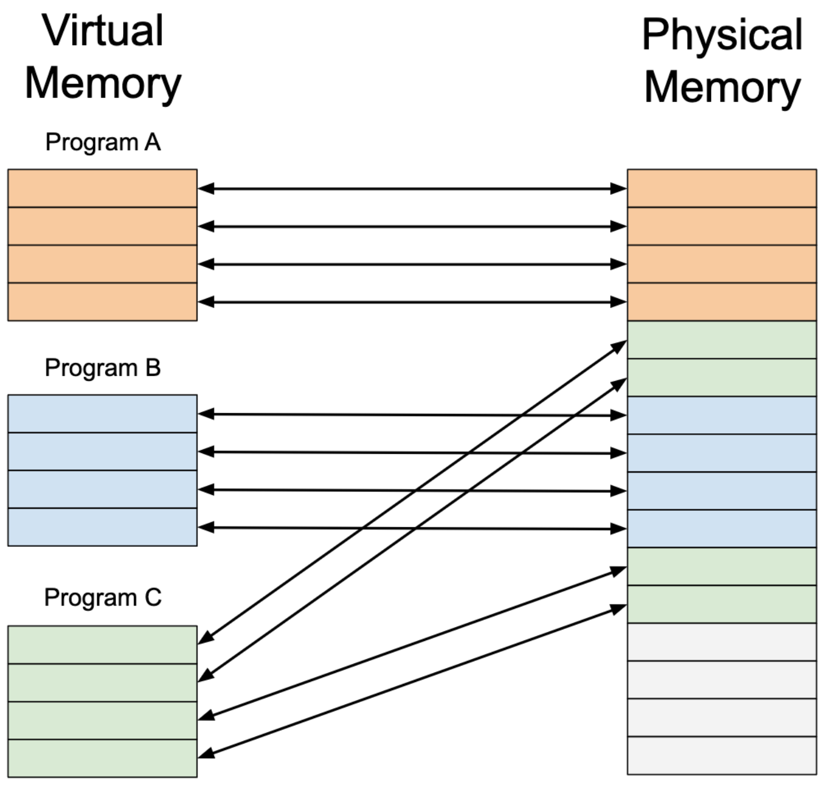


Figure 1: Translating virtual adresses to physical adresses

2.2 Printing To The VGA Buffer

A VGA (Video Graphics Array) buffer, also known as a frame buffer, is a region of memory in a computer's graphics system that stores the pixel data necessary to generate images on a display. It acts as an intermediary between the computer's software and the physical display hardware, allowing for efficient rendering and manipulation of graphical content.

To write to a VGA buffer, the software interacts with the memory addresses reserved for the frame buffer. Each memory location within the buffer corresponds to a pixel on the display. By modifying the pixel values stored in the buffer, the software can control the appearance and arrangement of visual elements on the screen.

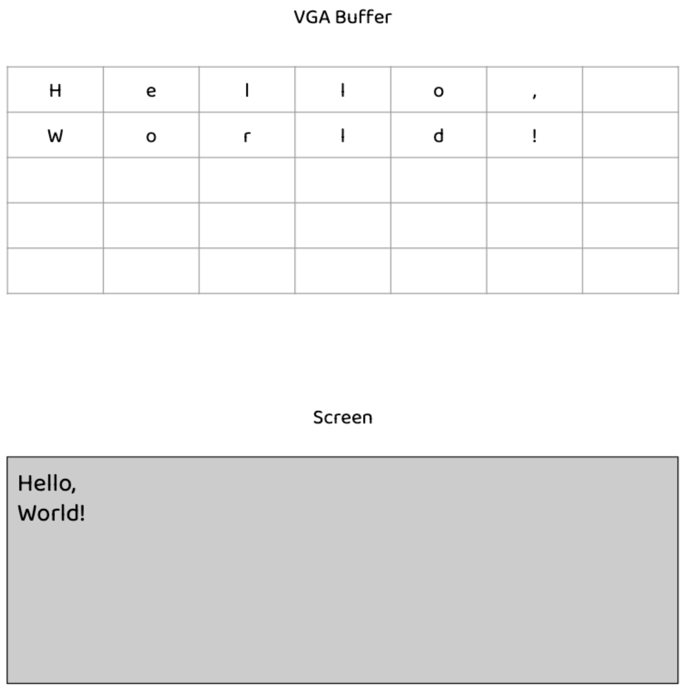


Figure 2: A comparison of the VGA buffer to a screen

We can also take peripheral inputs and print them to the screen using interrupts.

2.3 Interrups And Taking Keyboard Inputs

Interrupts are essential mechanisms in computer systems that handle immediate events, like keyboard inputs. When a key is pressed, the keyboard controller triggers a hardware interrupt to notify the CPU. The CPU pauses its current task and executes an interrupt handler specifically designed for keyboard input. The handler captures the keystrokes and prints them to the VGA buffer. The CPU then resumes its previous task or switches to another task. Interrupts enable efficient and timely management of keyboard inputs in computer systems.

2.4 Memory Management

My kernel manages dynamic memory by using a simple linked list allocator. For more information read the How Does My Kernel Manage Memory section.

3 What is Hardware and Software?

3.1 Hardware

Hardware refers to the physical components of a computer system, such as the motherboard, CPU, memory modules, storage devices, input/output devices, and peripherals. It encompasses the tangible, electronic, and mechanical parts that form the foundation of a computer system. Hardware components are responsible for executing instructions and performing calculations. They provide the necessary interfaces for communication with other hardware, firmware, and software components.

3.2 Firmware

Firmware sits between hardware and software. It is a type of software that is embedded into hardware devices or components during the manufacturing process. Firmware provides low-level control and instructions specific to the hardware it is installed on. It is typically stored in non-volatile memory, such as ROM (read-only memory) or flash memory which doesn't need constant power to be stored. Firmware controls hardware devices' basic functionalities and initialization processes, enabling them to work seamlessly with the rest of the computer system. Examples of firmware include the BIOS (**B**asic **I**nput/**O**utput **S**ystem) in a computer.

3.3 Software

Software refers to the programs, data, and instructions that run on a computer system. It is a collection of codes and algorithms written in programming languages that provide specific functionalities and services to users. Software can be classified into system software (such as operating systems and device drivers) that manages and controls the hardware resources and application software (such as word processors, web browsers, and games) that are designed for specific tasks or user requirements. Software is typically stored in non-volatile storage devices such as hard drives or solid-state drives and is loaded into the computer's memory for execution.

4 Integral Hardware Components

- **Central Processing Unit (CPU)**: The CPU is the "brain" of the computer that executes instructions, performs calculations, and manages data processing. It consists of the arithmetic logic unit (ALU), control unit, registers, and cache.
  - **Arithmetic Logic Unit (ALU)**: The ALU is responsible for performing arithmetic (such as addition, subtraction, multiplication, and division) and logical operations (such as AND, OR, and NOT) on data. It carries out calculations and comparisons for processing instructions and manipulating data within the CPU.
  - **Control Unit (CU)**: The Control Unit manages the execution of instructions within the CPU. It fetches instructions from memory, decodes them, and controls the flow of data between different CPU components. The Control Unit also coordinates the timing and sequencing of instructions, ensuring that they are executed in the correct order.
  - **Registers**: Registers are small, high-speed memory units located within the CPU. They store data and instructions that are currently being processed by the CPU. Registers provide quick access to data, allowing for faster execution of instructions. Common types of registers include the program counter (PC), which holds the memory address of the next instruction to be fetched, and general-purpose registers (like the accumulator) used for various calculations and data manipulation.

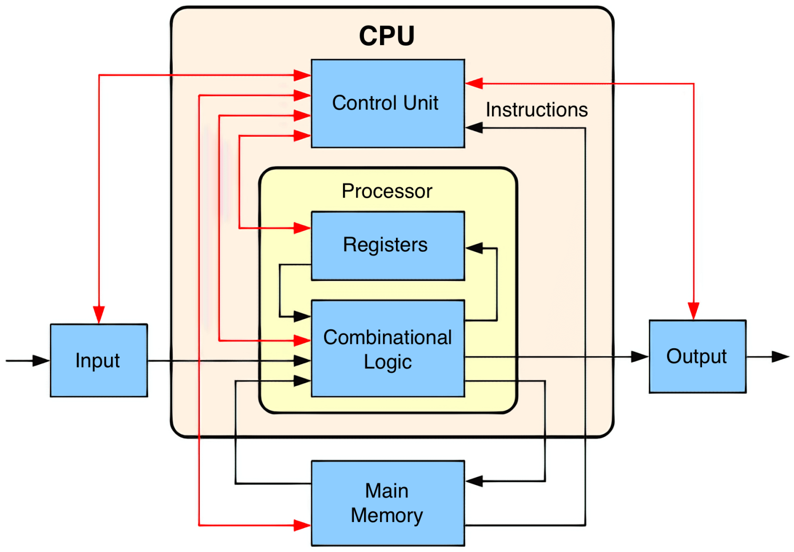


Figure 3: A diagram of the CPU

Gayde

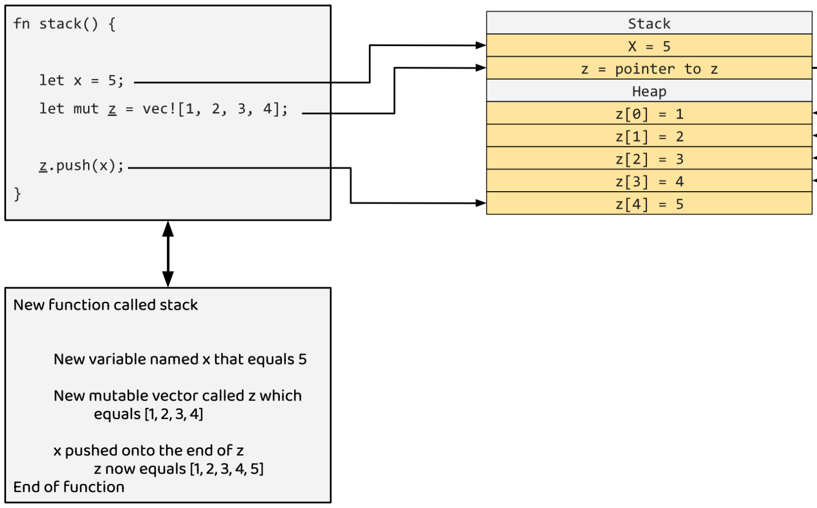
- **Random Access Memory (RAM)**: RAM is the primary memory of a computer system that stores data and instructions temporarily while the CPU is actively using them. It provides fast access to data, allowing for quick read and write operations.
- **Motherboard**: The motherboard is the main circuit board of a computer system. It connects and houses various components, including the CPU, memory modules, expansion cards, and other essential peripheral

5 What is Memory?

Memory(RAM) stores data, instructions, and program code for current or future use by the CPU (Central Processing Unit). It is a crucial part of a computer's architecture and plays a vital role in the overall performance and functionality of a system. There are two different ways memory is stored on your computer.

1. Stack: The stack is a region of memory used for storing local variables and function call information. It is managed by the CPU and operates in a Last-In, First-Out (LIFO) manner. When a function is called, its local variables and return address are pushed onto the stack, creating a new stack frame. As functions complete, their stack frames are removed from the top of the stack. The stack is typically faster to access and has a fixed size determined during program compilation.

2. Heap: The heap is a region of memory used for dynamic memory allocation. It is a pool of memory where objects and data can be allocated and deallocated as needed. Unlike the stack, memory allocation in the heap is not always managed automatically. Developers sometimes explicitly request memory from the heap using functions like ‘alloc()‘ and are responsible for freeing it when no longer needed using ‘dealloc()‘. Some languages, like java, have a built in heap manager called garbage collection that will automatically free memory when it is no longer needed. The heap provides greater flexibility in memory management but can be slower to access and requires manual memory management to avoid memory leaks and fragmentation.



```
fn stack() {

    let x = 5;
    let mut z = vec![1, 2, 3, 4];

    z.push(x);
}
```

New function called stack

New variable named x that equals 5

New mutable vector called z which equals [1, 2, 3, 4]

x pushed onto the end of z
z now equals [1, 2, 3, 4, 5]

End of function

Figure 4: A Diagram of the stack and heap

This previous code can be explained by the following sequence:

Function Entry: When the stack() function is called, the function’s stack frame is created. The stack frame includes space for storing local variables, function arguments, return addresses, and other bookkeeping information. The stack frame is typically limited in size and is managed automatically by the program’s execution environment.

Stack Variables: Inside the stack() function, two variables are declared: x and z. The x variable is a simple i32 type and will be allocated on the stack. It will consume a fixed amount of memory to store the value 5.

Vector z: The z variable is a Vec¡i32¿ type, which is a dynamically sized container. It will allocate memory on both the stack and the heap. On the stack, the Vec struct itself will be stored, which contains metadata about the vector, such as its length and capacity. On the heap, the actual elements of the vector will be stored. In this case, it will initially contain [1, 2, 3, 4].

Push Operation: The z.push(x) operation will add the value of x (which is 5) to the end of the vector. If the vector's capacity is not sufficient to hold the new element, it will trigger a reallocation on the heap to allocate additional memory. The existing elements of the vector will be copied to the newly allocated memory, and the new element (5) will be added at the end.

Function Exit: When the stack() function finishes executing, the stack frame will be deallocated, and the memory associated with local variables (x and z) will be freed. The vector z will deallocate its heap memory automatically, as it follows Rust's ownership and borrowing rules.

6 How Does My Kernel Manage Memory?

Dynamic memory is allocated at runtime for storing data. An allocator manages by tracking available space, and allocating memory blocks. It allows programs to handle varying data requirements and create dynamic data structures.

6.1 Bump Allocator:

The bump allocator is an efficient method for managing memory. It works by creating a pointer called *next*, which points to the beginning of unused memory. When a new block of memory is needed, the allocator moves *next* to the new beginning of the unused memory as seen in Figure 5.

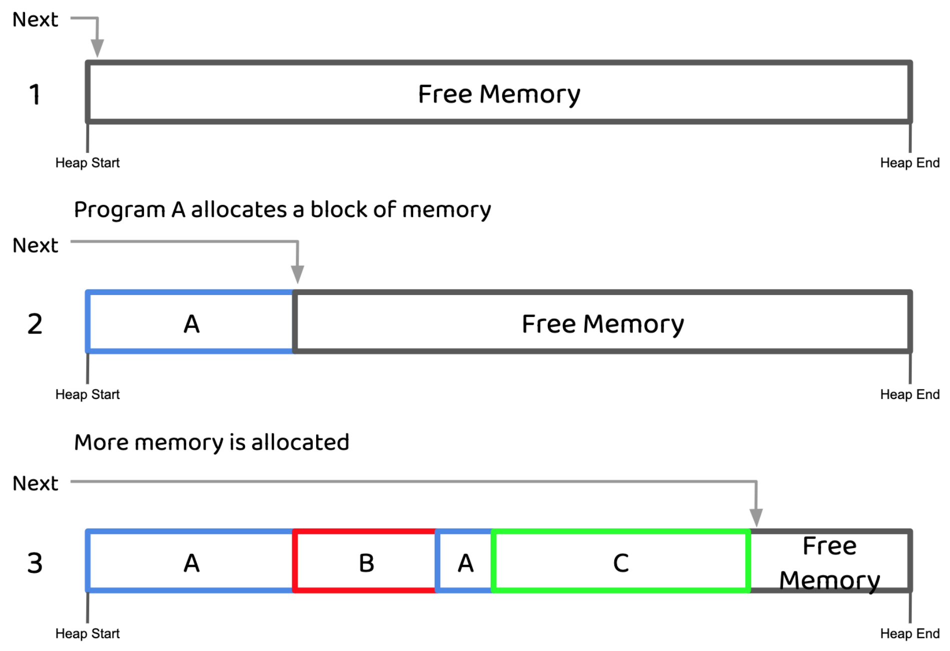


Figure 5: Bump Allocation

Memory can also be deallocated just by freeing that chunk of memory without changing the *next* pointer as see in Figure 6.

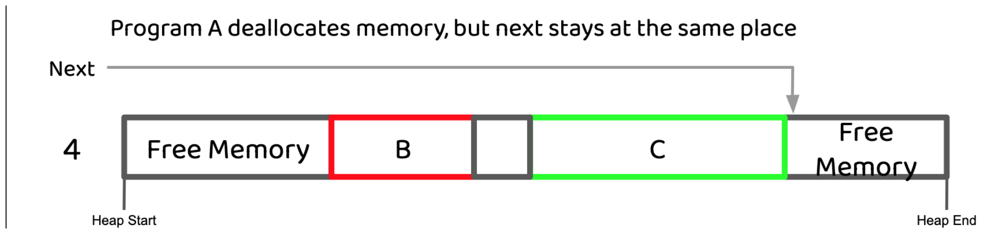

Program A deallocates memory, but next stays at the same place
Next
4
Free Memory
B
C
Free Memory
Heap Start
Heap End

Figure 6: Bump deallocation

However, there are limitations to this approach. Since the *next* pointer only moves forward, it cannot reuse memory that has been deallocated. This means that when the allocator reaches the end of the available memory in the heap, it runs out of space to allocate any more memory as seen in Figure 7. As a result, the following allocation request will result in an out-of-memory error. This approach can also lead to fragmentation that can make this memory allocation algorithm less efficient.

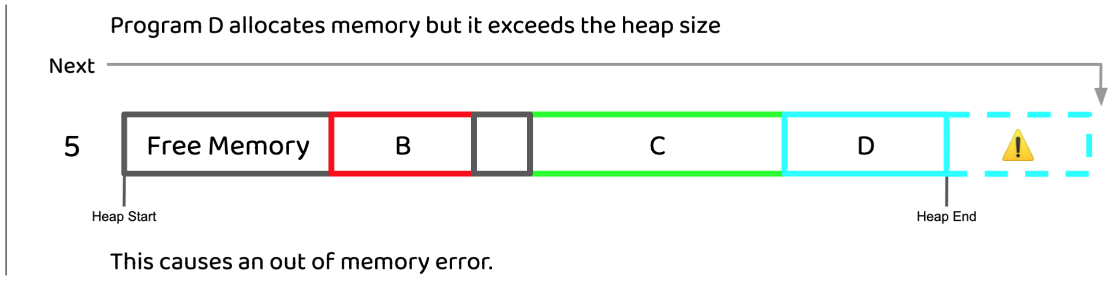

Program D allocates memory but it exceeds the heap size
Next
5
Free Memory
B
C
D
Heap Start
Heap End
This causes an out of memory error.

Figure 7: Out Of Memory Error

To mitigate this issue, an allocation counter can be implemented. This counter increases by one when memory is allocated and decreases by one when memory is deallocated. This allows the bump allocator to reset back to the beginning of the available memory when the counter reaches zero. However, there is still a possibility of encountering an out-of-memory error if the allocation counter is greater than one when it reaches the end of the available memory.

To fix some of these problems I used a Linked List Allocator Instead.

6.2 Linked List Allocator:

A simple linked list allocator is similar to a bump allocator. It uses a linked list, which is a chain of nodes, to allocate and deallocate memory as needed.

Initially, the entire memory block is treated as a single free block, as seen in Figure 8, and a linked list node is created to represent it. This node contains information regarding the block's size and a reference that points to the next node in the list.

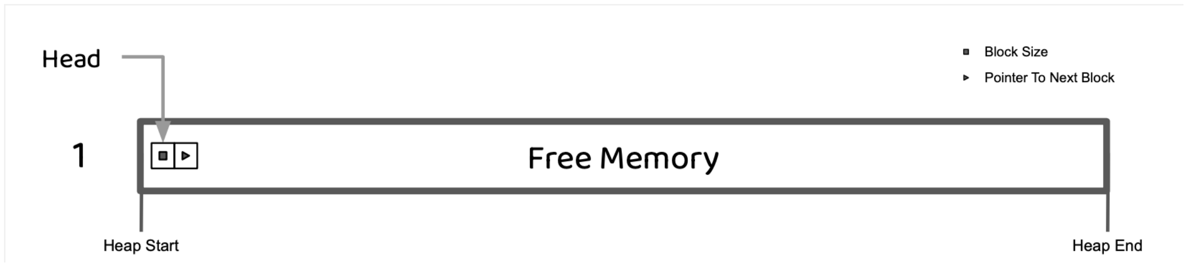


Figure 8: Empty Linked List

Upon receiving a memory allocation request, the allocator traverses the linked list, starting at the head node and following the references to the other nodes, to locate a suitable free block that can accommodate the requested size. Once found, the allocator divides the free block into two segments: one for the allocated memory and another for the remaining unused memory. The allocated block is marked as "in-use," while the unused block is transformed into a new free block by updating it with its own linked list node.

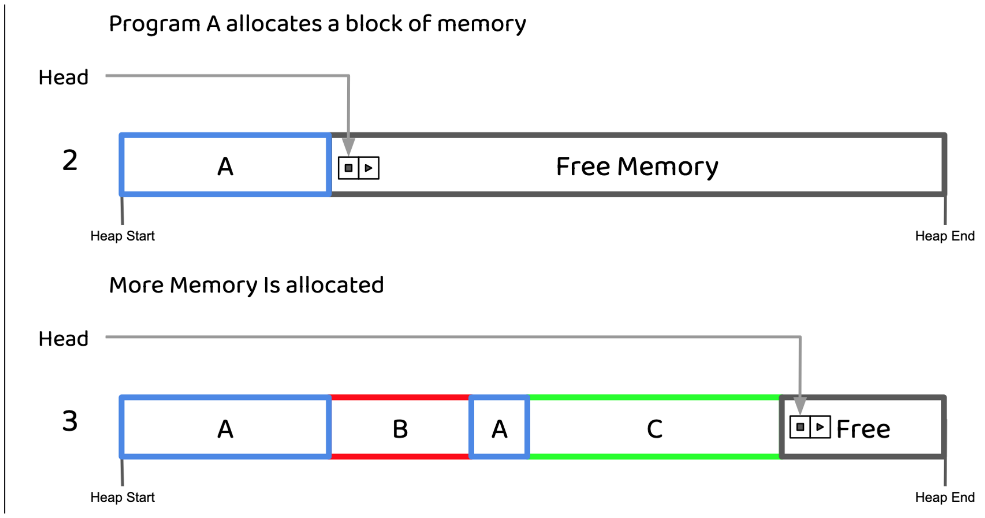


Figure 9: Linked List Allocation

When memory is no longer needed and is deallocated, the allocator marks that block as free again. It then creates a new node that the head points towards. Then, the new node points to the previous node the head was pointing towards to complete the list as shown below:

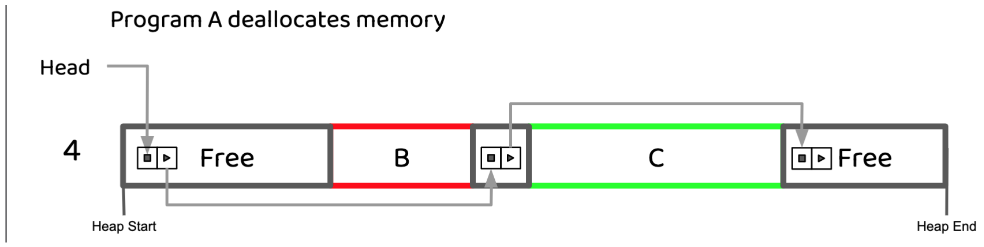


Figure 10: Linked List Allocation

While a linked list allocator offers flexibility in managing memory allocation and deallocation, it may encounter fragmentation issues over time if memory is allocated and deallocated irregularly, leading to a fragmented linked list of free blocks like shown:

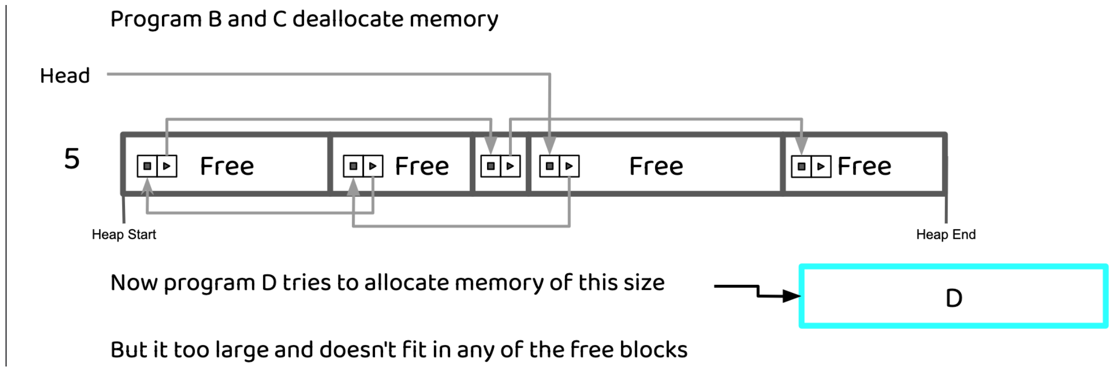

Program B and C deallocate memory
Head
5
Free
Free
Free
Free
Heap Start
Heap End
Now program D tries to allocate memory of this size
D
But it too large and doesn't fit in any of the free blocks

Figure 11: Fragmentation In Linked Lists

If the linked list is fragmented, whenever a program tries to allocate a single block of memory larger than any of the fragmented blocks, it results in an out of memory error because it cannot allocate any memory.

To fix this issue, a compactor can be implemented. The compactor is responsible for rearranging the linked list of free memory blocks by consolidating consecutive blocks into larger contiguous blocks. This helps to reduce fragmentation and maximize the utilization of available memory. By compacting the blocks, the allocator can ensure that larger chunks of continuous memory are available for allocation, improving overall efficiency.

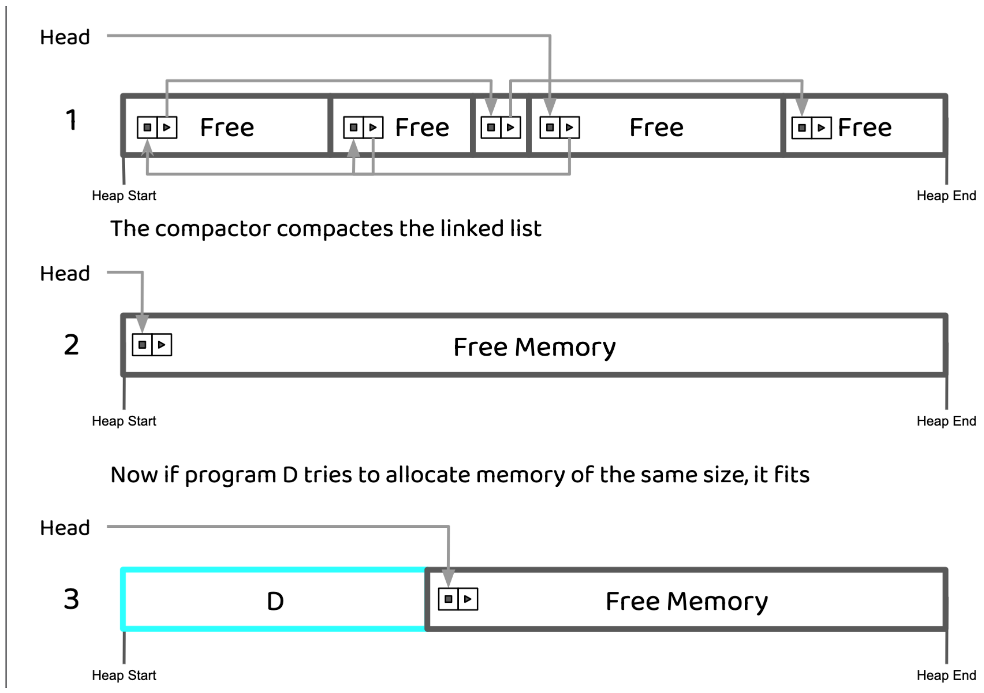


Figure 12: Linked List Compaction

Unfortunately this is a complicated algorithm to implement, so my linked list currently cannot do this. But it is something I would like to implement in the future.

A a simple linked list allocator provides a fundamental yet effective approach to dynamically manage memory allocation. It strikes a balance between simplicity and memory utilization efficiency, enabling variable-sized memory requests to be handled proficiently.

7 Current Limitations of My Kernel

My kernel has many limitations that I will improve on in the future, the most prominent being:

- Cannot compile programs
- Fragmentable heap management
- Lack of features

8 My Expert

My brother, Jack Maloney, who is a skilled software engineer, played a crucial role in this project. He helped me find and fix problems, add new features, and understand complex concepts about computer architecture. I am really grateful that he was there to support me throughout the project, and his contributions were a major factor in its success.

Works Cited

Gayde, William. “CPU Diagram”, 2020.