Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

Preface

These notes are based on the lecture slides from the course:
BU CS330: Introduction to Analysis of Algorithms

Presented by:
Dora Erdos, Adam Smith
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

— 1 —

Memory Management

1.1 Stacks and Heaps

To further understand how our algorithms run, we must understand ligtly how memory is passed around in our programs.

Definition 1.1: Machine Code & Compilation

Code is separated into two main areas of memory management, the program itself, and the data in transit during execution. The program itself is broken up such as follows:

- **Text Segment:** The part of the program which contains the executable code.
- **Data Segment:** The part of the program which contains global and static variables.
- **Machine Code:** The compiled code of the program, which is executed by the CPU.

Once the code compiles, we our data segment is further divided into two parts in memory:

- **Initialized Data:** This is data that has been given a value before the program starts running, such as global variables.
- **Uninitialized Data:** This is data that has not been given a value before the program starts running, such as local variables.

By memory we mean the **RAM (Random Access Memory)** hardware component, which stores temporary data, constantly communicating with the CPU or external storage (e.g., hard drive, SSD).

Let's talk about our first data structure, the stack:

Definition 1.2: Stack

A **stack data structure** is a collection of elements that follows a **Last In, First Out** (LIFO) principle. I.e., in a stack of plates, the last added plate is the first one to be removed, not the middle or bottom/first plate. Each *plate* in the stack is called a **stack frame**.

This text does not concern assembly code or low-level programmming, so **do not** get caught up on the specifics of this next example:

Example 1.1: Assembly Code

Here is an assembly code example that demonstrates both initialized and uninitialized data. Initialized data is placed in the '.data' section, while uninitialized data is placed in the '.bss' section:

```
section .data                   ; Initialized data section
    num1    dd  7               ; num1 is initialized to 7
    num2    dd  3               ; num2 is initialized to 3

section .bss                    ; Uninitialized data section
    temp    resd 1              ; temp is reserved (uninitialized)
    result  resd 1              ; result is reserved (uninitialized)

section .text                   ; Code section
    global _start

_start:
    mov eax, [num1]             ; Load num1 into eax
    mov [temp], eax             ; Store num1 in temp
    mov ebx, [num2]             ; Load num2 into ebx
    add eax, ebx                ; Add num2 to eax (eax = num1 + num2)
    mov [result], eax           ; Store the sum in result
; (Exit code omitted for brevity)
```

In this example, 'num1' and 'num2' are initialized before execution, while 'temp' and 'result' are uninitialized and only receive values during program execution. ■

Now let's look at how our programs utilize the stack:

Definition 1.3: Call Stack

A **call stack** is a stack which keeps track of function calls in a program as well as any local variables within such functions.

This is why we say a variable is in **scope**, as when a function is taken off the stack, or a new stack frame is placed ontop, the variables in the previous or discarded stack frame are **no longer accessible**.

Let's illstrate this with the following diagram:

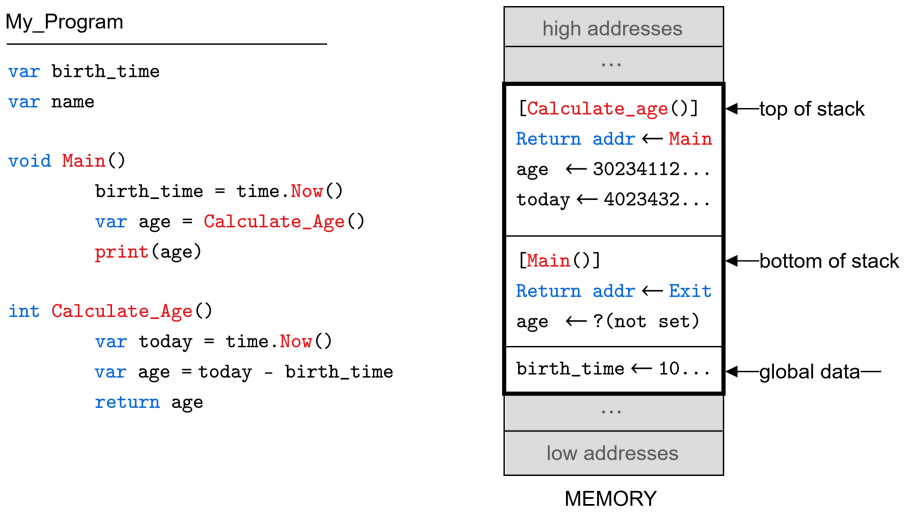

Figure 1.1: Here is a simplified look at how memory manages the stack. On the left is our program written in some abstract language, and on the right is the call stack in memory (simplified). The program has a global 'birth_time' variable, which is intialized in the `Main` function. The `Main` function then calls the `Calculate_Age` function which uses the 'birth_time' variable to calculate the 'age' via the difference of the current time and the 'birth_time.' Lookin at the memory, we see at the bottom of our memory contains global variables accessible to any frame. Next, is the bottom of the stack, containing a return address to exit the program, while awaiting the result of the function call for 'age'. The top of our stack contains another frame that we will return the value a new 'age' (not the same as the one before) not accessible from the main function. This new frame also contains a new local variable 'today.' Once this function returns, `Main` will have the result of its local variable 'age.'. Concretely, the 'age' variable in both the `Main` and `Calculate_Age` function are completely separate despite sharing the same *name*.

Please Note: The above figure is a simplified version; This presentation derivatives from what actually happens for teaching sake. In the following pages we define the stack frame in more detail.

Tip: A lot of demonstrations (including this text) will show the stack growing **upwards**; This is strictly because it's easier to visualize and does not accurately potray what a stack really does or looks like. In the following pages we will clear this up, and show how the stack actually grows from top-to-bottom. Of course, there is always room for deviation if a developer wishes to implement a stack in some other arbitrary way. Nonetheless, the following is what one might typically expect in a stack implementation.

Definition 1.4: Instructions

The CPU register (IP/EIP) is the **instruction pointer**, pointing to the next opreation to execute. All commands are baked into the CPU; This includes the **ALU (Arithmetic Logic Unit)**, **Memory Unit (load/store)**, and **Control Unit (branching/looping)**. All instructions are given a numeric ID called an **opcode** (operation code).

The CPU fetches the instruction from memory, decodes it, and executes it. This process is repeated until the program terminates. Languages like assembly interface this with human-readable mnemonics, such as 'MOV', 'ADD', 'SUB', etc (as seen in Example 1.1).

Definition 1.5: Stack Frame Anatomy

Under the x86-32 calling convention Two registers keep track our place in the stack:

- **Base Pointer (BP/EBP):** Points to the base (i.e. "bottom") of the current function's stack frame.
- **Stack Pointer (SP/ESP):** Points to the "top" of the current function's stack frame, i.e., the next free byte where a push would land.

When the program starts, the operating system *reserves* a contiguous region of memory for the stack. By convention, the *bottom* of that region lies at a higher address, and the stack "grows downward" toward lower addresses as data is pushed. If the stack pointer ever moves past the reserved limit—a **stack overflow** occurs.

A single **stack frame** itself is a contiguous block of memory in which the function stores:

- **Parameters:** The arguments passed in by the caller,
- **Return Address:** The address of the next instruction to execute after the function returns,
- **Old Base Pointer:** The caller's 'EBP', saved so that on return we can restore the previous frame,
- **Local Variables:** Space for any locals or temporaries that the function needs.

This is why variables in previous or new functions calls become "**out of scope**" (no longer accessible), as they belong to some other stack frame; When it comes to **Global Variables**, they live in a separate region of memory, defined by the **data segment** (1.1).

Moreover, a call to a new function invokes the **call** instruction, this automically pushes the return address to the current frame onto the stack. Additonally, the CPU reserves the **EAX** register for the return value (number or address) of a function. When the function returns, it can place its result in 'EAX', and the caller can retrieve it from there. During constant use the 'EAX' register may contain **garbage** data from previous use, unless explicitly set to zero or some other value.

High Addresses		
Contents	**Offset**	**Notes**
(Parameters 3, 4, ...)	$EBP + 16, +20, \ldots$	Third-and-onward arguments, if any.
Parameter 2	$EBP + 12$	Second argument passed on stack.
Parameter 1	$EBP + 8$	First argument passed on stack.
Return Address	$EBP + 4$	Auto-pushed by the `call` instruction.
Old EBP (Saved BP)	$EBP + 0$	The caller's base pointer
Current Frame (locals/temporaries)		
Local Variable 1	$EBP - 4$	First 4-byte local (or smallest slot).
Local Variable 2	$EBP - 8$	Next 4-byte local or part of a larger object.
...	⋮	(additional locals at $EBP - 12, -16, \ldots$)
Low Addresses		

Table 1.1: Typical x86-32 Stack-Frame Layout, where offsets are typically a multiple of 4 bytes.

My_Program

```
void Main()
   var birth_time = time.Now()
   var age = Calculate_Age(birth_time)
   print(age)

int Calculate_Age()
   var today = time.Now()
   var age = today - birth_time
   return age
```

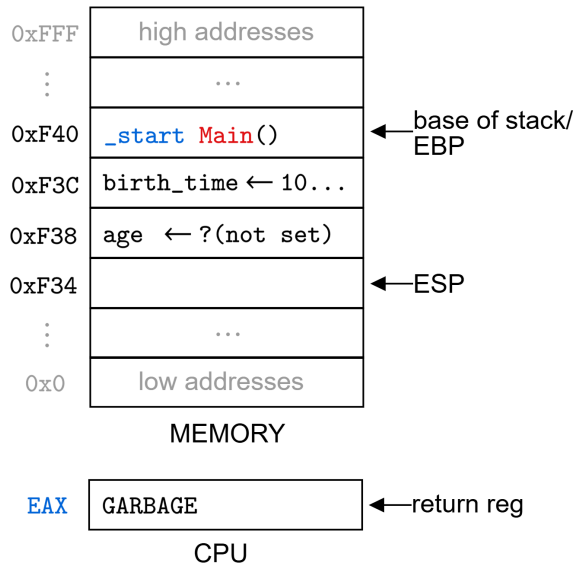

Figure 1.2: Revisiting Figure (1.1) with slight alterations to the code: This is a snapshot of the code executing right before `CalculateAge(birth_time)` is called. For simplicity sake, let's say the stack begins at address `0xF40` (Hexadecimal), growing downwards. Here the base of the stack and the EBP are one and the same. We include the CPU's EAX (return register), which contains garbage. Address `0xF38` is currently just reserved space for 'age'.

My_Program

```
void Main()
    var birth_time = time.Now()
    var age = Calculate_Age(birth_time)
    print(age)

int Calculate_Age()
    var today = time.Now()
    var age = today - birth_time
    return age
```

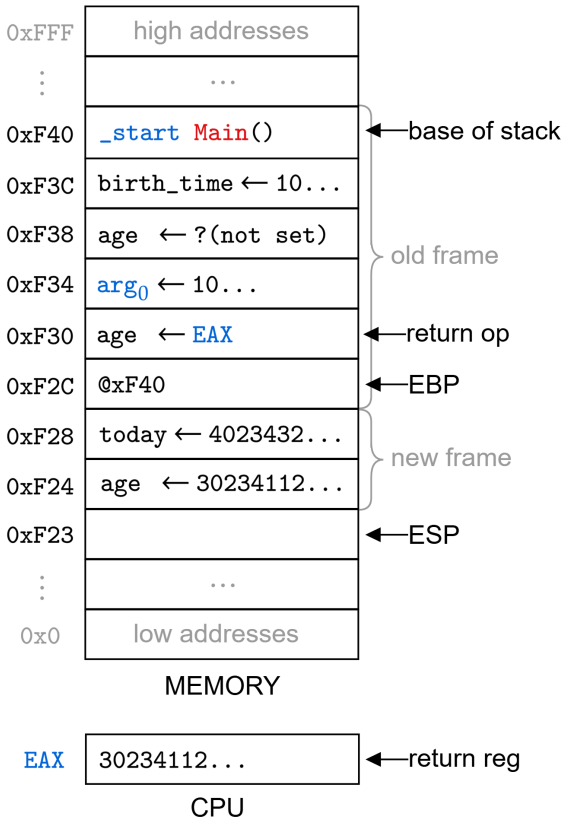

Figure 1.3: Revisiting Figure (1.2) at the moment the function `CalculateAge(birth_time)` has supplied its return value to the EAX register, and is about to return. We see that before calling `CalculateAge(birth_time)`: The old frame pushed it's arguments (`birth_time`) onto the stack, then the return address (IP/Next Instruction) onto the stack, and finally the old EBP (Base Pointer) onto the stack. The 'new frame' then sets the saved EBP address to the current EBP, concluding the old frame into the 'new frame'. Moreover, since the offset looks for local variables below `0xF40`, the above 'birth_time' and 'age' are **out of scope** for the 'new frame', vice-versa. **Note:** This is still a high-level absrtraction of what actually happens squentially with opcodes; Nonetheless, this is the fundemental idea of how a stack works.

This concludes our discussion on stack structures; We continue with the heap structure next.

Bibliography