libasm sources

Sadettin Fidan

July 10, 2024

1 Introduction

I have implemented a library in assembly language. The resources I used will be mentioned and presented in this document.

2 Functions in the library:

- ft_strlen (man 3 strlen)
- ft_strcpy (man 3 strcpy)
- ft_strcmp (man 3 strcmp)
- ft_write (man 2 write)
- ft_read (man 2 read)
- ft_strdup (man 3 strdup, you can call to malloc)

3 Bonus Functions in the library

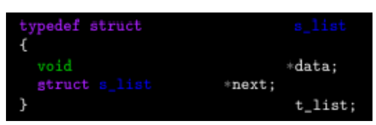


Figure 1: Struct list

- ft_atoi_base (like the one in the piscine)
- ft_list_push_front (like the one in the piscine)
- ft_list_size (like the one in the piscine)
- ft_list_sort (like the one in the piscine)
- ft_list_remove_if (like the one in the piscine)

4 Resources

4.1 General purpose Registers for Intel

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b (r8l)
r9	r9d	r9w	r9b (r9l)
r10	r10d	r10w	r10b (r10l)
r11	r11d	r11w	r11b (r11l)
r12	r12d	r12w	r12b (r12l)
r13	r13d	r13w	r13b (r13l)
r14	r14d	r14w	r14b (r14l)
r15	r15d	r15w	r15b (r15l)
r16 (with APX)	r16d	r16w	r16b (r16l)
r17 (with APX)	r17d	r17w	r17b (r17l)
...	...	...	...
r31 (with APX)	r31d	r31w	r31b (r31l)

Figure 2: General purpose Registers for Intel

You can find the image in 2 at: `https://stackoverflow.com/questions/20637569/assembly-registers-in-64-bit-architecture`

4.2 Virtual Memory

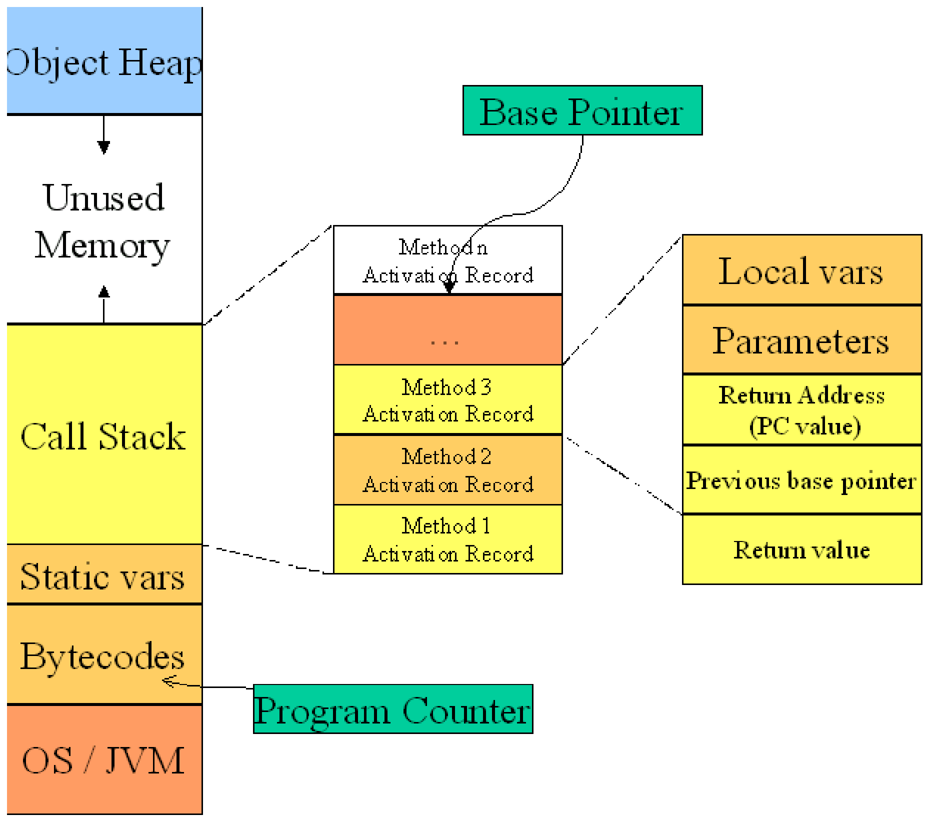


Figure 3: Virtual Memory

Using the figure 3 as a reference, I used local variables inside functions. I get it from `https://learn.microsoft.com/en-us/cpp/build/stack-usage?view=msvc-170`

4.3 x64 Intel Stack

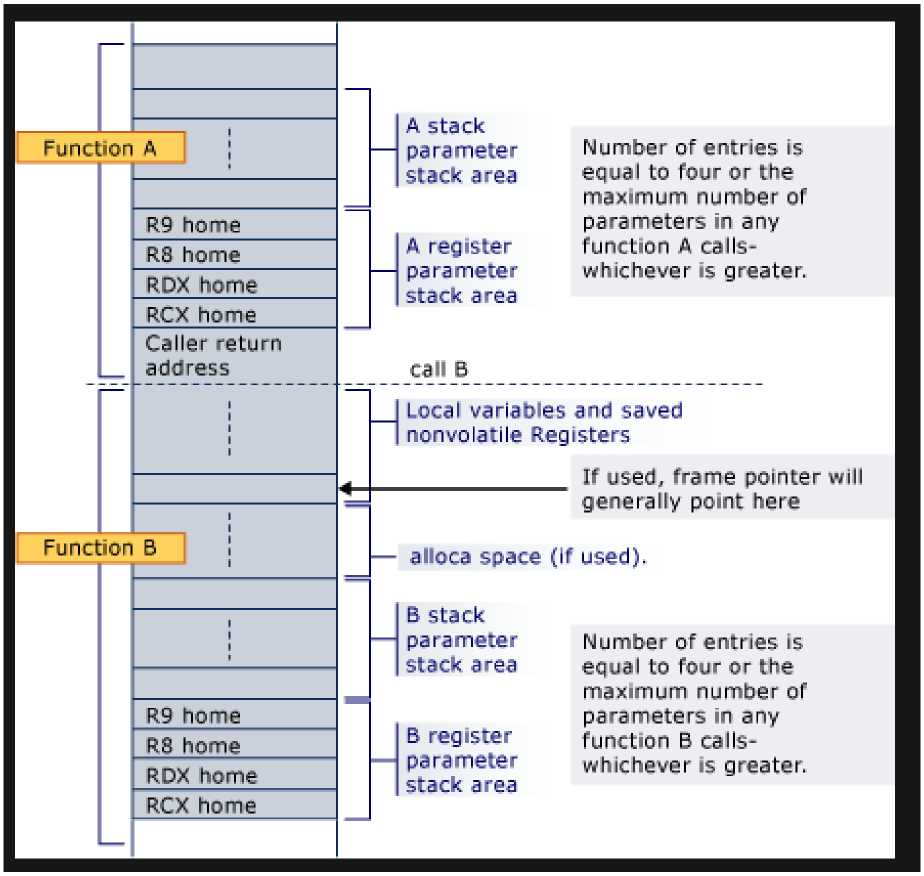
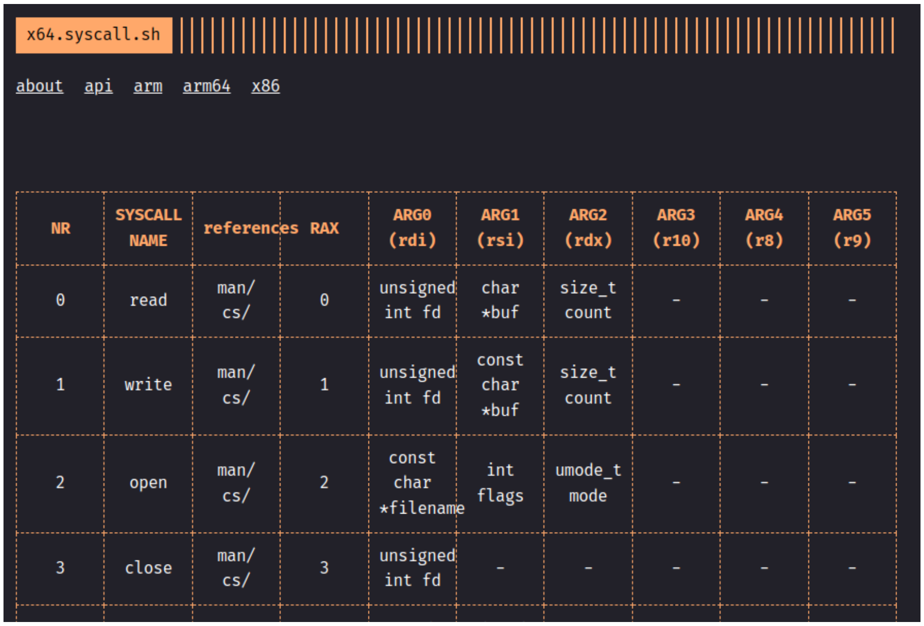


Figure 4: Stack in x64 Intel, as Function A and B

I used the figure 4 as a reference for the stack in x64 Intel. I get it from `https://en.wikipedia.org/wiki/X86_calling_conventions`

4.4 System Calls



x64.syscall.sh

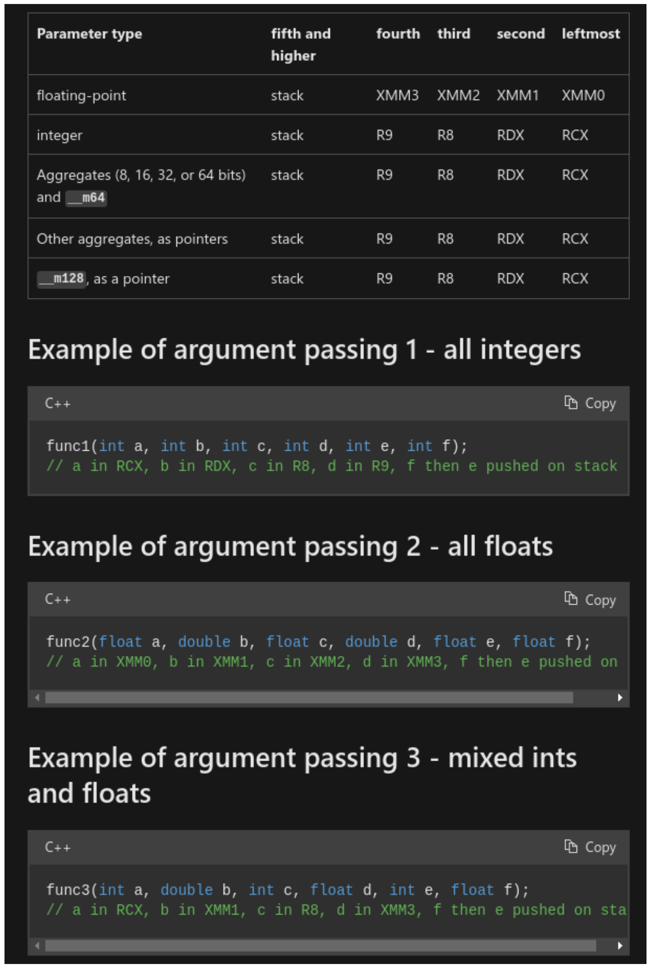
about api arm arm64 x86

NR	SYSCALL NAME	references	RAX	ARG0 (rdi)	ARG1 (rsi)	ARG2 (rdx)	ARG3 (r10)	ARG4 (r8)	ARG5 (r9)
0	read	man/ cs/	0	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	1	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	2	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	3	unsigned int fd	-	-	-	-	-

Figure 5: System Calls

I used the figure 5 as a reference for system calls. I get it from `https://x64.syscall.sh/`

4.5 Calling Convention



Parameter type	fifth and higher	fourth	third	second	leftmost
floating-point	stack	XMM3	XMM2	XMM1	XMM0
integer	stack	R9	R8	RDX	RCX
Aggregates (8, 16, 32, or 64 bits) and `__m64`	stack	R9	R8	RDX	RCX
Other aggregates, as pointers	stack	R9	R8	RDX	RCX
`__m128`, as a pointer	stack	R9	R8	RDX	RCX

Example of argument passing 1 - all integers

C++ Copy

```cpp
func1(int a, int b, int c, int d, int e, int f);
// a in RCX, b in RDX, c in R8, d in R9, f then e pushed on stack
```

Example of argument passing 2 - all floats

C++ Copy

```cpp
func2(float a, double b, float c, double d, float e, float f);
// a in XMM0, b in XMM1, c in XMM2, d in XMM3, f then e pushed on
```

Example of argument passing 3 - mixed ints and floats

C++ Copy

```cpp
func3(int a, double b, int c, float d, int e, float f);
// a in RCX, b in XMM1, c in R8, d in XMM3, f then e pushed on sta
```

Figure 6: x64 Intel calling convention

I used the figure 6 as a reference for calling convention. I get it from `https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170`

4.6 Assembly Code

"Hello world!" program for 64-bit mode Linux in NASM style assembly [edit]

This example is in modern 64-bit mode.

```
;  build: nasm -f elf64 -F dwarf hello.asm
;  link:  ld -o hello hello.o

DEFAULT REL             ; use RIP-relative addressing modes by default, so [foo] = [rel foo]

SECTION .rodata         ; read-only data should go in the .rodata section on GNU/Linux, like .rdata on
Windows
Hello:      db "Hello world!", 10   ; Ending with a byte 10 = newline (ASCII LF)
len_Hello:  equ $-Hello             ; Get NASM to calculate the length as an assembly-time constant
                                    ; the '$' symbol means 'here'. write() takes a length so that
                                    ; a zero-terminated C-style string isn't needed.
                                    ; It would be for C puts()

SECTION .rodata         ; read-only data can go in the .rodata section on GNU/Linux, like .rdata on Windows
Hello:      db "Hello world!",10        ; 10 = `\n`.
len_Hello:  equ $-Hello                 ; get NASM to calculate the length as an assemble-time constant
;;  write() takes a length so a 0-terminated C-style string isn't needed. It would be for puts

SECTION .text

global _start
_start:
    mov eax, 1              ; __NR_write syscall number from Linux asm/unistd_64.h (x86_64)
    mov edi, 1              ; int fd = STDOUT_FILENO
    lea rsi, [rel Hello]            ; x86-64 uses RIP-relative LEA to put static addresses into regs
    mov rdx, len_Hello      ; size_t count = len_Hello
    syscall                 ; write(1, Hello, len_Hello);  call into the kernel to actually do the system
call
     ;; return value in RAX.  RCX and R11 are also overwritten by syscall

    mov eax, 60             ; __NR_exit call number (x86_64) is stored in register eax.
    xor edi, edi            ; This zeros edi and also rdi.
                            ; This xor-self trick is the preferred common idiom for zeroing
                            ; a register, and is always by far the fastest method.
                            ; When a 32-bit value is stored into eg edx, the high bits 63:32 are
                            ; automatically zeroed too in every case. This saves you having to set
                            ; the bits with an extra instruction, as this is a case very commonly
                            ; needed, for an entire 64-bit register to be filled with a 32-bit value.
                            ; This sets our routine's exit status = 0 (exit normally)
    syscall                 ; _exit(0)
```

Figure 7: Assembly Code

I used the figure 7 as a reference for assembly code. I get it from `https://en.wikipedia.org/wiki/X86_assembly_language`

4.7 Debugging with GDB

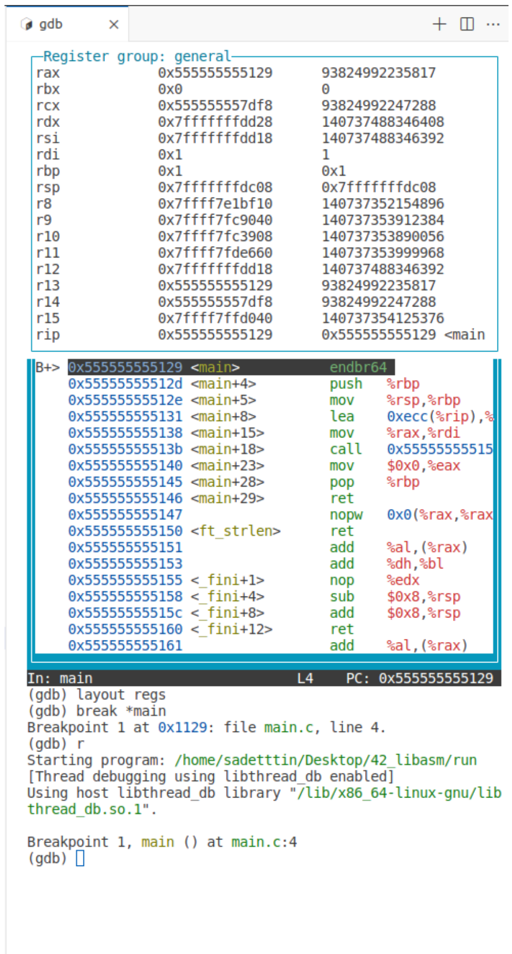


Figure 8: Debugging Assembly Code with GDB

I used the figure 8 as a reference for debugging assembly code with GDB.

I run the following commands to debug the assembly code:

```
make run
gdb ./run
(gdb) layout asm
(gdb) layout regs
(gdb) break *main
(gdb) run
```

5 Tricks

5.1 argument passing in x64 Intel

Figure 9: Argument passing for ft_strcpy

I noticed that when compiled with gcc (gcc main.c + libasm.a, libasm.a was compiled using nasm already) the first argument is passed in both rdx and rsi registers, and the second argument is passed in both rdi and rsi registers. I used the advantage this trick in my implementation by moving a register value and then substraction with fixed valued register and ta da.

5.2 debuggin malloc

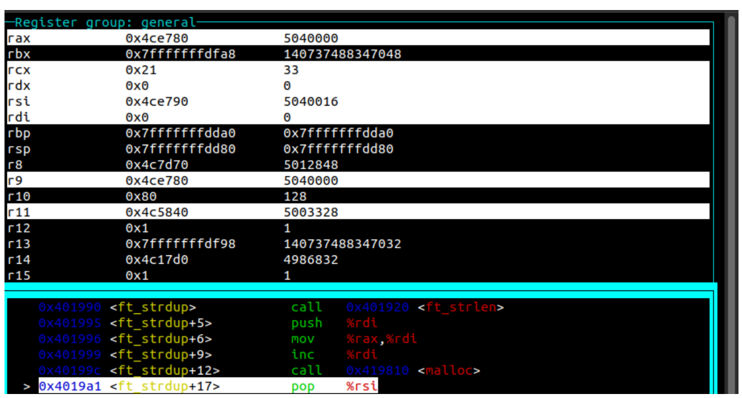


Figure 10: Registers affected by malloc

5.3 leave instruction

```
push rbp
mov rbp, rsp
sub rsp, 0x10
mov [rbp - 0x8], rdi
mov [rbp - 0x10], rsi
```

Figure 11: Leave instruction reverses this

I used the leave instruction in the end of the function to reverse the stored up memory. The stored up memory is allocated for the local variables in the function as seen in the figure 11.

5.4 offset calculation

```
[750]                   ; displacement only
[rbp]                   ; base register only
[rcx + rsi*4]           ; base + index * scale
[rbp + rdx]             ; scale is 1
[rbx - 8]               ; displacement is -8
[rax + rdi*8 + 500]     ; all four components
[rbx + counter]         ; uses the address of the variable 'counter' as the displacement
```

Figure 12: Offset calculation