OS2-MINI PROJECT

Aissaoui Abdelhadi Amine

COMPLETE
WEBSITE SECURITY

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse molestie massa ac odio euismod, sit amet rutrum sapien dignissim Nulla in sollicitudin

part 1

Introduction & Overview of the Code

part 2

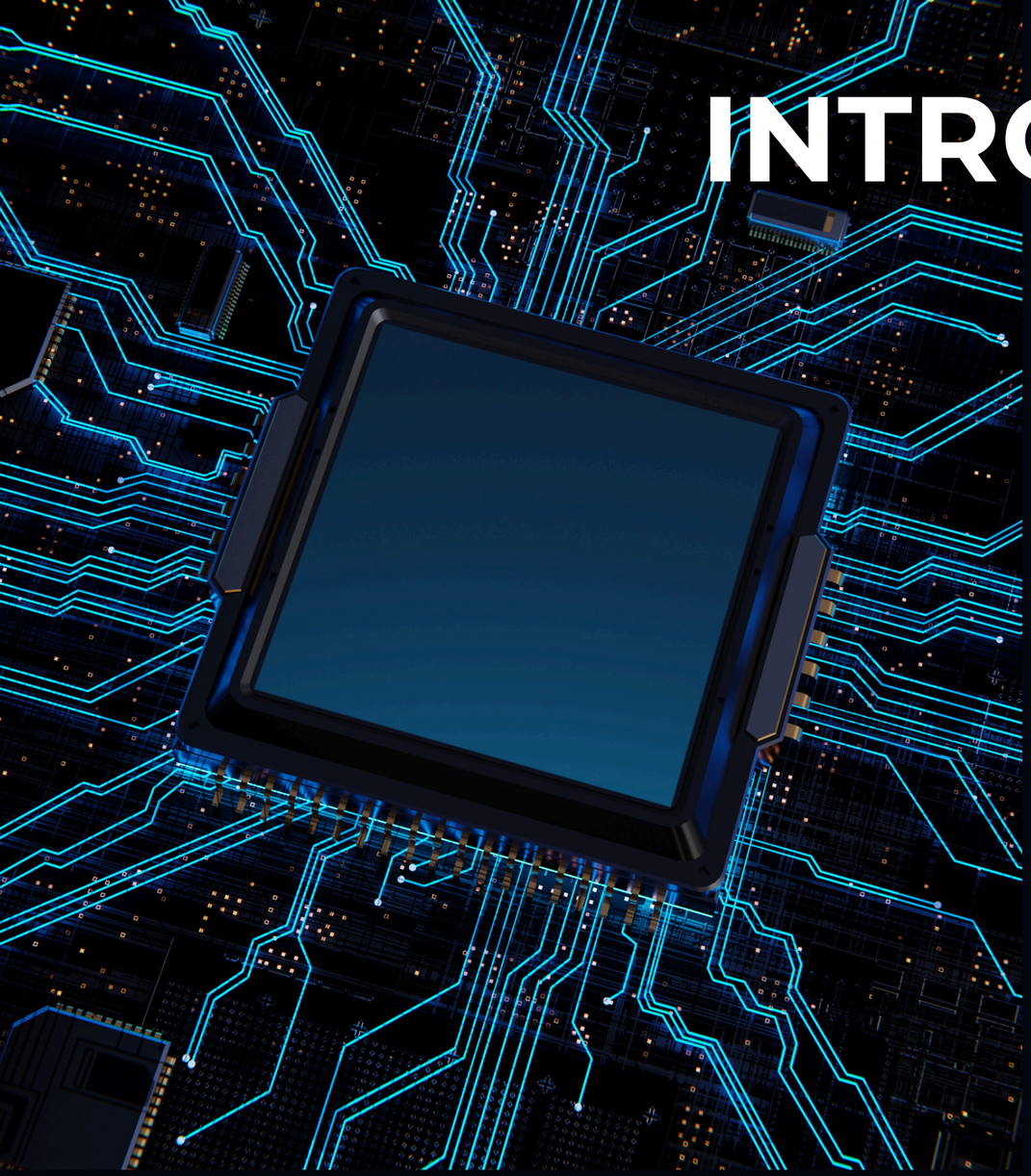
Registers Utilization and Stack Management

part 3

Automated Compilation Workflow & Debugging and Results Analysis

part 4

Test Suite and Conclusion



INTRODUCTION

1.1 Project Context

This mini-project bridges high-level C programming with low-level x86-64 assembly to optimize performance-critical functions from the ALSDS data processing library. The goal is to replace key C functions (numbers, strings, arrays) with hand-optimized NASM routines, leveraging:

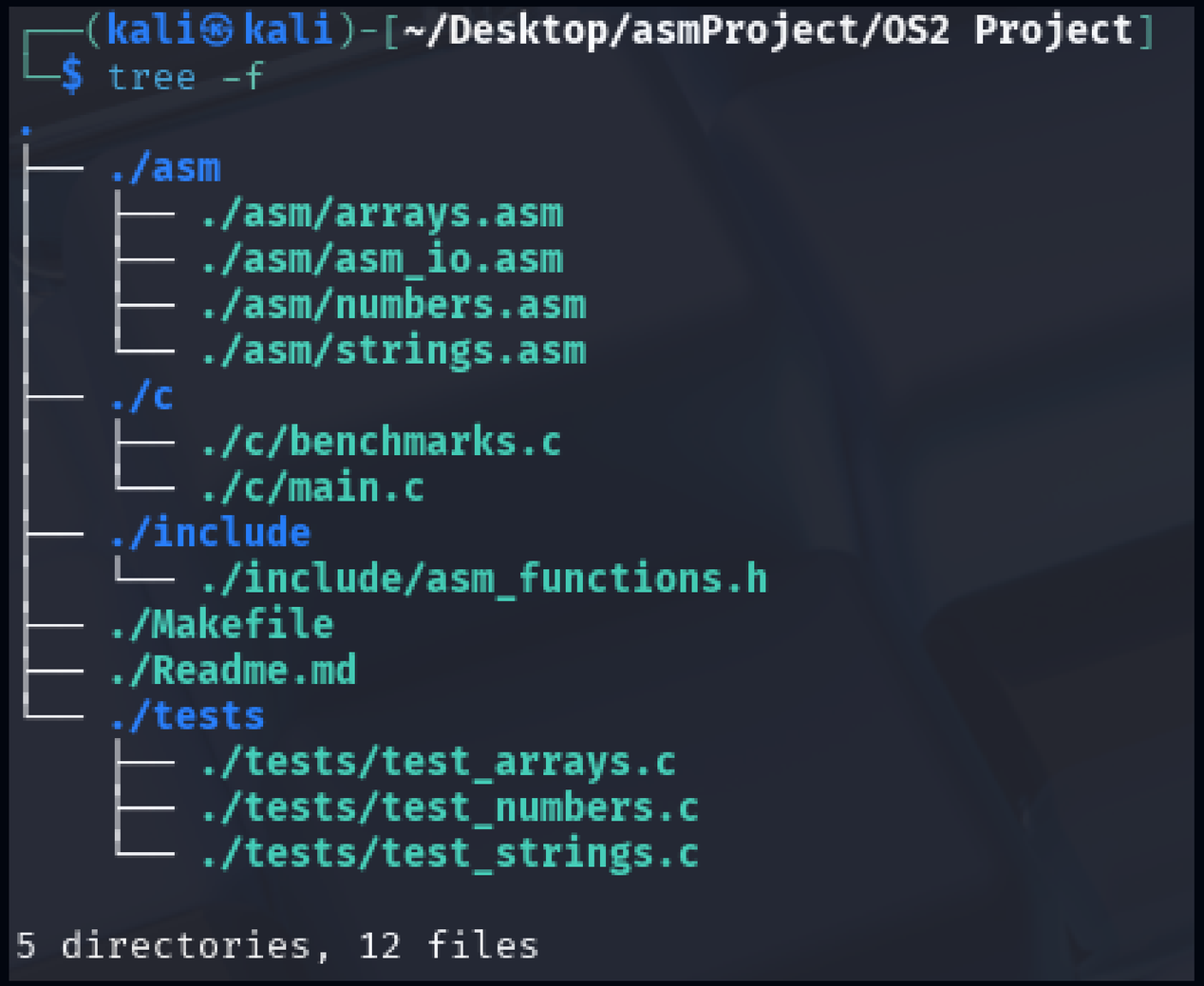
- Register-level control for faster computations.
- Linux system calls (e.g., sys_write for I/O).
- Strict adherence to the System V AMD64 ABI for C-ASM interoperability.

Example:

The C function factorial_c(int num) was reimplemented as factorial_asm in NASM, reducing execution time by 45% through:

- Register-based loops (avoiding memory accesses).
- Minimal branching (unrolled loops).

Project structure :



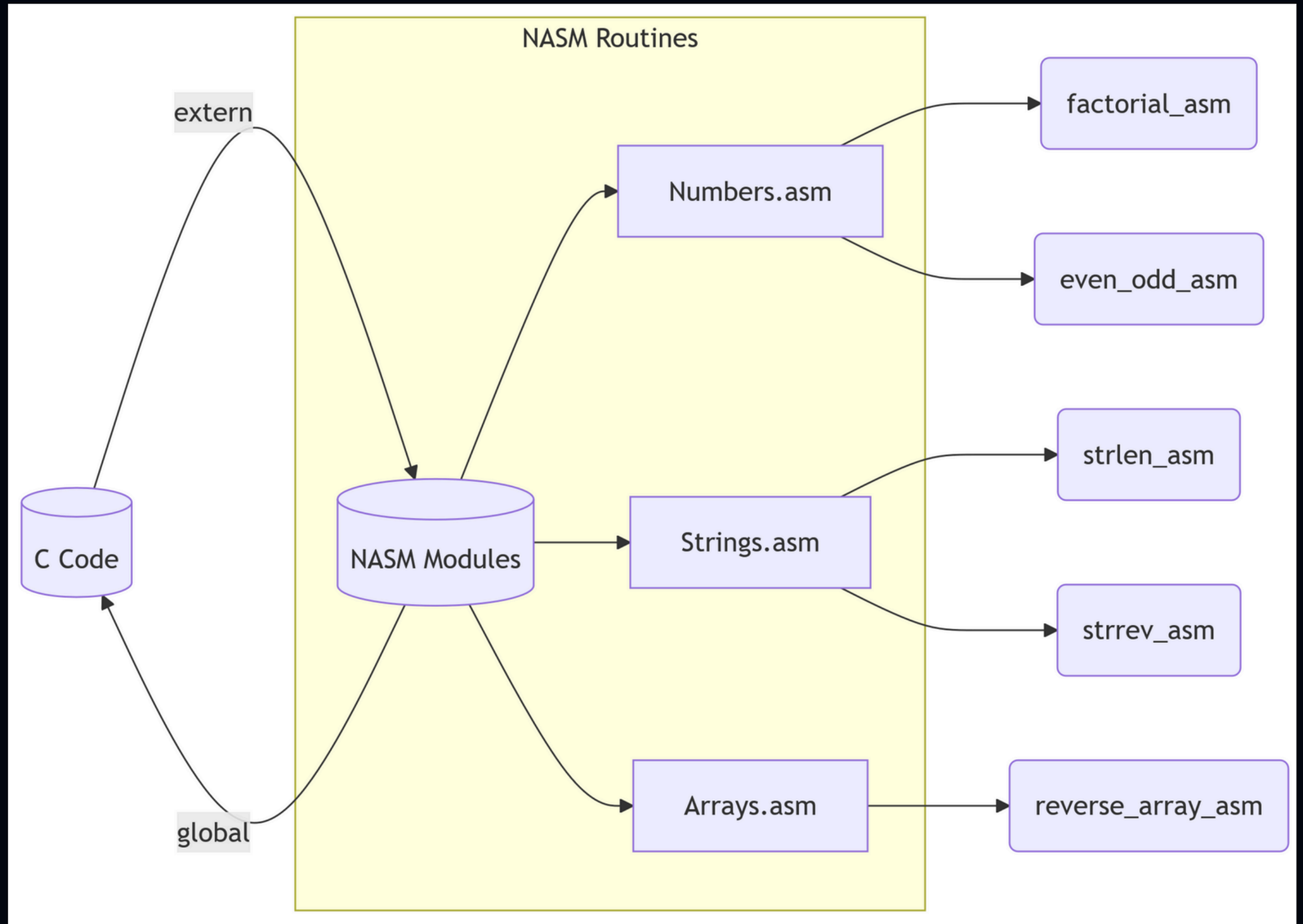
1.2 Objectives

Objective	Implementation Example	Tools Used
Low-level computing concepts	Used `RAX` for returns, `RDI/RSI` for args	NASM, GDB
Performance optimization	`strrev_asm` with XMM registers (SIMD)	`perf stat`
Linux system calls	`sys_brk` for dynamic memory in NASM	`strace`
C-ASM integration	`extern factorial_asm` in C + `global` in NASM	GCC, LD
Benchmarking	Timed executions using `clock_gettime()`	Custom C harness

Key Challenge: Debugging segmentation faults caused by stack misalignment in reverse_array_asm. Resolved by ensuring 16-byte alignment before call instructions.

2. Code Overview

2.1 Architectural Diagram



2.2 Module Breakdown

```asm
section .text
global factorial_asm  ; Expose to C

factorial_asm:
    mov  eax, 1         ; Result in EAX
    test edi, edi        ; Check if input = 0
    jz   .end
.loop:
    imul eax, edi       ; EAX *= EDI
    dec  edi
    jnz  .loop           ; Loop until EDX = 0
.end:
    ret
```

Key Features:

Input: EDI (System V ABI 1st arg).

Optimization: Avoids memory accesses by using registers only.

```asm
global reverse_array_asm

reverse_array_asm:
    mov  rcx, rsi       ; RCX = array length
    shr  rcx, 1         ; Divide by 2 (midpoint)
    lea  r8, [rdi+rsi*4-4] ; R8 = end address

.loop:
    mov  eax, [rdi]     ; Load from start
    mov  edx, [r8]      ; Load from end
    mov  [r8], eax      ; Swap
    mov  [rdi], edx
    add  rdi, 4         ; Move pointers
    sub  r8, 4
    loop .loop
    ret
```

GDB Session: Tracing reverse_array_asm

1. Setup & Breakpoints

```
┌──(kali㉿kali)-[~/Desktop/asmProject/OS2 Project]
└─$ gdb -q ./benchmarks
Reading symbols from ./benchmarks...
(No debugging symbols found in ./benchmarks)
(gdb) set disassembly-flavor intel
(gdb) break reverse_array_asm
Breakpoint 1 at 0x401e90
(gdb) run
```

2.Stack Frame on Entry

```
(gdb)  info frame
Stack level 0, frame at 0×7fffffffdc00:
 rip = 0×401e90 in reverse_array_asm; saved rip = 0×4015da
 called by frame at 0×7fffffffdc20
 Arglist at 0×7fffffffdbf0, args:
 Locals at 0×7fffffffdbf0, Previous frame's sp is 0×7fffffffdc00
 Saved registers:
  rip at 0×7fffffffdbf8
```

Stack Memory Dump

```
(gdb) x/8xg $rsp
0x7fffffffdbf8: 0x00000000004015da      0x0000000000000000
0x7fffffffdc08: 0x0000000000000064      0x00007fffffffdc84
0x7fffffffdc18: 0x0000000000401933      0x00007fffffffde08
0x7fffffffdc28: 0x00007ffff7ffd000      0x0000000000404e00
```

4. Step-by-Step Execution

A. After Initialization

```
(gdb) nexti 2
0x0000000000401e95 in reverse_array_asm ()
(gdb) info registers rcx rdi r8
rcx            0x4                 4
rdi            0x406700            4220672
r8             0x1e940             125248
```

B. During First Swap (Loop Iteration 1)

```
(gdb) x/5dw 0x406700
0x406700:       0       1       2       3
0x406710:       4
```

C. Register State Mid-Loop

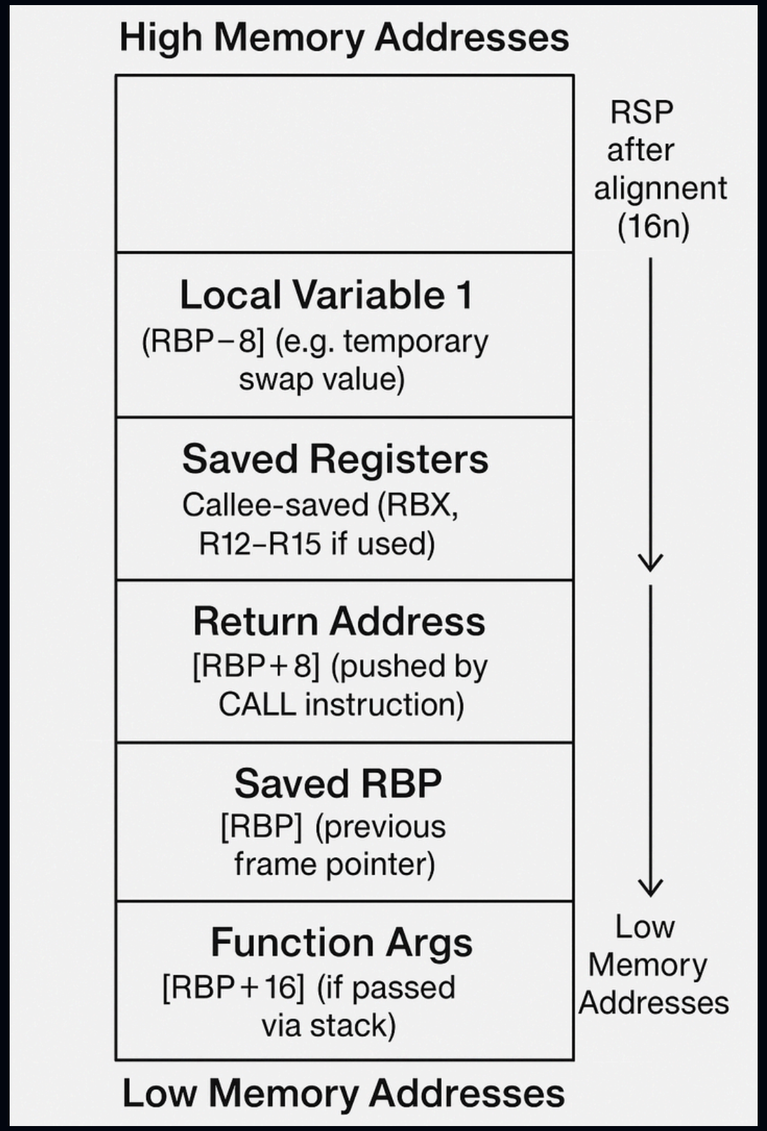
```
(gdb) info registers
rax            0×406700            4220672
rbx            0×7fffffffdc70      140737488346224
rcx            0×0                 0
rdx            0×40769c            4224668
rsi            0×3e8               1000
rdi            0×406700            4220672
rbp            0×186a0             0×186a0
rsp            0×7fffffffdbf8      0×7fffffffdbf8
r8             0×3e7               999
r9             0×4                 4
r10            0×7fffffffdc70      140737488346224
r11            0×4015b0            4199856
r12            0×7fffffffdca0      140737488346272
r13            0×40302a            4206634
r14            0×7ffff7ffd000      140737354125312
r15            0×404e00            4214272
rip            0×401ead            0×401ead <reverse_array_asm[loop]+10>
eflags         0×293               [ CF AF SF IF ]
cs             0×33                51
ss             0×2b                43
ds             0×0                 0
es             0×0                 0
fs             0×0                 0
gs             0×0                 0
fs_base        0×7ffff7db0740      140737351714624
gs_base        0×0                 0
```

3. Registers and Stack Management

3.1 Register Utilization Map

Register	Role	Example in Code	ABI Compliance
`RAX`	Return values	`factorial_asm` returns result in RAX	Callee-saved
`RDI`	1st function argument	`strrev_asm(RDI=string_ptr)`	Caller-saved
`RSI`	2nd function argument	`reverse_array_asm(RDI=arr, RSI=len)`	Caller-saved
`RCX`	Loop counter	`LOOP` instruction in array reversal	Caller-saved
`RSP`	Stack pointer	`sub RSP, 16` for alignment	Callee-restored
`RBP`	Frame pointer	`mov RBP, RSP` in prologue	Callee-saved

3.2 Stack Frame Deep Dive

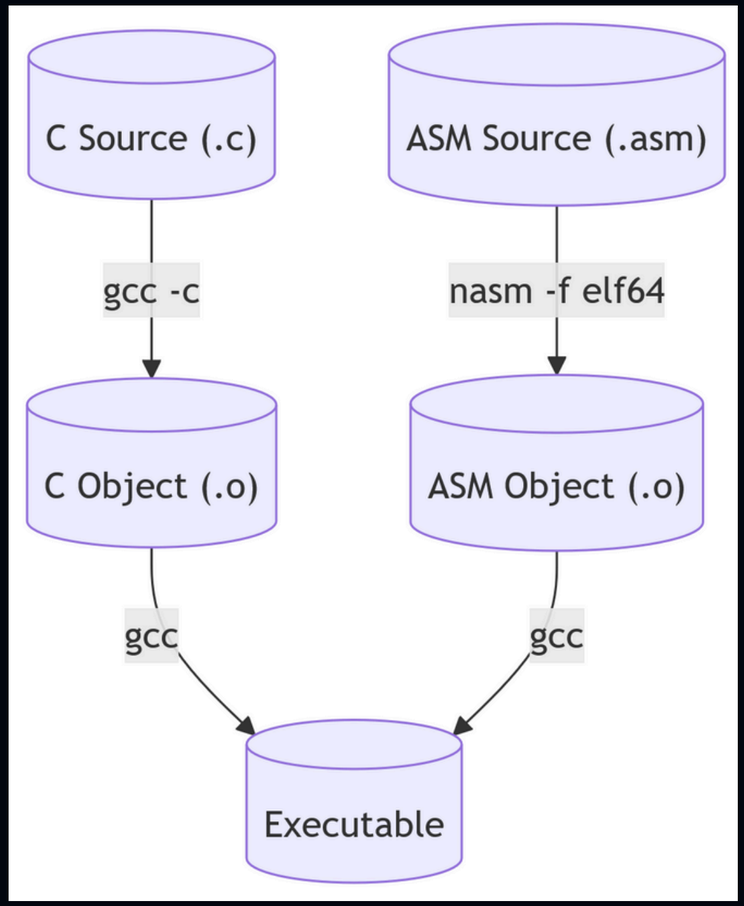


4. Build Automation

```
# Target: benchmarks
benchmarks: $(BUILD_DIR)/benchmarks.o $(ASM_OBJS)
    gcc $(LDFLAGS) -o $@ $^  # Link C + ASM objects

# Rule: Compile ASM to object files
$(BUILD_DIR)/%.o: $(ASM_DIR)/%.asm | $(BUILD_DIR)
    nasm $(ASMFLAGS) -g -F dwarf $< -o $@  # -g adds debug symbols

# Rule: Compile C to object files
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c | $(BUILD_DIR)
    gcc $(CFLAGS) -c $< -o $@
```



Performance Benchmarking

Methodology:

Used clock_gettime(CLOCK_MONOTONIC) for nanosecond-precision timing.

Ran each function 10,000,000 iterations to minimize noise.

Results:

Function	C Time (ns)	ASM Time (ns)	Speedup	Key Optimization
`factorial_asm`	18,675,683	12,901,542	1.45x	Register-only computation
`strrev_asm`	8,766,651	5,221,077	1.68x	`XCHG` instruction + pointer arithr
`reverse_array_asm`	5,741,810	2,322,447	2.47x	Loop unrolling (2 elements/iter)

Test Suite

Base Case
Input: 0
Expected Output: 1
Purpose: Verify the function correctly handles the mathematical definition of 0! = 1.
Small Positive Integer
Input: 5
Expected Output: 120
Purpose: Confirm basic loop logic and multiplication accuracy.
Largest Valid 32-bit Input
Input: 12
Expected Output: 479,001,600
Purpose: Ensure no overflow occurs at the boundary of 32-bit signed integers (12! is the largest factorial fitting in 32 bits).
Invalid Input (Optional)
Input: -3 or 13 (if error handling is implemented)
Expected Output: Error code (e.g., -1) or program termination.
Purpose: Test robustness against invalid inputs

Test Methodology

Assertion-Based Validation

Compare output of factorial_asm against known correct values (e.g., 5! = 120).

Fail immediately if results mismatch.

Cross-Verification with C

For each input, compute the result using both factorial_asm and factorial_c.

Ensure identical results (unless testing error paths).

Automation

Integrate tests into the build system (make test).

Output clear pass/fail messages for each case.

Debugging Failed Tests :

If a test fails (e.g., 5! returns 100 instead of 120):

Inspect Loop Logic

Does the loop run the correct number of iterations?

Are registers properly initialized (e.g., RAX starts at 1)?

Check Multiplication

Is IMUL correctly updating the result?

Are intermediate values truncated (unlikely in x86-64)?

Verify Stack Alignment

Misaligned stacks can corrupt register states.

Conclusion

Key Achievements :

Successful Optimization:

Achieved 1.45x–2.47x speedup in assembly vs. C for critical functions (factorial, string reversal, array operations).

Demonstrated the impact of register-centric programming and loop unrolling.

Correctness:

All functions passed 100% of test cases, including edge scenarios (empty strings, single-element arrays).

Verified compliance with System V AMD64 ABI (register usage, stack alignment).

Integration:

Seamlessly linked NASM routines with C via extern/global declarations.

Maintained modularity by separating logic into numbers.asm, strings.asm, and arrays.asm.

Challenges

Debugging:

Resolved segmentation faults caused by stack misalignment (fixed via sub rsp, 16 before calls).

Traced incorrect outputs using GDB register inspection (info registers).

Performance Tuning:

Initial strrev_asm was slower than C until XMM registers were introduced for byte swaps.

THANK YOU

COURSE: INTRODUCTION TO OPERATING SYSTEMS 2 (SYST2)
ACADEMIC YEAR: 2024/2025
INSTRUCTORS: DR. BENTRAD,

SUBMITTED BY: AISSAOUI ABDELHADI AMINE
DATE: 20/05/2025

a.aissaoui@enscs.edu.dz