Integrating Assembly in C with Makefile

Abstract

This document explains the process of integrating Assembly (ASM) code with C in a stable development environment using a `Makefile`. It details how a `Makefile` is configured to handle both C and NASM (Netwide Assembler) source files, automating the compilation, assembly, and linking steps. Furthermore, it demonstrates how C code can declare and call functions defined in Assembly, ensuring seamless interaction between the two languages.

1 Introduction

Integrating Assembly (ASM) code with C, especially when managed by a `Makefile`, is a common practice for tasks requiring high performance or direct hardware interaction. This section will elaborate on how the provided `Makefile` and `main.c` files facilitate this integration.

2 Makefile Configuration for C and Assembly

The provided `Makefile` is designed to manage both C and NASM source files, automating the entire build process from compilation and assembly to final linking.

2.1 Compiler and Assembler Definitions

- `CC = gcc`: Specifies `gcc` as the C compiler.
- `NASM = nasm`: Specifies `nasm` as the assembler.

2.2 Flags

- `CFLAGS = -Wall -O0 -g`: Compiler flags for C source files.
  - `-Wall`: Enables all warning messages.
  - `-O0`: Disables optimization (useful for debugging).
  - `-g`: Includes debugging information, crucial for debugging mixed C and Assembly code with tools like GDB.
- `NASMFLAGS = -f elf64 -g`: Assembler flags for NASM source files.
  - `-f elf64`: Specifies the output format as ELF64, suitable for 64-bit Linux systems.
  - `-g`: Includes debugging information.
- `LDFLAGS = -no-pie`: Linker flag.
  - `-no-pie`: Disables Position-Independent Executables (PIE). This is often used when linking non-PIC assembly code or when encountering issues with PIE.

2.3 Source and Object Files

- `C`$_SOURCES = main.c : ListstheCsourcefiles.$

2.4 Build Rules

- `all: $(TARGET)`: The default target that builds the final executable.
- `$(TARGET): $(C_OBJECTS) $(ASM_OBJECTS)`: This rule defines how to link the executable. It states that the `$(TARGET)` (which is `main_tester`) depends on all C object files (`$(C_OBJECTS)`) and all Assembly object files ($ASM_{O}BJECTS$)).

3 Calling Assembly Functions from C

The `main.c` file demonstrates how to declare and call functions defined in Assembly code, making them accessible within your C program.

3.1 Function Declarations

In `main.c`, assembly functions are declared using the `extern` keyword. This informs the C compiler that these functions are defined elsewhere (in your assembly code) and that the linker will resolve their addresses.

```c
// Declare the assembly functions as external
// Ensure the function signatures match the assembly (especially argument types and return values)
extern long long asm_function(long long n); // Factorial
extern long long asm_fibonachi(long long n); // Fibonacci (prints internally)
extern void toUpperCase(char* str);
extern void toLowerCase(char* str);
extern void reversString(char* str); // String reversal
extern bool isPalindrom(char* str);
extern void stringConcat(char* dest, const char* src); // String concatenation
extern void strgCopy(char* dest, const char* src);

// Array functions
extern void asm_sort_array(long long* arr, long long size); // Expects array pointer in RDI, size in RSI based on your asm
extern void asm_reverse_array(long long* arr, long long size); // Expects array pointer in RDI, size in RSI based on your asm
extern void asm_reversewithstack_array(long long* arr, long long size); // Expects array pointer in RDI, size in RSI based on your asm
extern long long asm_find_min_in_array(long long* arr, long long size); // Expects array pointer in RSI, size in RDI based on your asm
extern long long asm_find_max_in_array(long long* arr, long long size); // Expects array pointer in RSI, size in RDI based on your asm
extern bool linearSrch(long long* arr, long long size, long long target); // Expects array pointer in RDI, size in RSI, target in RDX based on your asm
```

Listing 1: External Assembly Function Declarations in C

- **Matching Signatures:** It is critical that the C function declarations precisely match the calling convention and argument types expected by your assembly functions. For 64-bit Linux (System V AMD64 ABI), integer and pointer arguments are typically passed in registers `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, and return values in `RAX`.
- **Data Types:** Ensure C data types (e.g., `long long`, `char*`, `bool`) correspond to how values are handled in assembly (e.g., `QWORD` for `long long`).

3.2 Function Calls

Once declared, you can call the assembly functions directly from your C code as if they were regular C functions.

```c
// Example calls from your main.c
printf("Result of asm_function(5) (factorial): %lld\n\n", asm_function(5)); // Calls assembly factorial

char my_string[] = "Hello World";
printf("Original string: %s\n", my_string);
toUpperCase(my_string); // Calls assembly toUpperCase
printf("Uppercase string: %s\n\n", my_string);

long long my_array[] = {5, 2, 8, 1, 9, 4, 7, 3, 6};
long long size = sizeof(my_array) / sizeof(my_array[0]);
print_long_long_array("Original Array", my_array, size);
asm_sort_array(my_array, size); // Calls assembly sort_array
print_long_long_array("Sorted Array", my_array, size);
```

Listing 2: Calling Assembly Functions from C

The C compiler generates calls that follow the standard calling convention, allowing your C code to seamlessly execute the assembly routines.

4 Summary of Integration

The integration of ASM into C using your `Makefile` environment is achieved through these key steps:

1. **Separate Compilation/Assembly:** C source files are compiled by `gcc`, and Assembly source files are assembled by `nasm`, both producing object files (`.o`).
2. **External Declarations:** In your C code, you declare assembly functions as `extern`, signaling to the C compiler that their definitions reside elsewhere.
3. **Unified Linking:** The `Makefile` orchestrates the linking phase, where `gcc` (acting as the linker) combines all the generated object files (from both C and Assembly) into a single executable. This resolves the external references and creates the final program.
4. **Standard Calling Conventions:** Adherence to the System V AMD64 ABI (or the appropriate ABI for your system) ensures that C and Assembly functions can pass arguments and return values correctly to each other.

This robust setup allows you to leverage the strengths of both C (for high-level logic and portability) and Assembly (for fine-grained control and performance optimization) within a single project.