Danmarks Tekniske Universitet

Softwareteknologi

02335 Operating systems, Fall 2024

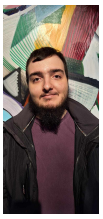
Group 17

Assignment 1 - Reintroduction to C — v.1

Hussein Jarrah - s235110

Marco Hansen - s194302

Ghaith Altayeb - s235090



Andrei Enache - s235089

September 18, 2024

Github: `https://github.com/HUS1904/OS-Assignment-1`

Contents

1 Task 1: I/O Implementation

1.1 Makefile Configuration and Efficiency

It is essential to understand build automation, highlighted through tools like makefiles that simplify program compilation and linking. The provided makefile illustrates its structure and use in system programming.

The makefile specifies the compiler (gcc) to adhere to the C11 standard and include debugging info. This ensures the code meets strict standards and flags potential issues during compilation.

It manages two applications, io_demo and cmd_int, compiling their source files into object files (*.o). A pattern rule applies to all .c files, showing the makefile's ability to generalize actions, which boosts maintainability and scalability. As an example, we will mention the `%.o:%.c` that works as a pattern rule in the makefile. It defines how to convert any .c file into an .o file using a specified recipe.

The linking process creates executable programs from the object files, taking the source files and converting them into object files and then linking them together as an executable file, showcasing how the makefile integrates components into a runnable form.

Utility targets like run-demo, run, and clean are included to execute programs and clean up the build environment, enhancing developer efficiency by allowing straightforward commands directly from the makefile.

1.2 Detailed Implementation of I/O Functions

The I/O functions in the io.c file are direct implementations using Unix system calls for input and output operations, bypassing the standard library functions in <stdio.h>.

read_char(): Implements the read system call, requesting 1 byte from the standard input (STDIN_FILENO). The function captures the result in a char variable and checks if the read was successful (i.e., one byte was read). On success, it casts the character to an integer to return; on failure or when the end of the file is reached, it returns EOF, defined as -1.

```
1  int read_char() {
2      char c;
3      ssize_t result = read(STDIN_FILENO, &c, 1);
4      return result > 0 ? (int)c : EOF;
5  }
```

Listing 1: Reading a byte from stdin with error handling.

write_char(char c): Uses the write system call to send a single byte (the character c) to the standard output (STDOUT_FILENO). It validates that the write operation successfully wrote one byte. If the write is successful, it returns 0, otherwise, it returns EOF to signal an error.

```
1  int write_char(char c) {
2      ssize_t result = write(STDOUT_FILENO, &c, 1);
3      return result == 1 ? 0 : EOF;
4  }
```

Listing 2: Writing a character to stdout using system calls.

write_string(char* s): Iterates over each character of the string s until the null terminator is encountered. Each character is written to standard output by invoking write_char(). If any call to write_char() fails (returns EOF), the write_string() function immediately returns EOF as well.

```
1 int write_string(char* s) {
2     for (char* p = s; *p != '\0'; ++p) {
3         ssize_t result = write(STDOUT_FILENO, p, 1);
4         if (result != 1) return EOF;
5     }
6     return 0;
7 }
```

Listing 3: Outputting a string to stdout, character by character.

write_int(int n): Manually converts an integer to its string representation. It accommodates negative numbers and constructs the string in reverse order. After reversing the string, it utilizes write_string() to output the entire integer. The function checks the return of write_string(); if it returns 0, indicating success, write_int() also returns 0. If not, it returns EOF.

```
1 buffer[i] = '\0';
2 for (int j = 0; j < i / 2; ++j) {
3     char temp = buffer[j];
4     buffer[j] = buffer[i - j - 1];
5     buffer[i - j - 1] = temp;
6 }
```

Listing 4: Reversing and writing an integer's string representation.

1.3 Analysis of Demo Program's Behavior

When the characters 'abcqls' are entered into the demo program, the processing of each character is dictated by predefined command interpretations. The characters 'a', 'b', and 'c' are processed according to the functions associated with these commands. Assuming that 'q' is designed to signify a termination command, the processing should ideally cease with its recognition.

If 'q' properly initiates termination, the characters 'l' and 's' following it should not be processed, suggesting that the demo program effectively ignores inputs post-termination command. This behavior is crucial as it directly reflects the program's capability to handle command sequences efficiently and halt processing when required.

The sequence 'abcx123' exemplifies a scenario where unexpected behavior might arise. If 'x' is not a recognized command and no robust error handling is implemented, the subsequent digits '123' could mistakenly be processed, leading to potential errors or undefined behavior.

Command Validation: Implementing a stringent validation mechanism for each input character against a set of permissible commands can prevent the processing of invalid commands. This approach ensures that only recognized commands influence the program's state, enhancing its reliability and predictiveness.

Robust Loop Management: Adjusting the program's main loop to include explicit checks for termination characters can bolster the program's ability to halt further processing immediately upon encountering an unrecognized command. This will not only secure the program against inadvertent executions but also streamline the command handling process.

2 Task 2: Command Interprete

2.1 Implementation of the Collection Management System

The core challenge involved developing an efficient and dynamic system for managing a collection of integers, per the specifications detailed in the assignment's requirements. The solution was implemented within the main.c file, employing dynamic memory allocation and manipulation techniques to ensure flexibility and robustness in handling an unknown number of integer inputs during runtime.

Data Structure Choice: The collection of integers is managed using a dynamically allocated array, leveraging the realloc function from ¡stdlib.h¿. This choice allows the array to grow as needed when new integers are added based on user commands, thereby accommodating an indefinite number of inputs which aligns with the requirement for dynamic memory usage.

```
1 int *collection = NULL;
2 int collection_size = 0;
```

Listing 5: Initializing dynamic array management.

Dynamic Memory Allocation: The dynamic array, named collection, starts with no allocated memory. Memory allocation is performed incrementally as the user inputs the 'a' command, which requires adding the current count to the collection. Each time this command is executed, realloc is called to resize the collection array to hold one additional integer. This method ensures that the program only uses as much memory as needed at any point in time, optimizing resource utilization and ensuring scalability.

```
1 collection = realloc(collection, (collection_size + 1) *
      sizeof(int));
2 if (collection != NULL) {
3     collection[collection_size++] = count;
4 } else {
5
6 }
```

Listing 6: Resizing the array dynamically.

Managing the Collection: The collection array is indexed directly to add integers at the next available position, which is tracked by collection_size. This indexing method minimizes the complexity of insert operations to O(1), as it directly accesses the end of the array. The removal of the most recently added element, triggered by the 'c' command, is managed by simply decrementing the collection_size, effectively ignoring the last element in subsequent operations without the need for memory reallocation at every step.

```
1 if (collection_size > 0) {
2     collection_size--;
3 }
```

Listing 7: Removing the latest integer efficiently.

Memory Cleanup: Proper memory management also involves cleaning up dynamically allocated memory to prevent leaks, which is handled at the end of the program's execution.

```
1 free(collection);
```

Listing 8: Cleaning up allocated memory.

This implementation strategy not only adheres to the assignment's requirements for using low-level system calls and dynamic memory management but also provides a robust framework for efficiently managing a dynamically sized collection of integers. The use of realloc ensures that the memory footprint is as small as possible while still providing the flexibility required for the unknown number of inputs, embodying both efficiency and practicality in system-level programming.

2.2 The 'make test' Command

The program will first check the status of the MAIN_EXECUTABLE, in our case this is cmd_int. If cmd_int does not yet exist it will be created using GCC, the flags we have set under CFLAGS and the MAIN_OBJECTS, the resulting command expands to something like this:

```
1 gcc -Wall -W -std=c11 -g main.o io.o -o cmd\_int
```

Listing 9: Command that creates the executable

Afterwards the ./test.sh is used to actually run the test file. The file test.sh contains several sets of inputs and expected outputs, cmd_int is run with the provided input, the script then echoes either "PASSED" or "FAILED", depending on whether or not the output matches the expected output. Echoed before each test is also the name of that given test.

2.3 Enhancements to Test Coverage with New Cases

To further test the program and verify that it functions as intended, 9 additional tests were added to the 'test.sh' file. These tests range from basic collection management to ensuring that the program can handle a large sequence of commands without any abnormalities.

```
1 in=$(printf 'a%.0s' {1..100})"q"
2 expected_output=$(seq -s ',' 0 99)";"
3 echo -n Test 7 (Large sequence of 'a' commands - 100 'a's):
4 [[ $(./cmd_int <<< "$in") == "$expected_output" ]] && echo "PASSED" || echo
      "FAILED"
```

Listing 10: Test 7: Large sequence of 'a' commands

Test 7, as shown above, tests the program with a large sequence of the command 'a' and checks if the output matches the expected result. The test gives feedback by echoing "PASSED" if the actual output matches the expected output and "FAILED" otherwise.

```
1 in="cccq"
2 out=";"
3 echo -n Test 5 ('c' with empty collection - cccq):
4 [[ $(./cmd_int <<< "$in") == "$out"* ]] && echo "PASSED" || echo "FAILED"
```

Listing 11: Test 5: 'c' command with empty collection

Test 5 checks how the program behaves when given the 'c' command in an empty collection. To pass this test, the program must produce a semicolon as output and avoid crashing when the collection is empty.

Listings