Building a Modular Shell in C Using Static and Dynamic Libraries: A System-Level Programming Approach

Hanan Kemal
Department of Computer Science
Bowie State University
Bowie, MD, USA
Email: hkemal@students.bowiestate.edu

Prof. Appolo Tankeh
Department of Computer Science
Bowie State University
Bowie, MD, USA
Email: atankeh@bowiestate.edu

***Abstract*—This project examines the design and implementation of a modular command-line shell written in the C programming language using core system-level programming techniques studied throughout the semester. These techniques include the use of static and shared libraries, automated build systems through Makefiles, runtime dynamic linking, and memory management principles. Drawing from Stephen Brennan's Lightweight Shell (LSH) tutorial, Dr. Duncan White's TEFEL (Tool-Enhanced Features for Existing Languages) methodology, and midterm course projects, the development goal was to transform a monolithic C shell program into a reusable, extensible library-based software system. Reusable functionality was encapsulated within a shared object (`libshelllib.so`) and dynamically linked to the main shell at runtime. This modular approach improves maintainability, promotes code reuse, and demonstrates how professional operating system techniques can be applied in C. This report documents the system architecture, code implementation, build process, runtime behavior, memory considerations, debugging strategies, validation testing, and recommendations for further development.**

***Index Terms*—Shell programming, C libraries, TEFEL, Makefile automation, dynamic linking, memory management, modular software systems**

I. Introduction

Modern operating systems rely on modular software architectures that emphasize maintainability, extensibility, and code reuse. Complex software systems are rarely constructed as single, monolithic programs; instead, they are organized into reusable libraries linked at runtime. Unix shells such as `bash` and `zsh` embody this philosophy by separating command parsing, job control, I/O handling, and built-in utilities into logically distinct modules.

This project adopts the same architectural philosophy to construct a student-developed shell using the C language. Core topics studied throughout the semester—including memory management, static and dynamic linking, modular compilation, Makefile automation, and TEFEL tools—were integrated into a single cohesive system. Rather than embedding all shell commands within a single executable file, reusable utility functions were relocated into a dedicated shared library loaded during program execution. This design reduces code duplication, improves scalability, and aligns closely with industry-standard practices in systems software development.

II. Background and Related Work

Stephen Brennan's *Lightweight Shell* tutorial served as an introductory guide for essential shell design concepts including input parsing, command routing, process control, and loop execution. While Brennan's approach demonstrates a functional shell design, it remains monolithic and difficult to extend.

Dr. Duncan White's TEFEL methodology builds upon this foundation by enabling the extension of existing programming languages through supporting tools and code generation practices rather than modifying core language syntax. White's work, particularly his exploration of Go-style interfaces in C, demonstrates how dynamic libraries and preprocessing tools can simulate advanced language features while preserving C's simplicity and portability.

These ideas guided the architectural transformation of the shell developed in this project: static shell logic combined with dynamically loaded modular functionality implemented as a shared library.

III. Project Overview

The completed system consists of the following primary components:

- A shell controller program: `myshell.c`
- Library interface definitions: `shelllib.h`
- Shared library implementation: `shelllib.c`
- Compiled dynamic library: `libshelllib.so`
- A Makefile for project automation
- Screenshot documentation demonstrating each build and execution stage

This separation of responsibility ensures that the shell remains lightweight and focused on user interaction while all functional behavior resides within the externally maintained library.

IV. System Architecture

The system follows a layered architectural model:

- **User Interface Layer:** Accepts input commands and displays output.
- **Shell Control Layer:** Interprets commands and invokes library calls.
- **Library Module Layer:** Implements all reusable command features via dynamically linked functions contained in `libshelllib.so`.
- **Operating System Layer:** Handles process creation, virtual memory management, and symbol resolution performed by the dynamic linker.

Unlike static libraries that are merged directly into the final executable, shared libraries are loaded only when needed at runtime. This allows multiple applications to reference a single instance of the library in physical memory, significantly reducing RAM usage while supporting future upgrades without entire program recompilation.

V. Library Code

A. Header File: `shelllib.h`

```
#ifndef SHELLLIB_H
#define SHELLLIB_H

void shell_hello(void);
void shell_help(void);
int shell_add(int a, int b);

#endif
```

B. Library Implementation: `shelllib.c`

```
#include <stdio.h>
#include "shelllib.h"

void shell_hello(void) {
    printf("Hello from the shared library!\n")
        ;
}

void shell_help(void) {
    printf("Library commands:\n");
    printf("  hello\n");
    printf("  help\n");
    printf("  add A B\n");
}

int shell_add(int a, int b) {
    return a + b;
}
```

VI. Shell Program

```
#include <stdio.h>
#include "shelllib.h"

int main(void)
{
    shell_hello();
    shell_help();
    printf("3 + 5 = %d\n", shell_add(3,5));
    return 0;
}
```

VII. Build System

The Makefile automates:

- Compilation of library source into object files
- Creation of the shared dynamic object
- Linking of the shell binary
- Cleanup of intermediate artifacts

VIII. Compilation Commands

```
make
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
./myshell
```

IX. Execution Output

```
Hello from the shared library!
Library commands:
  hello
  help
  add A B
3 + 5 = 8
```

X. Memory Management Considerations

Position-Independent Code (PIC) enables the operating system to map the shared library into any available virtual memory address region without modification. Read-only memory sections are shared across processes while writable sections employ copy-on-write semantics to preserve safety and isolation between running programs.

This dynamic linking design increases both memory efficiency and overall system stability compared to static linking.

XI. Debugging and Error Handling

Troubleshooting primarily involved resolving linker errors caused by missing PIC flags, incorrect library search paths, and unresolved external symbols. Compiler warnings were enforced using `-Wall` to prevent unsafe memory usage or improper function declarations. Runtime verification ensured that function calls were correctly resolved from the shared object.

XII. Evaluation

System validation involved:

- Building the project repeatedly via Makefile fences
- Verifying correct shell output execution
- Dependency tracking using linker inspection commands
- Runtime stability testing

All tests demonstrated stable and correct behavior across builds.

XIII. Future Extensions

Future enhancements to the shell architecture may include:

- Full parsing of user input
- Job execution via `fork()` and `exec()`
- Plugin-style command loading
- Signal handling
- Environment variable management

XIV. Conclusion

This project demonstrates professional system programmability through the use of modular architecture, TEFEL-based extensions, dynamic linking, and automated build pipelines. The work validates both theoretical and practical course objectives while providing a scalable framework for future shell development efforts.

Acknowledgment

I thank Professor Appolo Tankeh for instruction in operating systems and system-level C programming.