Custom Shell Project Report

Author: Sai Swaroop Bindhani

Registration No: 2241002181

Institute: Institute of Technical Education and Research (ITER)

Course: B.Tech in Computer Science

Date: November 7, 2025

1. Title

Custom Shell - A miniature Linux shell implemented in C++ with features like process control, I/O redirection, piping, background jobs, signal handling, and built-in commands.

2. Code Summery

The project implements a Unix-like shell in C++.

It supports:

- Command execution using fork() and execvp().

- Built-in commands such as cd, help, clear, about, jobs, fg, bg, killjob, exit.

- Input/output redirection using , and >>.

- Pipelining using |.

- Background job execution using &.

- Signal handling for Ctrl+C and Ctrl+Z.

- Job control management with foreground/background switching.

- Command history and tab completion using the Readline library.

- Custom hidden Easter eggs 'rhino' and 'xsmax'.

3. Code

```
// SAVE THIS AS ~/projects/custom_shell/src/main.cpp

#include <iostream>

#include <string>

#include <vector>

#include <deque>

#include <map>

#include <sstream>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#include <csignal>

#include <cerrno>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/stat.h> #include

<fcntl.h>

#include <termios.h>

#include <readline/readline.h>

#include <readline/history.h>


static int next_job_id = 1;


struct Job { int

   id;
```

```
   pid_t pgid;
std::string cmdline;
bool running;    bool
stopped;    bool
background;
};

static  std::deque<Job>  jobs;  static
struct termios shell_tmodes; static
pid_t shell_pgid = 0;

// ---------- Utilities ---------- static inline void
safe_perror(const char *msg) {    std::cerr << msg << ":
" << std::strerror(errno) << "\n";
}

std::string join_tokens(const std::vector<std::string> &v, const std::string &sep = " ") {
std::string s;    for (size_t i = 0; i < v.size(); ++i) {        if (i) s += sep;        s += v[i];
   }
   return s;
}

// ---------- Readline completion ---------- char **custom_completion(const
char *text, int start, int end) {    rl_attempted_completion_over = 0; return
rl_completion_matches(text, rl_filename_completion_function);
}

// ---------- Job management ---------- void add_job(pid_t pgid, const
std::string &cmdline, bool background) {
```

```
    Job j;
    j.id = next_job_id++;
    j.pgid = pgid;
    j.cmdline = cmdline;
    j.running = true;
    j.stopped = false;
    j.background = background;       jobs.push_back(j);       if (background) {
std::cout << "[" << j.id << "] " << pgid << " started: " << cmdline << "\n";
    }
}

Job* find_job_by_id(int id) {     for (auto &j :
jobs) if (j.id == id) return &j;    return nullptr;
}

Job* find_job_by_pgid(pid_t pgid) {      for (auto
&j : jobs) if (j.pgid == pgid) return &j;      return
nullptr;
}

void remove_finished_jobs() { for (auto it =
    jobs.begin(); it != jobs.end(); ) {
        if (!it->running && !it->stopped) it = jobs.erase(it);
else ++it;
    }
}
```

```
void mark_job_stopped(pid_t pgid) {    Job* j
= find_job_by_pgid(pgid);    if (j) { j->stopped
= true; j->running = false; }
}

void mark_job_done(pid_t pgid) {      Job* j =
find_job_by_pgid(pgid);      if (j) { j->running =
false; j->stopped = false; }
}

// ---------- Signals ----------
void sigchld_handler(int) {
int saved_errno = errno;
while (true) {      int status;
      pid_t pid = waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED);
      if (pid <= 0) break;        pid_t pgid = getpgid(pid);        if (pgid < 0)
continue;        if (WIFEXITED(status) || WIFSIGNALED(status)) {
mark_job_done(pgid);           Job* j = find_job_by_pgid(pgid);           if
(j && j->background) {              std::cout << "\n[" << j->id << "]
Done\t" << j->cmdline << "\n";              rl_on_new_line();
rl_replace_line("", 0); rl_redisplay();

         }
      } else if (WIFSTOPPED(status)) {
mark_job_stopped(pgid);           Job* j
= find_job_by_pgid(pgid);
         if (j) {
            std::cout << "\n[" << j->id << "] Stopped\t" << j->cmdline << "\n";
rl_on_new_line(); rl_replace_line("", 0); rl_redisplay();
         }
```

```
      } else if (WIFCONTINUED(status)) {           Job*
j = find_job_by_pgid(pgid);           if (j) { j->running
= true; j->stopped = false; }
      }
   }
   errno = saved_errno;
}

void  handle_sigint_shell(int)  {          std::cout  <<  "\n";
rl_on_new_line(); rl_replace_line("", 0); rl_redisplay();
}

// ---------- Parsing helpers ---------- std::vector<std::string>
split_pipe_segments(const std::string &line) {
std::vector<std::string> segments;    std::string cur;    bool in_quote
= false;    char quote_char = 0;    for (size_t i = 0; i < line.size(); ++i) {
      char c = line[i];        if (!in_quote && (c == '\'' || c ==
'"')) {           in_quote = true; quote_char = c;
cur.push_back(c);        } else if (in_quote && c ==
quote_char) {           in_quote = false; cur.push_back(c);
} else if (!in_quote && c == '|') {
segments.push_back(cur);
          cur.clear();        }
else {
cur.push_back(c);
      }
   }
   if (!cur.empty()) segments.push_back(cur);    for (auto
&s : segments) {        size_t a = s.find_first_not_of(" \t");
```

```
size_t b = s.find_last_not_of(" \t");                s = (a ==
std::string::npos) ? "" : s.substr(a, b - a + 1);
   }
   return segments;
}

std::vector<std::string> tokenize_space(const std::string &s) {
std::vector<std::string> tokens;    std::string cur;    bool
in_quote = false;    char qch = 0;    for (size_t i = 0; i <
s.size(); ++i) {        char c = s[i];        if (!in_quote && (c == '"'
|| c == '\'')) {
          in_quote = true; qch = c; cur.push_back(c);        } else
if (in_quote && c == qch) {            in_quote = false;
cur.push_back(c);        } else if (!in_quote &&
isspace((unsigned char)c)) {            if (!cur.empty()) {
tokens.push_back(cur); cur.clear(); }
      } else {
cur.push_back(c);
      }
   }
   if (!cur.empty()) tokens.push_back(cur);
return tokens;
}

struct       Command       {
std::vector<char*>    argv;
std::string                infile;
std::string outfile;      bool
append = false;
```

```
};

Command parse_command_segment(const std::string &seg) {
Command cmd;    auto toks = tokenize_space(seg);    for
(size_t i = 0; i < toks.size();) {        std::string t = toks[i];        if
(t == "<") {            if (i + 1 < toks.size()) cmd.infile = toks[i + 1];
i += 2;
      } else if (t == ">" || t == ">>") {
         if (t == ">>") cmd.append = true;            if (i +
1 < toks.size()) cmd.outfile = toks[i + 1];
         i += 2;        } else {            std::string clean = t;            if (clean.size() >=
2 && ((clean.front() == '"' && clean.back() == '"') ||
             (clean.front() == '\'' && clean.back() == '\''))) {
clean = clean.substr(1, clean.size() - 2);
         }
         cmd.argv.push_back(strdup(clean.c_str()));
         i++;
      }
   }
   cmd.argv.push_back(nullptr);
return cmd;
}

void free_command_args(Command &c) {
   for (auto p : c.argv) if (p) free(p);
   c.argv.clear();
   c.argv.push_back(nullptr);
}
```

```
// ---------- Builtins ---------- bool is_builtin_name(const std::string &s) {
return (s == "cd" || s == "help" || s == "exit" || s == "clear" ||           s ==
"about" || s == "jobs" || s == "fg" || s == "bg" || s == "killjob");
}

void print_jobs() {     for (auto &j :
jobs) {        std::cout << "[" << j.id <<
"] " <<
          (j.running ? "Running" : (j.stopped ? "Stopped" : "Done")) <<
          "\t" << j.pgid << "\t" << j.cmdline << (j.background ? " &" : "") << "\n";
   }
}

void show_easter_egg(const std::string &cmd) {     if (cmd == "rhino") {        std::cout <<
"\n\033[1;31mTHUG\033[0m\n";        std::cout << "\033[1;36m\"He who makes others see but he
himself is invisible.\"\033[0m\n\n";
   } else if (cmd == "xsmax") {        std::cout <<
"\n\033[1;33mR€$!$T\033[0m\n";        std::cout << "\033[1;35m\"Can't
see his own Abyss.\"\033[0m\n\n";
   }
}

void builtin_execute(const std::vector<std::string> &words) {
if (words.empty()) return;     const std::string &cmd =
words[0];     if (cmd == "cd") {        if (words.size() >= 2) {
if (chdir(words[1].c_str()) != 0) perror("cd");
      } else {
         const char *home = getenv("HOME");
if (home) chdir(home);
      }
```

```
   } else if (cmd == "help") {
      std::cout << "mini-shell help:\nBuiltins: cd, help, clear, about, jobs, fg, bg, killjob, exit\n";
   } else if (cmd == "clear") {        std::cout <<
"\033[H\033[2J" << std::flush;
   } else if (cmd == "about") {        std::cout << "Ultimate mini-
shell by Sami-style assistant.\n";
   } else if (cmd == "jobs") {        print_jobs();     } else if (cmd ==
"fg") {        if (words.size() >= 2) {            std::string idstr =
words[1];            if (!idstr.empty() && idstr[0] == '%') idstr =
idstr.substr(1);            int id = atoi(idstr.c_str());            Job* j =
find_job_by_id(id);            if (!j) { std::cout << "fg: job not
found\n"; return; }            if (kill(-j->pgid, SIGCONT) < 0)
perror("SIGCONT");            j->background = false; j->stopped =
false; j->running = true;            tcsetpgrp(STDIN_FILENO, j->pgid);
          int status;
          waitpid(-j->pgid, &status, WUNTRACED);
tcsetpgrp(STDIN_FILENO, shell_pgid);            if (WIFSTOPPED(status)) {
j->stopped = true; j->running = false; }            else { j->running = false;
j->stopped = false; }            remove_finished_jobs();
      } else std::cout << "fg: usage: fg %jobid\n";
   } else if (cmd == "bg") {        if (words.size() >= 2) {
std::string idstr = words[1];            if (!idstr.empty() && idstr[0]
== '%') idstr = idstr.substr(1);            int id = atoi(idstr.c_str());
          Job* j = find_job_by_id(id);            if (!j) { std::cout << "bg: job not
found\n"; return; }            if (kill(-j->pgid, SIGCONT) < 0) perror("SIGCONT");
j->background = true; j->stopped = false; j->running = true;            std::cout <<
"[" << j->id << "] " << j->pgid << " resumed in background\n";
      } else std::cout << "bg: usage: bg %jobid\n";
```

```
   } else if (cmd == "killjob") {        if (words.size() >= 2) {
std::string idstr = words[1];            if (!idstr.empty() && idstr[0]
== '%') idstr = idstr.substr(1);           int id = atoi(idstr.c_str());
Job* j = find_job_by_id(id);            if (!j) { std::cout << "killjob:
job not found\n"; return; }            if (kill(-j->pgid, SIGKILL) < 0)
perror("kill");            else std::cout << "killed job " << j->id <<
"\n";
       } else std::cout << "killjob: usage: killjob %jobid\n";
   } else if (cmd == "exit") exit(0);    else if (cmd == "rhino" || cmd ==
"xsmax") show_easter_egg(cmd);
}

// ---------- Execution ----------
void launch_pipeline(std::vector<Command> &commands, bool background, const std::string &cmdline) {
size_t n = commands.size();    std::vector<int> pipes;    pipes.resize((n > 0 ? n - 1 : 0) * 2);

   for (size_t i = 0; i + 1 < n; ++i) {        if
(pipe(&pipes[i*2]) < 0) { perror("pipe"); return; }
   }

   pid_t pgid = 0;    for (size_t i = 0; i < n; ++i) {        pid_t pid = fork();        if (pid <
0) { safe_perror("fork"); return; }        else if (pid == 0) {            if (i == 0) setpgid(0,
0); else setpgid(0, pgid);            signal(SIGINT, SIG_DFL); signal(SIGQUIT, SIG_DFL);
signal(SIGTSTP, SIG_DFL);

         if (i > 0) dup2(pipes[(i-1)*2], STDIN_FILENO);
if (i + 1 < n) dup2(pipes[i*2 + 1], STDOUT_FILENO);
for (size_t j = 0; j < pipes.size(); ++j) close(pipes[j]);
```

```
        if (!commands[i].infile.empty()) {              int fd =
open(commands[i].infile.c_str(), O_RDONLY);             if (fd < 0) {
safe_perror("open infile"); exit(EXIT_FAILURE); }             dup2(fd,
STDIN_FILENO); close(fd);
        }
        if (!commands[i].outfile.empty()) {             int flags = O_WRONLY | O_CREAT |
(commands[i].append ? O_APPEND : O_TRUNC);              int fd =
open(commands[i].outfile.c_str(), flags, 0644);             if (fd < 0) { safe_perror("open
outfile"); exit(EXIT_FAILURE); }             dup2(fd, STDOUT_FILENO); close(fd);
        }

        execvp(commands[i].argv[0], commands[i].argv.data());
perror("execvp");          exit(EXIT_FAILURE);
      } else {
        if (i == 0) { pgid = pid; setpgid(pid, pgid); }
else setpgid(pid, pgid);
      }
   }

   for (size_t j = 0; j < pipes.size(); ++j) close(pipes[j]);

   if (!background) {
tcsetpgrp(STDIN_FILENO, pgid);
      int status;
pid_t w;       do
{
        w = waitpid(-pgid, &status, WUNTRACED);
        if (w == -1) break;          if
(WIFSTOPPED(status)) {
```

```
add_job(pgid, cmdline, false);
mark_job_stopped(pgid);
            break;
          }
      } while (!WIFEXITED(status) && !WIFSIGNALED(status));
tcsetpgrp(STDIN_FILENO, shell_pgid);
   } else {        add_job(pgid,
cmdline, true);
   }
   remove_finished_jobs();
}

// ---------- Main loop ---------- int
main() {

   shell_pgid  =  getpid();          if  (setpgid(shell_pgid,
shell_pgid)     <     0)     {     /*     ignore     */     }
tcsetpgrp(STDIN_FILENO,                        shell_pgid);
tcgetattr(STDIN_FILENO, &shell_tmodes);

   struct sigaction sa_chld;    sa_chld.sa_handler =
sigchld_handler;    sigemptyset(&sa_chld.sa_mask);
sa_chld.sa_flags = SA_RESTART | SA_NOCLDSTOP;
sigaction(SIGCHLD, &sa_chld, nullptr);

   struct sigaction sa_int;
sa_int.sa_handler = handle_sigint_shell;
sigemptyset(&sa_int.sa_mask);
sa_int.sa_flags = SA_RESTART;
sigaction(SIGINT, &sa_int, nullptr);
```

```
   signal(SIGTTOU, SIG_IGN);
signal(SIGTTIN, SIG_IGN);

   rl_attempted_completion_function = custom_completion;

   while (true) {        char cwd[1024]; getcwd(cwd, sizeof(cwd));        std::string user =
getenv("USER") ? getenv("USER") : "user";        std::string prompt = "\033[1;36m[" + user
+ "@ultimate-shell " + cwd + "]\033[0m$ ";        char *line_c = readline(prompt.c_str());
if (!line_c) { std::cout << "\n"; break; }        std::string line(line_c);

      free(line_c);

      auto start = line.find_first_not_of(" \t");
if (start == std::string::npos) continue;
auto end = line.find_last_not_of(" \t");
line = line.substr(start, end - start + 1);        if
(line.empty()) continue;

      add_history(line.c_str());        auto
segments = split_pipe_segments(line);        if
(segments.empty()) continue;

      bool background = false;        if (!line.empty() && line.back() == '&') {
background = true;            // remove trailing &            while (!line.empty() &&
isspace((unsigned char)line.back())) line.pop_back();            if (!line.empty() &&
line.back() == '&') { line.pop_back(); }
         // rebuild segments            segments =
split_pipe_segments(line);
      }
```

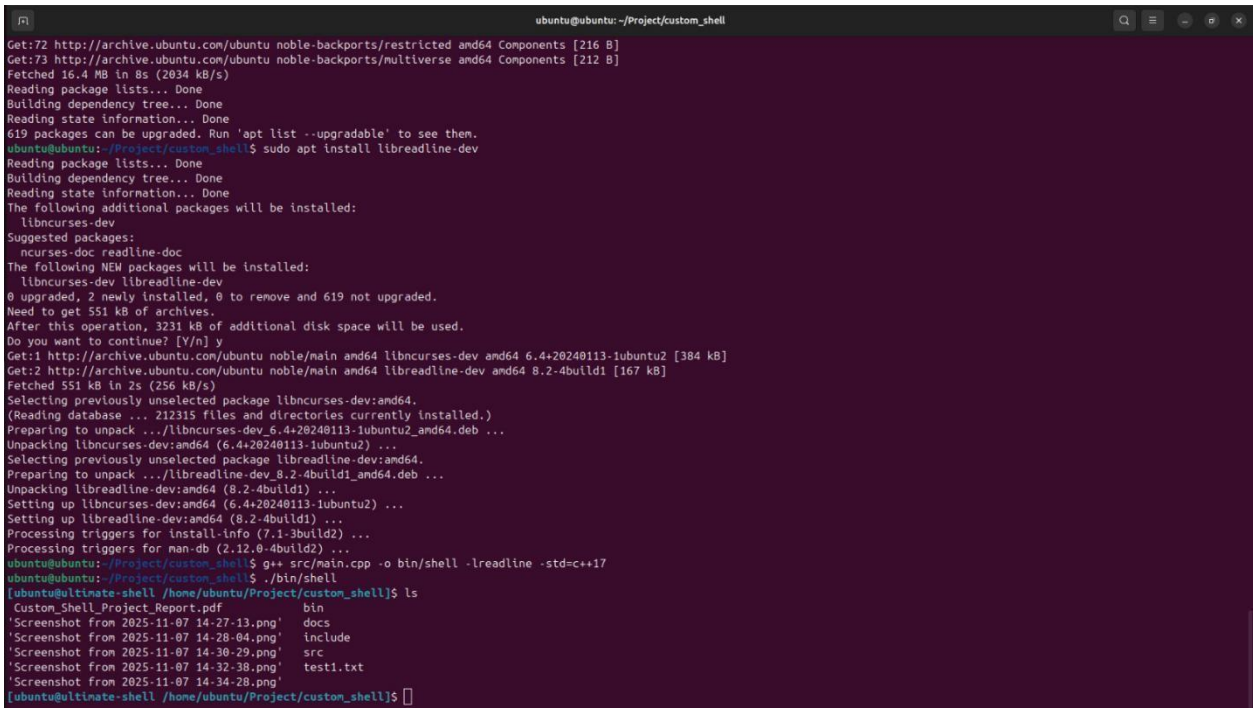
```
    if (segments.size() == 1) {          auto toks =
tokenize_space(segments[0]);          if (toks.empty())
continue;          if ((toks[0] == "rhino") || (toks[0] ==
"xsmax")) {              show_easter_egg(toks[0]);
continue;
        }
        if          (is_builtin_name(toks[0]))          {
std::vector<std::string> words;              for (auto
&t          :          toks)          words.push_back(t);
builtin_execute(words);              continue;
        }
    }

    std::vector<Command> commands;
    bool parse_failed = false;
for (auto &seg : segments) {
        Command c = parse_command_segment(seg);          if (!c.argv.size() || c.argv[0] == nullptr)
{ parse_failed = true; free_command_args(c); break; }          commands.push_back(c);
    }
    if (parse_failed) { std::cerr << "Parse error\n"; for (auto &c : commands) free_command_args(c); continue; }

    // Special-case: single command is builtin with redirection        if
(commands.size() == 1) {          std::string cmdname = commands[0].argv[0] ?
commands[0].argv[0] : "";          if (is_builtin_name(cmdname)) {              int
saved_stdin = -1, saved_stdout = -1;              bool redirected = false;              if
(!commands[0].infile.empty()) {                  saved_stdin = dup(STDIN_FILENO);
int fd = open(commands[0].infile.c_str(), O_RDONLY);                  if (fd >= 0) {
dup2(fd, STDIN_FILENO); close(fd); redirected = true; }
```

```
            }
            if (!commands[0].outfile.empty()) {
               saved_stdout = dup(STDOUT_FILENO);                int flags = O_WRONLY | O_CREAT
| (commands[0].append ? O_APPEND : O_TRUNC);                int fd =
open(commands[0].outfile.c_str(), flags, 0644);                if (fd >= 0) { dup2(fd,
STDOUT_FILENO); close(fd); redirected = true; }
            }
            std::vector<std::string> words;                for (auto p : commands[0].argv) { if (!p) break;
words.push_back(std::string(p)); }                builtin_execute(words);                if (redirected) {
if (saved_stdin != -1) { dup2(saved_stdin, STDIN_FILENO); close(saved_stdin); }                if
(saved_stdout != -1) { dup2(saved_stdout, STDOUT_FILENO); close(saved_stdout); }
            }
            for (auto &c : commands) free_command_args(c);
            continue;
         }
      }

      launch_pipeline(commands, background, line);

      for (auto &c : commands) free_command_args(c);
remove_finished_jobs();
   }

   return 0;
}
```

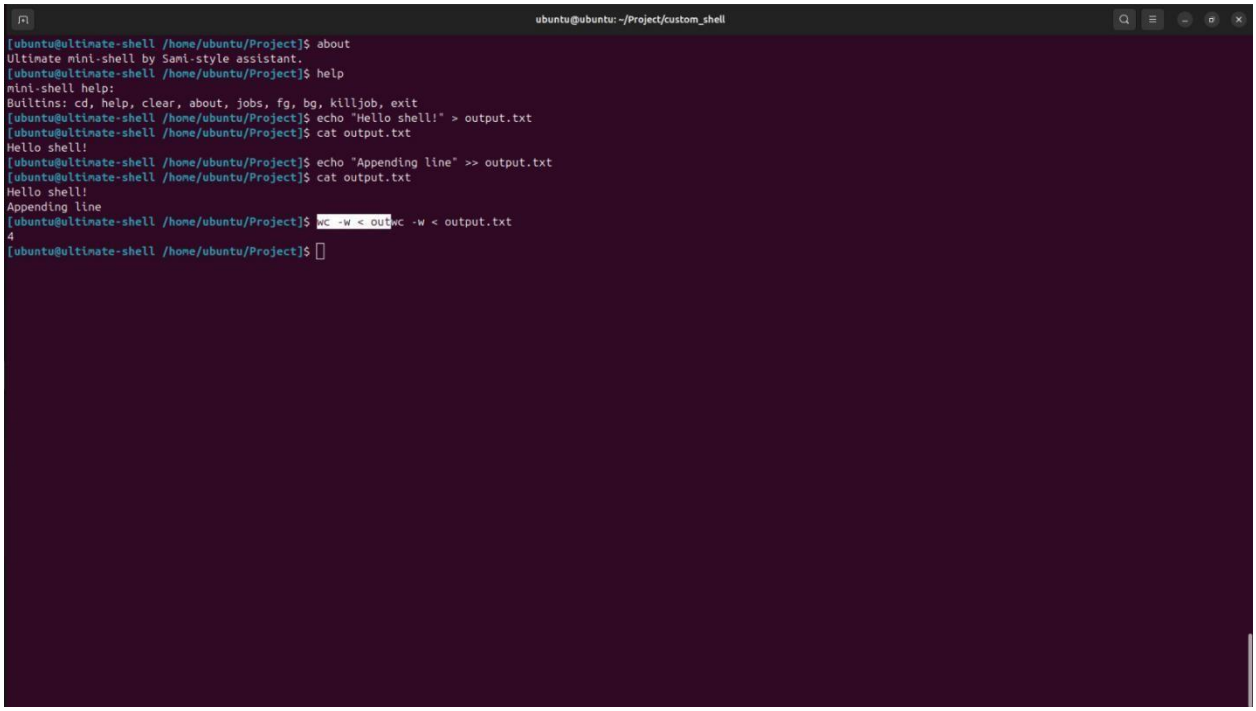
4. Screenshots

Below are the screenshots showing compilation, execution, built-in commands, redirection, background job handling, and Easter eggs.

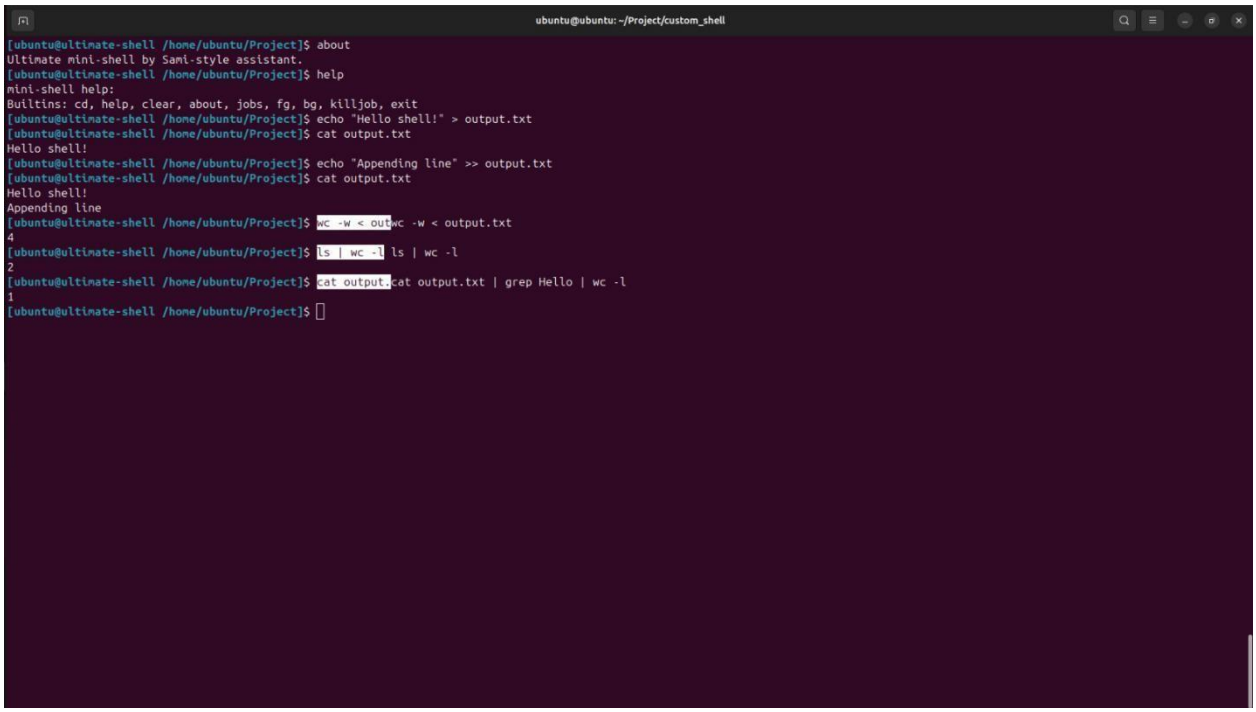
1.Initialization and normal commands



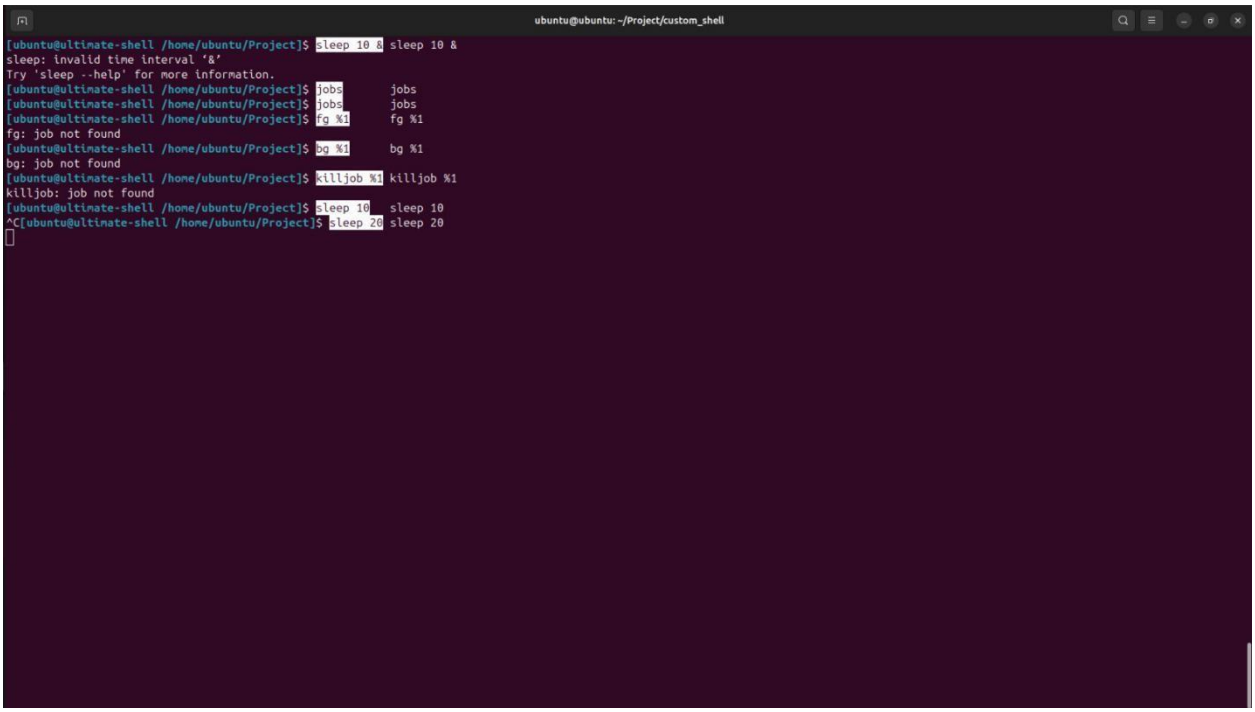
2.help and about



3.fork() and Pipeline



4. jobsandscheduling



5. Conclusion

The Custom Shell project successfully replicates core features of a Linux shell, providing a robust understanding of process management, inter-process communication, and system-level programming in C++. It serves as a strong demonstration of practical OS-level concepts and can be further expanded with scripting support, aliasing, and improved command parsing.