Assignment-2-LinuxOS-Custom-Shell-Implementation

Code:

```
#include <windows.h>
#include <bits/stdc++.h>
using namespace std;

vector<string> history;
const int HISTORY_LIMIT = 100;

struct Job {
    DWORD pid;
    PROCESS_INFORMATION pi; // keeps handles until we close them
    string cmd;
    bool done;
    DWORD exit_code;
};

vector<Job> jobs;

// Tokenizer: splits by whitespace and makes |, <, >, & separate tokens
vector<string> tokenize(const string &line) {
    vector<string> tokens;
    string cur;
    for (size_t i = 0; i < line.size(); ++i) {
        char c = line[i];
        if (isspace((unsigned char)c)) {
            if (!cur.empty()) { tokens.push_back(cur); cur.clear(); }
        } else if (c == '|' || c == '<' || c == '>' || c == '&') {
            if (!cur.empty()) { tokens.push_back(cur); cur.clear(); }
```

```
            tokens.push_back(string(1, c));
        } else {
            cur.push_back(c);
        }
    }
    if (!cur.empty()) tokens.push_back(cur);
    return tokens;
}

// helper: join tokens into a single command string
string join_tokens(const vector<string> &tokens, size_t start=0, size_t end_n=string::npos) {
    if (end_n == string::npos) end_n = tokens.size();
    string s;
    for (size_t i = start; i < end_n && i < tokens.size(); ++i) {
        if (!s.empty()) s += " ";
        s += tokens[i];
    }
    return s;
}

// Check job status: if a job finished, update its 'done' and close handles
void refresh_jobs_status() {
    for (auto &job : jobs) {
        if (job.done) continue;
        if (job.pi.hProcess == NULL) {
            job.done = true;
            continue;
        }
        DWORD res = WaitForSingleObject(job.pi.hProcess, 0);
        if (res == WAIT_OBJECT_0) {
            // finished
            DWORD code = 0;
            GetExitCodeProcess(job.pi.hProcess, &code);
```

```
            job.exit_code = code;
            job.done = true;
            if (job.pi.hProcess) { CloseHandle(job.pi.hProcess); job.pi.hProcess = NULL; }
            if (job.pi.hThread)  { CloseHandle(job.pi.hThread);  job.pi.hThread  = NULL; }
        }
    }
}

// Add a background job (stores PROCESS_INFORMATION as copy)
void add_job(const PROCESS_INFORMATION &pi, const string &cmd) {
    Job j;
    j.pid = pi.dwProcessId;
    j.pi = pi; // struct copy; handles remain valid
    j.cmd = cmd;
    j.done = false;
    j.exit_code = 0;
    jobs.push_back(j);
}

// Print jobs list
void builtin_jobs() {
    refresh_jobs_status();
    if (jobs.empty()) {
        cout << "(no background jobs)\n";
        return;
    }
    for (size_t i = 0; i < jobs.size(); ++i) {
        auto &j = jobs[i];
        cout << "[" << i+1 << "] ";
        cout << (j.done ? "Done    " : "Running ");
        cout << "PID:" << j.pid << "  ";
        cout << j.cmd;
        if (j.done) cout << "  (exit " << j.exit_code << ")";
```

```
        cout << "\n";
    }
}

// Bring a job to foreground (wait for it). jobIndex is 1-based
void builtin_fg(size_t jobIndex) {
    if (jobIndex == 0 || jobIndex > jobs.size()) {
        cerr << "fg: invalid job id\n";
        return;
    }
    refresh_jobs_status();
    Job &j = jobs[jobIndex-1];
    if (j.done) {
        cout << "fg: job " << jobIndex << " already finished (exit " << j.exit_code << ")\n";
        return;
    }
    if (j.pi.hProcess == NULL) {
        cerr << "fg: internal error (no handle)\n";
        return;
    }
    cout << "Bringing job [" << jobIndex << "] PID " << j.pid << " to foreground: " << j.cmd << "\n";
    // Wait until it finishes
    WaitForSingleObject(j.pi.hProcess, INFINITE);
    DWORD code = 0;
    GetExitCodeProcess(j.pi.hProcess, &code);
    j.exit_code = code;
    j.done = true;
    if (j.pi.hProcess) { CloseHandle(j.pi.hProcess); j.pi.hProcess = NULL; }
    if (j.pi.hThread)  { CloseHandle(j.pi.hThread);  j.pi.hThread  = NULL; }
    cout << "Job [" << jobIndex << "] finished with exit code " << j.exit_code << "\n";
}

// Simple bg: mostly informational for Windows version. If job already running, report; if finished, cannot bg.
```

```
void builtin_bg(size_t jobIndex) {
    if (jobIndex == 0 || jobIndex > jobs.size()) {
        cerr << "bg: invalid job id\n";
        return;
    }
    refresh_jobs_status();
    Job &j = jobs[jobIndex-1];
    if (j.done) {
        cerr << "bg: job " << jobIndex << " already finished\n";
        return;
    }
    // On Windows we don't have SIGCONT for general processes in the same way.
    // If the process is running, just report it and keep it in background.
    cout << "Job [" << jobIndex << "] PID " << j.pid << " is running in background: " << j.cmd << "\n";
}

// Run an external command using CreateProcessA. This version supports:
// - optional input file (inFile) for STDIN
// - optional output file (outFile) for STDOUT/STDERR
// - optionally run in background (background == true) -> don't wait and add to jobs
// Returns true on success (process created), false otherwise.
bool launch_process(const string &cmdline, const string &inFile, const string &outFile, bool background,
PROCESS_INFORMATION *outPI = nullptr) {
    // Prepare SECURITY_ATTRIBUTES for inheritable handles
    SECURITY_ATTRIBUTES sa;
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE;

    // Prepare STARTUPINFO
    STARTUPINFOA si;
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
```

```
    si.dwFlags = STARTF_USESTDHANDLES;

    HANDLE hIn = INVALID_HANDLE_VALUE;
    HANDLE hOut = INVALID_HANDLE_VALUE;

    // Input redirection
    if (!inFile.empty()) {
        hIn = CreateFileA(inFile.c_str(), GENERIC_READ, FILE_SHARE_READ, &sa, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
        if (hIn == INVALID_HANDLE_VALUE) {
            cerr << "Error: cannot open input file '" << inFile << "'.\n";
            return false;
        }
        si.hStdInput = hIn;
    } else {
        si.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
    }

    // Output redirection
    if (!outFile.empty()) {
        hOut = CreateFileA(outFile.c_str(), GENERIC_WRITE, FILE_SHARE_READ, &sa, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
        if (hOut == INVALID_HANDLE_VALUE) {
            cerr << "Error: cannot open output file '" << outFile << "'.\n";
            if (hIn != INVALID_HANDLE_VALUE) CloseHandle(hIn);
            return false;
        }
        si.hStdOutput = hOut;
        si.hStdError  = hOut;
    } else {
        si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
        si.hStdError  = GetStdHandle(STD_ERROR_HANDLE);
    }
```

```
    // Prepare modifiable command line buffer (CreateProcessA may modify it)
    vector<char> cmdBuf(cmdline.begin(), cmdline.end());
    cmdBuf.push_back('\0');

    PROCESS_INFORMATION pi;
    ZeroMemory(&pi, sizeof(pi));

    BOOL ok = CreateProcessA(
        NULL,
        cmdBuf.data(),
        NULL,
        NULL,
        TRUE,    // bInheritHandles -> TRUE so child gets handles we set
        0,
        NULL,
        NULL,
        &si,
        &pi
    );

    if (!ok) {
        cerr << "Error: CreateProcess failed for '" << cmdline << "'. Code: " << GetLastError() << "\n";
        if (hIn  != INVALID_HANDLE_VALUE) CloseHandle(hIn);
        if (hOut != INVALID_HANDLE_VALUE) CloseHandle(hOut);
        return false;
    }

    // If caller wants the PROCESS_INFORMATION, copy it out; otherwise, if background keep it; if foreground,
wait and close
    if (background) {
        if (outPI) *outPI = pi; // caller will manage handles
        else {
            // default: if not returning pi to caller, store it in jobs via add_job
```

```
            add_job(pi, cmdline);
        }
        // Close our copies? NO -- we must keep process/thread handles valid for later wait, so we DO NOT close handles here.
        // If we returned pi to caller, they will own/close handles.
    } else {
        // Foreground: wait until finished, then close handles
        WaitForSingleObject(pi.hProcess, INFINITE);
        DWORD exitcode = 0;
        GetExitCodeProcess(pi.hProcess, &exitcode);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }

    // Close the redirection handles in parent (they are duplicated/inherited by child)
    if (hIn  != INVALID_HANDLE_VALUE) CloseHandle(hIn);
    if (hOut != INVALID_HANDLE_VALUE) CloseHandle(hOut);

    return true;
}

// Main runCommand that handles tokens: piping, redirection, bg, builtins
void runCommand(vector<string> tokens) {
    if (tokens.empty()) return;

    // Check for builtins first: exit, history, jobs, fg, bg, cd, help, cls
    string first = tokens[0];

    if (first == "exit") {
        cout << "Exiting shell...\n";
        exit(0);
    }
```

```
if (first == "history") {
    for (size_t i = 0; i < history.size(); ++i) {
        cout << (i+1) << "  " << history[i] << "\n";
    }
    return;
}

if (first == "jobs") {
    builtin_jobs();
    return;
}

if (first == "fg") {
    if (tokens.size() < 2) { cerr << "Usage: fg <job_number>\n"; return; }
    size_t jid = stoul(tokens[1]);
    builtin_fg(jid);
    return;
}

if (first == "bg") {
    if (tokens.size() < 2) { cerr << "Usage: bg <job_number>\n"; return; }
    size_t jid = stoul(tokens[1]);
    builtin_bg(jid);
    return;
}

if (first == "cd") {
    if (tokens.size() < 2) { cerr << "Usage: cd <dir>\n"; return; }
    if (!SetCurrentDirectoryA(tokens[1].c_str())) {
        cerr << "cd: cannot change directory to " << tokens[1] << "\n";
    }
    return;
}
```

```
if (first == "cls") {
    system("cls");
    return;
}

if (first == "help") {
    cout << "Available commands:\n";
    cout << "  cd <dir>, cls, help, exit\n";
    cout << "  history, jobs, fg <n>, bg <n>\n";
    cout << "  Use '&' to run background jobs, '>' and '<' for redirection, '|' for piping\n";
    return;
}

// detect background '&' (if last token)
bool background = false;
if (!tokens.empty() && tokens.back() == "&") {
    background = true;
    tokens.pop_back();
}

// detect pipe
auto itPipe = find(tokens.begin(), tokens.end(), "|");
bool hasPipe = (itPipe != tokens.end());

if (!hasPipe) {
    // Handle simple command with possible < and > tokens
    string inFile, outFile;
    vector<string> cmdTokens;
    for (size_t i = 0; i < tokens.size(); ++i) {
        if (tokens[i] == "<") {
            if (i+1 < tokens.size()) { inFile = tokens[i+1]; i++; }
            else { cerr << "Syntax error: no input file\n"; return; }
```

```
        } else if (tokens[i] == ">") {
            if (i+1 < tokens.size()) { outFile = tokens[i+1]; i++; }
            else { cerr << "Syntax error: no output file\n"; return; }
        } else {
            cmdTokens.push_back(tokens[i]);
        }
    }
    if (cmdTokens.empty()) return;
    string cmdline = "cmd.exe /C " + join_tokens(cmdTokens);
    if (background) {
        // create process and add to jobs
        PROCESS_INFORMATION pi;
        ZeroMemory(&pi, sizeof(pi));
        // Use launch_process to create and fill pi
        // We want to store pi in jobs, so pass outPI
        bool ok = launch_process(cmdline, inFile, outFile, true, &pi);
        if (!ok) {
            cerr << "Failed to start background job\n";
            return;
        }
        // If launch_process returned true and filled pi, we must add to jobs (it didn't add since outPI provided)
        add_job(pi, cmdline);
        cout << "Started background job [" << jobs.size() << "] PID " << pi.dwProcessId << "\n";
    } else {
        launch_process(cmdline, inFile, outFile, false, nullptr);
    }
    return;
}

// If we have a pipe — only handle single pipe (left | right) for simplicity
// split tokens
vector<string> leftTok(tokens.begin(), itPipe);
vector<string> rightTok(itPipe + 1, tokens.end());
```

```
// handle redirection tokens inside left/right separately
string leftIn, leftOut, rightIn, rightOut; // we'll only treat < and > inside each side
auto process_side = [&](vector<string> sideTok, string &inF, string &outF, vector<string> &cmdTok) {
   cmdTok.clear();
   for (size_t i = 0; i < sideTok.size(); ++i) {
      if (sideTok[i] == "<") {
         if (i+1 < sideTok.size()) { inF = sideTok[i+1]; i++; }
      } else if (sideTok[i] == ">") {
         if (i+1 < sideTok.size()) { outF = sideTok[i+1]; i++; }
      } else cmdTok.push_back(sideTok[i]);
   }
};
vector<string> leftCmdTok, rightCmdTok;
process_side(leftTok, leftIn, leftOut, leftCmdTok);
process_side(rightTok, rightIn, rightOut, rightCmdTok);
if (leftCmdTok.empty() || rightCmdTok.empty()) {
   cerr << "Syntax error: invalid pipe command\n";
   return;
}

// Create anonymous pipe
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE hRead = NULL, hWrite = NULL;
if (!CreatePipe(&hRead, &hWrite, &sa, 0)) {
   cerr << "Error: CreatePipe failed\n";
   return;
}

// --- Launch left command (stdout -> pipe write)
string leftCmdline = "cmd.exe /C " + join_tokens(leftCmdTok);
```

```
    STARTUPINFOA siLeft;
    ZeroMemory(&siLeft, sizeof(siLeft));
    siLeft.cb = sizeof(siLeft);
    siLeft.dwFlags = STARTF_USESTDHANDLES;
    // left: stdin from leftIn if provided, else parent's stdin
    if (!leftIn.empty()) {
        // open leftIn
        HANDLE hLeftIn = CreateFileA(leftIn.c_str(), GENERIC_READ, FILE_SHARE_READ, &sa, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
        if (hLeftIn == INVALID_HANDLE_VALUE) { cerr << "Error: cannot open " << leftIn << "\n";
CloseHandle(hRead); CloseHandle(hWrite); return; }
        siLeft.hStdInput = hLeftIn;
        // we'll close hLeftIn after CreateProcess
        // ensure inherits
    } else {
        siLeft.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
    }
    // left stdout -> pipe write
    siLeft.hStdOutput = hWrite;
    siLeft.hStdError  = hWrite;

    PROCESS_INFORMATION piLeft;
    ZeroMemory(&piLeft, sizeof(piLeft));
    // create left process
    {
        vector<char> buf(leftCmdline.begin(), leftCmdline.end());
        buf.push_back('\0');
        BOOL ok = CreateProcessA(NULL, buf.data(), NULL, NULL, TRUE, 0, NULL, NULL, &siLeft, &piLeft);
        // if we opened leftIn file, close its handle in parent now
        if (!leftIn.empty() && siLeft.hStdInput && siLeft.hStdInput != GetStdHandle(STD_INPUT_HANDLE)) {
            CloseHandle(siLeft.hStdInput);
        }
        if (!ok) {
            cerr << "Error: failed to create left process. Code: " << GetLastError() << "\n";
```

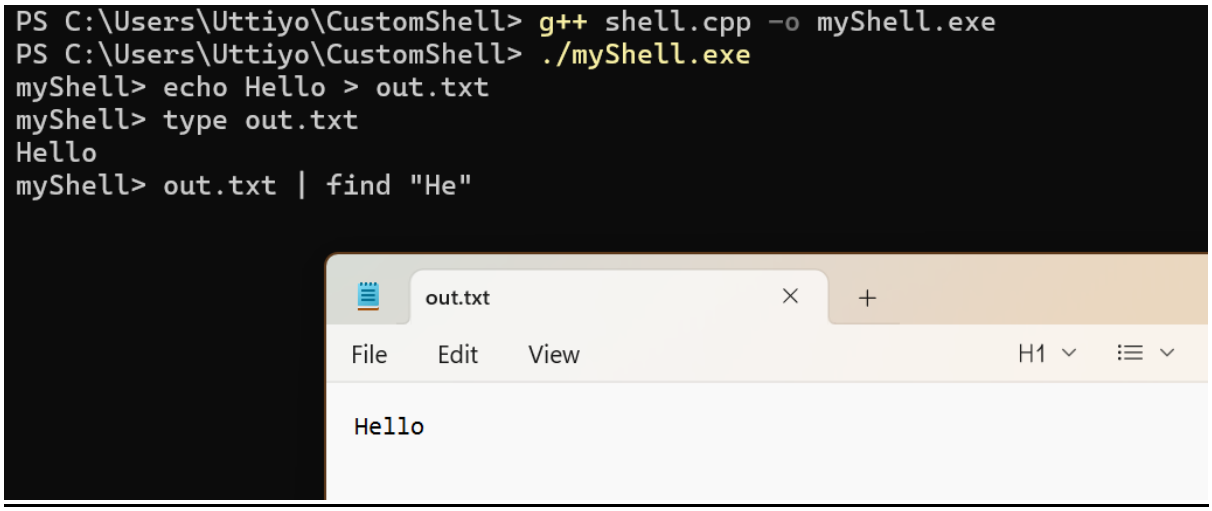
```
        CloseHandle(hRead); CloseHandle(hWrite);
        return;
    }
  }

  // --- Launch right command (stdin <- pipe read)
  string rightCmdline = "cmd.exe /C " + join_tokens(rightCmdTok);
  STARTUPINFOA siRight;
  ZeroMemory(&siRight, sizeof(siRight));
  siRight.cb = sizeof(siRight);
  siRight.dwFlags = STARTF_USESTDHANDLES;
  // right: stdin from pipe read
  siRight.hStdInput = hRead;
  // right: stdout to rightOut if exists, else parent's stdout
  if (!rightOut.empty()) {
    HANDLE hRightOut = CreateFileA(rightOut.c_str(), GENERIC_WRITE, FILE_SHARE_READ, &sa,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hRightOut == INVALID_HANDLE_VALUE) {
      cerr << "Error: cannot open " << rightOut << "\n";
      TerminateProcess(piLeft.hProcess, 1);
      CloseHandle(hRead); CloseHandle(hWrite);
      CloseHandle(piLeft.hProcess); CloseHandle(piLeft.hThread);
      return;
    }
    siRight.hStdOutput = hRightOut;
    siRight.hStdError  = hRightOut;
  } else {
    siRight.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    siRight.hStdError  = GetStdHandle(STD_ERROR_HANDLE);
  }

  PROCESS_INFORMATION piRight;
  ZeroMemory(&piRight, sizeof(piRight));
```

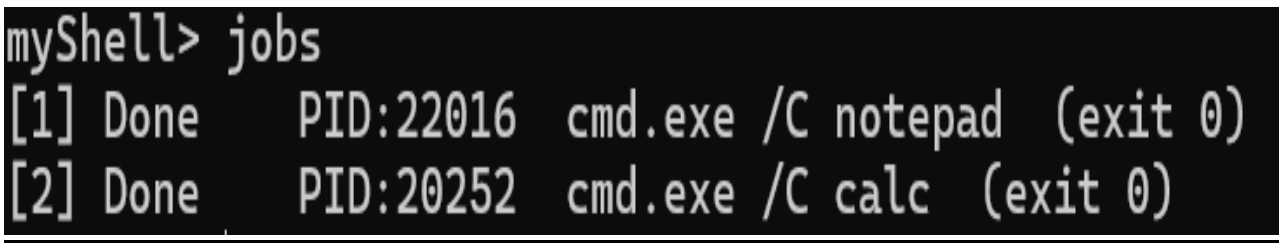
```
    {
        vector<char> buf(rightCmdline.begin(), rightCmdline.end());
        buf.push_back('\0');
        BOOL ok = CreateProcessA(NULL, buf.data(), NULL, NULL, TRUE, 0, NULL, NULL, &siRight, &piRight);
        if (!ok) {
            cerr << "Error: failed to create right process. Code: " << GetLastError() << "\n";
            TerminateProcess(piLeft.hProcess, 1);
            CloseHandle(hRead); CloseHandle(hWrite);
            CloseHandle(piLeft.hProcess); CloseHandle(piLeft.hThread);
            return;
        }
    }

    // After creating both processes, close pipe write in parent, close read in parent after waiting
    CloseHandle(hWrite);

    if (background) {
        // Add both processes as separate jobs (simple approach)
        add_job(piLeft, leftCmdline);
        add_job(piRight, rightCmdline);
        cout << "Started background pipeline jobs: [" << jobs.size()-1 << "] PID " << piLeft.dwProcessId
             << " and [" << jobs.size() << "] PID " << piRight.dwProcessId << "\n";
        // parent should not wait; keep handles open in job entries
        CloseHandle(hRead); // child has inherited read; parent can close local copy (child will have an inheritable duplicate)
        return;
    } else {
        // Foreground: wait both
        WaitForSingleObject(piLeft.hProcess, INFINITE);
        WaitForSingleObject(piRight.hProcess, INFINITE);
        // retrieve exit codes (optional)
        DWORD codeL = 0, codeR = 0;
        GetExitCodeProcess(piLeft.hProcess, &codeL);
```

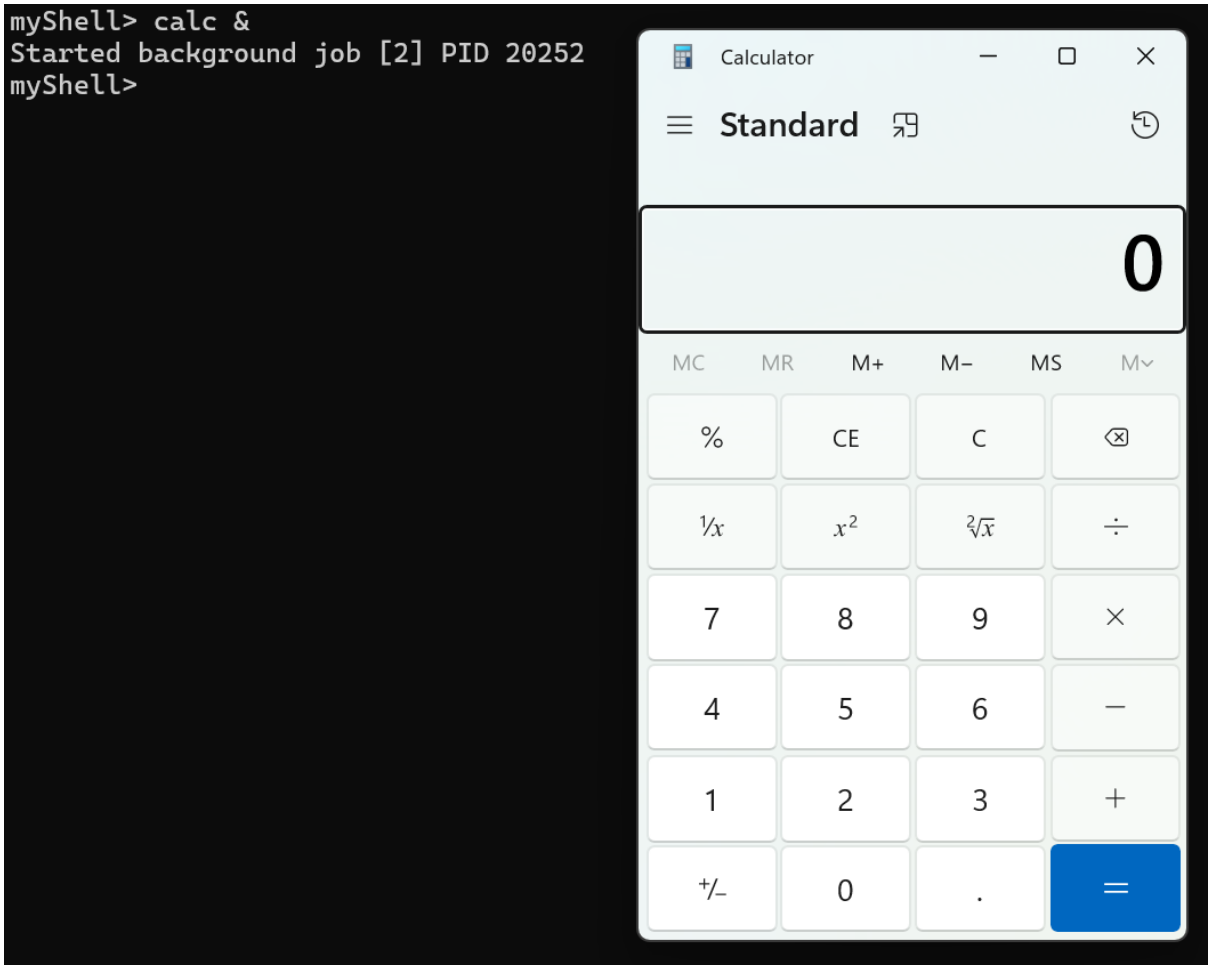
```
        GetExitCodeProcess(piRight.hProcess, &codeR);
        CloseHandle(piLeft.hProcess); CloseHandle(piLeft.hThread);
        CloseHandle(piRight.hProcess); CloseHandle(piRight.hThread);
        CloseHandle(hRead);
        return;
    }
}

int main() {
    string line;
    while (true) {
        cout << "myShell> " << flush;
        if (!getline(cin, line)) break;
        if (line.empty()) continue;

        // store in history
        history.push_back(line);
        if (history.size() > HISTORY_LIMIT) history.erase(history.begin());

        // If user wants to repeat a history entry like "!3" -> optional: implement simple support
        if (line.size() > 1 && line[0] == '!' && isdigit((unsigned char)line[1])) {
            // parse number
            size_t idx = stoi(line.substr(1));
            if (idx >= 1 && idx <= history.size()) {
                line = history[idx-1];
                cout << line << "\n";
            } else {
                cerr << "No such history entry\n";
                continue;
            }
        }
```

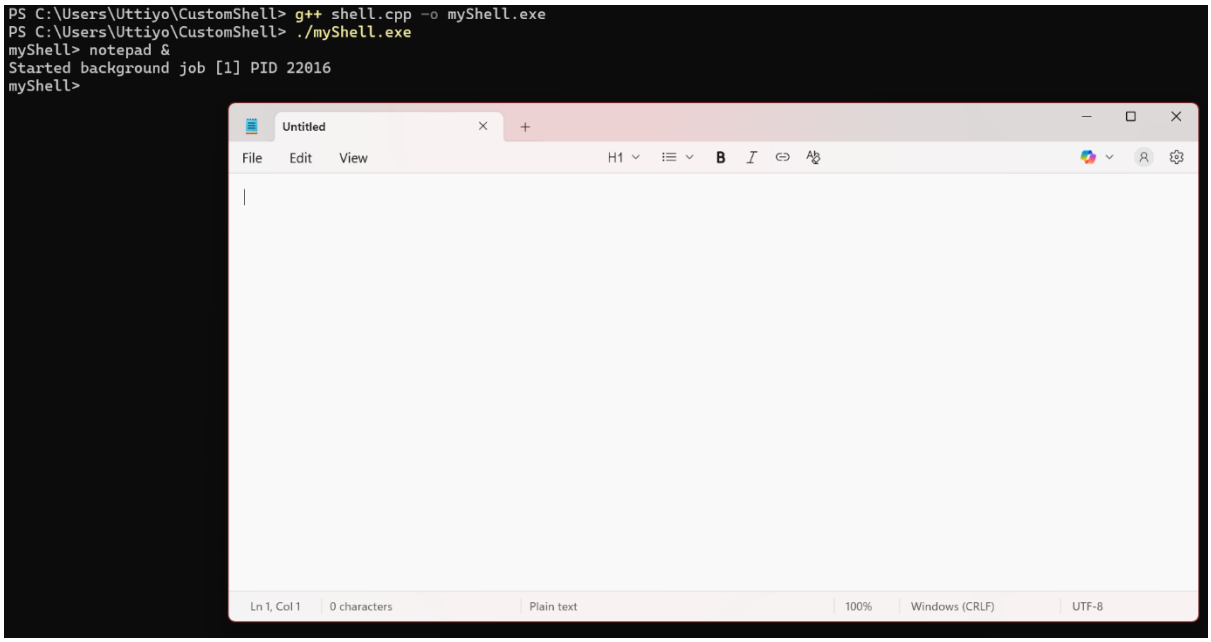
```
        // built-in quick check for "history" (we also handle it inside runCommand, but doing early allows direct printing)
        if (line == "history") {
            for (size_t i = 0; i < history.size(); ++i) {
                cout << (i+1) << "  " << history[i] << "\n";
            }
            continue;
        }

        auto tokens = tokenize(line);
        runCommand(tokens);

        // refresh job statuses after each command prompt
        refresh_jobs_status();
    }
    return 0;
}
```

Screenshots:






myShell> calc &
Started background job [2] PID 20252
myShell>
Calculator
Standard
0
MC MR M+ M− MS M
% CE C
1/x x² ²√x ÷
7 8 9 ×
4 5 6 −
1 2 3 +
+/− 0 . =


PS C:\Users\Uttiyo\CustomShell> g++ shell.cpp -o myShell.exe
PS C:\Users\Uttiyo\CustomShell> ./myShell.exe
myShell> notepad &
Started background job [1] PID 22016
myShell>
Untitled
File Edit View
H1
Ln 1, Col 1 0 characters Plain text 100% Windows (CRLF) UTF-8

```
PS C:\Users\Uttiyo\CustomShell> g++ shell.cpp -o myShell.exe
PS C:\Users\Uttiyo\CustomShell> ./myShell.exe
myShell> echo Test1
Test1
myShell> echo Test2
Test2
myShell> history
1  echo Test1
2  echo Test2
3  history
myShell>
```

```
PS C:\Users\Uttiyo\CustomShell> g++ shell.cpp -o myShell.exe
PS C:\Users\Uttiyo\CustomShell> ./myShell.exe
myShell> dir
 Volume in drive C is Windows
 Volume Serial Number is 666A-2AE9

 Directory of C:\Users\Uttiyo\CustomShell

08-11-2025  00:22    <DIR>          .
07-11-2025  23:52    <DIR>          ..
06-11-2025  20:07                30 input.txt
08-11-2025  00:22           228,789 myShell.exe
06-11-2025  21:13                41 out.txt
06-11-2025  20:08                14 output.txt
06-11-2025  21:04            19,668 shell.cpp
               5 File(s)        248,542 bytes
               2 Dir(s)  82,677,055,488 bytes free
myShell> echo  Hellow World
Hellow World
myShell> cd ..
myShell> exit
Exiting shell...
PS C:\Users\Uttiyo\CustomShell>
```

```
PS C:\Users\Uttiyo\CustomShell> g++ shell.cpp -o myShell.exe
PS C:\Users\Uttiyo\CustomShell> ./myShell.exe
myShell> notepad &
Started background job [1] PID 12348
myShell> calc &
Started background job [2] PID 28644
myShell> jobs
[1] Done    PID:12348  cmd.exe /C notepad  (exit 0)
[2] Done    PID:28644  cmd.exe /C calc  (exit 0)
myShell> fg 1
fg: job 1 already finished (exit 0)
myShell>
```