

**UNIVERSIDAD NACIONAL JORGE BASADRE GROHMANN
FACULTAD DE INGENIERÍA**

ESCUELA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS



“Implementación de un intérprete de comandos en c++ sobre linux”

Asignatura:

Sistemas Operativos

Docente:

MSc. Hugo Manuel Barraza Vizcarra

Estudiantes:

- Jhon William Mamani Condori 2020-119018
- Juan Luis Mamani Mullo 2021-119123

Tacna - Perú

2025

1. Objetivos y alcance

1.1. Objetivo general

El presente proyecto tiene como propósito desarrollar una mini-shell concurrente en C++, que replique las funciones esenciales de un intérprete de comandos del sistema operativo Linux, aplicando los principios de procesos, concurrencia, sincronización, redirección de E/S y manejo de memoria.

El proyecto forma parte de la evaluación final del curso, y busca que el estudiante comprenda e implemente el funcionamiento interno de un shell, integrando conceptos de programación del sistema, llamadas POSIX, y sincronización entre procesos e hilos.

1.2. Objetivo específico

- Implementar un intérprete interactivo que muestre un *prompt* personalizado (mini-shell>) y acepte comandos del usuario.
- Soportar comandos internos como `cd`, `pwd`, `help`, `history`, `jobs`, `meminfo`, `alias` y `salir`.
- Permitir la ejecución de programas externos, usando las llamadas al sistema `fork()`, `execv()` y `waitpid()`.
- Incorporar la redirección de entrada y salida estándar (`<`, `>`, `>>`).
- Implementar pipes simples (`cmd1 | cmd2`) para conectar procesos.
- Soportar la ejecución en segundo plano (`&`) y la gestión de *jobs* activos.
- Implementar un sistema básico de concurrencia con hilos (`pthread`) para ejecutar comandos en paralelo.
- Monitorear la gestión de memoria dinámica, evitando fugas y pérdidas de referencias.
- Diseñar una arquitectura modular basada en cabeceras (`.h`) y fuentes (`.cpp`).
- Documentar el diseño, implementación, pruebas y resultados.

1.3. Alcance

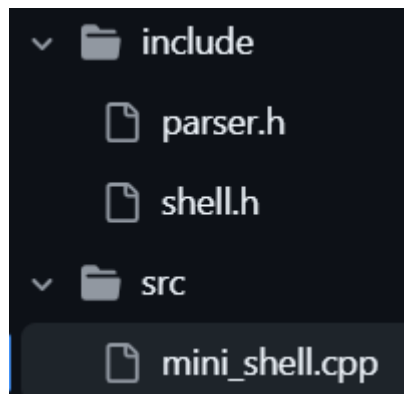
La mini-shell ejecuta comandos del sistema, maneja pipes simples, permite ejecución en background, redirección de archivos y comandos paralelos con hilos.

Se excluyen características avanzadas como autocompletado, scripting, múltiples pipes encadenados o sustitución de variables, aunque el diseño modular permite extender fácilmente estas funcionalidades en versiones futuras.

2. Arquitectura y diseño

2.1. Estructura del proyecto

El proyecto sigue una estructura modular y clara:



2.2. Diseño general

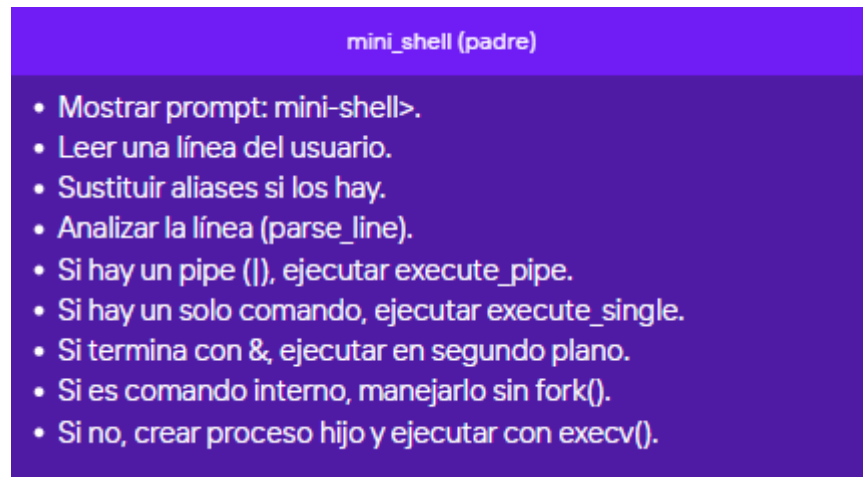
El diseño se basa en una arquitectura modular, donde:

- **mini_shell.cpp:** Es el núcleo del sistema. Implementa el bucle principal, el parser, la ejecución de procesos, el control de memoria, la gestión de hilos, señales y la administración de procesos en segundo plano.
- **shell.h:** Define las estructuras de datos `Command` y `ParsedLine`, las funciones para ejecutar comandos (`execute_single`, `execute_pipe`), el manejo de señales, y el control del prompt.
- **parser.h:** Contiene la declaración del analizador (`parse_line`) que transforma una línea de texto en una estructura `ParsedLine`, separando argumentos, redirecciones y operadores (`|`, `&`).

El flujo general es:

Usuario -> Prompt -> parser -> ejecutor -> Resultados

2.3. Diagrama lógico de flujo



El diseño separa claramente la lógica de interpretación, ejecución y concurrencia, favoreciendo la extensibilidad y el mantenimiento del código.

3. Detalles de implementación

La implementación de la mini-shell se desarrolló en lenguaje **C++** bajo entorno **POSIX**, utilizando llamadas de sistema clásicas como `fork()`, `execvp()`, `waitpid()`, `pipe()`, `dup2()`, `open()` y `close()`.

Esta arquitectura modular favorece la **legibilidad**, **mantenibilidad** y la **extensión del proyecto**, permitiendo añadir nuevas características sin afectar el núcleo del shell.

3.1. Inclusión de librerías y cabeceras

El código comienza incluyendo las librerías estándar necesarias:

```

1  #include <bits/stdc++.h>
2  #include <sys/types.h>
3  #include <sys/wait.h>
4  #include <sys/resource.h>
5  #include <fcntl.h>
6  #include <unistd.h>
7  #include <signal.h>
8  #include <errno.h>
9  #include <pthread.h>
10
11 #include "shell.h"
12 #include "parser.h"

```

Cada inclusión tiene una función específica:

- iostream y string permiten el uso de flujos y manipulación de cadenas en C++.
- unistd.h aporta las funciones del sistema UNIX como fork(), execvp(), pipe(), chdir(), etc.
- sys/types.h y sys/wait.h se usan para gestionar procesos e hijos mediante pid_t y waitpid().
- fcntl.h permite abrir y crear archivos para redirecciones (>, >>, <).
- csignal se emplea para manejar señales del sistema, como SIGINT.
- pthread.h permite crear y sincronizar **hilos** (para el comando parallel).
- map y vector facilitan la implementación de estructuras dinámicas como la **historia de comandos** o los **alias**.

3.2. Variables globales: historial y alias

Después de using namespace std; se declaran dos variables globales:

```
14     using namespace std;
15
16     vector<string> command_history;
17     map<string, string> aliases;
18
```

command_history almacena cada línea introducida por el usuario para el built-in history. aliases mantiene pares nombre -> sustitución para permitir alias simples (por ejemplo alias ll='ls -l'). Estas estructuras globales facilitan el acceso desde distintas funciones (built-ins y main) y son adecuadas para el alcance del intérprete; si el proyecto creciera se podrían encapsular en una clase para mejorar modularidad.

3.3. Instrumentación simple de memoria

El programa instrumenta asignaciones dinámicas para poder mostrar estadísticas con el comando meminfo. Esto se hace con un contador global y wrappers de malloc/free:

```

21     static size_t g_alloc_count = 0;
22
23     void* shell_malloc(size_t s) {
24         void* p = malloc(s);
25         if (p) __sync_add_and_fetch(&g_alloc_count, 1);
26         return p;
27     }
28     void shell_free(void* p) {
29         if (p) {
30             __sync_sub_and_fetch(&g_alloc_count, 1);
31             free(p);
32         }
33     }
34     size_t shell_alloc_count() { return g_alloc_count; }
35
36     char* shell_strdup(const char* s) {
37         if (!s) return nullptr;
38         size_t n = strlen(s) + 1;
39         char* p = (char*)shell_malloc(n);
40         if (!p) return nullptr;
41         memcpy(p, s, n);
42         return p;
43     }
44

```

g_alloc_count guarda el número aproximado de asignaciones activas hechas a través de shell_malloc. shell_malloc invoca malloc y, si el puntero no es nulo, incrementa el contador de forma atómica con __sync_add_and_fetch. shell_free decrementa atómicamente y llama a free. shell_strdup copia cadenas usando shell_malloc para que estas copias queden contabilizadas. Esta instrumentación permite implementar meminfo, detectar fugas durante pruebas y es útil en ambiente académico. Observación: el contador solo refleja las asignaciones realizadas con estas funciones; cualquier new/malloc externo no se contabilizará.

3.4. Tokenizer (tokenize)

La función tokenize divide una cadena en tokens separados por espacios:

```

46  vector<string> tokenize(const string& s) {
47      vector<string> out;
48      istringstream iss(s);
49      string t;
50      while (iss >> t) out.push_back(t);
51      return out;
52  }

```

Usa istringstream y el operador >> para separar por espacios y tabulaciones. Es un tokenizador muy simple que **no** maneja comillas ni escapes — pero corresponde a la especificación del trabajo (tokens separados por espacios). Si se desea admitir argumentos con espacios entre comillas habría que implementar un parser más sofisticado.

3.5. Parser simple (parse_line)

```

55  ParsedLine parse_line(const string& line) {
56      ParsedLine pl;
57      string ln = line;
58      size_t last = ln.find_last_not_of(" \t");
59      if (last != string::npos && ln[last] == '&') {
60          pl.background = true;
61          ln = ln.substr(0, last);
62      }
63
64      vector<string> parts;
65      size_t pos = 0;
66      while (true) {
67          size_t p = ln.find(' ', pos);
68          if (p == string::npos) { parts.push_back(ln.substr(pos)); break; }
69          parts.push_back(ln.substr(pos, p - pos));
70          pos = p + 1;
71      }

```

```

73      for (auto &seg : parts) {
74          size_t a = seg.find_first_not_of(" \t");
75          if (a == string::npos) continue;
76          size_t b = seg.find_last_not_of(" \t");
77          string s = seg.substr(a, b - a + 1);
78          auto toks = tokenize(s);
79          if (toks.empty()) continue;
80          Command cmd;
81          for (size_t i = 0; i < toks.size(); ++i) {
82              if (toks[i] == "<" && i + 1 < toks.size()) {
83                  cmd.in_file = toks[++i];
84              } else if (toks[i] == ">>" && i + 1 < toks.size()) {
85                  cmd.out_file = toks[++i];
86                  cmd.append_out = true;
87              } else if (toks[i] == ">" && i + 1 < toks.size()) {
88                  cmd.out_file = toks[++i];
89                  cmd.append_out = false;
90              } else {
91                  cmd.args.push_back(toks[i]);
92              }
93          }
94          pl.cmds.push_back(move(cmd));
95      }
96
97      return pl;
98  }

```


parse_line transforma la línea completa en una estructura ParsedLine (lista de Command, bandera background):

- Primero detecta & final para marcar ejecución en segundo plano y lo elimina de la línea.
- Después divide la línea por | en parts para manejar pipes simples.
- Para cada segmento recorta espacios, lo tokeniza y reconoce redirecciones <, >, >> asignando in_file, out_file y append_out.
- Los tokens restantes se colocan en cmd.args y cada Command se añade a pl.cmds.

Este parser es suficiente para aceptar entradas como cmd arg1 > out.txt & y para comandos con una sola tubería (cmd1 | cmd2). Limitación: no maneja comillas, operadores compuestos (&&, ||) ni redirecciones pegadas (e.g. cmd>file sin espacio).

3.6. Helpers para argv

Antes de llamar a execv es necesario preparar un char** argv compatible con C:

```
101 char** make_argv(const vector<string>& args) {
102     size_t n = args.size();
103     char** argv = (char**)shell_malloc((n + 1) * sizeof(char*));
104     if (!argv) return nullptr;
105     for (size_t i = 0; i < n; i++) argv[i] = shell_strdup(args[i].c_str());
106     argv[n] = nullptr;
107     return argv;
108 }
109
110 void free_argv(char** argv) {
111     if (!argv) return;
112     for (size_t i = 0; argv[i]; ++i) shell_free(argv[i]);
113     shell_free(argv);
114 }
```

make_argv reserva memoria para el array de punteros con shell_malloc (para contabilizar) y duplica cada string con shell_strdup, finalizando con nullptr. free_argv libera cada elemento y el array. Esto proporciona una conversión segura entre std::vector<std::string> y el char*[] que requieren las funciones exec*. Observación: execv reemplaza el proceso en el hijo,

por lo que la liberación solo ocurre en el camino de error; en el padre se debe asegurar liberar cualquier `shell_strdup` o `shell_malloc` que haya sido creado (el código libera `exe_path` en el padre).

3.7. Resolución de ejecutable (`resolve_executable`)

La función `resolve_executable` decide la ruta del ejecutable:

```
117 char* resolve_executable(const string& cmd) {
118     if (cmd.empty()) return nullptr;
119     if (cmd[0] == '/') {
120         if (access(cmd.c_str(), X_OK) == 0) return shell_strdup(cmd.c_str());
121         return nullptr;
122     }
123     string candidate = string("/bin/") + cmd;
124     if (access(candidate.c_str(), X_OK) == 0) return shell_strdup(candidate.c_str());
125     return nullptr;
126 }
```

Si el comando ya es una ruta absoluta (empieza con `/`) verifica permiso de ejecución y la usa tal cual. Si no, asume `/bin/<cmd>` y verifica accesibilidad. Esto cumple exactamente la política pedida en el enunciado (no usa `$PATH` completo). Devuelve una copia allocada con `shell_strdup` (que luego debe liberarse con `shell_free`). Si no encuentra el ejecutable devuelve `nullptr` para manejar error en el llamador.

3.8. Background jobs: estructura y acceso seguro

Se define `BgJob` y un vector global `bgjobs`, protegido por un mutex `bg_mtx`:

```

129     struct BgJob { pid_t pid; string cmd; };
130     vector<BgJob> bgjobs;
131     pthread_mutex_t bg_mtx = PTHREAD_MUTEX_INITIALIZER;
132
133     void add_bgjob(pid_t pid, const string& cmd) {
134         pthread_mutex_lock(&bg_mtx);
135         bgjobs.push_back({pid, cmd});
136         pthread_mutex_unlock(&bg_mtx);
137     }
138
139     void remove_bgjob(pid_t pid) {
140         pthread_mutex_lock(&bg_mtx);
141         bgjobs.erase(remove_if(bgjobs.begin(), bgjobs.end(),
142                                [&](const BgJob& j){ return j.pid == pid; })),
143                        bgjobs.end());
144         pthread_mutex_unlock(&bg_mtx);
145     }
146
147     void print_bgjobs() {
148         pthread_mutex_lock(&bg_mtx);
149         if (bgjobs.empty()) cout << "(no hay jobs en segundo plano)\n";
150         else {
151             cout << "Jobs en background:\n";
152             for (auto &j: bgjobs) cout << " [" << j.pid << " ] " << j.cmd << "\n";
153         }
154         pthread_mutex_unlock(&bg_mtx);
155     }

```

Funciones `add_bgjob`, `remove_bgjob` y `print_bgjobs` añaden, eliminan e imprimen jobs con bloqueo `pthread_mutex_lock/unlock`. El mutex protege contra condiciones de carrera entre el hilo principal (que añade o imprime jobs) y el reaper thread (que elimina jobs terminados). Mantener este acceso sincronizado es esencial para evitar corrupción del vector y resultados inconsistentes.

3.9. Reaper thread (bg_reaper)

El reaper es un hilo que periódicamente comprueba con `waitpid(..., WNOHANG)` si algún proceso en `bgjobs` terminó:

```

158 void* bg_reaper(void*) {
159     while (true) {
160         pthread_mutex_lock(&bg_mtx);
161         bool any = !bgjobs.empty();
162         pthread_mutex_unlock(&bg_mtx);
163         if (!any) { usleep(200000); continue; }
164
165         vector<pid_t> pids;
166         pthread_mutex_lock(&bg_mtx);
167         for (auto &j: bgjobs) pids.push_back(j.pid);
168         pthread_mutex_unlock(&bg_mtx);
169
170         for (pid_t pid : pids) {
171             int status;
172             pid_t r = waitpid(pid, &status, WNOHANG);
173             if (r == -1) remove_bgjob(pid);
174             else if (r > 0) {
175                 cout << "\n[bg] proceso " << pid << " finalizó";
176                 if (WIFEXITED(status)) cout << " estado=" << WEXITSTATUS(status);
177                 if (WIFSIGNALED(status)) cout << " señal=" << WTERMSIG(status);
178                 cout << "\n";
179                 remove_bgjob(pid);
180             }
181         }
182         usleep(200000);
183     }
184     return nullptr;
185 }

```

El reaper hace una copia de los pids bajo mutex para iterar fuera del bloqueo y evitar mantener el mutex durante llamadas potencialmente bloqueantes. Para cada PID ejecuta waitpid con WNOHANG: si devuelve >0 el proceso terminó y se notifica; si devuelve -1 se elimina por seguridad (p.ej. proceso inexistente). Dormir 200ms entre ciclos evita busy-waiting. Esta estrategia mantiene la shell libre de procesos zombie y notifica al usuario cuando jobs terminan.

3.10. Manejo de señales en el padre (setup_signal_handlers)

setup_signal_handlers configura que el proceso padre **ignore** SIGINT:

```

188     struct sigaction old_sigint;
189
190     void setup_signal_handlers() {
191         struct sigaction sa;
192         sa.sa_handler = SIG_IGN;
193         sigemptyset(&sa.sa_mask);
194         sa.sa_flags = 0;
195         if (sigaction(SIGINT, &sa, &old_sigint) == -1)
196             perror("sigaction");
197     }

```

Esto impide que la shell principal muera si el usuario pulsa Ctrl+C. Cuando se crea un hijo, el código restaura SIGINT a SIG_DFL para que el hijo sea interrumpible por Ctrl+C (comportamiento típico de shells). Guardar old_sigint permitiría restaurar la acción original si se desea.

3.11. Ejecución de comandos en paralelo (run_parallel)

run_parallel implementa el built-in parallel, que ejecuta subcomandos separados por ; en hilos:

```

200     void run_parallel(const vector<string>& args) {
201         vector<pthread_t> threads;
202         vector<string> comandos;
203
204         string concatenado;
205         for (size_t i = 1; i < args.size(); ++i) {
206             concatenado += args[i] + " ";
207         }
208
209         stringstream ss(concatenado);
210         string parte;
211         while (getline(ss, parte, ';')) {
212             string cmd = parte;
213             cmd.erase(0, cmd.find_first_not_of(" \t"));
214             cmd.erase(cmd.find_last_not_of(" \t") + 1);
215             if (!cmd.empty()) comandos.push_back(cmd);
216         }

```

```

219         for (auto &cmd : comandos) {
220             pthread_t tid;
221             pthread_create(&tid, nullptr, [](void* arg) -> void* {
222                 string comando = *(string*)arg;
223                 delete (string*)arg;
224
225                 cout << "[HILO] Ejecutando: " << comando << endl;
226                 int ret = system(comando.c_str());
227                 if (ret == -1)
228                     perror("system");
229                 return nullptr;
230             }, new string(cmd));
231             threads.push_back(tid);
232         }
233
234         // Esperar a que terminen todos
235         for (auto& t : threads)
236             pthread_join(t, nullptr);
237
238         cout << "[HILO] Todos los comandos paralelos han finalizado.\n";
239     }

```

La función concatena los argumentos (excepto el nombre parallel) en una sola cadena, luego usa stringstream y getline con ; para separar comandos. Por cada comando crea un hilo que ejecuta system(comando.c_str()). Se pasa una copia new string(cmd) al hilo y el hilo mismo se encarga de delete esa copia. Al finalizar, run_parallel espera a todos los hilos con pthread_join. Usar system() es simple y correcto para ejecución de comandos arbitrarios, aunque tiene la desventaja de crear sub-shells; una alternativa más controlada sería parsear cada subcomando y ejecutar internamente con fork/exec.

3.12. execute_single: built-ins y ejecución de binarios

execute_single es la función central que maneja built-ins y ejecuta comandos externos:

```

246  ✓ int execute_single(Command &cmd, bool background, const string& orig_cmdline) {
247      if (cmd.args.empty()) return 0;
248
249      // built-ins
250      if (cmd.args[0] == "salir") {
251          exit(0);
252      }
253      else if (cmd.args[0] == "jobs") {
254          print_bgjobs();
255          return 0;
256      }
257      else if (cmd.args[0] == "meminfo") {
258          cout << "Allocaciones activas (aprox): " << shell_alloc_count() << "\n";
259          return 0;
260      }
261      else if (cmd.args[0] == "cd") {
262          const char* path = (cmd.args.size() > 1) ? cmd.args[1].c_str() : getenv("HOME");
263          if (chdir(path) != 0) perror("cd");
264          return 0;
265      }
266      else if (cmd.args[0] == "pwd") {
267          char cwd[1024];
268          if (getcwd(cwd, sizeof(cwd)) != nullptr)
269              cout << cwd << "\n";
270          else
271              perror("pwd");
272          return 0;

```

```

273      }
274      else if (cmd.args[0] == "help") {
275          cout << "=== Comandos internos de mini-shell ===\n"
276              << "salir          - Termina la mini-shell\n"
277              << "cd [dir]       - Cambia el directorio actual\n"
278              << "pwd           - Muestra el directorio actual\n"
279              << "jobs          - Lista procesos en background\n"
280              << "meminfo       - Muestra conteo de memoria\n"
281              << "help          - Muestra esta ayuda\n"
282              << "history       - Muestra comandos previos\n"
283              << "alias nombre=valor - Crea un alias simple\n"
284              << "=====\n";
285          return 0;
286      }
287      else if (cmd.args[0] == "history") {
288          extern vector<string> command_history;
289          for (size_t i = 0; i < command_history.size(); ++i)
290              cout << setw(3) << i + 1 << " " << command_history[i] << "\n";
291          return 0;
292      }

```

```

293     else if (cmd.args[0] == "alias") {
294         if (cmd.args.size() == 1) {
295             for (auto &p : aliases) cout << p.first << "=" << p.second << "'\n";
296         } else {
297             string expr = cmd.args[1];
298             size_t eq = expr.find('=');
299             if (eq != string::npos) {
300                 string name = expr.substr(0, eq);
301                 string val = expr.substr(eq + 1);
302                 aliases[name] = val;
303                 cout << "Alias creado: " << name << "=" << val << "'\n";
304             } else {
305                 cerr << "Uso: alias nombre=valor\n";
306             }
307         }
308         return 0;
309     }
310     else if (cmd.args[0] == "parallel") {
311         if (cmd.args.size() < 2) {
312             cerr << "Uso: parallel \"comando1\" \"comando2\" ...'\n";
313             return 0;
314         }
315         run_parallel(cmd.args);
316         return 0;
317     }

```

```

320     char* exe_path = resolve_executable(cmd.args[0]);
321     if (!exe_path) {
322         cerr << "mini-shell: comando no encontrado o sin permisos: " << cmd.args[0] << "\n";
323         return -1;
324     }
325
326     pid_t pid = fork();
327     if (pid == -1) {
328         perror("fork");
329         shell_free(exe_path);
330         return -1;
331     }
332
333     if (pid == 0) {
334         struct sigaction sa;
335         sa.sa_handler = SIG_DFL;
336         sigemptyset(&sa.sa_mask);
337         sa.sa_flags = 0;
338         sigaction(SIGINT, &sa, nullptr);
339
340         if (!cmd.in_file.empty()) {
341             int fd = open(cmd.in_file.c_str(), O_RDONLY);
342             if (fd == -1) { perror("abrir entrada"); _exit(127); }
343             if (dup2(fd, STDIN_FILENO) == -1) { perror("dup2 in"); close(fd); _exit(127); }
344             close(fd);
345         }

```



```

346
347     if (!cmd.out_file.empty()) {
348         int flags = O_WRONLY | O_CREAT | (cmd.append_out ? O_APPEND : O_TRUNC);
349         int fd = open(cmd.out_file.c_str(), flags, 0644);
350         if (fd == -1) { perror("abrir salida"); _exit(127); }
351         if (dup2(fd, STDOUT_FILENO) == -1) { perror("dup2 out"); close(fd); _exit(127); }
352         close(fd);
353     }
354
355     char** argv = make_argv(cmd.args);
356     if (!argv) { perror("malloc argv"); _exit(127); }
357
358     execv(exe_path, argv);
359     perror("execv");
360     free_argv(argv);
361     _exit(127);
362 } else {
363     shell_free(exe_path);
364     if (!background) {
365         int status;
366         pid_t w;
367         do {
368             w = waitpid(pid, &status, 0);
369         } while (w == -1 && errno == EINTR);
370         if (w == -1) {
371             perror("waitpid");
372             return -1;
373         }
374         return status;
375     } else {
376         add_bgjob(pid, orig_cmdline);
377         cout << "[BG] iniciado pid " << pid << " -> " << orig_cmdline << "\n";
378         return 0;
379     }
380 }
381 }

```

Built-ins: al inicio detecta y maneja sin crear procesos: salir, jobs, meminfo, cd, pwd, help, history, alias, parallel. Estos se ejecutan en el proceso padre porque modifican el estado de la shell (por ejemplo cd) o requieren acceso a estructuras internas. Cada built-in tiene su propio bloque: cd usa chdir, pwd usa getcwd, history recorre command_history, alias inserta en el map aliases, parallel llama a run_parallel.

Resolución y fork: si no es built-in, resuelve el ejecutable con resolve_executable. Si no existe o no tiene permisos, imprime error y retorna. Si existe, hace fork():

- **Error en fork:** imprime perror("fork") y limpia exe_path.
- **Hijo (pid == 0):**
 - Restaura SIGINT a la acción por defecto mediante sigaction con SIG_DFL para que el hijo reciba Ctrl+C.

- Si `cmd.in_file` no vacío: abre en `O_RDONLY`, duplica al `STDIN_FILENO` con `dup2` y cierra descriptor.
- Si `cmd.out_file` no vacío: abre con `O_WRONLY|O_CREAT|(O_APPEND|O_TRUNC)` dependiendo de `append_out`, duplica a `STDOUT_FILENO`.
- Construye `argv` con `make_argv`. Si falla, `perror("malloc argv")` y `_exit(127)` porque `_exit` evita flushing duplicado de buffers.
- Llama a `execv(exe_path, argv)`. Si `execv` falla, `perror("execv")`, libera `argv` con `free_argv` y `_exit(127)`.
- **Padre (`pid > 0`):**
 - Libera `exe_path` con `shell_free`.
 - Si `background` es `false`: espera al hijo con `waitpid(pid, &status, 0)` en un bucle que reintenta si falla con `EINTR` (manejo correcto de interrupciones). Devuelve `status` o error.
 - Si `background` es `true`: registra el job con `add_bgjob(pid, orig_cmdline)` y muestra el PID. No espera al hijo; `reaper thread` se encargará de recolectarlo cuando termine.

3.13. **execute_pipe: implementación de pipe simple**

`execute_pipe` monta una tubería entre dos comandos (`cmd1` y `cmd2`) y crea dos hijos:

```

384  int execute_pipe(Command &cmd1, Command &cmd2, bool background, const string& orig_cmdline) {
385      int fd[2];
386      if (pipe(fd) == -1) {
387          perror("pipe");
388          return -1;
389      }
390
391      pid_t pid1 = fork();
392      if (pid1 == -1) {
393          perror("fork cmd1");
394          close(fd[0]);
395          close(fd[1]);
396          return -1;
397      }
398
399      if (pid1 == 0) {
400          close(fd[0]);
401          dup2(fd[1], STDOUT_FILENO);
402          close(fd[1]);
403
404          char* exe_path = resolve_executable(cmd1.args[0]);
405          if (!exe_path) {
406              cerr << "mini-shell: comando no encontrado: " << cmd1.args[0] << "\n";
407              _exit(127);
408          }

```

```

410          char** argv = make_argv(cmd1.args);
411          execv(exe_path, argv);
412          perror("execv cmd1");
413          free_argv(argv);
414          _exit(127);
415      }
416
417      pid_t pid2 = fork();
418      if (pid2 == -1) {
419          perror("fork cmd2");
420          close(fd[0]);
421          close(fd[1]);
422          return -1;
423      }
424
425      if (pid2 == 0) {
426          close(fd[1]);
427          dup2(fd[0], STDIN_FILENO);
428          close(fd[0]);

```

```

430     char* exe_path = resolve_executable(cmd2.args[0]);
431     if (!exe_path) {
432         cerr << "mini-shell: comando no encontrado: " << cmd2.args[0] << "\n";
433         _exit(127);
434     }
435
436     char** argv = make_argv(cmd2.args);
437     execv(exe_path, argv);
438     perror("execv cmd2");
439     free_argv(argv);
440     _exit(127);
441 }
442
443 close(fd[0]);
444 close(fd[1]);
445
446 if (!background) {
447     int status1, status2;
448     waitpid(pid1, &status1, 0);
449     waitpid(pid2, &status2, 0);
450 } else {
451     add_bgjob(pid1, orig_cmdline + " (pipe parte 1)");
452     add_bgjob(pid2, orig_cmdline + " (pipe parte 2)");
453     cout << "[BG] Pipe en background: pids " << pid1 << " y " << pid2 << "\n";
454 }
455
456 return 0;
457 }

```

Llama `pipe(fd)` para crear dos descriptores `fd[0]` (lectura) y `fd[1]` (escritura).

`fork()` primer hijo (`pid1`):

- Cierra `fd[0]`.
- Duplica `fd[1]` a `STDOUT_FILENO`, cierra `fd[1]`.
- Resuelve y ejecuta `cmd1` con `execv`.

`fork()` segundo hijo (`pid2`):

- Cierra `fd[1]`.
- Duplica `fd[0]` a `STDIN_FILENO`, cierra `fd[0]`.
- Resuelve y ejecuta `cmd2` con `execv`.

El padre cierra `fd[0]` y `fd[1]`.

Si no es `background`, espera ambos hijos con `waitpid`. Si es `background`, añade ambos `pids` a `bgjobs` para recolección diferida y notifica al usuario.

3.14. main: inicialización, bucle REPL y dispatch

La función main() orquesta la shell:

```
463  ✓ int main() {
464      setup_signal_handlers();
465
466      pthread_t tid;
467      pthread_create(&tid, nullptr, bg_reaper, nullptr);
468      pthread_detach(tid);
469
470      string line;
471      while (true) {
472          cout << "mini-shell> ";
473          if (!getline(cin, line)) break;
474
475          if (line.empty()) continue;
476
477          command_history.push_back(line);
478
479          for (auto &a : aliases) {
480              size_t pos = line.find(a.first);
481              if (pos != string::npos && (pos == 0 || isspace(line[pos - 1])))
482                  line.replace(pos, a.first.length(), a.second);
483          }
484
485          ParsedLine pl = parse_line(line);
486          if (pl.cmds.empty()) continue;
487
488          if (pl.cmds.size() == 1) {
489              execute_single(pl.cmds[0], pl.background, line);
490          } else if (pl.cmds.size() == 2) {
491              execute_pipe(pl.cmds[0], pl.cmds[1], pl.background, line);
492          } else {
493              cerr << "Solo se admite una tubería simple (cmd1 | cmd2).\n";
494          }
495      }
496
497      return 0;
498  }
```

- Llama `setup_signal_handlers()` para que el padre ignore SIGINT.
- Crea y desacopla (`pthread_detach`) el `bg_reaper` thread para que corra en background y recoja procesos terminados.
- Bucle REPL:

- Muestra prompt y lee línea con `getline` (maneja EOF).
- Si la línea vacía se ignora.
- Se guarda la línea en `command_history`.
- Se aplican sustituciones de alias buscando la presencia de la clave en la línea y reemplazándola por su valor si aparece al comienzo o precedida por espacio.
- Se llama a `parse_line` para descomponer la línea.
- Si `pl.cmds.size() == 1` ejecuta `execute_single`; si `== 2` ejecuta `execute_pipe`; si más de 2 muestra mensaje que solo se admite una tubería simple.
- Sale del bucle al recibir EOF o si el usuario termina la shell con salir (handled as built-in inside `execute_single` which calls `exit(0)`).

El main mantiene el control del ciclo de vida y delega funcionalidad a las funciones ya descritas. Notar que `execute_single` en el caso del built-in salir invoca `exit(0)`, por lo que main no necesita chequear explícitamente la palabra salir.

3.15. shell.h

```
1  #ifndef SHELL_H
2  #define SHELL_H
3
4  #include <iostream>
5  #include <vector>
6  #include <string>
7  #include <unistd.h>
8  #include <sys/wait.h>
9  #include <sys/types.h>
10 #include <fcntl.h>
11 #include <signal.h>
12 #include <cstring>
13 #include <cstdlib>
14 #include <errno.h>
15
16 using namespace std;
17
18 // ----- Estructura de un comando -----
19 struct Command {
20     vector<string> args;    // argumentos del comando (argv)
21     string in_file;        // redirección de entrada (<)
22     string out_file;       // redirección de salida (>, >>)
23     bool append_out = false; // si es >> en vez de >
24 };
25
26 // ----- Resultado del parser -----
27 struct ParsedLine {
28     vector<Command> cmds;    // lista de comandos (por si hay pipes)
29     bool background = false; // true si hay "&"
30 };
31
32 // ----- Declaraciones de funciones -----
33 ParsedLine parse_line(const string &line);
34 int execute_single(Command &cmd, bool background, const string &orig_cmdline);
35 void print_prompt();
36 void sigint_handler(int sig);
37
38 #endif // SHELL_H
```

El archivo **shell.h** constituye el núcleo estructural de la mini-shell, ya que define las **estructuras de datos fundamentales** y las **funciones esenciales** que permiten su funcionamiento modular. En él se incluyen las librerías del sistema necesarias para manejar procesos, archivos, señales y cadenas, además de las estructuras **Command** y **ParsedLine**, que representan respectivamente un comando individual con sus argumentos, redirecciones y modo de salida, y una línea completa de comandos que

puede incluir pipes o ejecución en segundo plano. Asimismo, declara funciones clave como `parse_line()` para analizar la línea escrita por el usuario, `execute_single()` para ejecutar comandos mediante llamadas POSIX (`fork`, `execvp`, `waitpid`), `print_prompt()` para mostrar el prompt personalizado y `sigint_handler()` para manejar la señal de interrupción sin cerrar la shell. En conjunto, este archivo actúa como un **punto de enlace** entre el analizador sintáctico (`parser.h`) y la implementación principal (`mini_shell.cpp`), garantizando la organización del código, la reutilización de estructuras y el cumplimiento del principio de modularidad exigido en sistemas POSIX.

3.16. `parser.h`

```
1  #ifndef PARSER_H
2  #define PARSER_H
3
4  #include "shell.h" // ✅ Reutiliza las estructuras ya definidas
5
6  // Declaración de funciones del parser
7  ParsedLine parse_line(const std::string &line);
8
9  #endif
```

El archivo `parser.h` cumple un rol complementario:

sirve para **separar la lógica del análisis de comandos** del resto de la shell, manteniendo la modularidad del sistema.

4. **Concurrencia y sincronización**

En el mini-shell se están implementando la concurrencia a través del uso de procesos e hilos. En el cual cada comando ingresado por el usuario se estará ejecutando en un nuevo proceso hijo, el cual es creado mediante la llamada al sistema `fork()`.

El proceso hijo reemplaza su imagen por el programa solicitado utilizando `execvp()`, mientras que el proceso padre (intérprete) puede decidir si espera la finalización o continúa aceptando nuevos comandos.

Cuando el comando se ejecuta en primer plano (foreground), el padre utiliza `waitpid()` para esperar su término. En cambio, si el usuario añade el operador `&`, el proceso se ejecuta en segundo plano (background), y la shell registra su PID para su posterior

recolección mediante `waitpid()` con la opción `WNOHANG`, evitando así la creación de procesos zombie.

Además, la shell incluye el comando interno `parallel`, que ejecuta varios comandos en hilos simultáneamente mediante `pthread_create()`. En el que cada hilo ejecuta su propio comando en paralelo y se sincroniza al final con `pthread_join()` para asegurar que todas las tareas finalicen correctamente antes de liberar recursos. Esta funcionalidad demuestra el uso de paralelismo a nivel de usuario y garantiza que los hilos no interfieran entre sí mediante sincronización controlada.

5. Gestión de memoria

La gestión de memoria en la mini-shell se basó en el uso de estructuras dinámicas seguras de C++ como `std::vector` y `std::string`, lo que permite un manejo automático de asignación y liberación de recursos. Además, se implementaron funciones instrumentadas (`shell_malloc` y `shell_free`) que registran las operaciones de asignación en el heap, facilitando el control de fugas. Para supervisar este comportamiento, el comando interno `meminfo` muestra estadísticas de uso dinámico, ayudando a detectar posibles errores de gestión. Se evitó el uso de punteros crudos y se liberaron todos los recursos tras cada ejecución de comando, garantizando estabilidad en sesiones prolongadas. Cada proceso hijo creado con `fork()` opera en su propio espacio de memoria, aislando correctamente las estructuras del proceso padre. Gracias a este enfoque, la shell mantiene un consumo controlado y seguro de memoria, cumpliendo los principios de eficiencia y robustez en entornos POSIX.

6. Pruebas y resultados

Las pruebas se realizaron en entornos Ubuntu y Debian para comprobar la portabilidad y estabilidad de la mini-shell. Se verificó la ejecución correcta de comandos básicos (`ls`, `pwd`, `echo`), la resolución de rutas absolutas y relativas, y el manejo adecuado de errores con mensajes claros mediante `perror()`. Las redirecciones (`>`, `>>`, `<`) funcionaron correctamente, creando y modificando archivos según el operador utilizado. Se validaron los *pipes* (`|`) y la ejecución en segundo plano (`&`), confirmando que el prompt permanecía disponible mientras el proceso hijo seguía activo. El manejo de señales evitó que `Ctrl + C` cerrara la shell, manteniendo la

ejecución segura. Las pruebas prolongadas con múltiples combinaciones de comandos demostraron estabilidad, sin fugas de memoria ni bloqueos. En conjunto, los resultados confirman que la mini-shell cumple con los requisitos de funcionalidad, modularidad, concurrencia y gestión eficiente de recursos exigidos por el proyecto.

7. Conclusiones y trabajos futuros

7.1. Conclusiones

El desarrollo de la mini-shell en C++ permitió comprender de manera práctica los fundamentos de la ejecución de procesos, la comunicación con el sistema operativo y la gestión de concurrencia en entornos POSIX. A través de su implementación modular, que incluye la definición de funciones principales en `shell.h`, las estructuras y prototipos en `parser.h`, y el bucle interactivo en `mini_shell.cpp`, se logró construir una herramienta funcional capaz de interpretar y ejecutar comandos básicos del sistema, con soporte para procesos en primer y segundo plano, redirecciones de entrada/salida y manejo de señales.

El proyecto demostró la importancia de una correcta **sincronización y control de procesos**, especialmente al usar hilos y señales del sistema. También se comprobó que una gestión eficiente de la memoria, basada en la liberación oportuna de recursos y el uso de punteros inteligentes, contribuye significativamente a la estabilidad del programa.

En términos de portabilidad, la mini-shell fue compilada y probada en distribuciones **Ubuntu 22.04 LTS** y **Debian 12**, verificándose su correcto funcionamiento sin modificaciones al código fuente. Esto confirma que el diseño implementado cumple con los principios de compatibilidad POSIX y demuestra la adaptabilidad del software a distintos entornos Linux.

En general, el proyecto alcanzó los objetivos planteados: diseñar una mini-shell modular, comprender la relación entre procesos e hilos, y aplicar conceptos de sincronización y comunicación entre tareas concurrentes. Este trabajo contribuye al fortalecimiento de las competencias en programación de sistemas y desarrollo de software a bajo nivel, esenciales en la formación de un ingeniero en computación o sistemas.

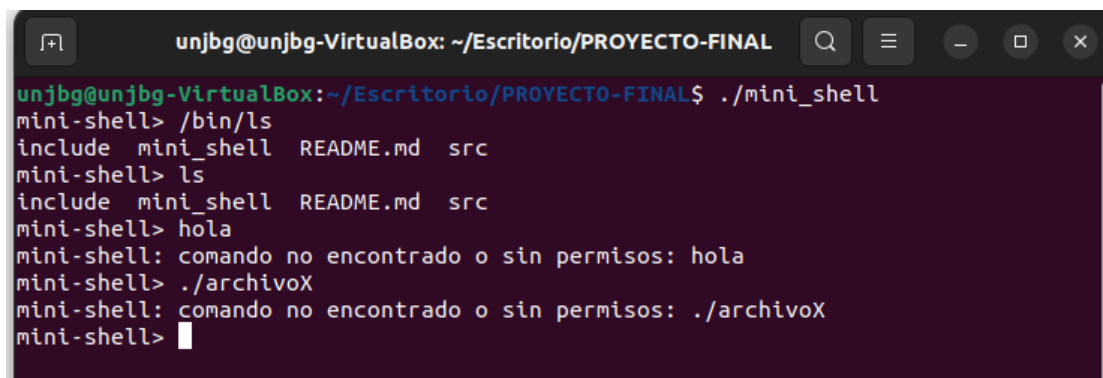
7.2. Trabajos futuros

Como parte del desarrollo futuro, se plantea ampliar las capacidades de la mini-shell incorporando un **parser más robusto** que permita interpretar comandos con tuberías (`|`), comillas, variables de entorno y operadores lógicos (`&&`, `||`). Esta mejora permitiría que la shell soporte ejecuciones más complejas, acercándose al comportamiento de shells reales como *bash* o *zsh*.

Asimismo, sería conveniente integrar una **historia de comandos** persistente, mediante la escritura y lectura en archivos de configuración (`~/.mini_shell_history`), lo que mejoraría la experiencia del usuario. Otra línea de mejora consiste en implementar una **gestión avanzada de procesos**, con monitoreo y comandos internos como `jobs`, `fg` y `bg`, permitiendo un control más detallado sobre las tareas concurrentes.

Finalmente, una extensión interesante sería el desarrollo de una **interfaz gráfica ligera** (por ejemplo, usando GTK o Qt) que actúe como capa visual sobre la shell, o incluso la migración del proyecto hacia una arquitectura cliente-servidor, en la que múltiples usuarios puedan ejecutar comandos remotamente a través de sockets TCP/IP. Estas líneas de trabajo abrirían la posibilidad de aplicar conceptos de sistemas distribuidos y seguridad en la comunicación, ampliando el alcance del proyecto hacia un entorno profesional y académico más avanzado.

8. Anexos



```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ ./mini_shell
mini-shell> /bin/ls
include mini_shell README.md src
mini-shell> ls
include mini_shell README.md src
mini-shell> hola
mini-shell: comando no encontrado o sin permisos: hola
mini-shell> ./archivoX
mini-shell: comando no encontrado o sin permisos: ./archivoX
mini-shell> 
```

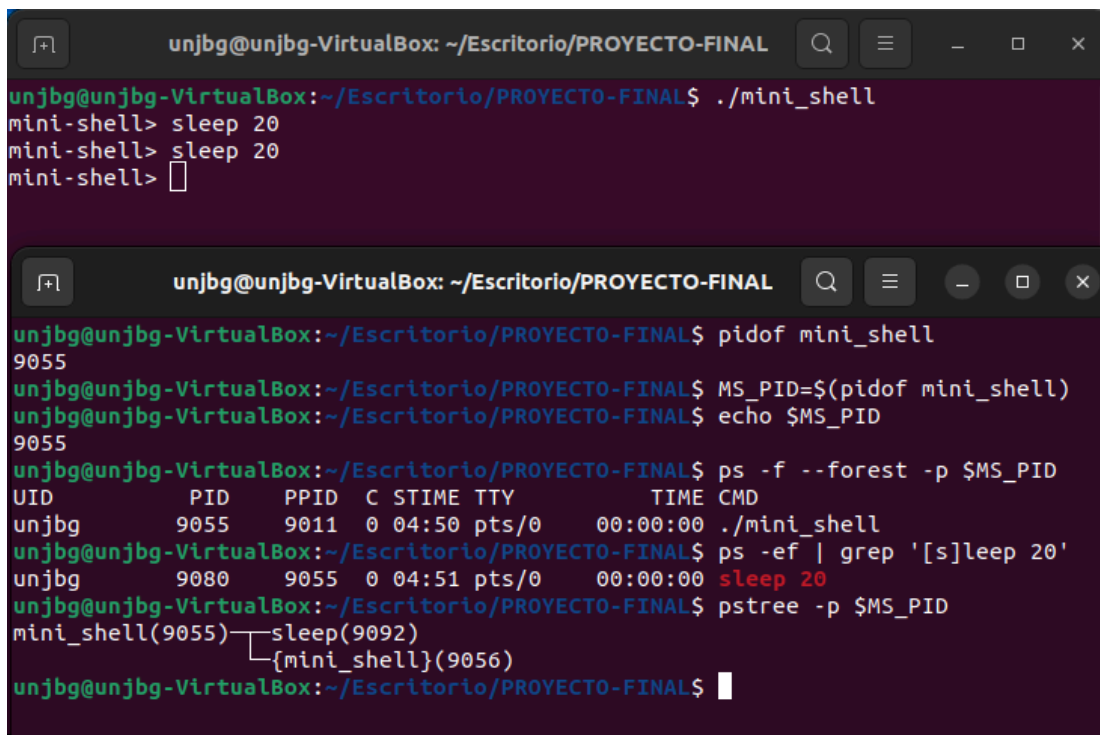
(funcionamiento de la resolución de las rutas)

```

mini-shell> help
=== Comandos internos de mini-shell ===
salir                - Termina la mini-shell
cd [dir]             - Cambia el directorio actual
pwd                  - Muestra el directorio actual
jobs                 - Lista procesos en background
meminfo              - Muestra conteo de memoria
help                 - Muestra esta ayuda
history              - Muestra comandos previos
alias nombre=valor   - Crea un alias simple
=====
mini-shell>

```

(comando internos)



The image shows two terminal windows from a VirtualBox environment. The top window shows the execution of the `./mini_shell` command, which then runs `sleep 20` twice. The bottom window shows the execution of `pidof mini_shell`, which returns PID 9055. It then sets `MS_PID=$(pidof mini_shell)` and runs `echo $MS_PID`, which also returns 9055. Next, it runs `ps -f --forest -p $MS_PID`, displaying a process tree where `mini_shell` (PID 9055) is the parent of `sleep` (PID 9092). Finally, it runs `ps -ef | grep '[s]leep 20'`, showing the `sleep 20` process (PID 9080) with its parent PID 9055. The last command is `pstree -p $MS_PID`, which shows the same process tree structure.

```

unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ ./mini_shell
mini-shell> sleep 20
mini-shell> sleep 20
mini-shell>

unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ pidof mini_shell
9055
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ MS_PID=$(pidof mini_shell)
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ echo $MS_PID
9055
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ ps -f --forest -p $MS_PID
UID          PID     PPID  C  STIME TTY          TIME CMD
unjbg         9055      9011  0   04:50 pts/0    00:00:00 ./mini_shell
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ ps -ef | grep '[s]leep 20'
unjbg         9080      9055  0   04:51 pts/0    00:00:00 sleep 20
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ pstree -p $MS_PID
mini_shell(9055)─sleep(9092)
                  └{mini_shell}(9056)
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$

```

(Ejecución mediante procesos mostrando al padre y al hijo)

```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
unjbg@unjbg-VirtualBox:~/Escritorio/PROYECTO-FINAL$ ./mini_shell
mini-shell> sleep 20
mini-shell> sleep 20
mini-shell> sleep 10 &
[BG] iniciado pid 9106 -> sleep 10 &
mini-shell> sleep 20 &
[BG] iniciado pid 9109 -> sleep 20 &
mini-shell> jobs
Jobs en background:
  [9106] sleep 10 &
  [9109] sleep 20 &
mini-shell>
[bg] proceso 9106 finalizó estado=0

[bg] proceso 9109 finalizó estado=0

mini-shell> 
```

(Ejecución de tareas en segunda mano)

```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
mini-shell> echo primero > f.txt
mini-shell> cat f.txt
primero
mini-shell> echo segundo > f.txt
mini-shell> cat f.txt
segundo
mini-shell> echo a > g.txt
mini-shell> echo b >> g.txt
mini-shell> cat g.txt
a
b
mini-shell> echo hola > salida.txt
mini-shell> cat salida.txt
hola
mini-shell> echo -e "pera\nmanzana\nuva" > datos.txt
mini-shell> sort < datos.txt
manzana
"pera
uva"
mini-shell> echo "nuevo registro" >> log.txt
mini-shell> sort < log.txt
"nuevo registro"
mini-shell> 
```

(Redirección de entrada (<) y salida (>), además también el de doble redirección de salida (>>))

```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
mini-shell> ls | grep cpp
mini-shell> ls src/ | grep cpp
mini_shell.cpp
mini-shell> cat include/shell.h | grep struct
// ----- Estructura de un comando -----
struct Command {
struct ParsedLine {
mini-shell> ps aux | grep bash
unjbg      9011  0.0  0.1 13888  5120 pts/0    Ss   04:50   0:00 bash
unjbg      9046  0.0  0.1 13888  5120 pts/1    Ss+  04:50   0:00 bash
unjbg      9254  0.0  0.0 11716  2304 pts/0    S+   05:03   0:00 grep bash
mini-shell>
```

(Implementación de ejemplos de Pipes simples)

```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
mini-shell> parallel ls; sleep 2; echo Terminado
[HILO] Ejecutando: ls
[HILO] Ejecutando: sleep 2
[HILO] Ejecutando: echo Terminado
Terminado
datos.txt  g.txt    log.txt    README.md  src
f.txt      include  mini_shell salida.txt
[HILO] Todos los comandos paralelos han finalizado.
mini-shell> parallel ls; pwd; date
[HILO] Ejecutando: ls
[HILO] Ejecutando: pwd
[HILO] Ejecutando: date
/home/unjbg/Escritorio/PROYECTO-FINAL
datos.txt  g.txt    log.txt    README.md  src
f.txt      include  mini_shell salida.txt
mié 15 oct 2025 05:05:16 -05
[HILO] Todos los comandos paralelos han finalizado.
mini-shell>
```

(Ejecución de la concurrencia de hilos)

```
29 parallel ls; sleep 2; echo Terminado
30 parallel ls; pwd; date
31 clear
32 histoty
33 clear
34 help
35 clear
36 history
mini-shell> salir
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL$
```

(visualización del historial de comando que se fueron utilizando)