



# Implementación de un intérprete de comandos en C++ sobre linux



- Jhon W. Mamani Condori
- Juan L. Mamani Mullo

# Introducción

Este proyecto presenta la implementación de una mini shell en C++ para Linux, capaz de ejecutar comandos, manejar procesos en segundo plano, tuberías y concurrencia mediante hilos.

Su desarrollo permitió aplicar conceptos clave de procesos, señales, sincronización y gestión de memoria, integrando los conocimientos teóricos de sistemas operativos en una experiencia práctica y funcional.



# Objetivos



## Objetivo General

Desarrollar una mini shell funcional en lenguaje C++ que permita ejecutar comandos del sistema, gestionar procesos en segundo plano, tuberías, redirecciones y aplicar conceptos de concurrencia y manejo de memoria.

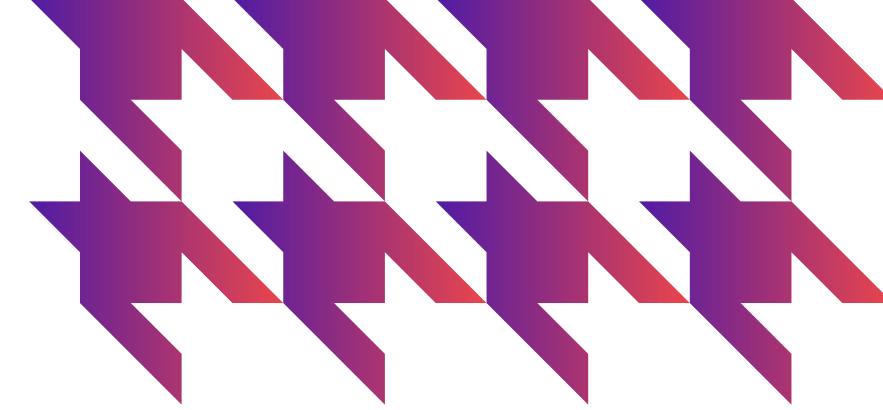


## Objetivos Específicos

- Implementar comandos básicos como cd, pwd, help, alias, history.
- Incorporar manejo de procesos en background y comunicaciones con pipes.
- Aplicar manejo de hilos para ejecución paralela.
- Añadir un módulo de gestión de memoria para monitorear asignaciones dinámicas.



# Arquitectura del Sistema y Flujo de Ejecución



Estructura modular:

- mini\_shell.cpp: lógica principal y control del shell.
- shell.h: definición de estructuras Command y ParsedLine.
- parser.h: análisis de línea de comandos.

El sistema sigue el modelo cliente–intérprete–sistema, donde el usuario envía comandos, el shell los interpreta y ejecuta a través de llamadas al sistema.

mini\_shell (padre)

- Mostrar prompt: mini-shell>.
- Leer una línea del usuario.
- Sustituir aliases si los hay .
- Analizar la línea (parse\_line).
- Si hay un pipe (|), ejecutar execute\_pipe.
- Si hay un solo comando, ejecutar execute\_single.
- Si termina con &, ejecutar en segundo plano.
- Si es comando interno, manejarlo sin fork().
- Si no, crear proceso hijo y ejecutar con execv().

# Desarrollo e Implementación

El archivo mini\_shell.cpp contiene la implementación completa.

- Se incluyen librerías de sistema (unistd.h, sys/wait.h, pthread.h).
- Se definen funciones como:
  - tokenize(): divide los comandos por espacios.
  - parse\_line(): analiza redirecciones, pipes y ejecución en background.
  - execute\_single(): ejecuta comandos simples y funciones internas.
  - execute\_pipe(): ejecuta dos comandos conectados por una tubería.
  - run\_parallel(): crea hilos para ejecución simultánea.

```
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  }
```

La función tokenize divide una cadena en tokens separados por espacios

# Desarrollo e Implementación

```
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      }
```

Este parser es suficiente para aceptar entradas como cmd arg1 > out.txt & y para comandos con una sola tubería (cmd1 | cmd2).

parse\_line transforma la línea completa en una estructura ParsedLine (lista de Command, bandera background)

```
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  }
```

# Desarrollo e Implementación

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     }
273 }
```

el hijo usa \_exit con códigos 127 para indicar fallos de ejecución, lo cual es una convención habitual. El padre trata waitpid con reintentos por EINTR lo cual evita retornar por interrupciones de señales. Es importante notar que free\_argv se llama en el hijo solo si execv falla; en el caso de éxito no hace falta porque el proceso es reemplazado.

```
375 } else {
376     add_bgjob(pid, orig_cmdline);
377     cout << "[BG] iniciado pid " << pid << " -> " << orig_cmdline << "\n";
378     return 0;
379 }
380 }
381 }
```

# Desarrollo e Implementación

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         }
409     }
```

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

```
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 }
```

# Desarrollo e Implementación

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      }  
217  }
```

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()).

```
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 << "[HIL0] 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 << "[HIL0] Todos los comandos paralelos han finalizado.\n";  
239  }
```

# Concurrencia y Sincronización

- Se usan procesos con **fork()** y **hilos** con **pthread**.
- La función **bg\_reaper()** elimina procesos terminados del vector **bgjobs**.
- Se implementa un **mutex** (pthread\_mutex\_t) para evitar condiciones de carrera.
- Los comandos **jobs** y **parallel** permiten visualizar y ejecutar tareas concurrentes.

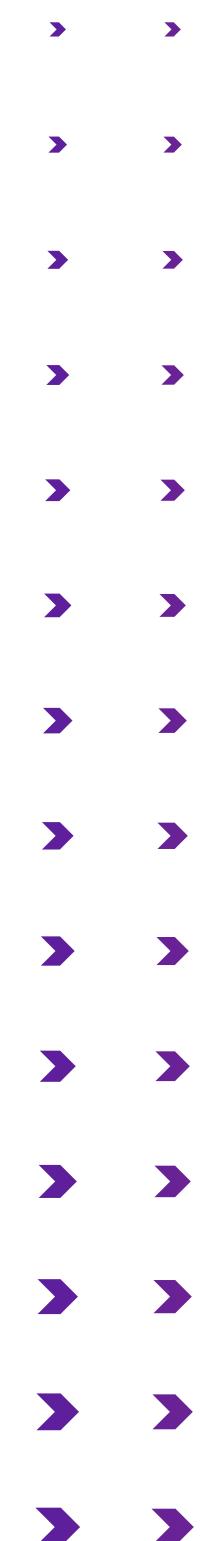
# Gestión de Memoria

- Se crea un sistema propio de seguimiento de memoria:
  - **shell\_malloc()** y **shell\_free()** controlan asignaciones.
  - **shell\_alloc\_count()** devuelve el número de bloques activos.
- Comando interno meminfo muestra la cantidad de asignaciones dinámicas en uso.
- Mejora la depuración y evita fugas de memoria durante la ejecución.

# Pruebas y Resultados

- Se probaron comandos básicos: pwd, cd, help, history.
- Se verificó ejecución en segundo plano (&) y pipes (|).
- Pruebas con parallel demostraron funcionamiento correcto de los hilos.
- Resultados: ejecución estable, sin fugas de memoria y correcta gestión de procesos.

```
jhon@jhon-VirtualBox:~/Documentos/producto_UI_SistemasOperativos$ ./mini_shell
mini-shell> pwd
/home/jhon/Documentos/producto_UI_SistemasOperativos
mini-shell> cd
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> history
  1  pwd
  2  cd
  3  help
  4  history
mini-shell> █
```



```

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>

```

Ingreso al mini-Shell y verificación del funcionamiento de las 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>

```

Visualización de los comandos internos

```

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

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$ ./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
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 (>>))

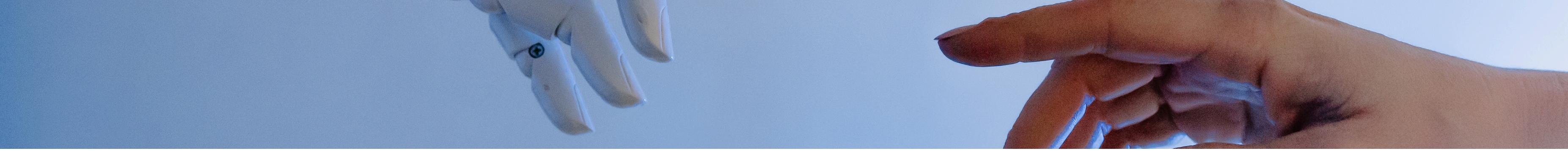
```
unjbg@unjbg-VirtualBox: ~/Escritorio/PROYECTO-FINAL
mini-shell> ls | grep cpp
mini-shell> ls src/ | grep cpp
mini_shell.cpp
mini-shell> cat include/shell.h | grep struct
// ----- Estructura de un comando -----
struct Command {
struct ParsedLine {
mini-shell> ps aux | grep bash
unjbg      9011  0.0  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
[HILLO] Ejecutando: ls
[HILLO] Ejecutando: sleep 2
[HILLO] Ejecutando: echo Terminado
Terminado
datos.txt  g.txt  log.txt  README.md  src
f.txt      include  mini_shell  salida.txt
[HILLO] Todos los comandos paralelos han finalizado.
mini-shell> parallel ls; pwd; date
[HILLO] Ejecutando: ls
[HILLO] Ejecutando: pwd
[HILLO] 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
[HILLO] 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$
```



# Conclusiones

El desarrollo de la mini shell permitió aplicar los principales conceptos de los sistemas operativos, como la creación y gestión de procesos, el manejo de señales, la sincronización mediante hilos y la redirección de flujos de entrada y salida. A través de la implementación en C++, se logró comprender cómo interactúan las llamadas al sistema POSIX con el hardware y el entorno del usuario, reforzando los conocimientos teóricos con una experiencia práctica de programación a bajo nivel.

Asimismo, el proyecto demostró la importancia de la modularidad y la gestión eficiente de recursos en el diseño de software de sistemas. La integración de funcionalidades como la ejecución en segundo plano, el uso de pipes, la memoria instrumentada y los comandos internos permitió construir una herramienta estable, funcional y extensible, capaz de replicar comportamientos esenciales de un intérprete de comandos real.



# Trabajos futuros

Se plantea optimizar el manejo de múltiples pipes encadenados, implementar un historial persistente de comandos, mejorar la detección y gestión de errores, y añadir soporte para autocompletado y variables de entorno. También se propone incorporar una interfaz gráfica básica y ampliar las capacidades de concurrencia para ejecutar varios comandos en paralelo con mayor eficiencia.

**¡Muchas  
gracias!**

