Universidad de Concepción

# Sistemas Operativos:
# Tarea 1: Implementación de una Shell

**Integrantes: Kurt Koserak Ortiz**

**Benjamín López Hermosilla**

**Ariel Fernández Fuentealba**

**Docente: Cecilia Hernández**

# Introducción:

Como equipo, nuestra shell se diseñó siguiendo el modelo clásico de intérprete de comandos: bucle de lectura–ejecución, manejando los procesos correspondientes mediante fork/exec, soporte para redirecciones vía pipes, y comandos internos para operaciones que afectan al proceso padre. Elegimos este enfoque porque nos acerca a cómo funciona una shell real en Unix/Linux, a diferencia de soluciones más simples que solo encapsulan system(). Nuestra implementación consideró varias etapas de decisión las cuáles comenzaremos a detallar en el presente informe.

## 0. Enlace a Repositorio: https://github.com/FurioStyle/Shell.git

## 1. Estructura base: el bucle principal

Básico, pero esencial es contar con un **loop infinito** (while(true)) en cualquier shell ya que esta debe permanecer activa esperando comandos del usuario todo el tiempo hasta que este sea quien decida hasta cuando el programa esté en proceso.

```
94  int main() {
95      while (true) {
96          // Mostrar prompt
97          char cwd[PATH_MAX];
98          if (!getcwd(cwd, sizeof(cwd))) { perror("getcwd"); continue; }
99          cout << COLOR_YELLOW "-SOShell-" COLOR_RESET << COLOR_BLUE << cwd << COLOR_RESET << "-$ ";
```

Con esto, garantizamos que la shell funcione de forma interactiva, no como un programa que se ejecuta una sola vez.

## 2. Personalización y comodidad: prompt y colores

Se consideró el buen manejo de comandos como “getcwd()”, obteniendo así el directorio actual y luego mostrándolo en el prompt. Además, en esta etapa consideramos usar **colores ANSI** (\033) con la finalidad de destacar tanto el nombre de la shell como el path.

```
-SOShell-/home/kurt/Shell-$ ls -la | grep
Modo de empleo: grep [OPCIÓN]... PATRONES [FICHERO]...
Pruebe 'grep --help' para más información.
-SOShell-/home/kurt/Shell-$ ls -la | grep hola
-SOShell-/home/kurt/Shell-$ ls -la | grep kurt
drwxrwxr-x  4 kurt kurt  4096 sep 26 18:10 .
drwxr-x--- 29 kurt kurt  4096 sep 25 17:50 ..
-rwxrwxr-x  1 kurt kurt 72360 sep 26 18:10 a.out
drwxrwxr-x  8 kurt kurt  4096 sep 26 18:06 .git
-rw-rw-r--  1 kurt kurt  3355 sep 26 17:18 miprof.cpp
-rw-rw-r--  1 kurt kurt   139 sep 26 17:18 miprof.h
```

Si bien, puede considerarse como detalles meramente estéticos, desde el perfil de usuarios decidimos mejorar estos aspectos ya que consideramos que una shell no solo debe ser funcional, sino también legible y cómoda.

### 3. Lectura de entrada

Para la entrada de comandos consideramos el uso de getline(cin, input), capturando así la línea de comandos completa, considerando también la excepción de entrada vacía, a nivel de procesos, evitamos ejecutar basura.

```
101             string input;
102             if (!getline(cin, input)) break;
103             if (input.empty()) continue;
104
105             // Separar la línea en tokens
```

Así, en este apartado consideramos importante que la Shell pueda leer toda la línea, permitiendo mejorar el manejo de comandos con argumentos, esto implica mayor facilidad para luego separar tokens y detectar pipes.

### 4. Tokenización

Para la comprensión de tokens decidimos separar la línea de entrada usando "istringstream". La ventaja de esto es permitirnos diferenciar el comando principal (ls, cd, etc.) de sus argumentos (-l, /home).

```
105             // Separar la línea en tokens
106             istringstream iss(input);
107             vector<string> tokens;
108             string token;
109             while (iss >> token) tokens.push_back(token);
110             if (tokens.empty()) continue;
```

En este punto, poder tokenizar es crucial debido a que execvp requiere un arreglo char*[] que reciba los argumentos de forma independiente.

### 5. Comandos internos (built-in)

La parte esencial de funcionalidad es ser capaces de implementar casos especiales para los siguientes comandos:

- exit: termina la shell.
- cd: cambia el directorio de trabajo con chdir().
- miprof: comando personal implementado, manejado por handleMiprof().

```
112             // Comando exit
113             if (tokens[0] == "exit") break;
114
115             // Comando cd
116             if (tokens[0] == "cd") {
117                 const char* path = (tokens.size() > 1) ? tokens[1].c_str() : getenv("HOME");
118                 if (chdir(path) != 0) perror("cd error");
119                 continue;
120             }
121
122             // Comando miprof
123             if (tokens[0] == "miprof") {
124                 handleMiprof(tokens);
125                 continue;
126             }
```

Algo importante de recalcar es que no todos los comandos deben ejecutarse con execvp (por ejemplo: cd modifica el proceso actual, no uno hijo). Como implementación básica (pero esencial) en una Shell real en su etapa más mínima de desarrollo debería reconocer al menos “cd” y “exit”.

## 6. Ejecución de comandos simples

Como parte de la implementación está la función “executeSimpleCommand()” la cuál es el esqueleto de ejecución conformado por estos aspectos:

- Se llama a fork() → crea un proceso hijo.
- En el hijo: execvp() reemplaza el proceso con el comando solicitado.
- En el padre: waitpid() espera a que termine el hijo.

```
76  void executeSimpleCommand(const string &command) {
77      vector<char*> args = parseCommand(command);
78      if (args.empty() || args[0] == nullptr) return;
79
80      pid_t pid = fork();
81      if (pid == 0) {
82          if (execvp(args[0], args.data()) == -1) cerr << "Comando no reconocido: " << args[0] << endl;
83          exit(EXIT_FAILURE);
84      } else if (pid > 0) {
85          int status;
86          waitpid(pid, &status, 0);
87      } else {
88          perror("fork");
89      }
90
```

Con esta implementación (estándar Unix) aseguramos aislamiento entre procesos: si un comando falla, aseguramos que la Shell no caiga por completa.

## 7. Ejecución de pipelines (|)

Como paso final en el flujo de ejecución detectamos si hay un pipe (|) en los tokens, y en base a esto definimos el flujo básico:

- Si lo hay:
    - Se divide la línea en comandos separados.
    - Creamos tuberías (pipe()), conectando la salida del comando anterior con la entrada del siguiente.
    - Usamos dup2() para redirigir correctamente stdin y stdout.
    - Manejamos de forma múltiples los pipes en cascada (ls | grep cpp | wc -l).

```
128             // ======= Verificar si hay pipes =======
129             bool hasPipe = false;
130             for (auto &t : tokens) if (t == "|") { hasPipe = true; break; }
131
132             if (hasPipe) {
133                 // Separar la línea en comandos por pipes
134                 vector<string> commands;
135                 string cmd;
```

En nuestra implementación, la detección de pipes permite ejecutar comandos encadenados de forma flexible, para cualquier shell moderna, esto es fundamental.

## 8. Manejo de errores

Por último, manejamos las excepciones posibles aun cuando el programa se encuentre el final del flujo básico de ejecución. Si un comando no existe, muestran un mensaje claro: “Comando no reconocido: nombre”.

```
80          pid_t pid = fork();
81          if (pid == 0) {
82              if (execvp(args[0], args.data()) == -1) cerr << "Comando no reconocido: " << args[0] << endl;
83              exit(EXIT_FAILURE);
84          } else if (pid > 0) {
85              int status;
86              waitpid(pid, &status, 0);
87          } else {
88              perror("fork");
89          }
```

- También se considera “perror()” en casos de fork, pipe, chdir, etc.

Algo fundamental en cualquier programa es el feedback claro y continuo al usuario, una shell, esto es aún más relevante, guiando así a corregir errores en la ejecución.

## 9. Implementación excepcional: comando “miprof”

Nuestra implementación contempló el comando “miprof” como una extensión de nuestra shell para ejecutar programas con medición de recursos. La implementación se basa en el mismo esquema de ejecución con fork y execvp, pero agregamos la recolección de métricas usando getrusage y chrono, esto quiere decir podemos hacer que datos propios del sistema o programa sean observables, lo cual nos permiten evaluar su funcionamiento.

Nuestra implementación nos permite reportar tiempos de CPU, tiempo real y uso de memoria. Para aumentar aún más su utilidad, incorporamos la posibilidad de limitar la ejecución mediante “alarm”, así como guardar resultados en un archivo si el usuario lo requiere.

Nótese lo siguiente, la lógica de ejecución se encuentra en la función “handleMiprof”. Con esta decisión nos podemos permitir mantener la shell de forma modular, aplicando principios como el encapsulamiento podemos hacer de nuestro código más consistente y extensible.

```
60    // ============================
61    // Procesar comando "miprof"
62    // ============================
63    void handleMiprof(const vector<string> &tokens) {
64        if (tokens.size() < 3) {
65            cerr << "Uso: miprof [ejec|ejcsave archivo|ejecutar maxtime] comando args...\n";
66            return;
67        }
68
69        if (tokens[1] == "ejec") {
70            vector<char*> args;
71            for (size_t i=2; i<tokens.size(); i++) {
72                args.push_back(strdup(tokens[i].c_str()));
73            }
74            args.push_back(nullptr);
75            runWithProfile(args);
```

## Conclusión:

Nuestra propuesta de implementación de una shell nos representa una aplicación directa de contenido teórico fundamental en el estudio de sistemas operativos. Tanto en la división de etapas de desarrollo como la tokenización, el manejo de procesos mediante llamadas a sistema (fork, exec) y una buena sincronización con “wait”, podemos traducir la teoría en una herramienta completamente funcional.

Haciendo mención del punto 6, el denominado “esqueleto de ejecución” de nuestra shell muestra con claridad la separación entre procesos: padre e hijo, lo que para nosotros es una reflexión evidente del rol e importancia del paralelismo: mientras un proceso principal continúa gestionando la interacción con el usuario, los procesos hijos trabajan en paralelo las instrucciones del usuario.

De este modo, nuestro trabajo concluye no solamente con la construcción de una shell básica, sino que, con una aplicación directa de principios de concurrencia y paralelismo, dejando así en evidencia que todos estos conceptos podemos materializarlos en un programa funcional, concreto y dinámico, lo que nos lleva a evaluar hasta que punto podríamos alcanzar en futuras implementaciones.