Resolucion de la tercer parte

Flujo estándar (redirecciones)

Como el esqueleto se encarga de parsear los nombres de archivos destino de las redirecciones solo es necesario implementar el comportamiento en `exec.c` :

```
void handle_redir(struct cmd *cmd) {
    struct execcmd *execcmd = (struct execcmd *) cmd;

    int in_fd = STDIN_FILENO;
    int out_fd = STDOUT_FILENO;
    int err_fd = STDERR_FILENO;
    if (strncmp(execcmd->in_file, "", strlen(execcmd->in_file)) != 0) {
        in_fd = open(execcmd->in_file, O_RDONLY);
        if (in_fd < 0) {
            perror("redir in_file open");
            return;
        }
    }

    if (strncmp(execcmd->out_file, "", strlen(execcmd->out_file)) != 0) {
        if (execcmd->out_file[0] == '>') {  // append
            out_fd = open(execcmd->out_file + 1, O_CREAT | O_APPEND | O_WRONLY, FILE_PERMISSIONS); // NOLINT
        } else {
            out_fd = open_redir_fd(execcmd->out_file);
        }

        if (out_fd < 0) {
            return;
        }
    }

    if (strncmp(execcmd->err_file, "", strlen(execcmd->err_file)) != 0) {
        if (strncmp(execcmd->err_file, "&1", strlen(execcmd->err_file)) == 0) {
            err_fd = STDOUT_FILENO;
        } else {
            err_fd = open_redir_fd(execcmd->err_file);
            if (err_fd < 0) {
                return;
            }
        }
    }
    dup2(in_fd, STDIN_FILENO);
    dup2(out_fd, STDOUT_FILENO);
    dup2(err_fd, STDERR_FILENO);
    handle_exec(cmd);
}
```

`handle_exec` es la función de la parte 1 para ejecutar un comando común.

```
$ ls -C /home /noexiste >out.txt 2>&1

$ cat out.txt
---????---
```

> Investigar el significado de este tipo de redirección y explicar qué sucede con la salida de `cat out.txt` . Comparar con los resultados obtenidos anteriormente.

La redirección `2>&1` implica redireccionar la salida de error a la salida estándar. La primera redirección, `>out.txt` , redirecciona la salida estándar al archivo `out.txt` . La combinación de ambos, entonces, redirecciona tanto la salida estándar como la de error a `out.txt` El resultado del comando `ls` , entonces, es el listado de contenidos correcto de `/home/` , más el mensaje de error de que el archivo `noexiste` no fue encontrado. El efecto de esto comparando con el inciso anterior es que toda salida fue redireccionada al mismo archivo.

> Challenge: investigar, describir y agregar la funcionalidad del operador de redirección >> y &>

El operador `>>` hace un *append* de la salida del comando a un archivo destino, es decir, no se trunca el archivo antes de escribir. `&>` es equivalente a `2>&1` seguido de una redirección `>`, es decir, se redirigen tanto salida de estándar como de error al mismo archivo. La implementación de `>>` puede ser vista en el código anterior, mientras que la funcionalidad de `&>` fue implementada modificando el archivo `parsing.c`, en `parse_redir_flow`:

```
      switch (outIdx) {
          // stdout redir
          case 0: {
              strcpy(c->out_file, arg + 1);
              break;
          }
              // stderr redir
          case 1: {
+             if (arg[0] == '&') {  // stdout redir too, '&>'
+                strcpy(c->out_file, &arg[outIdx + 1]);
+             }
              strcpy(c->err_file, &arg[outIdx + 1]);
              break;
          }
      }
```

Y `parse_cmd`:

```
          // checks if the background symbol is after
          // a redir symbol, in which case
          // it does not have to run in in the 'back'
          if ((idx = block_contains(buf_cmd, '&')) >= 0 &&
+                         buf_cmd[idx - 1] != '>' &&
+             buf_cmd[idx + 1] != '>')
```

Estas modificaciones logran levantar el caso de que un argumento pasado comience con los caracteres `&>`, se parsee como un comando de tipo `REDIR`, que tanto el `out_file` como el `err_file` apunten al mismo archivo.

Tuberías simples (pipes)

De manera similar al ejercicio anterior, el parseo ya viene dado por el esqueleto proporcionado, queda solo implementar el manejo de comandos tipo `PIPE`:

```
  void handle_pipe(struct cmd *cmd) {
      struct pipecmd *pipecmd = (struct pipecmd *) cmd;
      int pipefds[2] = {0};

      if (pipe(pipefds) < 0) {
          perror("pipecmd: error creating pipe");
          return;
      }

      pid_t first;
      if ((first = fork()) == 0) {
          dup2(pipefds[1], STDOUT_FILENO);
          exec_cmd(pipecmd->leftcmd);
      } else {
          waitpid(first, 0, 0);
          dup2(pipefds[0], STDIN_FILENO);
          close(pipefds[1]);
          exec_cmd(pipecmd->rightcmd);
      }
  }
```

Lo que se hace es hacer un fork adicional al hijo, en donde el primer comando se ejecute primero, redireccionando su salida estándar a una punta de un *pipe*, y luego leyendo de esta en el segundo comando.

> Investigar y describir brevemente el mecanismo que proporciona la syscall pipe(2), en particular la sincronización subyacente y los errores relacionados.

`pipe` devuelve dos *file descriptors*, que hacen de tubo unidireccional de lecturas y escrituras. El primero es la punta de lectura, y el segundo es la de escritura. Todo lo que se escriba al segundo, podrá ser leído por el primero.

El manejo interno del pipe por parte del kernel es un buffer de memoria, accesible por ambos lados hasta que se hayan cerrado. El tamaño de dicho buffer está fijado en linux en 16 páginas de memoria (64k) pero puede ser modificado a través de la syscall `fnctl`.

Sobre los errores que puede lanzar la syscall, existen los típicos relacionados a la creación de manejo de archivos, cuando se llega al límite de *file descriptors* máximos permitidos por proceso, o globales en todo el sistema, y además existe un error particular de pipes, cuando se alcanza el límite global del sistema de memoria alocada a pipes.