# Tarefa 4: Redirecionamento usando pipe

June 14, 2025

## 1 Implementação

Este relatório apresenta uma implementação em linguagem C que demonstra o redirecionamento de saída padrão de um processo filho para um pipe, onde um processo pai lê os dados até que o pipe seja fechado. O programa cria um processo filho que executa o comando "ls -l" e redireciona sua saída para o processo pai através de um pipe.

### 1.1 Código Implementado

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define BUFFER_SIZE 4096

int main() {
    int fd[2];  // File descriptors for pipe: fd[0] for reading, fd[1] for writing
    pid_t pid;
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;

    // Create the pipe
    if (pipe(fd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Create a child process
    pid = fork();

    if (pid == -1) {
```

```
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        printf("Child process executing 'ls -l'...\n");

        // Close the read end of the pipe in the child
        close(fd[0]);

        // Redirect stdout to the write end of the pipe
        if (dup2(fd[1], STDOUT_FILENO) == -1) {
            perror("dup2");
            exit(EXIT_FAILURE);
        }

        // Close the write end of the pipe as it's been duplicated
        close(fd[1]);

        // Execute the command
        execlp("ls", "ls", "-l", NULL);

        // If execlp returns, it must have failed
        perror("execlp");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        printf("Parent process waiting for output from child...\n");

        // Close the write end of the pipe in the parent
        close(fd[1]);

        // Read from the pipe until EOF
        printf("\nOutput from child:\n");
        printf("------------------\n");

        while ((bytes_read = read(fd[0], buffer, BUFFER_SIZE - 1)) >
    0) {
            buffer[bytes_read] = '\0';  // Null-terminate the string
            printf("%s", buffer);
        }

        printf("------------------\n");

        if (bytes_read == -1) {
            perror("read");
```

```
72              exit(EXIT_FAILURE);
73          }
74
75          // Close the read end of the pipe
76          close(fd[0]);
77
78          // Wait for the child to terminate
79          int status;
80          waitpid(pid, &status, 0);
81
82          if (WIFEXITED(status)) {
83              printf("\nChild process exited with status %d\n",
    WEXITSTATUS(status));
84          } else {
85              printf("\nChild process did not exit normally\n");
86          }
87      }
88
89      return 0;
90  }
```

Listing 1: Implementação do Redirecionamento usando pipe

# 2 Respostas às Perguntas

## 2.1 Como funciona o mecanismo de redirecionamento?

O mecanismo de redirecionamento implementado neste programa funciona através dos seguintes passos:

1. Um pipe é criado usando a chamada de sistema `pipe(fd)`, que gera dois descritores de arquivo: `fd[0]` para leitura e `fd[1]` para escrita.

2. Após criar o processo filho com `fork()`, o processo filho executa os seguintes passos para redirecionar sua saída padrão:

   - Fecha o descritor de leitura (`fd[0]`) pois não será usado pelo filho.
   - Utiliza a chamada de sistema `dup2(fd[1], STDOUT_FILENO)` para substituir o descritor da saída padrão (STDOUT_FILENO, valor 1) pelo descritor de escrita do pipe (`fd[1]`).
   - Fecha o descritor de escrita original (`fd[1]`) já que foi duplicado.
   - Quando o processo filho executa `execlp("ls", "ls", "-l", NULL)`, toda a saída que normalmente iria para o terminal é enviada para o pipe.

3. No processo pai:

- Fecha o descritor de escrita (`fd[1]`) pois não será usado.
- Lê os dados do pipe através do descritor de leitura (`fd[0]`) até encontrar o fim do arquivo (EOF), que ocorre quando todos os descritores de escrita do pipe são fechados.
- Processa os dados lidos (no caso, imprime na tela).
- Fecha o descritor de leitura quando termina.

Este mecanismo de redirecionamento é fundamental em sistemas Unix/Linux e permite conectar a saída de um processo à entrada de outro processo, possibilitando a criação de pipelines de comandos.

## 2.2 Por que o processo pai precisa fechar o lado da escrita da pipe?

O processo pai precisa fechar o lado de escrita do pipe (`fd[1]`) por duas razões principais:

1. **Detecção de EOF**: Quando o processo pai lê do pipe usando `read(fd[0], buffer, BUFFER_SIZE)`, ele precisa saber quando parar de ler. A função `read()` retorna 0 bytes lidos apenas quando o EOF (End of File) é encontrado no pipe. O EOF só é sinalizado quando todos os descritores de escrita do pipe são fechados. Se o pai mantiver seu descritor de escrita aberto, mesmo que não esteja escrevendo nada, o `read()` nunca receberá EOF e ficará bloqueado indefinidamente, esperando por mais dados.

2. **Gestão de recursos**: Manter descritores de arquivo abertos desnecessariamente ocupa recursos do sistema. Cada processo tem um limite para o número de descritores de arquivo que pode manter abertos simultaneamente. Fechar descritores não utilizados é uma boa prática de programação.

Por isso, imediatamente após o `fork()`, o processo pai fecha `fd[1]` porque sabe que não vai escrever no pipe, apenas ler dele. Se não fechasse, a chamada `read()` poderia ficar bloqueada indefinidamente, já que o próprio processo pai manteria um descritor de escrita aberto para o pipe.

## 2.3 O que mudaria se o processo pai quisesse redirecionar a saída de erro padrão?

Para redirecionar a saída de erro padrão (stderr) do processo filho para o pipe, seria necessário fazer uma pequena modificação no código do processo filho. Além de redirecionar STDOUT_FILENO (valor 1), também seria necessário redirecionar STDERR_FILENO (valor 2) para o descritor de escrita do pipe.

O código no processo filho mudaria de:

```
// Redirect stdout to the write end of the pipe
if (dup2(fd[1], STDOUT_FILENO) == -1) {
    perror("dup2");
    exit(EXIT_FAILURE);
}
```

Para:

```
// Redirect stdout to the write end of the pipe
if (dup2(fd[1], STDOUT_FILENO) == -1) {
    perror("dup2 for stdout");
    exit(EXIT_FAILURE);
}

// Redirect stderr to the write end of the pipe as well
if (dup2(fd[1], STDERR_FILENO) == -1) {
    perror("dup2 for stderr");
    exit(EXIT_FAILURE);
}
```

Com essa modificação, tanto a saída padrão quanto os erros gerados pelo comando executado no processo filho seriam redirecionados para o pipe e, consequentemente, lidos pelo processo pai. É importante notar que, neste caso, não há distinção entre o que vem da saída padrão e o que vem da saída de erro, pois ambos são mesclados no mesmo pipe.

Para manter a distinção entre stdout e stderr, seria necessário criar dois pipes separados e lidar com cada um deles de forma independente no processo pai.

## 2.4 E se quisesse redirecionar a entrada padrão?

Para redirecionar a entrada padrão, o processo seria essencialmente o contrário do que foi implementado. Neste caso, o processo pai escreveria dados no pipe e o processo filho leria esses dados como sua entrada padrão.

As modificações necessárias seriam:

1. No processo filho:
    - Fechar o descritor de escrita do pipe (`fd[1]`)
    - Redirecionar a entrada padrão para o descritor de leitura do pipe: `dup2(fd[0], STDIN_FILENO)`
    - Fechar o descritor de leitura original `fd[0]` (já que foi duplicado)
    - Executar o comando que receberá dados pela entrada padrão
2. No processo pai:
    - Fechar o descritor de leitura do pipe (`fd[0]`)
    - Escrever dados no descritor de escrita do pipe (`fd[1]`)
    - Fechar o descritor de escrita quando terminar de enviar dados
    - Esperar o processo filho terminar

Exemplo de código modificado para o redirecionamento da entrada padrão:

```c
if (pid == 0) {
    // Child process

    // Close the write end of the pipe in the child
    close(fd[1]);

    // Redirect stdin from the read end of the pipe
    if (dup2(fd[0], STDIN_FILENO) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
    }

    // Close the read end of the pipe as it's been duplicated
    close(fd[0]);

    // Execute a command that reads from stdin, like 'wc'
    execlp("wc", "wc", NULL);

    // If execlp returns, it must have failed
    perror("execlp");
    exit(EXIT_FAILURE);
} else {
    // Parent process

    // Close the read end of the pipe in the parent
    close(fd[0]);

    // Send data to the child process through the write end of the pipe
    const char *data = "This is a test string.\nIt has multiple lines.\n";
    write(fd[1], data, strlen(data));

    // Close the write end to signal EOF to the child
    close(fd[1]);

    // Wait for the child to process the data and terminate
    wait(NULL);
}
```

Neste exemplo, o processo pai envia uma string para o processo filho, que executa o comando `wc` (word count) que conta linhas, palavras e caracteres da entrada recebida.

# 3 Conclusão

O redirecionamento usando pipes é um mecanismo fundamental nos sistemas operacionais tipo Unix, permitindo a comunicação entre processos e a criação de pipelines de comandos.

Através da combinação de chamadas de sistema como `pipe()`, `fork()`, `dup2()` e `exec()`, é possível criar programas flexíveis que aproveitam a filosofia Unix de ter programas pequenos e especializados trabalhando juntos.

A compreensão desses mecanismos é essencial para o desenvolvimento de software em ambientes Unix/Linux e forma a base para muitos padrões de design de sistemas operacionais e aplicativos de linha de comando.