Tarefa 4: Redirecionamento usando pipe

17 de junho de 2025

1 Introdução

Este relatório descreve a implementação de um programa em linguagem C que demonstra o redirecionamento de saída padrão usando pipes. O programa executa um processo filho que roda o comando `ls -l` e redireciona sua saída padrão para um pipe, cujo outro lado é lido pelo processo pai até ser fechado.

2 Implementação

O programa implementa o seguinte fluxo:

1. Cria um pipe usando a chamada de sistema `pipe()`
2. Cria um processo filho usando `fork()`
3. No processo filho:
   - Fecha o lado de leitura do pipe (não utilizado pelo filho)
   - Redireciona a saída padrão para o lado de escrita do pipe usando `dup2()`
   - Executa o comando `ls -l` usando `execlp()`
4. No processo pai:
   - Fecha o lado de escrita do pipe (não utilizado pelo pai)
   - Lê dados do lado de leitura do pipe até que seja fechado (EOF)
   - Imprime os dados na tela
   - Espera a finalização do processo filho

3 Código Fonte

Abaixo está o código fonte do programa:

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

```c
#include <string.h>

#define BUFFER_SIZE 4096
char buffer[BUFFER_SIZE];

int pipe_fd[2];

enum PID {
    PID_PARENT,
    PID_CHILD,
    PID_FAIL,
};

enum PID check(pid_t pid) {
    if (pid == 0) {
        return PID_CHILD;
    } else if (pid > 0) {
        return PID_PARENT;
    }
    return PID_FAIL;
}

void parent(pid_t pid) {
    printf("Parent process waiting for output from child...\n");

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

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

    ssize_t nof_bytes_read;

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

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

    if (nof_bytes_read == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    // Close the read end of the pipe
    close(pipe_fd[0]);

    // Wait for theprograma child to terminate
```

```
56      int status;
57      waitpid(pid, &status, 0);
58
59      if (WIFEXITED(status)) {
60          printf("\nChild process exited with status %d\n", WEXITSTATUS
                (status));
61      } else {
62          printf("\nChild process did not exit normally\n");
63      }
64  }
65
66  void child(){
67      // Child process
68      printf("Child process executing 'ls -l'...\n");
69
70      // Close the read end of the pipe in the child
71      close(pipe_fd[0]);
72
73      // Redirect stdout to the write end of the pipe
74      if (dup2(pipe_fd[1], STDOUT_FILENO) == -1) {
75          perror("dup2");
76          exit(EXIT_FAILURE);
77      }
78
79      // Close the write end of the pipe as it's been duplicated
80      close(pipe_fd[1]);
81
82      // Execute the command
83      execlp("ls", "ls", "-l", NULL);
84
85      // If execlp returns, it must have failed
86      perror("execlp");
87      exit(EXIT_FAILURE);
88  }
89
90  int main() {
91      if (pipe(pipe_fd) == -1){
92          perror("pipe");
93          exit(EXIT_FAILURE);
94      }
95
96      pid_t pid = fork();
97
98      switch (check(pid)) {
99          case PID_FAIL:
100             perror("fork");
101             exit(EXIT_FAILURE);
102             break;
103         case PID_PARENT:
104             parent(pid);
105             break;
```

```
106             case PID_CHILD:
107                 child();
108                 break;
109             default:
110                 exit(1);
111         }
112 
113         return 0;
114 }
```

4 Respostas às Perguntas

4.1 Como funciona o mecanismo de redirecionamento?

O mecanismo de redirecionamento utilizando pipes funciona através da manipulação dos descritores de arquivos do processo. No Unix/Linux, cada processo tem pelo menos três descritores de arquivos padrão: entrada padrão (stdin, 0), saída padrão (stdout, 1) e erro padrão (stderr, 2).

O redirecionamento utilizando pipes segue estas etapas fundamentais:

1. Criação do pipe: A função `pipe(pipe_fd)` cria dois descritores de arquivo conectados, onde `pipe_fd[0]` é a extremidade de leitura e `pipe_fd[1]` a extremidade de escrita.

2. Duplicação de descritores: A função `dup2(pipe_fd[1], STDOUT_FILENO)` faz com que o descritor da saída padrão (STDOUT_FILENO ou 1) passe a se referir ao mesmo arquivo que `pipe_fd[1]` (extremidade de escrita do pipe).

3. Após esta operação, qualquer saída enviada para stdout (por exemplo, através de printf) na verdade será escrita no pipe.

4. Fechamento dos descritores não utilizados: Depois de duplicar os descritores necessários, fechamos os originais para manter a higiene do sistema.

Dessa maneira, quando o processo filho executa `ls -l`, sua saída que normalmente iria para o terminal é redirecionada para o pipe, onde o processo pai pode lê-la.

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

O processo pai precisa fechar o lado de escrita do pipe por várias razões importantes:

1. Detecção de EOF (End of File): Quando o processo pai lê do pipe usando a função `read()`, ele só receberá um retorno de 0 bytes (indicando EOF) quando todas as extremidades de escrita do pipe estiverem fechadas. Se o pai não fechar sua cópia do descritor de escrita, o `read()` nunca retornará EOF, mesmo se o processo filho terminar, pois ainda existe um descritor de escrita aberto.

2. Gerenciamento de recursos: Os descritores de arquivos são recursos limitados do sistema operacional. É uma boa prática fechar descritores que não serão utilizados.

3. Prevenção de deadlocks: Em situações mais complexas com múltiplos processos, não fechar descritores não utilizados pode levar a situações de deadlock, onde processos esperam indefinidamente por dados que nunca chegarão.

Portanto, é essencial que cada processo feche os lados do pipe que não irá utilizar: o processo filho fecha a extremidade de leitura, e o processo pai fecha a extremidade de escrita.

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

Para redirecionar a saída de erro padrão (stderr) em vez da saída padrão (stdout), seriam necessárias as seguintes alterações no código:

No processo filho, ao invés de:

```
dup2(pipe\_fd[1], STDOUT_FILENO);  // Redireciona stdout para o pipe
```

Usaríamos:

```
dup2(pipe\_fd[1], STDERR_FILENO);  // Redireciona stderr para o pipe
```

Onde `STDERR_FILENO` é a constante para o descritor de arquivo do erro padrão (geralmente 2).

Alternativamente, se quiséssemos redirecionar tanto stdout quanto stderr para o pipe, poderíamos usar ambas as chamadas:

```
dup2(pipe\_fd[1], STDOUT_FILENO);  // Redireciona stdout para o pipe
dup2(pipe\_fd[1], STDERR_FILENO);  // Redireciona stderr para o pipe
```

O restante do código permaneceria essencialmente o mesmo, já que o processo pai leria do pipe da mesma forma, independentemente de a saída vir de stdout ou stderr do processo filho.

4.4 E se quisesse redirecionar a entrada padrão?

Para redirecionar a entrada padrão, teríamos que inverter o fluxo de dados no pipe. No caso de redirecionamento da entrada padrão:

1. O processo pai escreveria dados no pipe
2. O processo filho leria esses dados como sua entrada padrão

As alterações necessárias no código seriam:

No processo filho:

```
// Fechar o lado de escrita (não será usado pelo filho)
close(pipe\_fd[1]);

// Redirecionar a entrada padrão para o lado de leitura do pipe
dup2(pipe\_fd[0], STDIN_FILENO);

// Fechar o descritor original de leitura após duplicação
close(pipe\_fd[0]);
```

```
9
10 // Executar um comando que leia da entrada padrão
11 execlp("sort", "sort", NULL);  // Exemplo: o comando sort lê da stdin
```

No processo pai:

```
1 // Fechar o lado de leitura (não será usado pelo pai)
2 close(pipe\_fd[0]);
3
4 // Escrever dados no pipe
5 const char *data = "linha 3\nlinha 1\nlinha 2\n";
6 write(pipe\_fd[1], data, strlen(data));
7
8 // Fechar o lado de escrita para sinalizar EOF para o filho
9 close(pipe\_fd[1]);
```

Neste cenário, o processo pai enviaria dados para o pipe e o processo filho receberia esses dados como sua entrada padrão. O comando executado pelo filho (no exemplo, o comando "sort") receberia esses dados como se fossem digitados no teclado.

5 Conclusão

O uso de pipes para redirecionamento de entrada e saída é um mecanismo fundamental em sistemas Unix/Linux, permitindo a comunicação interprocessos e a construção de pipelines de comandos. A implementação correta envolve a compreensão de como os descritores de arquivo funcionam e como eles podem ser manipulados para redirecionar fluxos de dados.

Os aspectos mais importantes a serem observados são:

- Fechar os descritores não utilizados em cada processo
- Usar `dup2()` para redirecionar os fluxos padrão
- Entender como a detecção de EOF funciona em pipes

Esta técnica é extensivamente utilizada em shells e outros programas que precisam construir pipelines de processamento ou capturar a saída de comandos para processamento adicional.