Laboratório de Sistemas Operacionais
PROJETO 2

743512 - Bianca Gomes Rodrigues
743588 - Pietro Zuntini Bonfim

Outubro de 2018

1 Descrições das Tarefas

1.1 Comandos com Argumentos

A primeira tarefa permite que o nosso interpretador de comandos receba argumentos nos comandos digitados, como:

```
> cp Arquivo1 Arquivo2
> kill -9 <pid>
> ls -a
```

Para que isso fosse possível primeiro precisamos criar um ponteiro de strings (char *argumentos[MAX]) para receber os argumentos. Por conseguinte, precisamos dividir cada argumento utilizando a função strtok() com delimitadores "espaço"e "\n".

Assim, foi colocado o nome do comando na primeira posição e cada argumento em argumentos[i], em que i é uma variável inteira iniciada em 1 e incrementada enquanto não acabarem os argumentos. Ao final do loop, teremos a posição do último argumento, que será utilizada posteriormente.

Tendo o nome do comando e todos os seus argumentos, podemos falar das três chamadas de sistema utilizadas para essa tarefa:

```
fork()
waitpid(pid_t pid, int *valor_retorno, int opcoes);
execvp(const char *arquivo, char *const argv[])
```

A chamada de sistema fork() é utilizada para criar um novo processo e armazenar o pid do processo filho na variável pid. Após o fork(), caso o pid encontrado seja igual a 0 então basta utilizar a chamada execvp(argumentos[0], argumentos) para executar o comando. Caso contrário − o pid encontrado seja diferente de 0 − significa que foi criado um processo pai e é necessário utilizar a chamada de sistema waitpid(pid, NULL, 0) para esperar a finalização do processo filho.

1.2 Comandos em Segundo Plano

A segunda tarefa adiciona um suporte para que seja possível executar comandos em segundo plano, utilizando a sintaxe padrão do interpretador Unix: &. Exemplo:

```
> gnome-calculator &
```

Para que fosse possível identificar que um comando em segundo plano foi solitado, precisamos primeiro verificar se o último argumento do comando inserido foi &. Utilizamos a função strcmp(argumentos[i-2], "&"), armazenando seu retorno em uma variável **Resultado** para realizar está verificação.

Caso seja encontrado o &, o valor da variável Resultado será 0, e por conseguinte basta retirá-lo atribuindo NULL na posição, visto que o shell não sabe interpretá-lo, e executar execvp(argumentos[0], argumentos).

1.3 Redireção de Entrada/Saída

A terceira e última tarefa solicitava que fosse possível realizar redireção de entrada e saída padrão, ou seja, permitir que um programa leia e escreva em uma arquivo como a sua entrada e saída padrão. A sintaxe padrão do interpretador Unix define < para entrada e > para saída. Permite a execução de comandos como:

```
> ls > arquivos.txt
> sort < arquivos.txt
> sort -r < arquivos.txt
```

Para que fosse possível a execução dessa tarefa, foram utilizadas (além de fork(), waitpid() e execvp()) as funções strcmp() para verficar a existência de um dos redirecionadores, e freopen() para redirecionar tanto para a entrada padrão stdin quanto para a saída padrão stdout.

Após verificar o último argumento do comando inserido e verificar que não se trata da solicitação de uma execução em segundo plano (&), verificamos o penúltimo argumento do comando inserido. Caso seja $>-$ saída $-$ precisamos utilizar a função freopen(argumentos[i-2], "w", stdout), e caso seja $<-$ entrada $-$ utilizamos a função freopen(argumentos[i-2], "r", stdin).

Escolhida a função para entrada ou saída, basta retirarmos tanto o > ou < dos argumentos atribuindo NULL naquela posição, e executar execvp(argumentos[0], argumentos).