Projetos de Programação

Projeto 1 — Shell UNIX e Recurso de Histórico

Esse projeto consiste em desenhar um programa em C para servir como uma interface de shell que aceita comandos de usuário e, então, executa cada comando em um processo separado. O projeto pode ser finalizado em qualquer sistema Linux, UNIX ou Mac OS X.

Uma interface de shell fornece um prompt ao usuário, após o qual o próximo comando é inserido. O exemplo a seguir ilustra o prompt `osh>` e o próximo comando de usuário: `cat prog.c`. (Esse comando exibe o arquivo `prog.c` no terminal usando o comando `cat` do UNIX.)

```
osh> cat prog.c
```

Uma técnica para a implementação de uma interface de shell é que, primeiro, o processo-pai leia o que o usuário insere na linha de comando (nesse caso, `cat prog.c`) e, então, crie um processo-filho separado para executar o comando. A menos que especificado de outra forma, o processo-pai espera que o filho saia antes de continuar. Isso é semelhante em funcionalidade à criação de um novo processo ilustrada na Figura 3.10. No entanto, os shells UNIX também permitem, normalmente, que o processo-filho seja executado em background ou concorrentemente. Para fazer isso, adicionamos um "e" comercial (**&**) ao fim do comando. Assim, se reescrevermos o comando acima como

```
osh> Cat prog.c &
```

o processo-pai e o processo-filho serão executados concorrentemente.

Um processo-filho separado é criado com o uso da chamada de sistema `fork ( )`, e o comando do usuário é executado com o uso de uma das chamadas de sistema da família `exec ( )` (como descrito na Seção 3.3.1).

Um programa em C que fornece as operações gerais de um shell de linha de comando é mostrado na Figura 3.36. A função `main ( )` apresenta o prompt `osh>` e descreve os passos a serem executados depois que a entrada do usuário tenha sido lida. A função `main ( )` executa um loop contínuo enquanto `should_run` for igual a 1; quando o usuário inserir `exit` no prompt, seu programa tornará `should_run` igual a 0 e terminará.

Esse projeto é organizado em duas partes: (1) criação do processo-filho e execução do comando nesse processo, e (2) modificação do shell para permitir um recurso de histórico.

Parte I — Criando um Processo-Filho

A primeira tarefa é modificar a função `main ( )` da Figura 3.36 para que um processo-filho seja criado e execute o comando especificado pelo usuário. Isso demandará a análise do que o usuário inseriu em tokens separados e o armazenamento dos tokens em um array de strings de caracteres (`args` na Figura 3.36). Por exemplo, se o usuário der entrada no comando `ps –ael` no prompt `osh>`, os valores armazenados no array `args` são:

```
args [0] = "ps"
args [1] = "-ael"
args [2] = NULL
```

Esse array `args` será passado à função `execvp ( )`, que tem o seguinte protótipo:

```
execvp (char *command, char *params[];
```

Aqui, `command` representa o comando a ser executado e `params` armazena os parâmetros para esse comando. Para esse projeto, a função `execvp ( )` deve ser invocada como `execvp (args[0], args)`. Certifique-se de que o usuário

incluiu um & para determinar se o processo-pai deve ou não esperar que o filho seja encerrado.

Parte II — Criando um Recurso de Histórico

A próxima tarefa é modificar o programa de interface de shell para que ele forneça um recurso de ***histórico*** que permita ao usuário acessar os comandos inseridos mais recentemente. O usuário poderá acessar até 10 comandos usando o recurso. Os comandos serão numerados consecutivamente começando em 1, e a numeração continuará depois de 10. Por exemplo, se o usuário tiver inserido 35 comandos, os 10 comandos mais recentes serão numerados de 26 a 35.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80 /* O comando de tamanho máximo */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* argumentos de linha de comando */
int should_run = 1; /* flag para determinar quando encerrar o programa */

  while (should_run) {
    printf("osh>");
    fflush(stdout);

    /**
    * Após a leitura da entrada do usuário, os passos são:
    * (1) cria um processo-filho usando fork ( )
    * (2) o processo-filho invocará execvp ( )
    * (3) se o comando incluir &, o pai invocará wait ( )
    */
  }

  return 0;
}
```

Figura 3.36 Descrição de um shell simples.

O usuário poderá listar o histórico de comandos inserindo o comando

```
history
```

no prompt `osh>`. Como exemplo, suponha que o histórico seja composto pelos comandos (do mais recente ao menos recente):

```
ps, ls, -1, top, cal, who, date
```

O comando `history` dará saída em:

```
6 ps
5 ls -1
4 top
3 cal
2 who
1 date
```

Seu programa deve suportar duas técnicas de recuperação de comandos do histórico de comandos:

1. Quando o usuário dá entrada em !!, o comando mais recente do histórico é executado.
2. Quando o usuário dá entrada em apenas um !, seguido por um inteiro *N*, o *N*-enésimo comando do histórico é executado.

Continuando nosso exemplo anterior, se o usuário dá entrada em !!, o comando `ps` é executado; se o usuário dá entrada em !3, o comando `cal` é executado. Qualquer comando executado dessa forma deve ser ecoado na tela do usuário. O comando também deve ser inserido no buffer do histórico como o próximo comando.

O programa também deve gerenciar um tratamento de erros básico. Se não há comandos no histórico, a entrada de !! deve resultar em uma mensagem **"`Nenhum comando no histórico`"**. Se não há comando correspondente ao número inserido com !, o programa deve exibir **"`Nenhum comando correspondente no histórico`"**.