jsh – Interface de Linha de Comando
Proj1

Informações

- Trabalho em **duplas**;
- Entregar em <`http://trab.dc.unifil.br/moodle/`>

1 Introdução

Neste trabalho, implementaremos uma Interface de Linha de Comando (CLI) em Java, nos moldes do *bash*, *tcsh*, *fish*, *zsh*, *cmd.exe*, *PowerShell*, dentre tantos outros. Obviamente a nossa versão será mais simples, porém completamente funcional para atividades básicas como: verificar o relógio; manipular e navegar pelo sistema de arquivos; lançar novos processos rodando outros programas.

O objetivo deste trabalho é nos aproximarmos das interfaces de programação (APIs) mais baixas de um sistema operacional, que são as **chamadas ao sistema**. Especificamente, vamos conhecer as principais chamadas oferecidas pela *Java Virtual Machine (JVM)*, que possuem a vantagem de serem traduzidas para o real sistema operacional hospedeiro sem precisarmos reprogramar ou recompilar o programa, enquanto provêm funcionalidade similar.

Este é um trabalho de caráter investigativo que contempla conteúdo extra-aula, o que significa que vocês deverão pesquisar, procurando livros, artigos e documentos da internet. Significa também que quem deixar para a última hora não terá tempo hábil para fazê-lo.

2 Ferramentas e Instruções

A correção deste trabalho será feita em apresentação e interrogatório oral, em que o programa deverá rodar como `.jar`, a partir de um terminal Linux com o seguinte comando: `java -jar jsh`.

2.1 Desconto de Pontos

O trabalho deve ser feito seguindo às normas estabelecidas nesta seção. Para cada norma não cumprida haverá um desconto de pontos de acordo com a tabela seguinte:

Regra quebrada	Desconto
Entrega em formato diferente de `jar`, incluindo os fontes	-20
Cada correção no trabalho durante a apresentação	-10
Não comparecer na apresentação	-50%

3 Bibliografia

Para este trabalho, recomenda-se a utilização das seguintes fontes para pesquisa e aprendizado dos tópicos pertinentes:

1. ORACLE. Lesson: Basic I/O. The Java™Tutorial > Essential Classes. Disponível em: <`https://docs.oracle.com/javase/tutorial/essential/io/`>.
2. ORACLE. ProcessBuilder. Java Platform SE 8. Disponível em: <`https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html`>.
3. ORACLE. Java Platform SE 8. Disponível em: <`https://docs.oracle.com/javase/8/docs/api/`>.
4. TANENBAUM, Andrew S. Sistemas operacionais modernos. Rio de Janeiro: Prentice-Hall, 2010. 672 p. ISBN 9788576052371.
5. SCHILDT, Herbert. C completo e total. 3. ed. São Paulo: Makron Books, 1996. 827 p. ISBN 8534605955

Os dois primeiros itens são de leitura obrigatória, ensinando como manipular arquivos e diretórios e como lançar novos processos de programas terceiros em Java, respectivamente.

4 Roteiro de Atividades

Este roteiro de trabalho inclui o pacote `jsh.zip`, que possui os arquivos de código-fonte básicos do *jsh*, nosso shell. Estes arquivos têm algumas funcionalidades e estrutura do programa pré-desenvolvidas. O que cada aluno deverá fazer é incrementá-los de acordo com o roteiro de atividades que se inicia a seguir:

1. (15 pontos) Implemente o método `exibirPrompt()`, que escreve o prompt de linha de comando do shell, que indica qual o usuário está utilizando e qual o diretório de trabalho atual. O prompt do nosso shell é da forma "`login#UID:workdir>`". Como exemplo, se tivermos um usuário cujo *UID* é `1337`, *nome de login* é "`alunos2012`", e o *diretório atual de trabalho*[1] do processo do shell é "`/home/shared/`", o prompt será da seguinte forma:

   ```
   alunos2012#1337:/home/shared/%
   ```

2. (15 pontos) O método `lerComando()` é responsável por ler o comando digitado pelo usuário e criar uma instância da classe `ComandoPrompt`, que será utilizada adiante pelo programa do shell para interpretar e executar o comando. As especificações desse método e do construtor de `ComandoPrompt` encontram-se como *javadocs* no código-fonte. Leia-as e implemente-as.

[1] *Current working directory.*

3. (30 pontos) O método `executarComando()` recebe o comando lido e seus parâmetros para realizar uma ação, desde que o comando seja entendido e suportado. A lista de comandos suportados é a seguinte:
   (a) `encerrar`: termina o programa retornando 0 (significa que terminou com sucesso, sem erros);
   (b) `relogio`: exibe a data e a hora atual do sistema. Implementar esta ação no método `exibirRelogio()`;
   (c) `la`: lista os nomes de todos os arquivos e diretórios do atual diretório de trabalho, um em cada linha. Implementar esta ação no método `escreverListaArquivos()`.
   (d) `cd`: cria um novo diretório com nome definido pelo primeiro argumento em `ComandoPrompt::getArgumentos`. Este diretório é colocado no diretório atual de trabalho. Implementar esta ação no método `criarNovoDiretorio`;
   (e) `ad`: apaga o diretório especificado pelo primeiro argumento em `ComandoPrompt::getArgumentos`. Implementar esta ação no método `apagarDiretorio`;
   (f) `mdt`: troca o atual diretório de trabalho pelo especificado pelo primeiro argumento em `ComandoPrompt::getArgumentos`. Implementar esta ação no método `mudarDiretorioTrabalho()`;

4. (40 pontos) O nosso shell deve ser capaz de executar comandos de terceiros, ou seja, outros programas instalados no computador. Essa funcionalidade é cumprida pelo método `executarPrograma()`, que é chamada por `executarComando()` caso o comando não seja nenhum dos pré-definidos pela questão 3. Para ficar mais claro, siga a sequência:
   1. `executarComando()` recebe o comando e argumentos digitados pelo usuário;
   2. `executarComando()` verifica se o comando digitado se refere a alguma das funcionalidades embutidas no shell, implementadas na questão 3;
   3. Se comando não for nenhum dos embutidos, `executarComando()` repassa o comando e argumentos para `executarPrograma()`;
   4. `executarPrograma()` obtém uma lista com todos os arquivos do diretório de trabalho atual e verifica se algum deles tem o mesmo nome do comando. Se não houver, o shell indica que o comando ou programa não existe e abre o prompt para o usuário;
   5. Se houver um arquivo com o mesmo nome de comando, `executarPrograma` verifica se esse arquivo tem permissão de execução. Se não houver, o terminal indica que o arquivo não tem permissão para execução;
   6. Se o arquivo tiver permissão, `executarPrograma()` cria um novo processo[2], que inicia execução enquanto o processo do shell espera por seu término para prosseguir para o próximo comando.

[2]Ver classe ProcessBuilder do Java.

7. Quando o processo lançado se encerrar, o shell volta ao prompt, lançando uma mensagem indicando erro na execução do processo filho caso seu valor de retorno seja diferente de 0.

5 Exemplo de uso do *jsh*

A seguir, a transcrição de um exemplo de uso de um jsh totalmente implementado e funcional, em um sistema com as seguintes características:

- Nome de login do usuário: `professor`
- PID deste usuário: `1001`
- Data e hora do uso: 20/05/2014, às 14:45
- Atual diretório de trabalho: `/home/professor/`
- Arquivos deste diretório:
  - `hobbit.txt`
  - `gabaritos.txt`
  - `Documentos`
  - `mesg_do_dia`
  - `mesg_do_dia.c`
  - `falha_arbitraria`
  - `falha_arbitraria.c`
- `mesg_do_dia` é um programa que escreve "The only way around is through." e retorna 0
- `falha_arbitraria` é um programa que escreve "Invalid arguments. Please, RTFM." e retorna 1 (erro!)

Transcrição do uso:

```
professor#1001:/home/professor/% relogio
Sao 14:45 de 20/05/2014.
professor#1001:/home/professor/% la
Documentos
falha_arbitraria
falha_arbitraria.c
gabaritos.txt
hobbit.txt
mesg_do_dia
mesg_do_dia.c
```

```
professor#1001:/home/professor/% cd Musicas
professor#1001:/home/professor/% cd Sandbox
professor#1001:/home/professor/% ad Musicas
professor#1001:/home/professor/% la
Documentos
falha_arbitraria
falha_arbitraria.c
gabaritos.txt
hobbit.txt
mesg_do_dia
mesg_do_dia.c
Sandbox
professor#1001:/home/professor/% mdt Sandbox
professor#1001:/home/professor/% la
professor#1001:/home/professor/Sandbox/% mdt /home/professor
professor#1001:/home/professor/% teste
Comando ou programa 'teste' inexistente.
professor#1001:/home/professor/% hobbit.txt
Arquivo 'hobbit.txt' não tem permissão para execução.
professor#1001:/home/professor/% mesg_do_dia
The only way around is through.
professor#1001:/home/professor/% falha_arbitraria
Invalid arguments. Please, RTFM.
ERRO: o programa indicou termino com falha!
professor#1001:/home/professor/% encerrar
```

5.1 mesg_do_dia e falha_arbitraria

Ambos programas são escritos em linguagem C. O código de `mesg_do_dia.c` é:

```
1 #include <stdio.h>
2
3 int main() {
4     printf("The only way around is through.\n");
5     return 0;
6 }
```

O código de `falha_arbitraria.c` é:

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Invalid arguments. Please, RTFM.\n");
5     return 1;
6 }
```

Para compilar ambos, é necessário ter instalado no computador um compilador C (`gcc` ou `clang`, como sugestão) e executar os comandos

```
gcc -o mesg_do_dia mesg_do_dia.c
gcc -o falha_arbitraria falha_arbitraria.c
```

ou

```
clang -o mesg_do_dia mesg_do_dia.c
clang -o falha_arbitraria falha_arbitraria.c
```