EXERCÍCIO-PROGRAMA 1: MAC422SHELL

SISTEMAS OPERACIONAIS – MAC0422

RENATO LUI GEH
NUSP: 8536030

1. Introdução

O EP foi feito em um Minix 3.1.2a simulado pela VM VirtualBox. Os arquivos fonte estão localizados em `/usr/local/src/mac422shell` e o binário compilado em `/usr/local/bin/` com o nome de `mac422shell`.

Ao rodar o shell `mac422shell` o prompt do shell terá o seguinte aspecto:

```
[r] pwd $>
```

Onde `r` é o valor de saída do último comando executado e `pwd` é o caminho absoluto em que `mac422shell` foi executado.

Existem nove comandos pré-existentes na shell:

- ls
  - Lista os conteúdos do diretório `pwd`. Similar (mas não igual) a `rodeveja \bin\ls`.
  - Uso: `ls`
- pwd
  - Retorna o diretório atual. Similar (mas não igual) a `rodeveja \bin\pwd`.
  - Uso: `pwd`
- exit
  - Sai do shell.
  - Uso: `exit`
- true
  - Retorna um valor de saída 0.
  - Uso: `true`
- false
  - Retorna um valor de saída 1.
  - Uso: `false`
- protegepracaramba

Date: 23 de agosto de 2016.

  - Dá ao arquivo dado como parâmetro proteção `000`.
  - Uso: `protegepracaramba filename`
- liberageral
  - Dá ao arquivo dado como parâmetro proteção `777`.
  - Uso: `liberageral filename`
- rodeveja
  - Roda um executável e em seguida retorna o valor de saída na saída padrão.
  - Uso: `rodeveja command [arguments]`
- rode
  - Roda um executável em background e imprime a saída na saída padrão.
  - Uso: `rode command [arguments]`

Além destes comandos, podemos também rodar um comando externo dando o nome do executável. Por exemplo:

```
$ /bin/ls -la
```

Rodar o comando externo tem um comportamento semelhante a usar `rodeveja`, com a diferença de que rodar apenas o comando não irá imprimir o valor de retorno do comando na saída padrão[1].

2. Organização

Iremos agora enumerar os arquivos fontes presentes em `/usr/local/src` e explicar como foi organizado o código.

O código foi separado nos seguintes arquivos:

- `args.{c,h}`
- `builtin.{c,h}`
- `proc.{c,h}`
- `prompt.{c,h}`
- `ustring.{c,h}`
- `utils.{c,h}`
- `main.c`

O código em `ustring` refere-se ao tipo `string_t`, uma estrutura para facilitar a passagem de argumentos. Guarda uma cadeia de caracteres e um inteiro indicando o tamanho. Em `utils` encontram-se algumas funções de uso geral, como macros para imprimir mensagens de erro, achar o mínimo ou máximo entre dois inteiros ou retornar o diretório atual com a chamada de sistema `getcwd`. Os arquivos `args` referem-se a estrutura que guarda argumentos dinamicamente. As funções presentes nestes arquivos manipulam a estrutura `args_t`, como veremos mais adiante.

[1] Note que usar `rodeveja` ou rodar o comando diretamente irá, em ambos os casos, modificar o valor de `r` no prompt.

Os arquivos fontes `builtin`, `proc` e `prompt` contêm o código responsável pela shell. O primeiro contém as funções que executam os comandos pré-existentes como enumerados na seção anterior. O segundo trata da criação de novos processos, seja por meio de `rodeveja`, `rode` ou executando um comando diretamente. O último cuida da entrada e da atualização do prompt.

O código em `main.c` apenas roda `prompt_readline` e `prompt_print` até que o usuário peça que a shell seja terminada.

3. Argumentos

Iremos apresentar nesta seção a estrutura de dados `args_t` que contem uma lista de argumentos que passaremos a cada função.

```
1  typedef struct {
2    /* List of string to be passed as arguments. */
3    string_t **s;
4    /* Argument count. */
5    int c;
6  } args_t;
```

Cada lista de argumentos `args_t` contém um número `(args_t).c` de argumentos. Para facilitar o uso de `args_t`, as seguintes funções de manipulação de `args_t` foram criadas:

```
1  /* Pops the first string_t from args_t *a. */
2  args_t *args_pop(args_t *a);
3
4  /* Pops the back of the list of arguments. */
5  args_t *args_pop_back(args_t *a);
6
7  /* Pushes argument s to the front of list of arguments a. */
8  args_t *args_push(args_t *a, string_t *s);
9
10 /* Pushes argument s to the back of list of arguments a. */
11 args_t *args_push_back(args_t *a, string_t *s);
```

Estas funções agem como se `args_t` fosse uma fila dupla. Iremos usar estas funções para mandarmos a shell rodar um comando em *background* ou *foreground*.

4. Comandos

Nesta seção iremos descrever cada comando pedido no enunciado do EP, explicitando quais foram as chamadas de sistema utilizados e como foram feitos.

O código para estes comandos estão em `builtin.c`. Todas estas funções contêm o prefixo `builtin_` antes do nome do comando e seguem o seguinte protótipo:

```
static int builtin_nomecomando(args_t *args);
```

4.1. *protegepracaramba*

Este comando toma como argumento um arquivo `filename`. Em seguida, `protegepracaramba` chama a chamada de sistema `chmod` com parâmetro `000`. Se a função de sistema teve sucesso, o arquivo `filename` terá agora permissão `000`. Senão, o comando `protegepracaramba` retornará o valor `1` e irá imprimir, na saída padrão, uma mensagem de erro dada pela chamada `chmod`. Se não houver erros, o comando retornará `0`.

```
1  static int builtin_protegepracaramba(args_t *args) {
2    if (args->c < 2)
3      puts("Usage: protegepracaramba filename\n  "
4           "Sets permissions to 000.");
5
6    if (chmod(args->s[1]->str, 0000) < 0) {
7      PRINT_ERR();
8      return 1;
9    }
10   return 0;
11 }
```

4.2. *liberageral*

Assim como `protegepracaramba`, o comando `liberageral` toma um arquivo `filename` e usa `chmod` para dar permissão `777` para `filename`. Se houver algum erro, o comando irá imprimir a mensagem e número de erro na saída padrão e retornar `1`. Senão, apenas retorna `0`.

```
1  static int builtin_liberageral(args_t *args) {
2    if (args->c < 2)
3      puts("Usage: liberageral filename\n  "
4           "Sets permissions to 777.");
5
6    if (chmod(args->s[1]->str, 0777) < 0) {
7      PRINT_ERR();
8      return 1;
9    }
10   return 0;
11 }
```

4.3. *rodeveja*

O comando `rodeveja` aceita como argumentos um comando `command` e argumentos para `command`.

```
$ rodeveja command [arguments]
```

Onde [·] indica uma lista de argumentos de tamanho arbitrário (e possivelmente zero). Esta função irá rodar o comando dado como parâmetro e imprimir o resultado.

```
1  static int builtin_rodeveja(args_t *args) {
2    int res;
3
4    if (args->c < 2)
5      puts("Usage: rodeveja cmd [args]\n  "
6           "Runs command cmd with arguments args "
7           "and outputs its exit status.");
8
9    args_pop(args);
10   res = proc_exec(args);
11   printf("=> programa '%s' retornou com codigo "
12          "%d.\n", args->s[0]->str, res);
13   return res;
14 }
```

Assim que passarmos `args_t *args`, teremos uma lista de argumentos cuja cabeça da lista é a string `rodeveja`. Para passarmos o comando certo dado por `command [arguments]`, temos de tirar o primeiro elemento da fila dupla `args`. Para isso usaremos `args_pop`.

Após eliminarmos a cabeça da lista, passamos o restante dos argumentos para a função `proc_exec`, que veremos na próxima seção. Note que esta função não roda nenhuma chamada de sistema explicitamente. Cabe a `proc_exec` fazer tais chamadas.

4.4. *rode*

O comando `rode` é semelhante a `rodeveja`, contendo apenas duas diferenças. Como o executável dado como argumento será rodado em *background*, iremos dizer a `proc_exec` para não esperar o processo filho acabar para retornar o valor. Além disso, não escreveremos o resultado do comando na saída padrão como fizemos em `rodeveja`.

```c
static int builtin_rode(args_t *args) {
  if (args->c < 2)
    puts("Usage: rode cmd [args]\n  "
         "Runs command cmd with arguments args.");

  args_pop(args);
  args_add(args, copy_string("&", 2));
  return proc_exec(args);
}
```

Note que, além de chamarmos `args_pop(args)` como fizemos em `rodeveja`, também adicionamos um novo argumento no final da fila `args`: a string `"&"`. Essa string diz a `proc_exec` para não esperar pelo processo filho.

5. Rodando comandos externos

Nesta seção analisaremos o módulo `proc`. Este módulo contém apenas uma função:

```c
int proc_exec(args_t *args);
```

Seja uma lista de argumentos A, e $|A| = n$. Então a função `proc_exec` segue o seguinte algoritmo:

1. Toma a string $A[0]$ como o comando a ser rodado.
2. Toma a lista de strings $A[1..n-2]$ como argumentos para $A[0]$.
3. Cria um clone do processo com `fork`.
4. Se for processo filho:
   4.1. Roda $A[0]$ dados $A[1..n-2]$ com `execve`.
   4.2. Se deu erro, imprime erro.
   4.3. Termina o processo.
5. Senão, é processo pai:
   5.1. Se $A[n-1] =$ "&", estamos rodando em *background*:
      5.1.1. Retorna o valor fixo de 0.
   5.2. Senão, temos de esperar pelo filho:
      5.2.1. Se houve erro, imprime erro e retorna o número do erro.
      5.2.2. Senão, retorna o valor de `execve`$(A[0..n-1])$.

Passos 4 e 5 estarão rodando concorrentemente. A princípio poderíamos achar que isso pode trazer algum problema, mas ao analisarmos é fácil ver que os dois eventos são independentes. Caso o filho acabe antes, então o pai saber se deve esperar ou não não importará. Se o filho acabar depois, o pai então deverá esperar pelo filho, e portanto ficará "preso" em 5.2.

Usamos nesta função duas chamadas de sistema: `fork` e `execve`. A primeira cria um clone do nosso processo no passo 3. Se o resultado desta chamada for igual

a zero, então estamos rodando no processo filho e portanto entraremos no bloco do passo 4. Caso contrário estaremos no processo pai e portanto entraremos no bloco do passo 5.

6. Compilando

Para compilar, basta rodar o Makefile em `/usr/local/src/mac422shell`. O Makefile usa o `gcc`, que no Minix que usei coloquei em `/bin/`. O binário resultante é um executável `mac422shell`, que deve ser equivalente ao já presente em `/usr/local/bin/`.

A shell `mac422shell` foi compilada com as seguintes flags:

```
-Wall -ansi -pedantic
```

Não imprimindo nenhum erro durante a compilação.