CONTEÚDO

1 INTRODUÇÃO

Neste projeto, cada equipe deverá realizar a implementação de um programa que simula o *shell* do sistema operacional e oferece os comandos para gerenciar programas de usuário. Além disso, terá a oportunidade de trabalhar com chamadas de sistemas para manipulação de processos pai e filhos.

1.1 Objetivos

- Aprender o relacionamento entre o kernel, o shell e os programas de usuário;
- Aprender como usar as chamadas de sistema do Unix para gerenciamento de processos.
- Ganhar experiência na manipulação de erros.

1.2 Requisitos Essenciais

Escrever um programa chamado *myshell*, o qual é capaz de executar, gerenciar e monitorar programas do nível de usuário. Esse programa será similar, em termos de objetivos e projeto, aos shells utilizados no dia a dia (e.g., bash, tcsh etc.), mas usará uma sintaxe um pouco diferente.

O programa *myshell* deverá ser invocado sem quaisquer argumentos e deverá suportar diferentes comandos. Ao ser executado, o programa deve imprimir um prompt similar a *myshell>* quando estiver pronto para aceitar comandos de entrada. Para isso, deve ler uma entrada de linha de comando, aceitando vários possíveis parâmetros. Por exemplo, o comando *start* iniciará outro programa com argumentos de linha de comando, imprimirá o ID do processo do programa que está rodando, e aceitará outra entrada de linha de comando:

myshell> start emacs myfile.c
myshell: processo 357 iniciado.
myshell>

Note que *emacs* irá assumir o controle do console. Esse é um comportamento aceitável se o usuário não estiver usando o console de linha de comando. Tente imaginar outros comandos que não assumam o controle do console.

O comando *wait* não precisa de argumentos e faz com que o *shell* aguarde até que o processo finalize. Quando isso acontecer, indique se o término foi normal ou não, incluindo o código de status de término (*exit code*) ou o número e nome do sinal, respectivamente. Se não existir processos que o *shell* deva aguardar, imprima uma mensagem e volte a aceitar novos comandos de entrada. Por exemplo:

myshell> wait
myshell: processo 346 finalizou normalmente com status 0.

myshell> wait
myshell: processo 347 finalizou de forma anormal com sinal 11: Segmentation fault.

myshell> wait
myshell: não há processos restantes.
myshell>

O comando *run* combina o comportamento dos comandos *start* e *wait*. O comando *run* deve iniciar um programa, possivelmente com argumentos de linha de comando, esperar que tal processo finalize e imprimir o status de término. Por exemplo:

myshell> run date
Mon Mar 19 10:52:36 EST 2019
myshell: processo 348 finalizou normalmente com status 0.
myshell>

Os comandos *kill*, *stop* e *continue* usam o ID do processo como argumento e envia um sinal, SIGKILL, SIGSTOP e SIGCONT, respectivamente, para o processo indicado. Note que um processo que é "morto" (killed) ainda requer um *wait* para que seja coletado seu status de término. Por exemplo:

myshell> kill 349
myshell: processo 349 foi finalizado.

myshell> wait
myshell: processo 349 finalizou de forma anormal com sinal 9: Killed.

myshell> stop 350
myshell: processo 350 parou a execução.

myshell> continue 350
myshell: processo 350 voltou a execução.

Depois que cada comando é concluído, seu programa deve continuar a imprimir um prompt e a aceitar outras entradas de linha de comando. O shell deve finalizar com status zero se o comando é *quit* ou *exit* ou quando alcançar um *end-of-file* (ver dicas técnicas mais adiante). Se o usuário digitar uma linha em branco (i.e., teclar *[enter]*), simplesmente deverá ser exibido o prompt aceitando uma nova entrada de linha de comando. Se o usuário digitar qualquer comando não identificado, o shell deverá imprimir uma mensagem de erro legível:

myshell> blabla ls -la
myshell: comando desconhecido: blabla

O seu programa shell deve aceitar entradas de linha de comando de até 4096 caracteres e deve manipular até 100 palavras em cada linha de comando.

1.3 Dicas Técnicas

Para realizar está atividade, cada equipe deverá pesquisar no "man pages" sobre: chamadas de sistema de gestão de processos – *fork, execvp, wait, waitpid, kill e exit;* e funções de bibliotecas diversas – *printf; fgets; strtok, strcmp, strsignal, atoi*.

Use fgets para ler uma linha de texto depois de imprimir o prompt. Note que quando você imprimir o prompt com um printf sem um salto de linha no final, o mesmo pode não ser impresso imediatamente – chame *fflush(stdout)* para forçar a saída em tela.

Quebrar uma entrada de linha de comando em palavras separadas é um pouco complicado, mas com apenas algumas linhas de código isto pode ser resolvido. Chame strtok(linha, " \t\n") na primeira vez para obter a primeira palavra e, então, chame strtok(0, "\t\n") nas demais vezes para obter as palavras restantes, até que a função retorne null. Declare um array de ponteiros char * *palavras*[100], então para cada palavra encontrada com *strtok*, armazene um ponteiro para a palavra em uma posição do vetor *palavras*. Mantena um contador (*npalavras*) para o número de palavras encontradas, então defina *palavras[npalavras] = 0*, quando a última palavra for encontrada.

Uma vez que a entrada de linha de comando foi quebrada em palavras, é possível checar, em *palavras[0],* o nome do comando. Quando necessário use *strcmp* para comparar strings e atoi para converter uma string em um inteiro.

Use *fork* e *execvp* para implementar o comando *start*. Use *wait* para implementar o comando *wait*, *waitpid* para implementar o comando *run* e *kill* para implementar *kill*, *stop* e *continue*. Procure a "man 7 signal" para obter uma lista dos sinais e suas descrições.

Certifique-se de parar quando *fgets* retornar *null*, indicando end-of-file. Isto permite rodar myshell e ler comandos de um arquivo. Por exemplo, crie um arquivo *myscript* contendo os seguintes comandos

```
start /s
wait
start date
wait
```

Então, execute o arquivo como entrada, usando:

```
./myshell < myscript
```

Nesta versão do Projeto 2, para execução de scripts, não existe a obrigação de haver um controle da ordem com que as informações irão aparecer na tela. Isto porque não é necessário ter um controle sobre qual processo (pai ou filhos) estará executando em um determinado momento.

1.4 Testes

- Certifique-se de testar o programa em uma variedade ampla de condições.
- Tente executar múltiplos programas simultaneamente.
- Crie programas simples, que finalizem com falha ou normalmente, e certifique-se que os comandos *wait* e *run* irão reportar corretamente o status de término.
- Tente executar programas iterativos como o *emacs* e use os comandos *stop*, *continue* e *kill* para ver o que acontece.
- Tenha certeza de manipular cuidadosamente todas as possíveis condições de erro. Toda chamada de sistema pode falhar de várias formas distintas. Você deve tratar todos os possíveis erros com uma mensagem de erro legível. É obrigação sua ler as "man pages" cuidadosamente e aprender quais são os erros possíveis.

2 CONSIDERAÇÕES GERAIS

- As atividades devem ser realizadas por equipes com até 5 membros.
- Durante o semestre serão marcadas apresentações nas quais os estudantes realizarão a apresentação das implementações desenvolvidas. Nestas apresentações, os estudantes serão avaliados individualmente.
- As notas serão concedidas de acordo com o desempenho em termos dos aspectos teóricos e práticos de cada estudante nas apresentações.
- Como as apresentações serão feitas a posteriori, é interessante que as equipes preparem relatórios com nível de detalhamento adequado para permitir que os conceitos e decisões de projetos utilizados sejam recordados para serem discutidos durante as apresentações.