Universidade Federal da Bahia
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Sistemas Operacionais (MATA58 e MAT566) – Semestre Letivo Suplementar 2020
Prof. Alirio Sá

PRÁTICA 02. PROCESSOS E O SHELL

traduzido e adaptado da atividade prática disponível em https://www3.nd.edu/~dthain/courses/cse30341/spring2020/project2/*, acessada em 14 de Setembro de 2020.*

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-like para gerenciamento de processos.
- Ganhar mais experiência na manipulação de erros.

1.2 Ambiente de Implementação

- Sistema operacional Linux (sugestão: Xubuntu 18.04+);
- Compilador C padrão C99
    - Compilar /usr/bin/gcc, usando opção – *std=c99 -Wall*

1.3 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.) – contudo, utilizará 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. Tal programa 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 cp myfile.c myfile2.txt
myshell: processo 357 iniciado.
myshell>

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 waitfor é similar ao comando wait, mas espera que um processo específico finalize:

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

myshell> waitfor 346
myshell: processo inexistente.

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>

O comando **watchdog** recebe um tempo limite (em segundos) e um comando que deve ser executado, então executa o comando da mesma forma que **run**. Contudo, se o comando demorar mais que o tempo limite, então o shell deve enviar um sinal SIGKILL para o processo filho, esperar até que o mesmo finalize e, então, apresentar o status da execução do mesmo. Por exemplo:

myshell> watchdog 3 sleep 10
…
myshell: finalizando processo 70, pois excedeu o tempo limite
myshell: processo 70 finalizou de forma anormal com sinal 9: terminado.
myshell>

O comando **chdir** deve fazer com que o shell mude seu diretório de trabalho para o diretório informado:

myshell> chdir /tmp

O comando pwd deve fazer com que o shell imprima seu diretório de trabalho atual:

myshell> pwd
myshell: /usr/home/so

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.

2 ASPECTOS DE IMPLEMENTAÇÃO

2.1 Dicas Técnicas

Para realizar está atividade, cada equipe deverá pesquisar no "man pages" sobre as seguintes chamadas de sistema e de bibliotecas: *fork; execvp; wait; waitpid; kill; exit; signal; printf; fgets; strtok; strcmp; strsignal; atoi; chdir; e getwd*.

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.

O comando watchdog é um pouco complicado, porque você deseja que o shell pare e espere o tempo limite expirar ou que o processo filho finalize. Nesse caso, a dica é capturar o sinal SIGCHLD, o que é entregue quando um processo filho finaliza. Use **signal** para configurar uma função como um manipulador de sinal que é chamada toda vez que SIGCHLD é entregue. Inicie o processo filho e então durma usando *sleep*. Se o sinal chegar enquanto o shell estiver dormindo, a chamada de sistema sleep será interrompida e retornará um erro indicando isso.

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* com os seguintes comandos

start ls
wait
start date
wait

Então, execute o arquivo como entrada, usando:

./myshell < myscript

2.2 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.
- 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, da mesma forma que foi solicitado no Projeto 1. É obrigação sua ler as "man pages" cuidadosamente e aprender quais são os erros possíveis.

3 Considerações Gerais

- As atividades devem ser realizadas por equipes com até 4 membros.
- Durante o semestre, serão agendados encontros online nos 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.
- A não entrega dos códigos-fontes e relatório deste laboratório prático implicará em zero na nota da avaliação. Da mesma forma, a entrega sem realização da apresentação também implicará em zero na nota da avaliação.

4 Entrega

Entregável	Data
Upload no Moodle de Arquivos fontes e relatório	29/09/2020
Encontro online para defesas dos aspectos de implementação	13/10/2020

5 Pontuação

Item	Pontuação
Funcionamento correto do shell, de acordo com a especificação	50%
Verificação e manipulação correta de todas as condições de erro	40%
Bom estilo de codificação (e.g., boa formatação do código, nomes de variáveis significativos, comentários adequados etc.)	10%