Sistemas Operacionais - Trabalho de Programação

Profª. Roberta Lima Gomes - email: soufes@gmail.com

Período: 2025/2

Data de Entrega: 18/01/2026 Composição dos Grupos: até 3 pessoas

Material a entregar

- Por email: enviar um email para **soufes@gmail.com** seguindo o seguinte formato:
  - Subject do email: "**Trabalho SO**"
  - Corpo do email: lista contendo os nomes completos dos componentes do grupo em ordem alfabética
  - Em anexo: um arquivo compactado com o seguinte nome "**nome-do-grupo.extensão**" (*ex: joao-maria-jose.rar*). Este arquivo deverá conter todos os arquivos (incluindo os *makefile*) criados com o código muito bem comentado.

Serão pontuados: clareza, indentação e comentários no programa. Além disso, o grupo será chamado para uma entrevista, em que um dos membros será sorteado para explicar como foi implementado o trabalho. Horários a serem definidos posteriormente.
Desconto por atraso: 1 ponto por dia.

Descrição do Trabalho

Vocês deverão implementar uma nova shell na linguagen C, denominada lsh (*Lasy Shell*). Ao contrário de uma shell convencional, que é um interpretador de programas responsável por lançar processos de *background* ou *forground* a cada comando, a lsh tem um pequeno "problema":

- Sendo uma shell "preguiçosa", a cada comando que o usuário digita, ela não o executa imediatamente; ela vai enfileirando os comandos em um buffer, sem executá-los. Esse buffer dever ser capaz de armazenar até 5 linhas de comando (acima desses limites, as linhas de comando serão simplesmente descartadas);
- O usuário, que vai perdendo a paciência, pode tentar "matar" a lsh por meio de um "Ctrl-c"; se isso acontecer, a lsh não deverá finalizar... mas para deixar o usuário mais calminho ela deverá executar a sequência de comandos digitados até então pelo usuário, esvaziando assim o buffer de comandos;

  Obs.1: Os comandos que são armazenados no buffer podem ser **comandos internos** (que não implicam na criação de novos processos, como o comando "cd") e **comandos externos** (que implicam na criação de novos processos para executar os executáveis correspondentes, como o comando "ls");
  Obs.2: Em se tratando de comandos externos, a lsh criará os processos dentro de um **mesmo grupo de background** criado para esse conjunto de processos. Com isso, não receberão nenhum Sinal gerado pelo usuário por meio de comandos "Ctrl-..." no Terminal. Além disso, como herança dessa shell, esses processos deverão **Ignorar** o sinal SIGINT.
- Mas se o usuário digitar "Ctrl-c" e não tiver nenhum comando enfileirado no buffer, mas a lsh tiver processos filhos em execução (*i.e.* vivos!), ela lsh deverá exibir a mensagem:

“Não posso morrer... sou lenta mas sou mãe de família!!!”

- Mas, se em até 3s o usuário digitar “Ctrl-c” novamente

  “Ok... você venceu! Adeus!”

  … finalizando sua execução em seguida

- Outra habilidade da nossa shell, é que ela usa o caracter especial ‘#’ para criar pipes entre processos (este conceito ainda será trabalhado em sala de aula). Assim, uma linha de comando pode receber até 2 comandos externos , unidos por um ‘#’ :

  “comando externo 1” # “comando externo 2”

… nesse caso, serão criados dois processos (em bg), e a shell deverá criar um pipe e redirecionar a saída padrão do “comando extrno 1” para esse pipe, assim como a entrada padrão de “comando externo 2” para este mesmo pipe.

Abaixo temos exemplos de sequências de execução da lsh:

```
lsh> ls -l  # grep cocada   //2 comandos externos
lsh> cd ..                  //comando interno
lsh> firefox                //comando externo
lsh>
                    //usuário digita Ctrl-c
                    //a lsh cria processos em background para executar os
                    //comandos “ls -l” e “grep cocada”, com uso de pipe, executa
                    //o comando “cd”, cria processo em background para executar o
                    // comando “firefox”(nessa ordem)
                    // ... ls, grep e firefox pertencem a um mesmo “grupo x” (bg)
...                //saída do comando “grep cocada”
...                //saída do comando “firefox”

lsh> xcalc        //comando externo
lsh>
                    //usuário digita Ctrl-c
                    //a lsh cria processo em background para executar o
                    //comando “xcalc”, que irá pertencer a um outro “grupo y” (bg)

lsh>
                    //usuário digita Ctrl-c
lsh> Não posso morrer... sou lenta mas sou mãe de família!!!
lsh>
                   //usuário digita Ctrl-c novamente e menos de 3s após o Ctrl-c anterior
lsh> “Ok... você venceu! Adeus!”
                     //FIM DA lsh!!
```

```
lsh> ls -l          //comando externo
lsh> cd ..          //comando interno
lsh> firefox        //comando externo
lsh> ps  aux        //comando externo
lsh>
                    //usuário digita Ctrl-c
                    //a lsh cria processo em background para executar o
                    //comando “ls -l”, executa o comando “cd”, cria processos
                    // em background para executar os comandos “firefox” e “ps”
                    //(nessa ordem) ls, firefox e ps pertencem a um mesmo “grupo z” (bg)

...                //saída do comando “ls -l”
...                //saída do comando “firefox”
...                //saída do comando “ps”

lsh>
                    //usuário digita Ctrl-c
lsh> Não posso morrer... sou lenta mas sou mãe de família!!!
lsh>                //o usuário fecha o firefox... agora lsh não tem mais filhos
lsh>
                    //usuário digita Ctrl-c 10s após o Ctrl-c anterior
lsh> “Ok... você venceu! Adeus!”
                     //FIM DA lsh!!
```

Linguagem da lsh

A linguagem compreendida pela lsh é bem simples. Cada sequência de caracteres diferentes de espaço é considerada um termo. Termos podem ser:

(i) comandos internos da shell,
(ii) nomes de programas que devem ser executados (e seus argumentos), eventualmente separados ou por um '#'

- *Comandos internos da shell* são as sequências de caracteres que devem sempre ser executadas pela própria shell e não resultam na criação de um novo processo. Na lsh as operações internas são: cd, wait e exit. Essas operações devem sempre terminar com um sinal de fim de linha (*return*) e devem ser entradas logo em seguida ao *prompt* (isto é, devem sempre ser entradas como linhas separadas de quaisquer outros comandos).

`cd`: Muda o diretório corrente da *shell*. Isso terá impacto sobre os arquivos visíveis sem um caminho completo (*path*).

`wait`: Faz com que a *shell* libere todos os processos filhos que estejam no estado "Zombie" antes de exibir um novo *prompt*. Cada processo que seja "encontrado" durante um wait deve ser informado por meio de uma mensagem na linha de comando, que deverá identificar o pid do processo que morreu , assim como a "causa da morte". Caso não haja mais processos no estado "Zombie", uma mensagem a respeito deve ser exibida e lsh deve continuar sua execução.

`exit`: Este comando permite terminar propriamente a operação da *shell*. Ele faz com que todos os seus herdeiros vivos (herdeiros diretos e indiretos) morram também ... e a lsh só deve morrer após todos eles terem sido "liberados" do estado "Zombie".

- *Programas a serem executados* são identificados pelo nome do seu arquivo executável e podem ser seguidos por um número máximo de 3 argumentos (parâmetros que serão passados ao programa por meio do vetor argv[]). Até nois comandos externos podem estar presentes na mesma linha de comando, separados por um '#'. Na ocorrência de um "Ctrl-c", os processos devem ser criados e executados em backgroud e a *shell* deve continuar sua execução. Isso significa retornar imediatamente ao *prompt*.

ALGUNS CONCEITOS IMPORTANTES

Processos em Background no Linux

No linux, um processo pode estar em *foreground* ou em *background*, ou seja, em primeiro plano ou em segundo plano. A opção de se colocar um processo em *background* permite que a shell execute tarefas em segundo plano sem ficar bloqueada, de forma que o usuário possa passar novos comandos para o ele.

Quando um processo é colocado em *background*, ele ainda permanece associado a um

terminal de controle. No entanto, em algumas implementações Unix, quando um processo tenta ler ou escrever no terminal, o kernel envia um sinal SIGTTIN (no caso de tentativa de leitura) or SIGTTOU (no caso de tentativa de saída). Como resultado, o processo é suspenso.

Por fim, um processo de *background* não recebe sinais gerados por combinações de teclas, como Ctrl-C (SIGINT), Ctrl-\ (SIGQUIT), Ctrl-Z (SIGTSTP). Esses sinais são enviados apenas a processos em foreground criados pela shell.

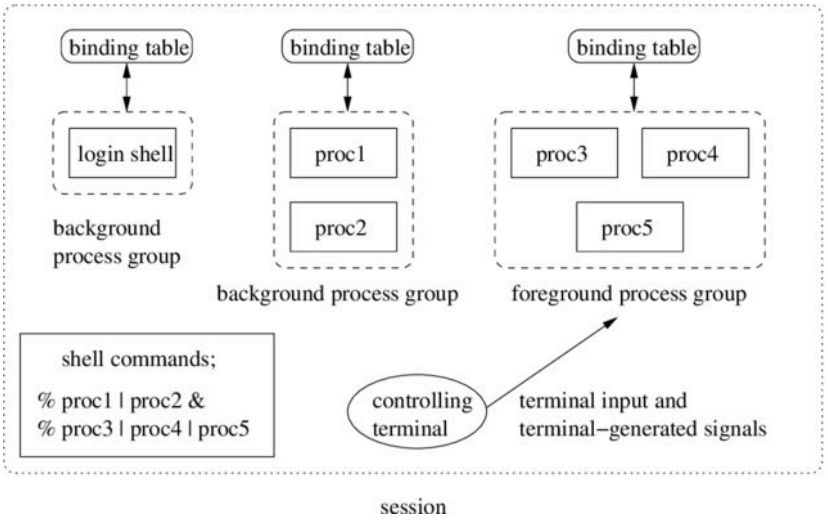

Fig 1: Relação entre processos, grupos, sessões e terminal de controle

Grupos e Sessões no Linux

Como vocês já viram em laboratórios passados, o Unix define o conceito de ***Process Group***, ou Grupo de Processos. Um grupo nada mais é do que um conjunto de processos. Isso facilita principalmente a vida dos administradores do sistema no envio de sinais para esses grupos. É que usando a chamada `kill()` é possível não somente enviar um sinal para um processo específico, mas também enviar um sinal para todos os processos de um mesmo *Process Group*. Como também já foi visto, quando um processo é criado, automaticamente ele pertence ao mesmo *Process Group* do processo pai (criador), sendo possível alterar o grupo de um processo por meio da chamada `setpgid()`. A bash, por exemplo, quando executa um comando de linha, ela faz `fork()` e logo em seguida é feita uma chamada a `setpgid()` para alocar um novo *Process Group* para esse processo filho. **Numa shell convencional (como a bash)** se o comando for executado sem o sinal '&', esse *process group* é setado para *foreground*, enquanto o grupo da bash vai para *background*. A figura acima ilustra como ficam os grupos após os comandos ilustrados no quadro "shell commands".

- Após a linha de comando "proc1 | proc2 &", a bash cria dois processos em *background* e um *pipe*, e redireciona a saída padrão de proc1 para o *pipe*, e a entrada padrão de proc2 para esse mesmo *pipe*.
- Após a linha de comando "proc1 | proc2 | proc3 ", a bash cria três processos em *foreground* e dois *pipes*, e redireciona a saída padrão de proc1 para o 1o. *pipe*, e a entrada padrão de proc2 para esse mesmo *pipe*; também redireciona a saída padrão de proc2 para o 2o *pipe*, e a entrada padrão de proc3 para esse 2o. *pipe*.

Mais informações sobre grupos de **foreground e background aqui:**

- https://www.gnu.org/software/libc/manual/html_node/Foreground-and-Background.html
- https://man7.org/linux/man-pages/man3/tcsetpgrp.3.html

Sessões no Linux!

Agora que vocês já estão feras em *Process Groups*, vamos ao conceito de ***Session***, ou Sessão. Uma sessão é uma coleção de grupos. Uma mesma sessão pode conter diferentes grupos de *background*, mas no máximo 1 (um) grupo de *foreground*. Com isso, uma sessão pode estar associada a um terminal de controle que por sua vez interage com os processos do grupo de *foreground* desta sessão. Quando um processo chama **`setsid()`**, é criada uma nova sessão (sem nenhum terminal de controle associado a ela… e portanto, **todos os processos em background**) e um novo grupo dentro dessa sessão. Esse processo se torna o líder da nova sessão e do novo grupo.

OUTRAS Dicas Técnicas

Outras funções que podem ser úteis são aquelas de manipulação de strings para tratar os comandos lidos da entrada. Há basicamente duas opções principais: pode-se usar `scanf("%s")`, que vai retornar cada sequência de caracteres delimitada por espaços, ou usar `fgets` para ler uma linha de cada vez para a memória e aí fazer o processamento de seu conteúdo, seja manipulando diretamente os caracteres do vetor resultante ou usando funções como `strtok`.

Ao consultar o manual, notem que as páginas de manual do sistema (acessíveis pelo comando `man`) são agrupadas em seções numeradas. A seção 1 corresponde a programas utilitários (comandos), a seção 2 corresponde às chamadas do sistema e a seção 3 às funções da biblioteca padrão. Em alguns casos, pode haver um comando com o mesmo nome da função que você procura e a página errada é exibida. Isso pode ser corrigido colocando-se o número da seção desejada antes da função, por exemplo, "`man 2 fork`". Na dúvida se uma função é da biblioteca ou do sistema, experimente tanto 2 quanto 3. O número da seção que está sendo usada aparece no topo da página do manual.

Verificação de erros

Muitos problemas podem ocorrer a cada chamada de uma função da biblioteca ou do sistema. Certifique-se de testar cada valor de retorno das funções e, em muitos casos, verificar também o valor do erro, caso ele ocorra. Isso é essencial, por exemplo, no uso da chamada `wait`. Além disso, certifique-se de verificar erros no formato dos comandos, no nome dos programas a serem executados, etc. Um tratamento mais detalhado desses elementos da linguagem é normalmente discutido na disciplina de compiladores ou linguagens de programação, mas a linguagem usada neste trabalho foi simplificada a fim de não exigir técnicas mais sofisticadas para seu processamento.

Bibliografia Extra: Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition* (Cap 1-3).