

# Sistemas Operacionais

## 1º Trabalho de Programação

Período: 2024/1

**Data de Entrega:** *ADIADA PARA 10/SET*

**Composição dos Grupos:** até 3 pessoas

### Material a Enviar

- Por email: enviar um email para **soufes@gmail.com** seguindo o seguinte formato:

**Subject do email:** “Trabalho 1”

**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.zip**” (ex: *joao-maria-jose.zip*). Este arquivo deverá conter todos os arquivos (incluindo o *makefile*) criados com o código muito bem comentado.

**Valendo ponto:** clareza, indentação e comentários no programa.

**Desconto por atraso:** 1 ponto por dia

### Motivação

Insatisfeitos com todos os tipos de shells que vocês já usaram, vocês decidiram criar a sua própria shell, chamada **fsh (first shell)**. Essa shell deve tratar: execução de programas (em background e em foreground), comandos internos e tratamento de sinais.

### Objetivos

Se familiarizar com chamadas básicas de sistemas, sinais, grupos de foreground/background.

### Descrição do Trabalho

Vocês devem implementar na linguagem C uma shell denominada **fsh (first shell)** para colocar em prática os princípios de manipulação de processos.

Ao iniciar, **fsh** exibe seu *prompt* “**fsh>**” (os símbolos no começo de cada linha indicando que a *shell* está à espera de comandos). Quando ela recebe uma linha de comando do usuário, é iniciado o processamento desse comando. Primeiramente, a linha deve ser interpretada em termos da linguagem de comandos definida a seguir e cada comando identificado deve ser executado. Essa operação possivelmente levará ao disparo de novos processos.

A **fsh**, assim como outras shells UNIX, permite que na mesma linha de comando o usuário possa solicitar a criação de um conjunto de processos:

```
fsh> comando1 # comando2 # comando3
```

(Exemplo 1)

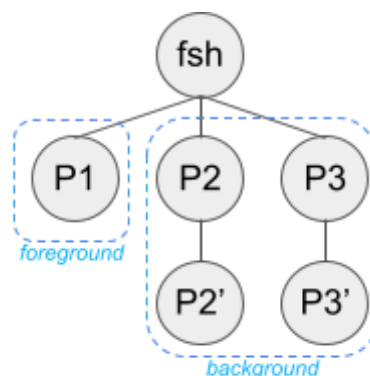
A *shell* poderá receber até 5 comandos na mesma linha. Neste exemplo, o programa deverá criar 3 processos – **P1**, **P2** e **P3** – para executar os comandos **comando1**, **comando2** e **comando3** respectivamente (**comandoX** corresponde a um arquivo executável do sistema, tratando-se de um “comando externo” que eventualmente pode receber parâmetros, como “**ls -l**”). No entanto, apenas

o primeiro processo (no exemplo, o processo P1, que vai executar “comando1”) será criado em *foreground*, enquanto os demais processos serão criados como processos de *background* (no exemplo, P2 e P3).

Com isso, quando a *fsh* recebe apenas um comando como abaixo, ela cria o processo em *foreground* normalmente:

```
fsh> ls -l
```

Além disso, essa shell tem um pequeno problema de consciência... para cada processo criado em *background*, é criado um **processo secundário** filho (também em *background*) rodando o mesmo comando. Com isso, no exemplo dado, os processos P2 e P3 terão cada um filho P2' e P3' respectivamente (chamados aqui de “processos secundários”. que também rodarão os “comando2” e “comando2” respectivamente). A *fsh* espera com isso compensar o fato de ter criado o processo em *background*. A árvore de processo no caso do exemplo 1, a árvore de processos ficaria assim:



Ainda não foi implementada a lógica de criar os filhos secundários. Então essa parte está parcialmente feita

Outra particularidade da *fsh* é que quando um processo morre ou é suspenso devido a um sinal, os demais processos **criados na mesma linha de comando** que o processo que morreu/suspendeu também devem morrer/suspender (incluindo os processos secundários  $P_x'$ ). **No exemplo 1, se P1 (ou P2 ou P3) terminarem/suspendirem porque receberam um sinal, P2, P2', P3 e P3' (ou seja, os demais processos) também devem ser finalizados/suspensos. Mas atenção... devem ser finalizados recebendo O MESMO SINAL.** Mas se um dos processos secundários  $P_x'$  morrer devido a um sinal, nada acontece com os demais processos do grupo. Por fim, se os processos morrem normalmente (por um *return* ou *exit*), também nada acontece com os demais processos.

## SOBRE O TRATAMENTO SINAIS...

Nossa *fsh* não quer saber de morte súbita enquanto ela tiver descendentes ainda vivos... (muito responsável!). Com isso, quando o usuário digitar Ctrl-C (SIGINT), caso ela ainda tenha descendentes vivos (ela NÃO vai considerar nessa conta os processos secundários, isto é, os  $P_x'$ ), ela deve imprimir uma mensagem perguntando ao usuário se ele tem certeza que ele deseja finalizar a shell. Caso o usuário confirme, a shell é finalizada. Mas se a shell não tiver nenhum descendente vivo, ela pode ir descansar em paz caso o usuário faça um Ctrl-C.

**IMPORTANTE:** durante a execução do tratador do sinal SIGINT, todos os demais sinais devem ser BLOQUEADOS. Dica: pesquise a chamada de sistema `sigaction()`...

TODO: testar pra ver se tá certo

TODO: testar pra ver se tá certo

Quanto aos descendentes da shell, TODOS devem IGNORAR o SIGINT... sejam eles processos de *foreground* ou *background*.

Por fim, caso o usuário digite Ctrl-Z (SIGTSTP), a shell em si não será suspensa, mas ela deverá suspender todos os seus descendentes (incluindo processos de *foreground* e *background*)...

TODO: testar pra ver se tá certo

## LINGUAGEM DA SHELL

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

- (i) operações internas da shell,
- (ii) operadores especiais,
- (iii) nomes de programas que devem ser executados,
- (iv) argumentos a serem passados para os comandos ou programas.

- *Operações internas 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 `fish` as operações internas são:

TODO: rever a função `waitall` e testar pra ver se funciona.

- `waitall`: faz com que a shell libere todos os seus descendentes filhos que estejam no estado “Zombie” antes de exibir um novo prompt. Aqui vocês podem desconsiderar os processos *secundários*... afinal, ela já fez uma boa ação criando eles!

**ATENÇÃO:** Em algumas implementações Unix, caso um processo receba um sinal enquanto ele esteja bloqueado em uma chamada `wait` ou `waitpid`, ele pode retornar da chamada sem que de fato um filho tenha morrido (ou suspenso). Nesse caso a chamada `wait/waitpid` retorna -1 e a variável global `errno` (definida em `errno.h`) é setada com o código "EINTR". Com isso, o programador deve tratar esse caso para evitar um comportamento errado do programa.

- `die`: deve terminar a operação da shell, mas antes de morrer, esta deve tomar providências para que todos os seus descendentes vivos morram também (TODOS!)...

TODO: ainda não foi implementada a lógica de quando tiver filhos secundários da função `die`

Essas operações internas, quando digitadas pelo usuário, 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).

TODO: importante, o jeito que eu fiz é diferente, tem que mudar

- *Operadores especiais*: Existe apenas um tipo de *operador*: o símbolo ‘#’ ao qual vocês já foram apresentados. Os demais operadores conhecidos de shell convencionais, como os símbolos ‘|’, ‘&’, etc., não serão tratados neste trabalho.

- *Programas a serem executados* são identificados pelo nome do seu arquivo executável e podem ser seguidos por um número máximo de dois argumentos (parâmetros que serão passados ao programa por meio do vetor `argv[]`). Cada comando do tipo (i) ou (iii) seguido ou não de argumentos deve terminar com um fim de linha. No caso (i) o comando é executado diretamente pela `fish`. No caso (iii), o processo (ou processos) devem ser criados conforme explicado anteriormente. **ATENÇÃO!** Cada vez que um processo `Px` criado em *foreground* retorna, a `fish` deve exibir imediatamente o prompt.

TODO: essas variações eu não cheguei a implementar. Precisa fazer a análise dessas regras e modificar a execução desses diferentes tipos de comandos no trabalho

## 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.

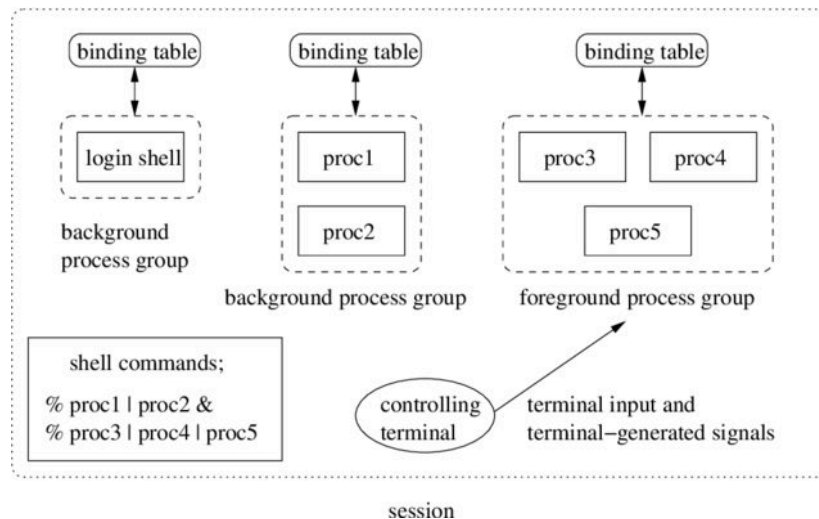


Fig 1: Relação entre processos, grupos, sessões e terminal de controle

## 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://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\*](#), 2<sup>nd</sup> Edition (Cap 1-3).