Insper

**Sistemas Hardware-Software, 2017-1**

**Shell Lab: Escreva seu próprio *shell*!**

**Início: 4/maio, Entrega: 5/junho, 11:59PM**

## Introdução

O propósito desta atividade é torná-los mais proficientes no uso de mecanismos de controle de processos e sinalização, através da construção de um mini-*shell*, o **`tsh`** (*tiny shell*)!

## Entrega

Você deverá submeter o arquivo **`tsh.c`** **via Blackboard** até a data de entrega (5/6).

Esta atividade será feita em duplas, sendo permitido apenas um grupo de 3 alunos em razão do número ímpar de alunos na sala.

## Instruções

Copie o arquivo **`shlab-handout.tar`** para sua máquina Linux e use o seguinte comando para descompactar este arquivo:

```
$ tar xvf shlab-handout.tar
```

Um diretório **`shlab-handout`** será criado, contendo vários arquivos. ***VOCÊ DEVERÁ MODIFICAR E ENTREGAR APENAS O ARQUIVO*** **`tsh.c`**

O arquivo **`tsh.c`** contém um esqueleto funcional de um *shell* Unix simples. Algumas funções auxiliares já estão implementadas. Sua tarefa é completar as

funções vazias listadas abaixo. (Para sua referência temos o número aproximado de linhas de código em cada caso, em uma implementação de referência)

- **`eval`**: Rotina principal que analisa e interpreta a linha de comando (70 linhas)
- **`builtin_cmd`**: Reconhece e interpreta os comandos do *shell*: **`quit`**, **`fg`**, **`bg`**, **`jobs`** (25 linhas)
- **`do_bgfg`**: Implementa os comandos **`bg`** e **`fg`** (50 linhas)
- **`waitfg`**: Espera o término de um *job* em *foreground* (20 linhas)
- **`sigchld_handler`**: Trata sinais **`SIGCHLD`** (80 linhas)
- **`sigint_handler`**: Trata sinais **`SIGINT`**, ou seja, **`ctrl-c`** (15 linhas)
- **`sigtstp_handler`**: Trata sinais **`SIGTSTP`**, ou seja, **`ctrl-z`** (15 linhas)

## Especificação do `tsh`

Seu *shell* **`tsh`** deve ter as seguintes características:

- O *prompt* deve ser a *string* **`"tsh> "`**.
- A linha de comando digitada pelo usuário deve consistir de um comando seguido opcionalmente por argumentos, todos separados por um ou mais espaços em branco. Se o comando for um dos comandos internos do *shell*, então o *shell* deve cumprir o comando e voltar a mostrar o *prompt*. Caso contrário, o comando se refere a um arquivo executável (com o *path* completo), que deve ser carregado e executado em um processo filho. (Usamos o termo *job* para nos referirmos a este processo filho)
- Se a linha de comando termina em '&' então tsh deve rodar o *job* em *background*, liberando o *prompt* para novos comandos. Caso contrário, deverá rodar o *job* em *foreground*.
- Se o usuário digitar **`ctrl-c`** então o *shell* deverá enviar o sinal **`SIGINT`** para o processo atualmente em *foreground* e para todos os seus descendentes (no caso de um programa que contenha bifurcações – **`fork()`**). Se não tiver nenhum processo em *foreground*, então **`ctrl-c`** não fará nada.
- Se o usuário digitar **`ctrl-z`** então o *shell* deverá enviar o sinal **`SIGTSTP`** para o processo atualmente em *foreground* e para todos os seus descendentes (no caso de um programa que contenha bifurcações – **`fork()`**). Se não tiver nenhum processo em *foreground*, então **`ctrl-z`** não fará nada.

- Cada *job* deverá ser identificado pelo seu *process ID* (PID), que como sabemos é atribuído pelo próprio sistema operacional, ou pelo seu *job ID* (JID), que será um inteiro positivo atribuído ao *job* pelo seu **`tsh`**.
  Quando usamos o JID numa linha de comando, devemos precede-lo por '%'. Por exemplo: '5' significa PID=5 e '%5' significa JID=5.
  (Todas as funções de manipulação da lista de jobs já estão implementadas para você!)
- Os comandos internos (built-in) a seguir devem ser implementados:
  - **`quit`**: termina execução do *shell*
  - **`jobs`**: lista os processos em *background*
  - **`bg <job>`**: reinicia em *background* o processo recém-interrompido por **`ctrl-z`**, através do envio do sinal **`SIGCONT`**. O argumento **`job`** pode ser um PID ou JID.
  - **`fg <job>`**: reinicia em *foreground* o processo recém-interrompido por **`ctrl-z`**, através do envio do sinal **`SIGCONT`**. O argumento **`job`** pode ser um PID ou JID.
- O *shell* deve recolher todos os processos filhos zumbis. Se algum job terminar porque recebeu algum sinal, então tsh deve reconhecer este evento e imprimir uma mensagem com o PID do job e qual sinal terminou o processo. Exemplo: "Job (9721) terminated by signal 2".

## Verificando seu trabalho

### Implementação de referência

O programa **`tshref`** é uma solução de referência para o seu *shell*. **Seu `tsh` deve se comportar de modo idêntico ao programa `tshref`: ambos devem gerar a mesma saída** (a menos dos PIDs, que obviamente mudam de execução para execução).

### *Driver* de teste

O programa **`sdriver.pl`** executa seu *shell* como um processo-filho, envia comandos e sinais a ele conforme indicado em um arquivo de registro (*trace file*), captura e mostra a saída do seu *shell*.

```
$ ./sdriver.pl -h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
```

```
Options:
-h Print this message
-v Be more verbose
-t <trace> Trace file
-s <shell> Shell program to test
-a <args> Shell arguments
-g Generate output for autograder
```

Dentre os arquivos providenciados a vocês estão 16 *trace files* (**trace01.txt** a **trace16.txt**) que devem ser usados em conjunto com o *driver* para testar a correção de seu *shell*. Por exemplo, para rodar o *driver* em seu **tsh** usando o arquivo **trace01.txt** você deve executar o seguinte comando:

```
$ ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

(O argumento **-a "-p"** informa o seu *shell* de que ele não deve imprimir o *prompt*)

## Dicas

- Você deverá revisar minuciosamente o capítulo 8 (*Exceptional Control Flow*) do seu livro!
- Use os *trace files* para guiar seu desenvolvimento: comece implementando funcionalidades requeridas para **trace01.txt**, depois **trace02.txt**, etc.
- As chamadas **waitpid**, **kill**, **fork**, **execve**, **setpgid** e **sigprocmask** serão muito úteis no seu desenvolvimento, bem como as opções **WUNTRACED** e **WNOHANG** de **waitpid**.
- Na hora de criar um processo-filho, você deve bloquear **SIGCHLD** antes do **fork()**, e desbloqueá-lo no processo-pai após adicionar o novo processo-filho à lista de jobs. Ademais, como o processo-filho herda uma cópia da máscara de bloqueio do processo-pai, lembre-se de desbloquear **SIGCHLD** no filho também, antes executar o novo programa com **execve()**. Leia com atenção a seção 8.5.6 do livro, que trata deste assunto!
- Ao criar um processo-filho, este terá o mesmo *group ID* do processo-pai. Quando enviamos **SIGINT** para o grupo do processo-filho, este grupo incluirá o processo-pai também! Ou seja, toda vez que tentarmos finalizar o grupo todo do processo-filho, acabaremos terminando nosso *shell* também! Para resolver este problema lembre-se de alterar o *group ID* do processo-filho com

Insper

a chamada **`setpgid(0, 0)`**, que vai colocar o processo-filho em um novo grupo cujo *group ID* é o PID do processo-filho.

## Desafios

- Implemente pipes!

```
/bin/ls | /usr/bin/less
```

- Implemente redirecionamento de arquivo!

```
/bin/ls > lista_arquivos.txt
```

- Implemente aliases e variáveis de ambiente!

## Avaliação

I – Implementação errônea ou ausente

D – Implementou apenas **`quit`** e **`jobs`**

C – Implementou processos em *foreground* e tratamento de **`ctrl-c`**

B – Implementou todas as funcionalidades, com erros pequenos

A – Implementou todas as funcionalidades, sem erros, com código legível e bem documentado

A+ : Nível A, e implementou algum dos desafios