TP 1

SHELL + PS(TOP) + SINAIS + MODULOS

Cesar Nascimento & Maximiliam Araujo
| DCC605 | Flavio Figueiredo
May 6, 2017

Especificação do Projeto

O trabalho proposto foi adaptado do material do Professor Ítalo Cunha. Conceitos como Pipe, Estrutura de Processos do Kernel e Sinais, dentro da disciplina de Sistemas Operacionais foram abordados.

O trabalho foi dividido em 4 partes. A seguir um descritivo de cada parte:

1. Desenvolvendo um Shell Básico: Baixar o esqueleto do shell e implementar o código para execução dos comandos como Redirecionamento de entrada/saída; Sequenciamento de Comandos
2. Lendo o /proc/ para fazer um OS-Tree: Imprimir a árvore de processos usando tab para separa-los.
3. Uma TOP Simples (topzera): Modificar o comando PS para imprimir os processos em sequência (remover os tabs). Além disto, alterar o mesmo para identificar o PID do programa, o usuário que está executando o mesmo e o estado do processo. Com isto, imprimir os programas em execução em uma tabela.
4. Sinais: Permitir que o comando TOP envie sinais, ou seja, criar uma função no TOP que envia sinais para um PID. Tal função pode ser apenas um "PID SINAL".

A seguir, para cada parte será apresentado uma breve descrição de como foi feita sua implementação.

Parte 1: Desenvolvendo um Shell Básico

Case ' ':

```
execvp(ecmd->argv[0], ecmd->argv);
```

Os comandos da família exec substituem a imagem do processo corrente com a imagem do novo processo. O primeiro argumento da função exec é o nome do arquivo que será executado. A utilização do execvp ocorreu por uma questão de facilidade no entendimento de seu funcionamento. Apenas passamos o arquivo a ser executado por parâmetro e a função se encarrega de criar o processo.

case '>':

case '<':

```
int file = open(rcmd->file, rcmd->mode, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

dup2(file, rcmd->fd);

close(file);
```

Esta primitiva (dup2) cria uma cópia de um descritor de arquivo existente (oldfd) e fornece um novo descritor (newfd) tendo exatamente as mesmas características que aquele passado como argumento na chamada. Dup2 determina que newfd será a cópia de oldfd, fechando antes newfd se ele já estiver aberto.

case '|':

```
int pipefd[2];
  if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
       }
  int pid = fork1();
       if(pid == -1){
        perror("fork");
        exit(EXIT_FAILURE);
       }
  if(pid == 0){
   close(pipefd[0]);
   close(1);
   dup(pipefd[1]);
   close(pipefd[1]);
   runcmd(pcmd->left);
  }
  else if(pid != 0){
     close(pipefd[1]);
     close(0);
     dup(pipefd[0]);
     close(pipefd[0]);
     wait(NULL);
```

```
    runcmd(pcmd->right);
  }
```

O comando acima descrito cria um pipe, em seguida, fork1() cria um processo filho. Após o fork1(), cada processo fecha os descritores que não são necessários para o pipe. O pai então a utiliza o dup(pipefd[1]) para se comunicar com o filho. Já o filho utiliza o dup(pipefd[0]) para se comunicar com pai.

Parte 2: Lendo o /proc/ para fazer um PS-Tree

Para resolver essa parte do trabalho prático, algumas funções foram criadas. Abaixo seguem breves explicações destas:

void retornarNomeProcesso(char str[], char *str2): função responsável por retornar o nome do processo e armazena-lo na str2. Como referência, passa-se uma string str, com o formato /proc/[pid]/task/[pid]/comm, onde o [pid] é o id do processo que se quer descobrir o nome.

int func(char str[], int espaco): função responsável por caminhar na árvore de processos a partir do pai init (pid = 1), passado como primeiro parâmetro. Para se descobrir os filhos de um processo, faz-se uso do arquivo /proc/[pid]/task/[pid]/children. Esse arquivo contém todo os filhos dos processos [pid] que se deseja buscar. Para cada processo, a func é novamente chamada recursivamente. Para controlar os tabs, a cada chamada da func é passado como argumento a quantidade de tabs de acordo com o nível hierárquico que o processo se encontra, através do parâmetro espaço.

Parte 3: Uma TOP Simples (topzera)

Para construção da terceira parte do trabalho, a segunda parte foi utilizada como base. A identação do programa foi removida e o número de [pid] encontrado para construir a árvore foi utilizado como base para encontrar as outras informações especificadas no trabalho.

A seguir serão detalhados o funcionamento de cada função considerada importante para esta parte do trabalho.

void arquivoNomeProcesso(char *str, char processo[]): constrói a string responsável em encontrar o arquivo certo para o nome de cada processo utilizando o [pid]. A string tem o formato "/proc/[pid]/task/[pid]/comm".

void arquivoNomeEstado(char *str, char processo[]): Esta função constrói a string "/proc/[pid]/status" que contém diversas informações de um processo. Em específico, nesse arquivo buscamos o estado do processo, que se for lido de forma linear, encontra-se na sexta posição do arquivo. Em seguida a função imprime na tela o estado do processo através do [pid] em questão.

void imprimiOwner(char *str, char processo[], char *owner): Para chegarmos ao nome do dono do processo, podemos obter o UID real do programa e traduzir esse número para o login, com ajuda

do arquivo password. Usando a função: struct passwd *getpwuid(uid_t uid); que retorna o endereço de uma struct password na qual contém o campo pw_name que aponta para o nome do usuário. A função também ficou responsável em imprimir esse nome na tabela do top.

void arquivoFilhos(char *str, char processo[]): função já havia sido utilizada para construir a árvore da parte dois do trabalho. Os pids filhos de cada processo foram retirados do arquivo children. Esta função contraí o caminho para esse arquivo. "/proc/[pid_pai]/task/[pid_pai]/children". O primeiro pid_pai é o 1.

A função "int func(char str[], int *numProc)" e "void retornarNomeProcesso(char str[], char *str2)" são omitidas nesta parte da documentação porque já foram abordadas na parte do dois do trabalho.

Parte 4: Sinais

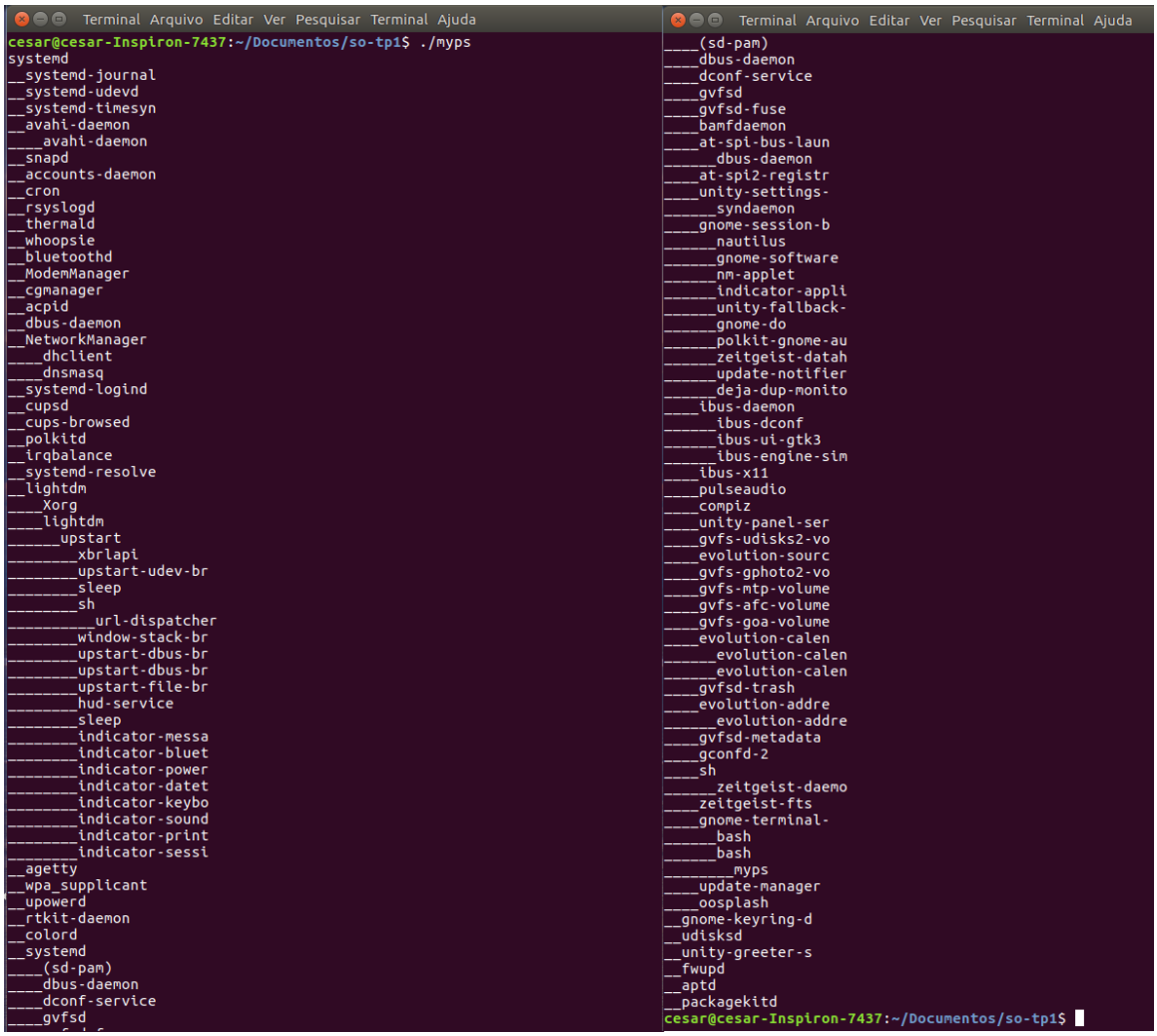
Para desenvolver esta parte do trabalho, utilizamos a biblioteca signal.h dentre as bibliotecas c. Com ela foi possível ler uma variável do tipo pid_t e utilizar a função kill(pid_t pid, int sinal) da biblioteca que faz o que foi requisitado na especificação. A função enviar um sinal int para um programa com pid determinado.

Decisões de implementação

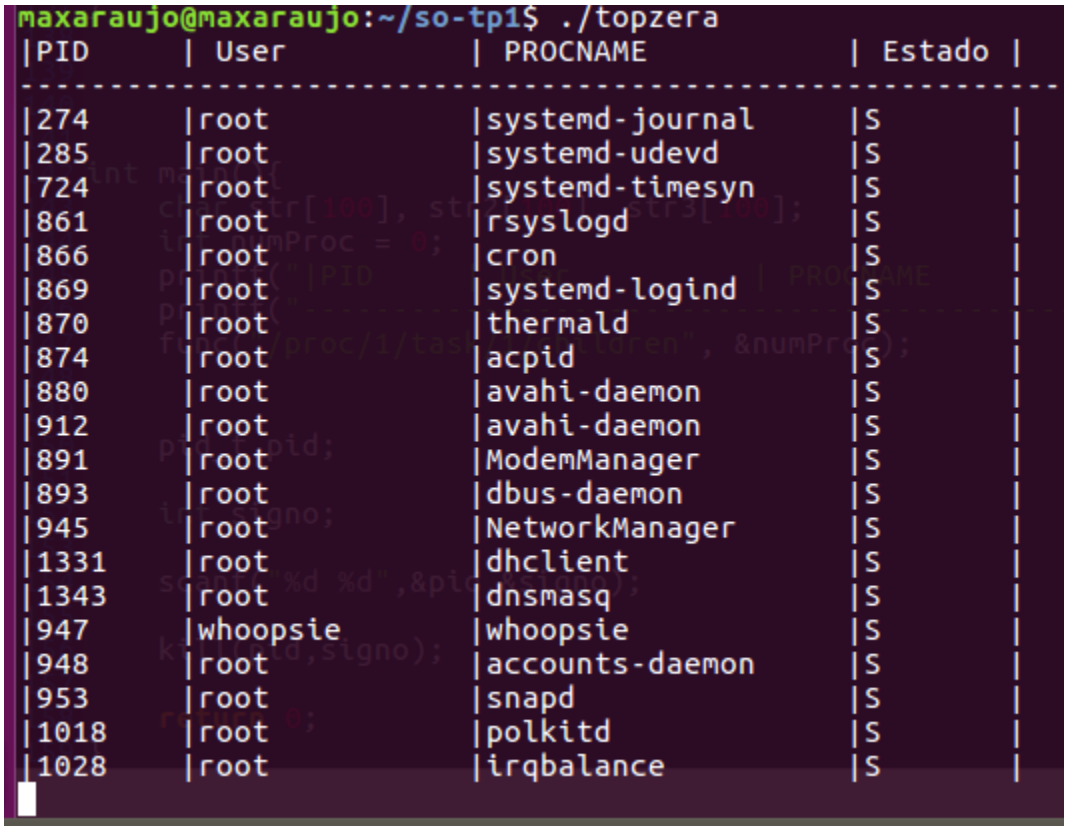
Preferimos não fazer a parte 5, que no caso era ponto extra. Além disso, optamos por não implementar as atualizações de 1 em 1 segundo para simplificar a quarta parte do trabalho.

Testes

Parte 2:



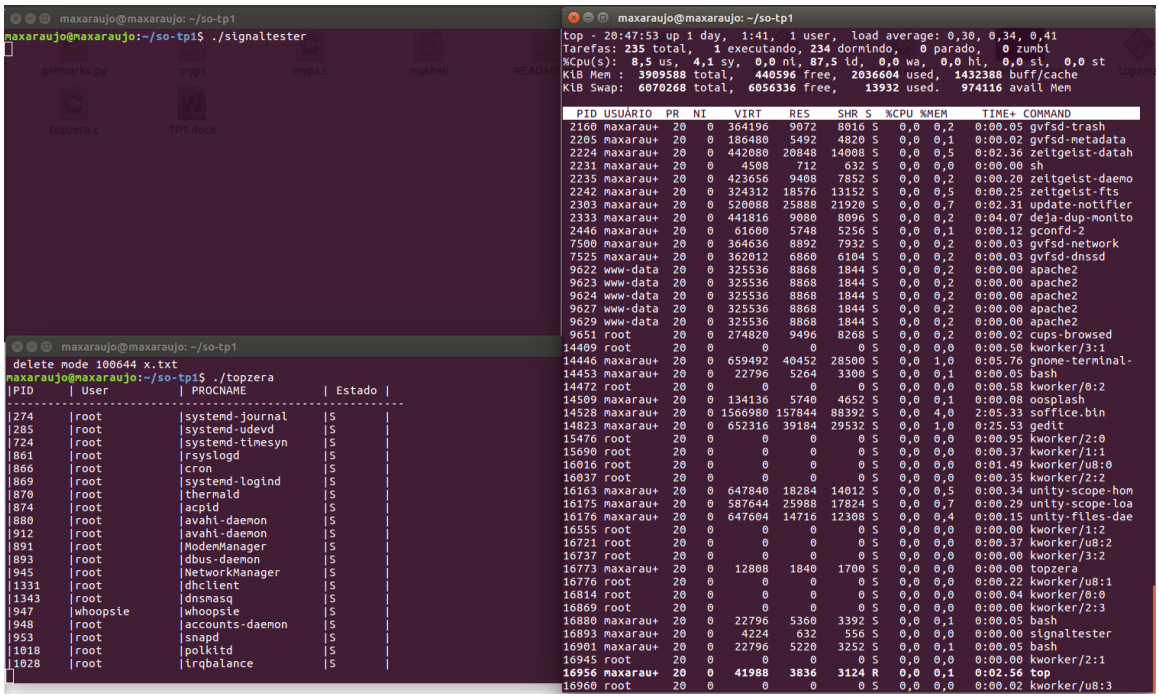
Parte 3: Impressão dos 20 primeiros processos em execução

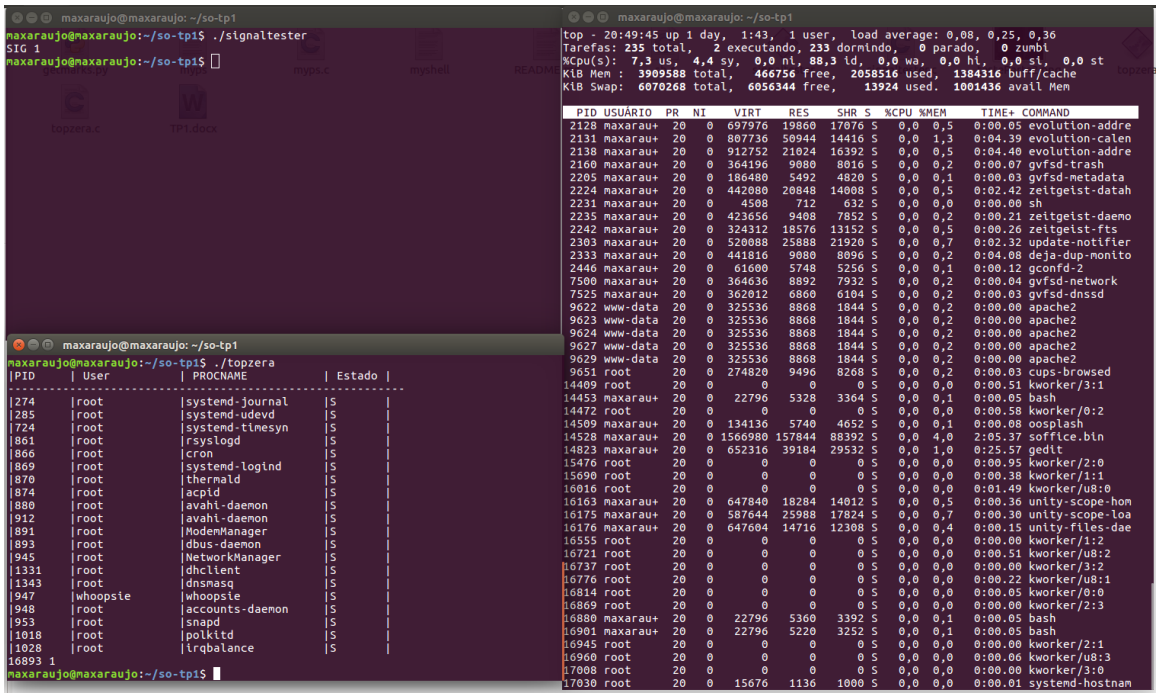


O campo no final da tabela é a espera pela leitura do pid e um sinal que é especificado na quarta parte do trabalho.

Parte 4:

Para imprimir este teste, utilizamos o TOP original para visualizar o PID do processo SIGNALTESTER.





O processo signaltester é o quarto processo de baixo para cima no terminal a direita. Enquanto isso o topzera aguarda a leitura de um pid e um sinal

Conclusão

Foi possível implementar todas as especificações mais importantes do trabalho, possibilitando assim o ganho de conhecimento sobre como funciona algumas funções do Linux.

Referências bibliográficas

https://linux.die.net/man/3/execvp

http://www.dca.ufrn.br/~adelardo/cursos/DCA409/node22.html

https://linux.die.net/man/2/pipe

http://man7.org/linux/man-pages/man5/proc.5.html