INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM225 – Sistemas Operacionais

Trabalho 01
Processos em Linux

Prof. Marco Furlan

Abril/2021

Instruções

- Esta **atividade** deverá ser **resolvida** em **equipe** – a mesma formada em sala de aula;
- **Responder** todas as **questões na ordem apresentada** em um arquivo Word ou LaTeX e depois **exportar** para **PDF**;
- **Identificar** claramente no **início do documento** o **nome** e **RA** dos integrantes da equipes;
- **Enviar o PDF** criado e **o arquivo de programa C** da **última questão** para o **link do trabalho** no **MyOpenLMS da disciplina**;
- Apenas um integrante da equipe precisa enviar;
- Esta atividade estará **disponível** a partir de **15/04/2020**;
- A solução deverá ser **enviada** até o dia **22/04/2020**.

1. (**até 2,0 pontos**) Utilizando o **comando** `ps`.

   (a) Abrir um **terminal no Linux** e então digitar, no seu diretório de usuário, o **comando** `ps`. **Explicar** o **significado** de cada **coluna** de dado que foi exibida.

   (b) Experimentar agora com o **comando** `ps --forest`. **Qual é a diferença** em relação ao item anterior?

   (c) O **comando** `ps -ef` permite **exibir todos os processos** que estão em execução na sua máquina. **Pergunta**: foi seu usuário logado que criou todos esses processos? Escreva o nome de um programa (coluna CMD) que não foi você (coluna UID) que criou. **Nota**: se a listagem for longa, passe a saída para o comando `less`, assim: `ps -ef | less`. Para navegar na saída, utilize as teclas de movimentação e tecle q para sair.

   (d) Como saber se um **programa específico** está em **execução**? Passe a **saída** de `ps` para o **filtro** `grep` (que pesquisa a ocorrência de uma palavra em um texto) assim: `ps -ef | grep nome_programa`. Escrever o comando para verificar se um programa (à sua escolha) está em execução.

   (e) Como saber **quais processos foram criados** pelo **usuário** `X`? **Executar** `ps -u X`. Desse modo, apresente quais são os **processos** do **seu usuário** e do **usuário** `root`.

2. (**até 0,5 pontos**) Utilizando o **comando** `top` (ou `htop`, se preferir). Executar `top` (tecle `q` para terminar) na linha de comando e **responda**, em relação a seu sistema Linux: qual é o **nome do processo** que está utilizando **mais CPU**? Justifique com a captura da tela do terminal do Linux.

3. (**até 2,0 pontos**) O **diretório** `/proc` **parece** ser um **diretório comum**, como `/usr` ou `/etc`, **mas não é**. O **diretório** `/proc` é um **pseudosistema** de **arquivos** mantido na **memória do computador**. O diretório `/proc` **contém um subdiretório** para **cada processo** em execução no sistema. **Programas** tais **como** `ps` e `top` **leem as informações** sobre processos em execução **desses diretórios**. O **diretório** `/proc` **também** contém **informações** sobre o **sistema operacional** e seu **hardware em** arquivos como `/proc/cpuinfo`, `/proc/meminfo` e `/proc/devices`. **Pede-se**:

   (a) **Listar**, com o máximo de detalhes, o diretório `/proc`.

   (b) Executar algum programa (exemplo: Firefox). Abrir uma janela de terminal e então descobrir o **ID** do processo escolhido usando o comando `ps -ef`. Depois, verificar se existe em `/proc` um **diretório** com tal número. Listar esse diretório e anotar os nomes dos arquivos apresentados. **Pesquisar e explicar** para que **serve** cada um dos **arquivos** listados.

4. (**até 2,5 pontos**) **Gerenciamento de processos**.

   (a) O **comando** `ping` implementa o protocolo de eco: permite enviar pacotes TCP/IP por uma interface de rede a alguma máquina e esta responde com os mesmos pacotes – útil para verificar se uma máquina está "no ar". **Executar** o comando `ping localhost > /dev/null` (ping local redirecionando a saída para não aparecer na tela);

   (b) Para **terminar** este comando, tecle `CTRL+C`;

   (c) Agora, **executar** este comando em **segundo plano**: `ping localhost > /dev/null &` (o símbolo `&` faz com que o **processo** seja **executado** em **segundo plano**). Na tela será apresentado o **PID** (identificador de processo) do **processo criado**.

   (d) **Como saber** quais são os **processos em segundo plano** criados por nesse terminal? Executar o comando `jobs`. Anotar o estado do processo.

(e) **Repetir** o passo (c) no mesmo terminal. Depois, repita o passo (c). Anotar quais são os estados apresentados.

(f) **Trazer** o job de número 1 (veja o número na saída de jobs) **para a frente** com o **comando** `fg %1`.

(g) **Interromper** este job com `CTRL+Z` – ele irá para o segundo plano. Executar `jobs` e anotar o que ocorreu.

(h) **Executar** o comando `bg %1` para **retornar** o job de número 1 para **segundo plano**. Reexecute `jobs` e anote o que ocorreu.

(i) **Terminar** o job de número 1 com o **comando** `kill %1` (o padrão de sinal de `kill` é `SIGTERM` – termina "graciosamente" um processo). Executar `jobs` e anotar os resultados.

(j) **Terminar** o job de número 2. Executar jobs e anotar os resultados.

5. (**até 3,0 pontos**) **Elaborar** um **programa** em **C** que crie **alarmes assíncronos**. O programa deverá executar como um laço infinito e permitir que o **usuário digite** um **tempo** (interpretado como segundos) e uma **mensagem** que será **exibida** após o término do tempo. Segue uma ilustração do funcionamento que se deseja deste programa, a ser **denominado de** `alarm-fork.c` (digita-se um número, seguido de espaço em branco e depois a mensagem):

```
$ ./alarm-fork
Alarme> 10 Acordar
Alarme> 20 Levantar
(10) Acordar
(20) Levantar
Alarme>
```

Notar que o programa de alarme continua executando, aguardando uma entrada a ser realizada pelo usuário. Quando o tempo de um alarme termina, uma mensagem é apresentada exibindo o tempo do alarme e a mensagem do alarme. Para terminar o programa terminar, teclar `CTRL+Z`, que gera um caractere de fim de arquivo e termina a entrada do usuário. A base deste programa está listada a seguir e os **comentários indicam o que deve ser programado para completar o programa**:

```c
/* Cabeçalhos necessários -> não precisa adicionar mais nada */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define TRUE 1

int main(int argc, char *argv[]) {
    int seconds;        /* para armazenar os segundos do alarme */
    char line[128];     /* para ler uma linha de entrada        */
    char message[64];   /* para armazenar a mensagem do usuário */
    pid_t pid;          /* para obter o resultado de fork()     */

    while (TRUE) {
        printf("Alarme> ");
        /* Se teclou CTRL + D -> fim de arquivo e termina */
```

```
            if (fgets(line, sizeof(line), stdin) == NULL)
                exit(EXIT_SUCCESS);
            /* Só ENTER -> repete a entrada */
            if (strlen(line) <= 1)
                continue;
            /* Senão, usar sccanf() para decompor a linha em número e mensagem
               o especificador 64[^\n] representa "aceite até 64 caracteres que
               não sejam '\n'". Retorna o número de argumentos analisados.*/
            if (sscanf(line, "%d %64[^\n]", &seconds, message) < 2)
                fprintf(stderr, "Comando inválido!\n");
            else {
                /* É APENAS ESTE ELSE QUE PRECISA SER TERMINADO */
                /* Execute fork() */
                /* SE o resultado de fork() for negativo, exibir uma mensagem
                   de erro e terminar com falha*/
                /* SE o resultado de fork() for ZERO, escrever o código do
                   PROCESSO-FILHO assim:
                        - Dormir a quantidade de segundos especificada
                        - Exibir os segundos passados e a mensagem associada
                        - Terminar normalmente
                   SENÃO escrever o código do PROCESSO-PAI asssim:
                        - Faça
                            - Aguarde o PID de um processo filho
                            - Se este PID tiver valor -1,
                                - Apresente uma mensagem indicando erro na espera
                                  de processo-filho
                                - Termine o processo com falha
                          Enquanto o PID obtido seja diferente de ZERO
                */
            }
        }
    }
```

Utilizar as seguintes funções:

- `fork()`: para criar um processo-filho por alarme;
- `sleep(n)`: para fazer o processo-filho dormir por `n` segundos;
- `exit(val)`: para terminar o processo pai ou filho com `val=0` (sucesso) ou `val=1` (falha);
- `waitpid()`: para aguardar o término de um processo-filho. Usar assim, dentro do laço indicado nos comentários: `pid = waitpid(-1, NULL, WNOHANG)`, que aguarda ou retorna automaticamente o PID de um processo-filho que terminou.

Para compilar este programa, executar:

```
gcc -Wall -g alarm-fork.c -o alarm-fork
```

E para executar:

```
./alarm-fork
```

Sugestão: executar o programa como está apresentado para entender a dinâmica de entrada de dados. Depois, complete as tarefas pedidas.