CES-33
Relatório 1 – Gerência de Processos
Aluno: Alyson Fernandes Basílio

Data: 05/04/2017

Seção 1

O objetivo deste laboratório é a familiarização com gerência de processos no Linux .

Introdução

A dinâmica de execução e gerenciamento de processos é parte extremamente importante do funcionamento do Sistema Operacional.

O Shell é uma interface do usuário com o SO que permite que o usuário dê comandos para o SO executar processos. Neste experimento, o aluno pode conhecer e entender melhor o funcionamento de um Shell.

O Shell, basicamente, criava processos, nos quais ele era o pai, e esperava esses processos encerrarem. A identificação do pai dos processos criados pelo Shell era feita através da recuperação do ppid de cada processo.

Dois conceitos aprendidos durante o experimento foram os conceitos de processo órfão e processo zumbi.

O nome Processo Órfão é dado para aqueles processos que não terminaram sua execução, porém seus respectivos Processos Pais terminaram suas execuções e morreram. Contudo, o Linux trata de fazer o chamado *reparenting* e passar o processo órfão para outro processo adotar, normalmente o processo *init*.

O nome Processo Zumbi é dado para aqueles processos que terminaram suas respectivas execuções, mas por algum motivo não foram mortos. Isso geralmente acontece por causa de um erro do processo pai que foi incapaz de processar o código de saída do processo filho. Para encerrar esse tipo de processo precisa-se matar o processo pai dele, para ele então virar um processo órfão e o init remover ele da tabela de processos.

Seção 2

Shell com History

A solução adotada para a implementação da função history pode ser dividida em três tópicos:

- No método *read_command* passou a fazer a identificação do comando digitado pelo usuário do Shell. Primeiro se havia o sinal de '!' no primeiro caractere. Neste caso, verifica-se primeiro se há histórico. Em segundo lugar verifica se a opção escolhida é a de repetir o último comando. Por último verifica qual comando da lista o usuário deseja acessar. Em cada uma dessas verificações, no caso de resultado positivo em uma delas, a variável *argv* é alterada e a execução volta para o método principal.
- No caso de ser um comando, o método *read_command* faz apenas o que ele fazia antes das mudanças: passar o comando e os seus parâmetros para a variável *argv* e voltar para a função principal.
- Na função principal, a única alteração foi feita no caso em que o comando ainda será executado. Foi criado um caso para o uso do *history* e um caso para um erro de comando inválido, por exemplo, acessar uma posição que não existe no history.

Código

```
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define TRUE 1

int n = 0; int pid;
char *argv[50];char temp[256];
int argc;
char hist[100][256];
int nhist = 0;

void type_prompt()
{
  printf("[shell-PID=%i]$ ", getpid());
}

void read_command(char *argv[])
{
  n = 0;
  gets(temp);
  if(temp[0]=='!'){
        if(nhist == 0)
                printf("Não há histórico a ser acessado.");
        else {
                if(temp[1]=='!'){
```

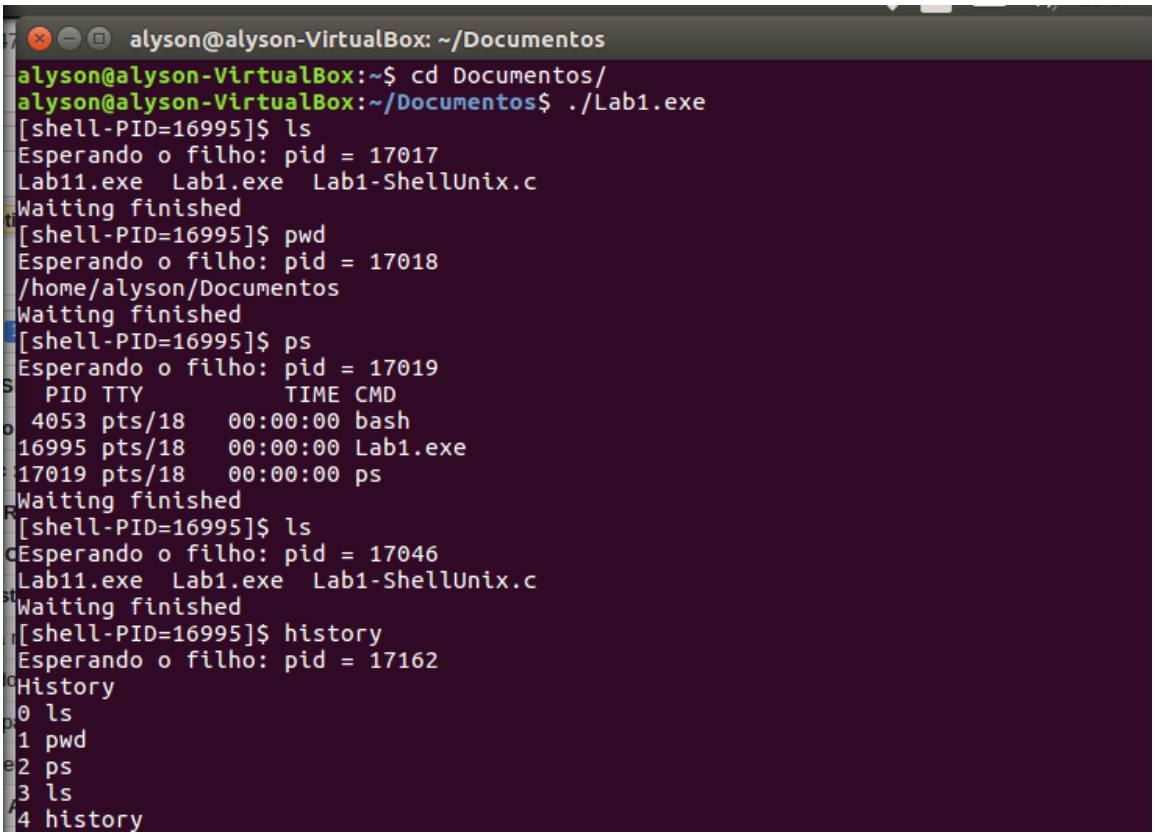
```
                        strcpy(hist[nhist],hist[nhist-1]);
                        nhist++;
                        argv[n++] = strtok (hist[nhist-1]," ");
                        while (argv[n-1] != NULL){
                                argv[n++] = strtok (NULL, " ");
                        }
                } else{
                        argv[n] = strtok (temp,"!");
                        int j = atoi(argv[n]);
                        if(j<nhist){
                                argv[n++] = strtok(hist[j]," ");
                                strcpy(hist[nhist++],hist[j]);
                                while (argv[n-1] != NULL){
                                        argv[n++] = strtok (NULL, "!");
                                }
                        }
                        else
                                strcpy(argv[n++],"Erro");
                }
        }
  }
  else{
          strcpy(hist[nhist++],temp);
          argv[n++] = strtok (temp," ");
          while (argv[n-1] != NULL){
                argv[n++] = strtok (NULL, " ");
          }
  }
}

int main()
{
  int status;
  argv[0] = NULL;
  while (TRUE) /* repeat forever */ {
    type_prompt(); /* display prompt on screen */
    read_command(argv); /* read input from terminal */
    if ((pid = fork()) != 0) /* fork off child process */ {
      printf("Esperando o filho: pid = %i\n", pid);
      waitpid(-1, &status, 0); /* wait for child to exit */
      printf("Waiting finished\n");
    }
    else {
      if(strcmp(argv[0],"history")==0){
                printf("History\n");
                for(int i =0; i<nhist; i++){
                        printf("%d %s \n",i, hist[i]);
```

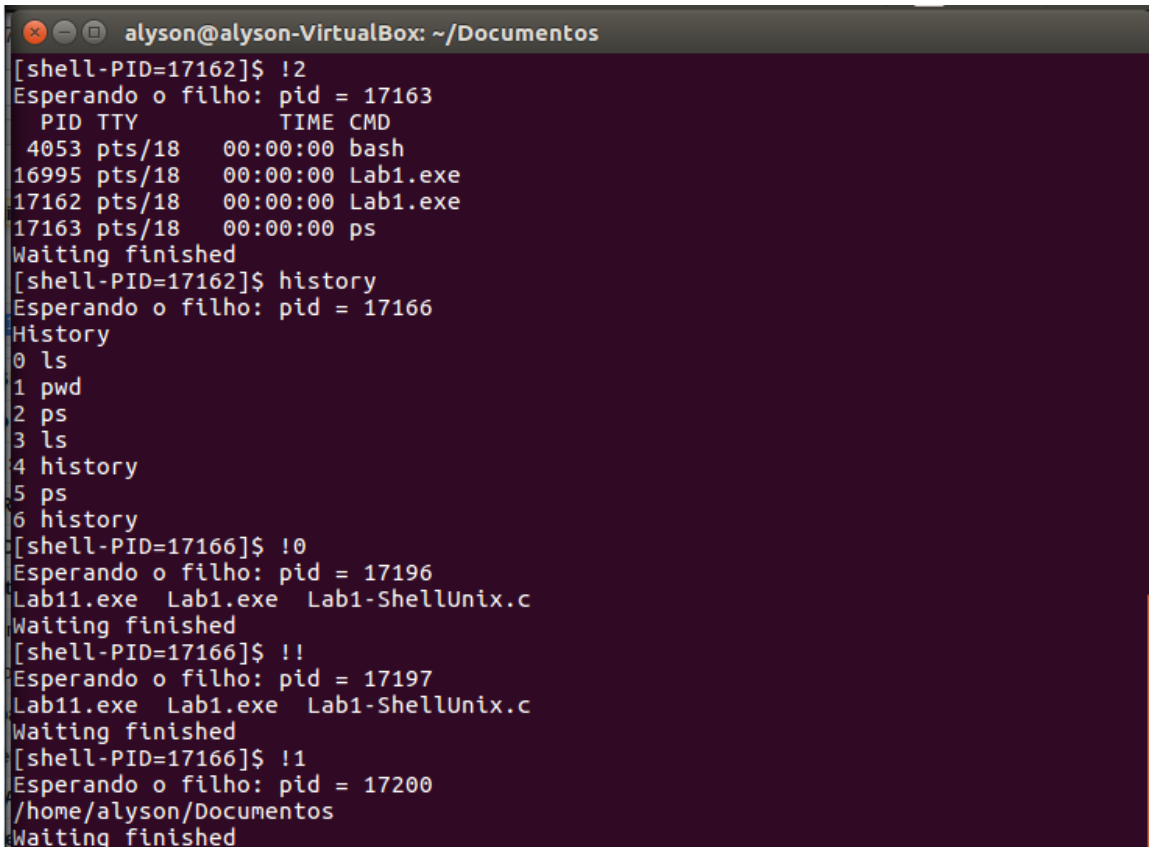
```
                }
        }
        else if (strcmp(argv[0], "Erro") == 0)
                printf("Não executou. Erro: Comando inválido\n");
        else if (execvp(argv[0], argv) == -1) /* execute command */
       printf("Não executou. Erro: %s\n", strerror(errno));
  }
}
return 0;
                                                }
```

Seção 3

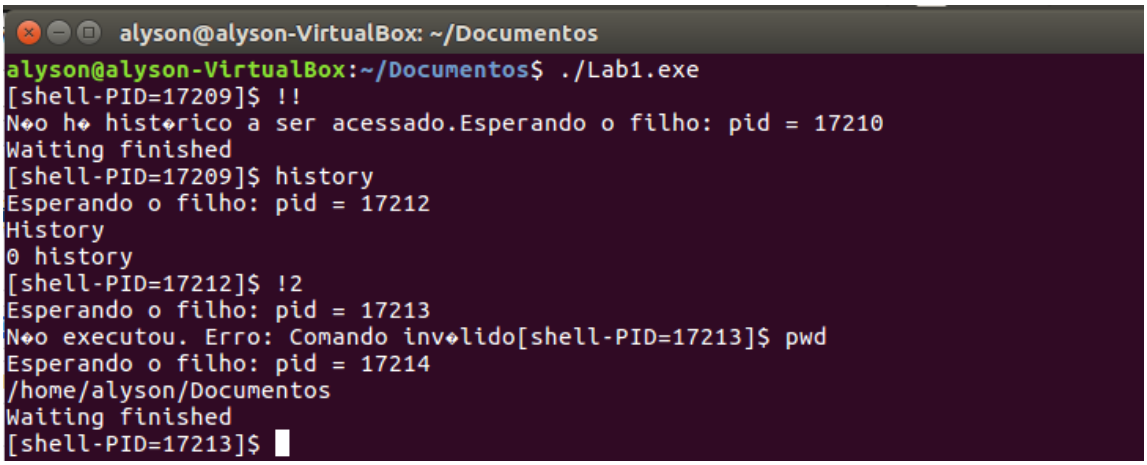
Testes

Foram executados testes básicos de funcionamento das funções implementadas e testes de casos de erro.





Testando erros:



Seção 4

Conclusão

Inicialmente, houve dificuldade de entender como funcionava métodos básicos de instruções com processos, como *waitpid* e execvp. Após o estudo e o bom entendimento do que estava acontecendo na execução do código, não houveram problemas na hora de implementar a solução. Algumas dificuldades foram encontradas no uso da linguagem C, pois há algum tempo não a utilizava.

Finalmente, observou-se que a solução cumpre os requisitos do problema.