RELATÓRIO EQUIPE 4

CONTEÚDO

1 RESUMO

Projeto apresentado pela equipe 4 formada por Alberto Santos, Pedro Santana, Renata Ribeiro e Rodrigo Cerqueira.

O projeto, escrito na linguagem C, consiste basicamente na implementação de um programa que simula o shell do sistema operacional contando com os seguintes comandos para gerenciamento de programas pelo usuário: startProcess, waitProcess, killProcess, stopProcess, continueProcess e runProcess.

O projeto também conta com um arquivo para tratamento de erros buscando validar a correta execução do gerenciamento dos programas pelo usuário. O arquivo de tratamento de erros contém funções para quantidade de argumentos inválida, permissão negada, memória insuficiente, executável inexistente e PID que não corresponde à processo ativo.

O projeto também conta com um script básico para validação contendo, respectivamente:

start ls - que inicia e executa o processo de listar os componentes do diretório em questão

wait - que mostra ao usuário se o processo foi finalizado de forma normal ou anormal e se não há processos restantes

start date - que inicia e executa o processo de mostrar a data e horário em questão

e, finalmente, **wait** novamente.

A seguir serão destrinchados aspectos gerais das funções de gerenciamento e de tratamento de erros.

1.1 FUNÇÕES DE GERENCIAMENTO

Função startProcess: inicia outro programa com argumentos de linha de comando, imprime o ID do processo do programa que está rodando, e aceita outra entrada de linha de comando:

Código-fonte:

```
15  void startProcess(char *argv[], int argc) {
16          char * args[argc];
17          pid_t processId;
18
19
20          for (int i = 0; i < argc; i++) {
21                  args[i] = argv[i];
22          }
23          args[argc] = NULL;
24
25          if ((processId = fork()) == 0) {
26                  if (execvp(args[1], &args[1]) == -1) {
27                          switch (errno) {
28                                  case EACCES:
29                                          errorPermission();
30                                          break;
31                                  case ENOMEM:
32                                          errorMemory();
33                                          break;
34                                  case ENOENT:
35                                          errorNotSuchExecutable();
36                                          break;
37                          }
38                  }
39          } else {
40                          printf("myshell: processo %d iniciado.\n", processId);
41          }
42  }
43
```

Função waitProcess: aguarda até que o processo finalize. Quando isso acontece, indica se o término foi normal ou anormal, incluindo o *exit code* e o PID. Se não existem processos que o *shell* deva aguardar, imprime uma mensagem e volte a aceitar novos comandos de entrada.

Código-fonte:

```c
void waitProcess() {
        int corpse;
        int status;
        int childrenCount = 0;

        while ((corpse = waitpid(0, &status, 0)) > 0) {
                childrenCount++;

                if (status == 0) {
                        printf("myshell: processo %d finalizou normalmente com status %d.\n", corpse, status);
                } else {
                        char * signalDescription;
                        switch (status) {
                                case 1:
                                        signalDescription = "SIGHUP";
                                        break;
                                case 2:
                                        signalDescription = "SIGINT";
                                        break;
                                case 3:
                                        signalDescription = "SIGQUIT";
                                        break;
                                case 4:
                                        signalDescription = "SIGILL";
                                        break;
                                case 5:
                                        signalDescription = "SIGTRAP";
                                        break;
                                case 6:
                                        signalDescription = "SIGABRT";
                                        break;
                                case 7:
                                        signalDescription = "SIGEMT";
                                        break;
                                case 8:
                                        signalDescription = "SIGFPE";
                                        break;
                                case 9:
                                        signalDescription = "SIGKILL";
                                        break;
                        }

                        printf("myshell: processo %d finalizou de forma anormal com sinal %d: %s.\n", corpse, status, signalDescription);
                }
        }

        if (childrenCount == 0) {
                printf("myshell: não há processos restantes.\n");
        }
}
```

Função killProcess: usa o PID do processo como argumento e envia um SIGKILL para o processo indicado. Retorna se o processo foi finalizado e o PID do mesmo.

Função stopProcess: usa o PID do processo como argumento e envia um SIGTERM para o processo indicado. Retorna se o processo parou e o PID do mesmo.

Código-fonte:

```c
void stopProcess(pid_t pid){
        int stoppingProcess = kill(pid, SIGSTOP);
        if (stoppingProcess == -1) {
                        switch (errno) {
                                case EPERM:
                                        errorPermission();
                                        break;
                                case ESRCH:
                                        errorPidNotExist();
                                        break;
                        }
                } else if (stoppingProcess == 0) {
                        printf("myshell: processo %d parou a execução.\n", pid);
                }
}
```

como argumento e envia um SIGTERM para o processo indicado. Retorna se o processo foi finalizado e o PID do mesmo.

Código-fonte:

```c
void killProcess(pid_t pid) {
                int killResult = kill(pid, SIGTERM);

                if (killResult == -1) {
                        switch (errno) {
                                case EPERM:
                                        errorPermission();
                                        break;
                                case ESRCH:
                                        errorPidNotExist();
                                        break;
                        }
                } else if (killResult == 0) {
                        printf("myshell: processo %d foi finalizado.\n", pid);
                }
}
```

Função continueProcess: usa o PID do processo como argumento e envia um SIGTERM para o processo indicado. Retorna se o processo parou e o PID do mesmo.

Código-fonte:

```c
void continueProcess(pid_t pid){
        int continueProcess = kill(pid, SIGCONT);
        if (continueProcess == -1) {
                        switch (errno) {
                                case EPERM:
                                        errorPermission();
                                        break;
                                case ESRCH:
                                        errorPidNotExist();
                                        break;
                        }
                } else if (continueProcess == 0) {
                        printf("myshell: processo %d voltou a execução.\n", pid);
                }
}
```

Função runProcess: inicia um programa com argumentos de linha de comando, espera que tal processo finalize e imprime o *exit code*.

Código-fonte:

```c
void runProcess(char *argv[], int argc){
    char * args[argc];
    pid_t processId;


    for (int i = 0; i < argc; i++) {
        args[i] = argv[i];
    }
    args[argc] = NULL;

    if ((processId = fork()) == 0) {
        if (execvp(args[1], &args[1]) == -1) {
            switch (errno) {
                case EACCES:
                    errorPermission();
                    break;
                case ENOMEM:
                    errorMemory();
                    break;
                case ENOENT:
                    errorNotSuchExecutable();
                    break;
            }
        }
    } else {
        int corpse;
        int status;
        int childrenCount = 0;

        while ((corpse = waitpid(processId, &status, 0)) > 0) {
            childrenCount++;

            if (status == 0) {
                printf("myshell: processo %d finalizou normalmente com status %d.\n", corpse, status);
            } else {
                char * signalDescription;
                switch (status) {
                    case 1:
                        signalDescription = "SIGHUP";
                        break;
                    case 2:
                        signalDescription = "SIGINT";
                        break;
                    case 3:
                        signalDescription = "SIGQUIT";
                        break;
                    case 4:
                        signalDescription = "SIGILL";
                        break;
                    case 5:
                        signalDescription = "SIGTRAP";
                        break;
                    case 6:
                        signalDescription = "SIGABRT";
                        break;
                    case 7:
                        signalDescription = "SIGEMT";
                        break;
                    case 8:
                        signalDescription = "SIGFPE";
                        break;
                    case 9:
                        signalDescription = "SIGKILL";
                        break;
                }

                printf("myshell: processo %d finalizou de formal anormal com sinal %d: %s.\n", corpse, status, signalDescription);
            }
        }

        if (childrenCount == 0) {
            printf("myshell: não há processos restantes.\n");
        }
    }
```

Sobre o Shell: o programa aceita entradas de linha de comando de até 4096 caracteres e manipula até 100 palavras em cada linha de comando, como evidenciado a seguir:

```c
#define _POSIX_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include "systemfunctions.h"
#include "errors.h"

int main() {
        char str[4096];
        char * palavras[100];
        int npalavras = 0;

        char *token;

        printf("myshell> ");

        while (fgets(str, 256, stdin) != NULL) {
                // Codigo abaixo relativo a separacao de palavra por palavra
                token = strtok(str, " \t\n");
                while (token != NULL) {
                        palavras[npalavras] = token;
                        npalavras++;
                        token = strtok(NULL, " \t\n");
                }
                palavras[npalavras] = 0;

                if (strcmp(palavras[0], "start") == 0) {
                        startProcess(palavras, npalavras);
                        sleep(1);
                } else if (strcmp(palavras[0], "wait") == 0) {
                        waitProcess();
                } else if(strcmp(palavras[0], "kill") == 0) {
                        killProcess(atoi(palavras[1]));
                        sleep(1);
                } else if(strcmp(palavras[0], "stop") == 0) {
                        stopProcess(atoi(palavras[1]));
                        sleep(1);
                } else if(strcmp(palavras[0], "continue") == 0) {
                        continueProcess(atoi(palavras[1]));
                        sleep(1);
                } else if (strcmp(palavras[0], "exit") == 0 || strcmp(palavras[0], "quit") == 0) {
                        exit(0);
                } else if (strcmp(palavras[0], "run") == 0) {
                        runProcess(palavras, npalavras);
                        sleep(1);
                } else {
                        printf("myshell: Comando desconhecido: %s\n", palavras[0]);
                }

                npalavras = 0;
                printf("myshell> ");
        }

        return EXIT_SUCCESS;
}
```

1.2 FUNÇÕES DE TRATAMENTO DE ERROS

Função errorInvalidArguments: que indica que o número de argumentos é inválido.

Função erro Permission: que indica que a permissão foi negada.

Função errorMemory: que denota memória insuficiente.

Função errorNotSuchExecutable: que indica que o executável não existe.

Função errorPidNotExist: que indica que o PID não corresponde a um processo ativo.

2 INSTALAÇÃO E EXECUÇÃO

O projeto foi alocado em um repositório no GitHub. Para ser utilizado o projeto deve ser clonado e, no terminal, estando no diretório correspondente ao projeto o usuário deve escrever na linha de comando:

- make ; para compilar e
- ./myshell para executar

Daí o shell pode ser acessado com os seguintes comandos:

- start {comando a ser executado}
- run {comando a ser executado}
- stop {inserir PID}
- kill {inserir PID}
- continue {inserir PID}
- wait

3 DIFICULDADES E APRENDIZADOS

Dentre os principais aprendizados estão o aprendizado sobre o funcionamento básico de um shell, sobre aplicações da linguagem C, sobre o funcionamento de um processo e como ele ocorre em um shell.