Relatório Lab 1

Lucas Toscano Pimentel Appolinário Cerqueira - 2110695

Objetivo

O objetivo principal é trabalhar com o paralelismo por meio da criação de processos filhos.

Estrutura do programa e Solução

Breve descrição de cada módulo e as funções implementadas;

- Ex1

```c
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <sys/wait.h>


int main(){
    int n = 3;
    pid_t pid;

    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Erro ao criar um processo filho\n");
        return 1;
    }
    else if (pid == 0) {
        // Filho
        for (int i = 0; i < 10000; i++) {
            n += 10;
        }
        printf("processo filho, pid=%d, n=%d\n", getpid(), n);
    }
    else {
        // Pai
        for (int i = 0; i < 10000; i++) {
            n += 1;
        }
        printf("processo pai, pid=%d, n=%d\n", getpid(), n);

        int status = 0;
        waitpid(-1, &status, 0);
    }


    return 0;
}
```

Após o fork(), são criados dois processos com os mesmos valores de n e pid na memória virtual de cada um deles. Os if-else's seguintes verificam o valor do pid e com isso separam o trecho de código que deve ser executado em cada processo. No processo pai, o waitpid() é utilizado para que o processo pai espere o processo fil

```
[lucas@fedora lab1]$ ./ex1
processo pai, pid=6063, n=10003
processo filho, pid=6064, n=100003
```

Os valores de pid são diferentes porque o pai e o filho são processos diferentes.
Os valores de n são diferentes porque, no momento do fork, é criada uma memória virtual que armazena o valor de n = 3. Com isso, os processos pai e filho podem acessar o valor original de n sem que as operações de um afetem o outro.

- Ex2.1

```c
#include  <stdio.h>
#include  <stdlib.h>

int main(int argc, char *argv[]){

    for(int i = 1; i < argc; i++){
        printf("%s ", argv[i]);
    }

    printf("\n");

    return 0;
}
```

Os parâmetros do meuecho são recebidos como argumentos da main e parseados por um for loop que imprime cada um deles, pulando o primeiro elemento do argv[] pois ele é o nome do comando.

```
[lucas@fedora lab1]$ ./meuecho abc 123
abc 123
```

O output é igual ao da função echo padrão.

- Ex2.2

```c
#include  <stdio.h>
#include  <stdlib.h>

int main(int argc, char *argv[]){
    FILE *fp;
    char line[256];

    if ((fp=fopen (argv[1],"r"))==NULL){
        printf ("Erro na abertura do arquivo.");
    }
    else{
        while (fgets(line, sizeof(line), fp) != NULL) {
            printf("%s", line);
        }
    }
    fclose(fp);

    printf("\n");
    return 0;
}
```

O path do arquivo é passado como argumento da main e é criado um vetor de 256 chars para armazenar cada linha do arquivo. O if-else tenta abrir o arquivo e em caso de sucesso ele percorre cada linha, a armazenando na variável line e, em seguida, imprimindo-a.

```
[lucas@fedora lab1]$ ./meucat meuecho.c
//Lucas Toscano Pimentel Appolinário Cerqueira
//2110695


#include  <stdio.h>
#include  <stdlib.h>

int main(int argc, char *argv[]){

    for(int i = 1; i < argc; i++){
        printf("%s ", argv[i]);
    }

    printf("\n");

    return 0;
}
```

O output condiz com o do comando cat padrão

- Ex3

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

void type_prompt();
void read_command(char **cmd, char *params[]);
void isOriginalCmd(char **cmd);
void fix_path(char **cmd);

int main() {
    char *command;
    char *parameters[256];

    while (TRUE) {
        type_prompt();
        read_command(&command, parameters);
        if (fork() != 0) {
            int status = 0;
            waitpid(-1, &status, 0);
        } else {
            int err = execve(command, parameters, 0);
            if (err == -1) {
                perror("execve");
                exit(1);
            }
        }
        free(command);
    }

    return 0;
}
```

O while loop é usado para que a minhashell seja executada continuamente. A cada ciclo é impresso o caracter identificador da linha de comando e lido o prompt do usuário. Neste momento, cria-se um processo filho para que o comando possa ser executado sem que o processo que executa a minhashell seja interrompido.

```c
void type_prompt() {
    printf("$ ");
}
```

Esta função imprime o character indentificador da linha de comando.

```c
void read_command(char **cmd, char *params[]) {
    char input[256];

    if (fgets(input, sizeof(input), stdin) != NULL) {
        char *token = strtok(input, " \t\n");
        int i = 0;

        if (token != NULL) {
            *cmd = strdup(token);

            isOriginalCmd(cmd);

            params[i] = strdup(token);
            while ((token = strtok(NULL, " \t\n")) != NULL) {
                params[++i] = strdup(token);
            }

            params[++i] = NULL;
        }
        else {
            printf("Command not found.\n");
            *cmd = NULL;
        }
    } else {
        printf("Error reading input.\n");
        *cmd = NULL;
    }
}
```

Esta função recebe os parâmetros passados na linha de comando e separa cada um deles em um vetor (token). O primeiro elementos do token é tido como o comando e o resto são os parâmetros referentes a esse comando. Verifica-se se o comando é o padrão ou o criado para definir o path que a minhashell deve passar para ser executado.

```c
void isOriginalCmd(char **cmd) {
    char *original_cmds[] = {"echo", "cat", "ls", "more", NULL};

    for (int i = 0; original_cmds[i] != NULL; i++) {
        if (strcmp(*cmd, original_cmds[i]) == 0) {
            fix_path(cmd);
            return;
        }
    }
    return;
}
```

Esta função existe para verificar se o comando é o padrão (echo e cat) ou o que foi criado (meuecho e meucat). Caso seja o padrão, o path do comando é redirecionado para o diretório onde estão os comandos padrões.

```c
void fix_path(char **cmd) {
    const char *path_base = "/bin/";

    size_t path_length = strlen(path_base);
    size_t cmd_length = strlen(*cmd);

    char *combined = (char *)malloc(path_length + cmd_length + 1);

    if (combined == NULL) {
        perror("Memory allocation failed");
        return;
    }

    strcpy(combined, path_base);

    strcat(combined, *cmd);

    free(*cmd);
    *cmd = combined;
}
```

Esta função foi criada para direcionar o path para o diretório comandos padrões, presentes no “/bin”.

Observações e conclusões

Não houveram dificuldades que não foram superadas e os testes realizados foram descritos acima.