

Laboratório 1

Hanna Epelboim Assunção - 2310289

Érica Oliveira Regnier - 2211893

1 - Elaborar programa para criar 2 processos hierárquicos (pai e filho) onde é declarado um vetor de 10 posições inicializado com 0. Após o fork() o processo pai faz um loop de somando 1 às posições do vetor, exibe o vetor e espera o filho terminar. O processo filho subtrai 1 de todas as posições do vetor, exibe o vetor e termina. Explique os resultados obtidos (por quê os valores de pai e filho são diferentes? Os valores estão consistentes com o esperado?)

Bibliotecas: stdio.h, unistd.h, sys/wait.h, stdlib.h Função que retorna o pid do processo: int getpid()

Código fonte:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void){
    int vetor[10] = {0};
    int pid;
    pid = fork();
    if(pid<0){
        perror("Erro ao criar o processo");
    }
    else if(pid>0){ //pai
        for(int i = 0; i < 10; i++){
            vetor[i] += 1;
        }
        printf("[");
        for(int i = 0; i < 10; i++){
            printf("%d ",vetor[i]);
        }
        printf("]\n");

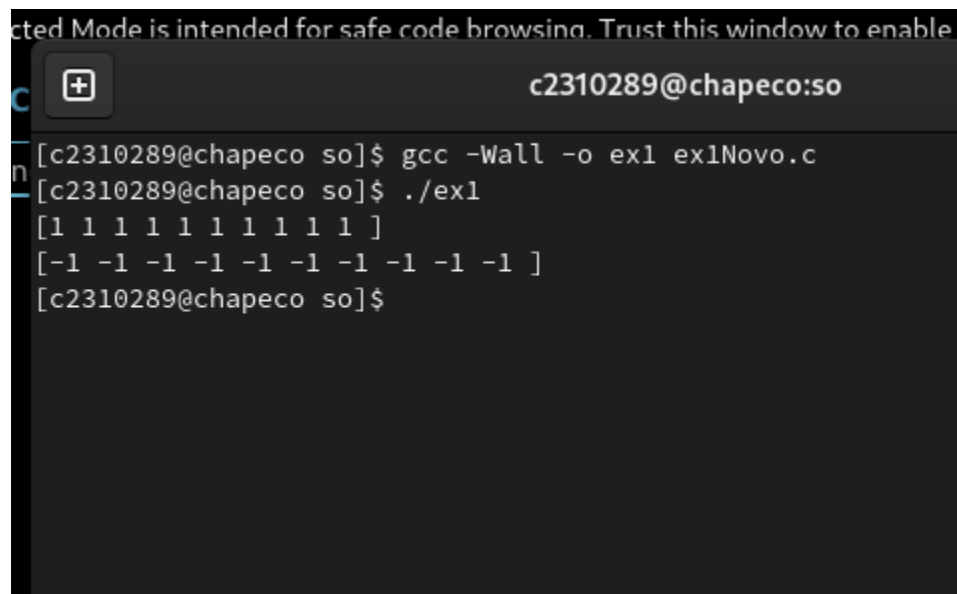
        waitpid(-1,NULL,0);
    }
}
```

```

        exit(0);
    }
    else{ //filho
        for(int i = 0; i < 10; i++){
            vetor[i] -= 1;
        }
        printf("[");
        for(int i = 0; i < 10; i++){
            printf("%d ",vetor[i]);
        }
        printf("]\n");
        exit(0);
    }
}

```

Saída:



```

c2310289@chapeco:so
[c2310289@chapeco so]$ gcc -Wall -o ex1 ex1Novo.c
[c2310289@chapeco so]$ ./ex1
[1 1 1 1 1 1 1 1 1 1 ]
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ]
[c2310289@chapeco so]$

```

Explicação:

O código inicializa um vetor de 10 posições com 0. Isso acontece uma vez, antes da bifurcação do processo. Devido ao `fork()` o processo é dividido em dois, e ambos os processos (pai e filho) continuam na linha do `fork()`. O processo filho (`pid == 0`) altera o vetor, subtraindo uma unidade de cada elemento do vetor e imprime esse valor. Essa alteração de valor afeta apenas o processo filho, pois após o `fork()`, o processo filho tem sua própria cópia da variável (vetor). O processo pai (`pid > 0`) altera o vetor, somando uma unidade em cada elemento do vetor e imprime esse valor e depois espera o filho terminar (execução do `waitpid()`). Como o processo

pai e o processo filho têm suas próprias cópias da variável após o fork(), a alteração feita pelo filho não afeta o valor de variável no pai. Portanto, era esperado que o pai imprimisse um vetor só com 1 e o filho com -1.

Observação: Se pid != 0, isso significa que o processo atual é o pai. O pai chama waitpid(-1, NULL, 0), que faz com que o processo pai espere até que o processo filho termine. Já se pid == 0 se trata de um processo filho.

2 - Programar funcionalidades dos utilitários do unix - “echo”

Ex: \$meuecho bom dia /* exhibe os parâmetros de meuecho */ bom dia

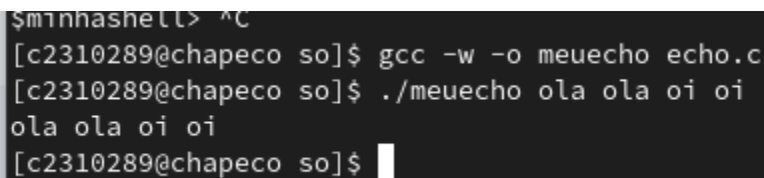
Código fonte:

Meuecho.c

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

```
int main(int argc, char *argv[]) {  
    for (int i = 1; i < argc; i++) {  
        printf("%s ", argv[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

Saída:



```
$minhashell> ^C  
[c2310289@chapeco so]$ gcc -w -o meuecho echo.c  
[c2310289@chapeco so]$ ./meuecho ola ola oi oi  
ola ola oi oi  
[c2310289@chapeco so]$
```

Explicação:

O programa repete a função do utilitário do unix “echo”, que repete a mensagem digitada no terminal. O programa faz um loop para imprimir todos os argumentos passados, com exceção do primeiro que é o próprio comando.

3 - Programar funcionalidades do utilitário do unix “cat”

Ex: \$meucat echo.c cat.c /* exibe os arquivos echo.c e cat.c */

Código fonte:

Meucat.c

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

```
int main(int argc, char *argv[]) {
    FILE *file;
    char ch;

    for (int i = 1; i < argc; i++) {
        file = fopen(argv[i], "r");
        if (file == NULL) {
            perror("Erro ao abrir o arquivo");
            return 1;
        }

        while ((ch = fgetc(file)) != EOF) {
            putchar(ch);
        }
        fclose(file);
    }
    return 0;
}
```

Saída:

```

[c2310289@chapeco so]$ gcc -w -o meucat cat.c
[c2310289@chapeco so]$ ./meu
meucat*  meuecho*
[c2310289@chapeco so]$ ./meucat echo.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}
[c2310289@chapeco so]$ █

```

Explicação:

O programa repete a função do utilitário do unix “cat”, que mostra o conteúdo de um arquivo. O programa faz um loop para pegar todos os argumentos (que seriam os nomes dos arquivos), exceto argv[0], que é o nome do comando, entrando em outro loop para ler os arquivos.

4 - Programar uma shell e executar os seus programas meuecho, meucat e os utilitários do Unix echo, cat, ls

Ex: \$minhashell meuecho alo alo Realengo aquele abraço /* executa meuecho */
 alo alo Realengo aquele abraço

Ex> \$minhashell meucat echo.c cat.c /* executa meucat */

Para executar os utilitários do unix é necessário indicar os diretórios onde eles estão.

Código fonte:

Minhashell.c

```

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

```

```
#include <sys/wait.h>

void executar_comando(char **args) {

    char comando_path[1024];
    snprintf(comando_path, sizeof(comando_path), "%s", args[0]);
    if (execvp(comando_path, args) == -1) {
        perror("Erro ao executar o comando");
    }
    exit(EXIT_FAILURE);
}

int main(void) {
    char comando[1024];
    char *args[64];
    char *token;
    char **arg_ptr;

    while (1) {
        printf("$minhashell> ");
        fgets(comando, sizeof(comando), stdin);

        comando[strlen(comando) - 1] = '\0';
        arg_ptr = args;
        token = strtok(comando, " ");

        while (token != NULL) {
            *arg_ptr = token;
            arg_ptr++;
            token = strtok(NULL, " ");
        }
        *arg_ptr = NULL;

        pid_t pid = fork();
        if (pid < 0) {
            perror("Erro ao criar o processo");
        }
    }
}
```

```

    } else if(pid > 0) { // Processo pai
        wait(NULL); // Espera o processo filho terminar
    } else if(pid == 0){
        if (strcmp(args[0], "meuecho") == 0) {
            args[0] = "./meuecho";
            executar_comando(args);
        } else if (strcmp(args[0], "meucat") == 0) {
            args[0] = "./meucat";
            executar_comando(args);
        } else if(strcmp(args[0], "cat") == 0){
            args[0] = "/usr/bin/cat";
            executar_comando(args);
        }else if(strcmp(args[0], "echo") == 0){
            args[0] = "/usr/bin/echo";
            executar_comando(args);
        }else if(strcmp(args[0], "ls") == 0){
            args[0] = "/usr/bin/ls";
            executar_comando(args);
        }
    }
}

return 0;
}

```

Comandos:

```

[c2310289@chapeco so]$ gcc -w -o minhashell minhashell.c
[c2310289@chapeco so]$ ./minhashell

```

Saída:


```
[c2310289@chapeco so]$ ./minhashell
$minhashell> cat echo.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}

$minhashell> cat cat.c echo.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *file;
    char ch;

    for (int i = 1; i < argc; i++) {
        file = fopen(argv[i], "r");
        if (file == NULL) {
            perror("Erro ao abrir o arquivo");
            return 1;
        }

        while ((ch = fgetc(file)) != EOF) {
            putchar(ch);
        }
        fclose(file);
    }
    return 0;
}

#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}

$minhashell> █
```

```

}
$minhashell> meucat echo.c cat.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return 0;
}
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *file;
    char ch;

    for (int i = 1; i < argc; i++) {
        file = fopen(argv[i], "r");
        if (file == NULL) {
            perror("Erro ao abrir o arquivo");
            return 1;
        }

        while ((ch = fgetc(file)) != EOF) {
            putchar(ch);
        }
        fclose(file);
    }
    return 0;
}
$minhashell>

```

```

}
$minhashell> meuecho oi oi Hanna e Érica
oi oi Hanna e Érica
$minhashell> echo Érica e Hanna
Érica e Hanna
$minhashell> ls
alo alo.c cat.c echo.c ex1 ex1Novo.c lle1 lle2 lle3 l2 lablex1.c lablex2 lablex2.c lablex3.c lablex4 lablex4.c meucat meuecho minhashell minhashell.c
$minhashell>

```

Explicação:

A função executar_comando é responsável por executar o comando especificado. Ela constrói o caminho do comando e usa execvp para substituir o processo atual pelo novo comando. Já na main, o programa entra em um loop infinito onde espera por comandos do usuário. A shell exibe um prompt \$minhashell> para indicar que está pronta para

receber comandos. O comando digitado pelo usuário é lido com `fgets` e armazenado na variável “comando” e o caractere de nova linha no final da string é removido. O comando é dividido em partes usando “`strtok`”, separando cada palavra pelos espaços em branco. Esses pedaços são armazenados no array `args`. O array `args` será usado para passar os argumentos para o comando a ser executado. Um novo processo é criado com `fork()`. Dependendo do comando digitado, o programa define o caminho do comando apropriado e chama a função `executar_comando` para executar o comando. Após a execução do comando e término do processo filho, o loop recomeça, esperando por mais comandos do usuário.

Dificuldade:

As questões 1, 2 e 3 foram feitas com facilidade, visto que havíamos feito exercícios parecidos na outra turma (antes do DE PARA). A questão 4 gerou um pouco mais de dificuldade, pois tivemos que lembrar como se chamam os utilitários (com parte do `path`) e os nossos programas (com `./`). Todos os programas saíram conforme o esperado.