



INF1316 - Laboratório 01 - `fork()` e `exec()`

Nome: Pedro de Almeida Barizon

Matrícula: 2211350

Professor: Luiz Fernando Seibel

Turma: 3WA

Data: 01/09/2024

Objetivo: implementar e compreender processos em sistemas Unix, com foco nas funções `fork()` e `exec()`.

Observações preliminares

- Para mais detalhes, todos os códigos fonte comentados encontram-se em anexo. Por isso, para evitar redundâncias, não serão aqui exibidos.
- Todos os arquivos fonte foram compilados com auxílio do *GNU C Compiler* (GCC) fazendo `gcc -Wall -o programa fonte1.c fonte2.c...` e executados com `./programa arg1 arg2...`. Caso haja particularidades, estas serão abordadas ao longo do relatório.

Exercício 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?)

1.1 Estrutura

Para que fosse enxuto, criou-se apenas um módulo, o `ex1.c`, que continha as funções:

- `main`: executa o programa;
- `exibe_vetor`: exibe o vetor de inteiros passado como argumento;

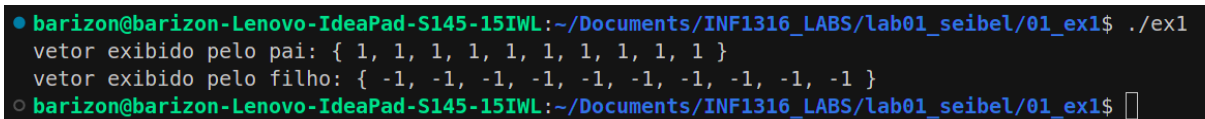
- `exibe_vetor_ln`: *wrapper* da função anterior que adiciona um `'\n'` ao fim da exibição.

1.2 Solução

Quando se executa o programa, o processo atual dá origem a um filho graças ao `fork()`. Como `fork` retorna 0 na instância do filho e o PID do filho na instância do pai (valor positivo), é possível controlar o bloco de código a ser executado por cada um via condicionais. Para tanto, foi necessário armazenar o valor de retorno na variável inteira `pid` e compará-la com zero. Cabe mencionar que, se houver falha na criação, `fork` retorna um valor negativo, permitindo-se o tratamento a erros.

No bloco executado pelo pai, incrementa-se cada entrada do vetor, que passa de 0 a 1. Em seguida, exibe-se o vetor resultante, e, por fim, espera-se pelo filho com `waitpid`. No bloco do filho, decrementa-se uma cópia do vetor original, cujas entradas passam de 0 a -1. A seguir, exibe-se o resultado, e finaliza-se com `exit(EXIT_SUCCESS)`. O processo pai, então, encerra o programa com `return 0`.

A saída do programa encontra-se abaixo.



```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel/01_ex1$ ./ex1
vetor exibido pelo pai: { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
vetor exibido pelo filho: { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 }
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel/01_ex1$
```

Figura 1.1 - Saída de `ex.1`

1.3 Observações e conclusões

Como cada processo possui área de endereçamento própria, o vetor manipulado pelo processo pai não foi o mesmo manipulado pelo filho. Por isso, em vez de o filho exibir um vetor repleto de zeros (já que teria desfeito o incremento do pai), decrementou diretamente uma cópia do vetor original, este sim repleto de zeros. O resultado, vimos, foi um vetor com entradas -1. Logo, o resultado saiu como esperado.

Exercício 2) Programar funcionalidades dos utilitários do unix - “echo”.

Exercício 3) Programar funcionalidades do utilitário do unix “cat”.

Exercício 4) Programar uma shell e executar os seus programas `meuecho`, `meucat` e os utilitários do Unix `echo`, `cat`, `ls`.

Foram agrupados todos os últimos três exercícios, porque compartilham de algo muito pertinente: para que pudessem ser executados como comandos nativos da *shell*, cada executável precisou ser colocado em um diretório — o qual foi chamado arbitrariamente `mybin` —, que passou a ser referenciado pela variável de ambiente da *shell* `PATH`. Dessa forma, os comandos puderam ser escritos sem “./” e a partir de qualquer pasta.

2.1 Estrutura

Para cada exercício, foi criado apenas um módulo: respectivamente, `meuecho.c`, `meucat.c` e `minhashell.c`, cada qual composto apenas por uma rotina principal, isto é, a `main`.

Em `meuecho.c`, a `main` exibe na *shell* os argumentos passados via teclado à exceção do próprio nome do comando (“`meuecho`”) separados entre si por um caractere de espaço ‘ ’ e encerrados com ‘\n’.

Em `meucat.c`, a `main`, segundo a codificação da tabela ASCII, exibe na *shell* o conteúdo dos arquivos passados como argumentos. Caso nenhum arquivo seja enviado, a rotina é simplesmente encerrada.

Em `minhashell.c`, a `main` cria um processo filho que executará o comando passado via teclado, como em `minhashell comando arg1 arg2`.

2.2 Solução

2.2.1 `meuecho.c`

Inicialmente, os argumentos transmitidos via teclado são passados como parâmetros da `main` sob a forma do vetor de *strings* `argv`, cujo último elemento é sempre `NULL`. Isso é usado como artifício pelo sistema operacional para contar a quantidade de elementos em `argv`: atribuindo 0 a uma certa variável inteira —

digamos, `argc` —, basta percorrer `argv` até encontrar `NULL`, incrementando `argc` a cada novo elemento. Desse modo, `argc` guardará a quantidade de elementos em `argv` antes de `NULL`. Dada a utilidade de saber o tamanho de `argv`, `argc` também é parâmetro da `main`.

Assim, considerando que o nome do comando ("meuecho") sempre ocupa a posição 0 de `argv`, a `main` percorre `argv` a partir do índice 1 até `argc - 1`, exibindo cada elemento na *shell*. Ao final, acrescenta um '`\n`'. Caso `argc` seja 1, isto é, caso nenhum argumento tenha sido passado para `meuecho`, ignora-se o elemento de índice `argc - 1` ($= 1 - 1 = 0$), já que este, por ocupar a posição 0, corresponderá ao nome do comando, que não deve ser impresso.

2.2.2 `meucat.c`

Considerando os parâmetros da `main` `argc` e `argv` como no item anterior, primeiramente se percorre `argv` a partir do índice 1 até o índice `argc - 1` e, com `fopen`, tenta-se abrir em modo leitura o arquivo de texto com caminho `argv[i]`. Caso haja falha na abertura, uma mensagem de erro é exibida na tela com `perror`, e o programa é encerrado com `exit(EXIT_FAILURE)`. Se foi aberto com sucesso, é lido e exibido um caractere por vez com auxílio de, respectivamente, `fgetc` e `fputc` até o final do arquivo (EOF), quando se fecha o arquivo atual com `fclose`, passando ao próximo.

2.2.3 `minhashell.c`

Quando se executa o programa, o processo atual dá origem a um filho graças ao `fork()`. O controle do bloco de código a ser executado ora pelo pai ora pelo filho é feito como no exercício 1. Enquanto o pai apenas espera seu filho terminar com `waitpid`, o filho usa `execvp` para executar o comando `argv[1]` procurado a partir da variável de ambiente `PATH` e cujo vetor de argumentos tem endereço inicial `argv + 1`. Terminada a execução do filho, o pai encerra o programa.

2.2.4 Saídas

```
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel$ meuecho echo echo echo
echo echo echo
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel$
```

Figura 2.1 - Teste do comando meuecho

```
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel/01_ex3$ meucat meucat.c
/* Pedro de Almeida Barizon 2211350 */

/* DEPENDENCIAS EXTERNAS */
#include <stdio.h>
#include <stdlib.h>

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

    for (int i = 1; i < argc; i++)
    {
        if (!(file = fopen(argv[i], "r"))) // Se falha de arquivo, encerra
        {
            fprintf(stderr, "meucat: %s: ", argv[i]); // Necessario usar printf por causa da string de formato, ausente em perror
            perror("");
            exit(EXIT_FAILURE);
        }
        while ((ch = fgetc(file)) != EOF)
            putc(ch, stdout); // Exibe um caractere por vez para evitar overload de buffer

        fclose(file);
    }

    return 0;
}
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel/01_ex3$
```

Figura 2.2 - Teste do comando meucat

```
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel$ minhashell meuecho meu echo eh tambem seu?
meu echo eh tambem seu?
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel$ minhashell /bin/echo nao, soh echo
nao, soh echo
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab01_seibel$
```

Figura 2.3 - Teste do comando minhashell com meuecho e echo

```
● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/lab02$ minhashell /bin/ls
main.c meutestecat.txt raiz.c raiz.h
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/lab02$
```

Figura 2.4 - Teste do comando minhashell com ls

```

● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/Lab02$ minhashell meucat raiz.c raiz.h
// Pedro de Almeida Barizon 2211350

#include "raiz.h"
#include <math.h>

#define EPSILON 5.0E-9
#define ABS_THRESH 1.0E-12

int bissecao(double a, double b, double (*f) (double x), double* r)
{
    double fa, fb, c, fc, err = (b - a) / 2.0;
    int iter = 0;

    fa = f(a);
    fb = f(b);

    if (fa * fb > 0.0)
        return -1;

    while (err > EPSILON)
    {
        iter++;
        c = (a + b) / 2.0;
        fc = f(c);

        if (fabs(fc) < ABS_THRESH)
            break;

        if (fa * fc < 0.0)
            b = c;
        else
        {
            a = c;
            fa = fc;
        }

        err = (b - a) / 2.0;
    }

    *r = (a + b) / 2.0;
    return iter;
}
// Pedro de Almeida Barizon 2211350

int bissecao(double a, double b, double (*f) (double x), double* r);
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/Lab02$ █

```

Figura 2.6 - Teste do comando minhashell com meucat

```

● barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/Lab01$ minhashell /bin/cat taylor.c taylor.h
#include "taylor.h"
#include <stdio.h>
#include <stdlib.h>

double avalia_taylor(int n, double* c, double x0, double x)
{
    double soma = 0.0, fat = 1.0, monomio = 1.0, h = x - x0;
    int i = 0;

    while (i < n)
    {
        soma += (c[i] / fat) * monomio;
        monomio *= h;
        fat *= ++i;
    }

    return soma;
}

double avalia_seno(int n, double x)
{
    double* coef = (double*)malloc(n * sizeof(double)),
        coef_aux[4] = {0.0, 1.0, 0.0, -1.0},
        aprox_taylor;

    if (!coef)
    {
        perror("malloc");
        return 0.0;
    }

    for (int i = 0; i < n; i++)
        coef[i] = coef_aux[i % 4];

    aprox_taylor = avalia_taylor(n, coef, 0.0, x); // Usamos a serie centrada em x0 = 0.0

    free(coef);
    return aprox_taylor;
}#pragma once

double avalia_taylor(int n, double* c, double x0, double x);
○ barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1608_LABS/Lab01$ █

```

Figura 2.7 - Teste do comando minhashell com cat

3.1 Conclusões finais

Os programas foram compilados sem erro usando-se o GCC, o ambiente de desenvolvimento do Visual Studio Code e o sistema operacional Ubuntu. Além disso, conforme se observa na saída do programa, todos os testes obtiveram êxito.