Identificação:

Felipe Cancella - 2210487

Diego Miranda - 2210996

23.2 - INF 1316 - 3WB - Profº.: Luiz Fernando Seibel

Compilando:

```
gcc -o ex1 ex1.c

gcc -o meucat meucat.c

gcc -o meuecho meuecho.c

gcc -o minhashell minhashell.c
```

Objetivo do EX1:

Criação de um programa que utilize processos hierárquicos para execução de duas funcionalidades, e entender o que ocorre durante esta divisão.

Explicação dos resultados:

Ao executar o programa, percebemos que os resultados obtidos são diferentes, isso ocorre pois pai e filho são execuções diferentes, o que quer dizer que mesmo usando a mesma variável, elas são armazenadas em endereços de memória diferentes, assim tendo independência quanto às operações realizadas.
Na prática, o processo pai usaria a variável n = 3 e o filho usaria n' = 3

Estrutura do programa:

main(void) - *A função main em questão começa inicializando as variáveis n=3 e status, e depois disso inicia um novo processo ao fazer fork(). O fork() é utilizado para criar um novo processo que é uma cópia exata do processo pai, incluindo o estado das variáveis e a sequência de execução. No processo do pai, ele realiza o loop:*

```
for(i=0;i<MAX;i++)
{
        n++;
}
```

E termina printando o resultado:

```
printf("processo pai-> pid=%d, n=%d\n",getpid(),n);
```

Depois disso, recebe o comando:

```
waitpid(-1,&status,0);
```

Que faz ele aguardar o processo do filho terminar de rodar. Quando o filho termina de rodar o seu loop:

```
for(i=0;i<MAX;i++)
{
        n+=10;
}
```

Ele também resulta em um print:

```
printf("processo filho-> pid=%d, n=%d\n",getpid(),n);
```

Solução:

Para solucionar a questão, criamos os dois processos a partir da função fork{}, e dentro de cada processo realizamos oque foi pedido (loop somando 1 ou 10 na variável n 10000 vezes). Por fim, pintamos oque a questão pedia.

```c
#define MAX 10000

int main(void) {
  int n=3, i, status;

  int pidfilho=fork();      //cria o novo processo
  if (pidfilho!=0){         //É PAI
    for(i=0;i<MAX;i++){
      n++;
    }
    printf("processo pai-> pid=%d, n=%d\n",getpid(),n);
    waitpid(-1,&status,0); //espera que o processo do filho acabe
  }
  else{                     //É FILHO
    for(i=0;i<MAX;i++){
      n+=10;
    }
    printf("processo filho-> pid=%d, n=%d\n",getpid(),n);
    exit(1);                //sem o exit ele vai executar mais uma vez o que segue no codigo
  }
}
```

```
~/lab1-so$ ./ex1
processo pai-> pid=647, n=10003
processo filho-> pid=648, n=100003
```

Observações e Conclusões:

Concluímos que mesmo ambos os códigos tendo a mesma estrutura e execução simultânea, eles são considerados como processos diferentes pelo sistema, e justamente por isso a variável n se “divide”/duplica em espaços de memória diferentes, sendo considerada “3” pelos dois.
Dificuldades encontradas: Conseguir compreender na prática como funciona a função fork() e a divisão de execuções pai e filho, porém uma vez passado esse ponto a conclusão se torna mais clara

Objetivo do EX2-Echo:

Criar uma adaptação da instrução ***echo*** do linux, que exibe os próprios argumentos de sua chamada

Explicação dos resultados:

Ao executar o programa, recebemos como resultado os parâmetros passados na chamada da função em questão (versão da echo do Linux)

Estrutura do programa:

main(int argc, char* argv[]) - *A função main recebe como parâmetro através da ordem de execução, strings que são armazenadas em um vetor (char* argv[]). Além disso, ela recebe também um valor inteiro (int argc) que representa a quantidade de parâmetros/strings passadas. Com tudo isso, exibimos todas as strings recebidas por meio de um loop "for", que itera sobre o vetor "argc" vezes.*

Solução:

Para solucionar a questão pedida, utilizamos os parâmetros argc e char* argv[] na função main, que servirão para armazenar os parâmetros recebidos, e assim printar ("echo") eles novamente no terminal.

```c
int main(int argc, char* argv[]){
  int i;
  for(i=1;i<=argc-1;i++)          //Para ler cada argumento passado
    printf("%s%s",argv[i]," ");   //Exibir o argumento em questao
  printf("\n");
  return 0;
}
```

```
~/lab1-so$ ./meuecho testando 1 2 3
testando 1 2 3
```

Observações e Conclusões:

A partir do desenvolvimento, conseguimos compreender de maneira mais clara como funciona a passagem de parâmetros para a função main, algo que foi uma dificuldade durante a criação, principalmente quanto a maneira como é armazenada, e como realizar operações sobre.

Objetivo do EX2-Cat:

Criar uma adaptação da instrução ***cat*** do linux, que exibe o que está escrito dentro dos arquivos passados como parâmetro.

Explicação dos resultados:

Ao executar o programa, recebemos como resultado as linhas escritas nos arquivos passados como parâmetros da função em questão (versão da cat do Linux)

Estrutura do programa:

main(int argc, char* argv[]) - *A função main recebe como parâmetro através da ordem de execução, strings que são armazenadas em um vetor (char* argv[]). Além disso, ela recebe também um valor inteiro (int argc) que representa a quantidade de parâmetros/nome de arquivos passados. Com tudo isso, acessamos todos os arquivos com a função:*

arq=fopen(argv[i],"r");

E por meio de um loop "while", iteramos sobre cada linha de cada arquivo com:

fgetc(arq);

imprimindo cada linha no terminal.

Solução:

Para solucionar a questão pedida, utilizamos os parâmetros argc e char* argv[] na função main, que servirão para armazenar os parâmetros recebidos, que serão os arquivos a serem "printados".

```c
int main(int argc, char* argv[]){
  int i;
  char c;
  FILE* arq;
  for(i=1;i<=argc-1;i++){     /*Loop para percorrer todos os argumentos passados*/
    arq=fopen(argv[i],"r");   /*Para cada argumento, abrir o arquivo a qual ela referencia*/
    c=fgetc(arq);             /*Pegar cada um dos caracteres presentes no arquivo*/
    while(c!=EOF){            /*Enquanto nao chegar no fim do arquivo, exibir*/
      printf("%c",c);
      c=fgetc(arq);
    }
    printf("\n");
    fclose(arq);
  }
  return 0;
}
```

```
~/lab1-so$ ./meucat ex1.c meuecho.c
//Diego Miranda (2210996)
//Felipe Cancella (2210487)
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAX 10000

int main(void) {
  int n=3, i, status;

  int pidfilho=fork();      //cria o novo processo
  if (pidfilho!=0){         //É PAI
    for(i=0;i<MAX;i++){
      n++;
    }
    printf("processo pai-> pid=%d, n=%d\n",getpid(),n);
    waitpid(-1,&status,0); //espera que o processo do filho acabe
  }
  else{                     //É FILHO
    for(i=0;i<MAX;i++){
      n+=10;
    }
    printf("processo filho-> pid=%d, n=%d\n",getpid(),n);
    exit(1);                //sem o exit ele vai executar mais uma vez
o que segue no codigo
  }
}
//Diego Miranda (2210996)
//Felipe Cancella (2210487)
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  int i;
  for(i=1;i<=argc-1;i++)          //Para ler cada argumento passado
    printf("%s%s",argv[i]," ");  //Exibir o argumento em questao
  printf("\n");
  return 0;
}
```

Observações e Conclusões:

Assim como para realizar a instrução echo, a principal dificuldade foi compreender a maneira de manipular os parâmetros passados na execução, além de relembrar a leitura de arquivo.

Objetivo do EX3 - Shell:

Desenvolver uma adaptação da ***shell*** do sistema linux, de forma em que passamos a instrução a ser executada e os parâmetros desejados.

Explicação dos resultados:

Ao executar o programa, percebemos que executamos em loop um próprio terminal, com acesso a diversos comandos do Linux (através do comando execvp), além de poder executar os criados por nós mesmos.

Estrutura do programa:

main(int argc, char* argv[]) **-** A main() é responsável por alocar os espaços de memória dos parâmetros de cada instrução, além de dentro de um loop de execução, chamar as funções type_prompt() e read_command(), executar a divisão de processos pai e filho, e liberar a memória onde eram guardados os parâmetros.

type_prompt() **-** imprime na tela a "setinha" (->>) indicando onde o usuário deve escrever seu input

read_command(char *cmd, char *params[]) **-** Lê o input do usuário e já armazena corretamente nas devidas variáveis. O char "command" recebe em formato de string o comando que a shell vai executar (ex.: echo, cat, ls, dir, etc), e parameters é um vetor que armazena todos os parâmetros do usuário para o comando passado (ex.: "bom", "dia", OU "main.c")

Solução:

Para executarmos a shell, a função main() roda em um loop onde ela deve funcionar em conjunto com as funções auxiliares type_prompt() e read_command() para identificar o input do usuário corretamente. Em cada execução/loop a main() irá obter, com ajuda das auxiliares, o input do usuário, e executar em seguida os comandos necessários por meio da função execvp(). Esses comandos serão executados dentro de um processo filho, que roda juntamente do pai. Ainda no mesmo loop, todos os parâmetros passados pelo usuário, que estavam armazenados em parameters, serão liberados por meio da função free, e assim o loop volta para o começo.

```c
void type_prompt(void){
  printf("\n ->> ");
}

void read_command(char *cmd, char *params[]) {  /*Guarda cmd em argv[0], e params em argv[...]*/
  int cont=0;
  char input[81],*token;
  fgets(input,sizeof(input),stdin);             /*Obtem toda a linha escrita*/
  token=strtok(input," \n");                    /*Separa os textos em ' ' e em '\n'*/
  strcpy(cmd,token);

  while(token!=NULL){                           /*Copiando os tokens para params[]*/
    params[cont++]=strdup(token);
    token=strtok(NULL," \n");
  }
  params[cont]=NULL;
}

int main(int argc, char *argv[]) {
  char comando[256],*parametros[10];
  int i,status,pidfilho;

  for(i=0;i<=argc;i++)                          /*Aloca espaco para os parametros*/
   parametros[i]=malloc(sizeof(char)*256);

  while (1){                                    /*Executa em loop*/
    type_prompt(); /* display prompt */
    read_command(comando,(char**)parametros);   /*Obtem os parametros*/
    /*comando é a instrucao (echo, cat, etc)*/
    /*parametro é o argv*/
    pidfilho=fork();
    if(pidfilho!=0){                            /*É pai*/
      waitpid(-1,&status,0);                    /*Espera o filho*/
    }
    else{
      execvp(comando,parametros);              /*Executa o comando com os parametros*/
    }
    for(i=0;parametros[i]!=NULL;i++)            /*Libera a memoria alocada dos parametros*/
      free(parametros[i]);
  }
  return 0;
}
```

```
~/lab1-so$ ./minhashell

 ->> ls
ex1    Makefile  meucat.c  meuecho.c   minhashell.c
ex1.c  meucat    meuecho   minhashell  replit.nix

 ->> ./meuecho testando 1 2 3
testando 1 2 3
```

Observações e Conclusões:

As principais dificuldades foram conseguir compreender como armazenar da forma correta os inputs do usuário e dividir o comando (command) e os parâmetros (parameters[]), além de como utilizar a função de execução execvp(). Além disso, fomos capazes de descobrir melhor o processo de funcionamento de uma shell, algo que utilizamos, mas sem se preocupar com como ela executa.