

INF1316 - Laboratório 1

Sistemas Operacionais

Dupla: Thiago Henrique e Julia Simão

Exercícios:

1) Faça um programa para criar dois processos, o pai escreve seu pid e espera o filho terminar e o filho escreve o seu pid e termina.

- Código fonte do programa(s):

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henrique - Exercício 1

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

int main(void) {
    int mypid, pid, status; //mypid: pid do processo atual (PAI); pid: pid do processo
    FILHO; status: status do processo FILHO.

    mypid = getpid();
    printf("Pai (PID %d) está executando.\n", mypid);

    pid = fork(); //criando o processo FILHO. obtendo o pid do FILHO (ou seja, 0).

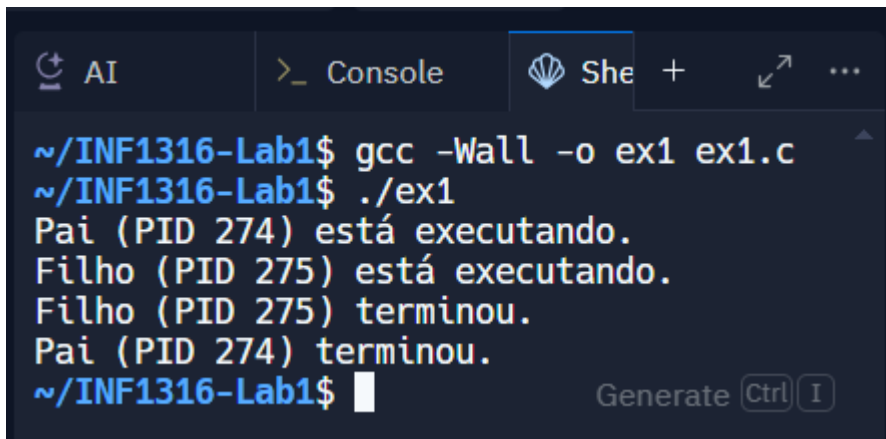
    if (pid != 0) { //Pai
        waitpid(-1, &status, 0); //esperando o processo FILHO (-1: espera algum processo
        FILHO acabar, o primeiro ou o mais rápido) terminar para executar.
        printf("Filho (PID %d) terminou.\n", pid);
    }

    else { //Filho
        pid = getpid();
        printf("Filho (PID %d) está executando.\n", pid);
        exit(3);
    }

    printf("Pai (PID %d) terminou.\n", mypid);
    return 0;
}
```

- Linha de comando p/ compilação e execução do programa e a saída gerada:

```
~/INF1316-Lab1$ gcc -Wall -o ex1 ex1.c
~/INF1316-Lab1$ ./ex1
Pai (PID 274) está executando.
Filho (PID 275) está executando.
Filho (PID 275) terminou.
Pai (PID 274) terminou.
~/INF1316-Lab1$
```



```
~/INF1316-Lab1$ gcc -Wall -o ex1 ex1.c
~/INF1316-Lab1$ ./ex1
Pai (PID 274) está executando.
Filho (PID 275) está executando.
Filho (PID 275) terminou.
Pai (PID 274) terminou.
~/INF1316-Lab1$
```

(tela do shell do replit)

- Um texto com uma reflexão sobre a razão de ter obtido esse resultado:

Quando executamos o programa que cria dois processos (um pai e um filho), observamos a seguinte sequência de eventos:

Pai (PID 1172) está executando:

O processo pai começa sua execução. Ele imprime seu próprio PID (1172) para identificação.

Filho (PID 1173) está executando:

O processo pai cria um novo processo filho (PID 1173) usando a função `fork()`.

O processo filho começa sua execução. Ele também imprime seu próprio PID: 1173 (`numeroDoProcessoDoPai + 1`)

Filho (PID 1173) terminou:

O processo filho conclui sua tarefa (nesse caso, apenas imprime seu PID) e sai com sucesso.

Pai (PID 1172) terminou:

O processo pai aguarda até que o processo filho termine (usando `waitpid()`).

Após o término do filho, o processo pai continua sua execução e imprime que ele próprio (PID 1172) também terminou.

Essa sequência de eventos ilustra a criação e execução de processos em um ambiente multitarefa. Cada processo tem seu próprio espaço de memória e é independente dos outros. O uso de PIDs permite que o sistema operacional gerencie e controle esses processos de maneira eficiente.

Logo simulamos a visualização de um gerenciador de processos de um Sistema Operacional.

2) Agora, usando a mesma estrutura de processos pai e filho, declare uma variável visível ao pai e ao filho, no pai inicialize a variável com 1 e imprima seu valor antes do fork(). No filho, altere o valor da variável para 5 e imprima o seu valor antes do exit(). Agora, no pai, imprima novamente o valor da variável após o filho ter alterado a variável - após a waitpid(). Justifique os resultados obtidos.

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henriques - Exercício 2

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

int main(void) {
    int mypid, pid, status; //mypid: pid do processo atual (PAI); pid: pid do processo
    FILHO; status: status do processo FILHO.

    //declarando e inicializando a variável com o valor 1 no PAI e FILHO (antes do fork).
    int var = 1;
    printf("Valor da variável (bloco PAI antes do waitpit): %d\n", var);

    mypid = getpid();
    printf("Pai (PID %d) está executando.\n", mypid);

    pid = fork(); //criando o processo FILHO. obtendo o pid do FILHO (ou seja, 0).

    if (pid != 0) { //Pai

        waitpid(-1, &status, 0); //esperando o processo FILHO (-1: espera algum processo
        FILHO acabar, o primeiro ou o mais rápido) terminar para executar. imprimindo o
        valor da variável após o FILHO terminar (após a alteração do valor no FILHO).

        printf("Valor da variável (bloco PAI depois do waitpit): %d\n", var);
        printf("Filho (PID %d) terminou.\n", pid);
    }
}
```

```

else { //Filho
    pid = getpid();
    printf("Filho (PID %d) está executando.\n", pid);

    //alterando o valor da variável no FILHO (antes do exit).
    var = 5;
    printf("Valor da variável (bloco FILHO): %d\n", var);

    exit(3);
}

printf("Pai (PID %d) terminou.\n", mypid);
return 0;
}

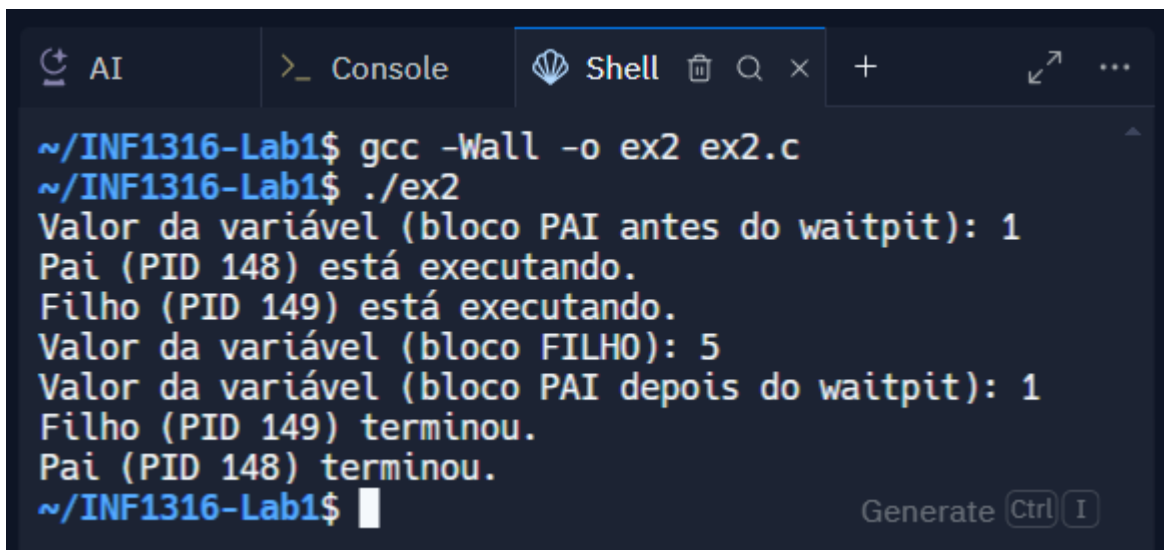
```

- Linha de comando p/ compilação e execução do programa e a saída gerada:

```

~/INF1316-Lab1$ gcc -Wall -o ex2 ex2.c
~/INF1316-Lab1$ ./ex2
Valor da variável (bloco PAI antes do waitpit): 1
Pai (PID 148) está executando.
Filho (PID 149) está executando.
Valor da variável (bloco FILHO): 5
Valor da variável (bloco PAI depois do waitpit): 1
Filho (PID 149) terminou.
Pai (PID 148) terminou.
~/INF1316-Lab1$

```



```

~ AI  >_ Console  Shell  Q x +  ↗ ...
~/INF1316-Lab1$ gcc -Wall -o ex2 ex2.c
~/INF1316-Lab1$ ./ex2
Valor da variável (bloco PAI antes do waitpit): 1
Pai (PID 148) está executando.
Filho (PID 149) está executando.
Valor da variável (bloco FILHO): 5
Valor da variável (bloco PAI depois do waitpit): 1
Filho (PID 149) terminou.
Pai (PID 148) terminou.
~/INF1316-Lab1$
Generate Ctrl I

```

(tela do shell do replit)

- Justificativa do resultado obtido:

O resultado do programa se deve ao comportamento do sistema operacional em relação ao 'fork()', que cria uma cópia exata do processo pai para o filho, incluindo as variáveis. No entanto, após a criação, cada processo possui seu próprio espaço de memória, o que significa que mudanças feitas em uma variável no processo filho não afetam o pai. Por isso, mesmo que o filho altere o valor de 'var' para 5, o pai continua com o valor original 1. O uso de 'waitpid()' garante que o pai espere o término do filho antes de continuar, o que permite ao pai exibir o valor de 'var' após o término do processo filho.

3) Use o programa anterior para ler e ordenar um vetor de 10 posições. O filho ordena o vetor e o pai exibe os dados do vetor antes do fork() e depois do waitpid(). Eles usarão o mesmo vetor na memória? Justifique.

- Código fonte do programa(s):

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henriques - Exercício 3

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

#define SIZE 10

void bubble_sort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

/*
void bubble_sort (int vetor[], int n) {
    int k, j, aux;

    for (k = 1; k < n; k++) {
        printf("\n[%d] ", k);

        for (j = 0; j < n - 1; j++) {
            printf("%d, ", j);

            if (vetor[j] > vetor[j + 1]) {
```

```

        aux = vetor[j];
        vetor[j] = vetor[j + 1];
        vetor[j + 1] = aux;
    }
}
}
}
}

```

código base para a função bubble_sort:

<http://devfuria.com.br/logica-de-programacao/exemplos-na-linguagem-c-do-algoritmo-bubble-sort/>

```
*/
```

```

int main(void) {
    int mypid, pid, status;
    int arr[SIZE] = {9, 4, 7, 3, 2, 8, 5, 1, 6, 0}; //vetor de 10 posições.

    printf("Vetor antes do fork():\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    mypid = getpid();
    printf("Pai (PID %d) está executando.\n", mypid);

    pid = fork(); //criando o processo FILHO.

    if (pid != 0) {          //Pai
        waitpid(-1, &status, 0); //espera algum FILHO terminar para executar.
        printf("Vetor no PAI depois do waitpid():\n");
        for (int i = 0; i < SIZE; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
        printf("Filho (PID %d) terminou.\n", pid);
    } else { //Filho
        pid = getpid();
        printf("Filho (PID %d) está executando.\n", pid);

        bubble_sort(arr, SIZE); //FILHO ordena o vetor
        printf("Vetor no FILHO (ordenado):\n");
        for (int i = 0; i < SIZE; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");

        exit(0);
    }
}

```

```

}

printf("Pai (PID %d) terminou.\n", mypid);
return 0;
}

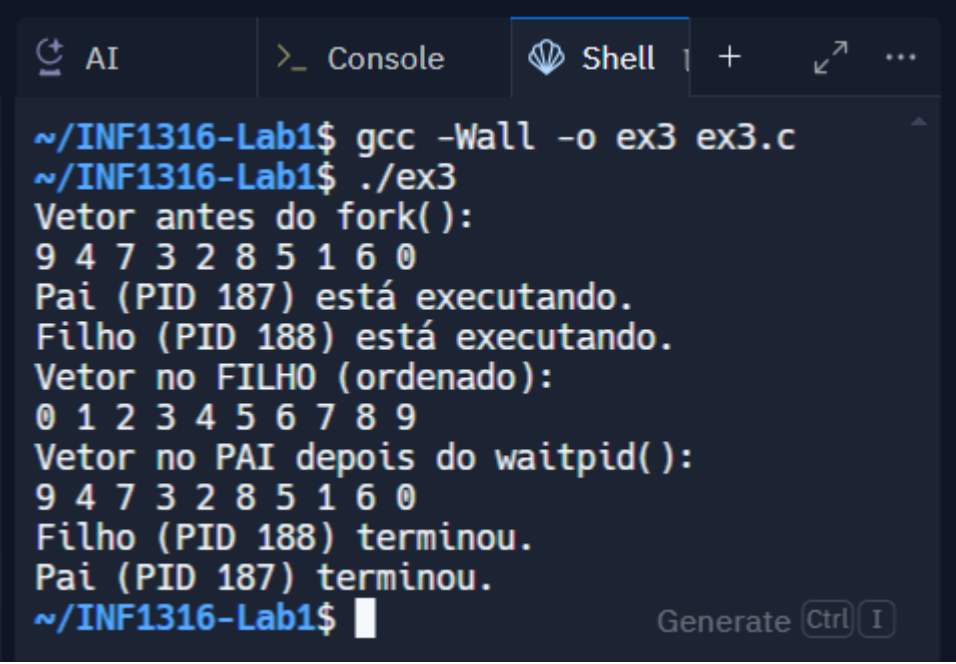
```

- Linha de comando p/ compilação e execução do programa e a saída gerada:

```

~/INF1316-Lab1$ gcc -Wall -o ex3 ex3.c
~/INF1316-Lab1$ ./ex3
Vetor antes do fork():
9 4 7 3 2 8 5 1 6 0
Pai (PID 187) está executando.
Filho (PID 188) está executando.
Vetor no FILHO (ordenado):
0 1 2 3 4 5 6 7 8 9
Vetor no PAI depois do waitpid():
9 4 7 3 2 8 5 1 6 0
Filho (PID 188) terminou.
Pai (PID 187) terminou.
~/INF1316-Lab1$

```



```

~/INF1316-Lab1$ gcc -Wall -o ex3 ex3.c
~/INF1316-Lab1$ ./ex3
Vetor antes do fork():
9 4 7 3 2 8 5 1 6 0
Pai (PID 187) está executando.
Filho (PID 188) está executando.
Vetor no FILHO (ordenado):
0 1 2 3 4 5 6 7 8 9
Vetor no PAI depois do waitpid():
9 4 7 3 2 8 5 1 6 0
Filho (PID 188) terminou.
Pai (PID 187) terminou.
~/INF1316-Lab1$

```

(tela do shell do replit)

- Justificativa do resultado obtido:

Após o 'fork()', o vetor 'arr' no processo pai e no processo filho são independentes. Mesmo que o filho ordene o vetor, essa alteração não reflete no processo pai, pois ambos têm suas

próprias cópias do vetor em suas respectivas memórias. Portanto, quando o pai exibe o vetor após o 'waitpid()', ele mostrará os valores originais, não os ordenados.

Isso ocorre porque o 'fork()' cria uma cópia do processo, incluindo toda a memória associada ao processo pai no momento do 'fork()'. Como resultado, pai e filho possuem cópias distintas do vetor 'arr', e as mudanças em um processo não afetam o outro.

4) Modifique o programa anterior para que o filho execute um programa elaborado por você, que mande imprimir uma mensagem qualquer no vídeo, por exemplo, “alo mundo”. Em seguida, altere o programa do item 4 para o filho executar o programa echo da shell.

- **Código fonte do programa(s):**

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henriques - Exercício 4 (alomundo)

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

```
int main(void) {  
    printf("Alô Mundo\n");  
    return 0;  
}
```

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henriques - Exercício 4 (principal)

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

```
int main(void) {  
    int mypid, pid, status;
```

```
    mypid = getpid();  
    printf("Pai (PID %d) está executando.\n", mypid);
```

```
    pid = fork(); //criando o processo FILHO.
```

```
    if (pid != 0) { //Pai  
        waitpid(-1, &status, 0); //espera algum FILHO terminar.  
        printf("Filho (PID %d) terminou.\n", pid);
```

```
    } else { //Filho  
        pid = getpid();  
        printf("Filho (PID %d) está executando o programa 'alomundo'.\n", pid);
```

```
        //executa o programa 'alomundo'.
```



```

    execl("./alomundo", "alomundo", (char *)NULL);

    //caso o execl falhe.
    perror("execl falhou");
    exit(1);
}

printf("Pai (PID %d) terminou.\n", mypid);
return 0;
}

```

//Laboratório 1 - 20/08/2024 - Julia Simão e Thiago Henriques - Exercício 4 (echo)

```

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

int main(void) {
    int mypid, pid, status;

    mypid = getpid();
    printf("Pai (PID %d) está executando.\n", mypid);

    pid = fork(); //criando o processo FILHO.

    if (pid != 0) { //Pai
        waitpid(-1, &status, 0); //espera algum FILHO terminar.
        printf("Filho (PID %d) terminou.\n", pid);
    } else { //Filho
        pid = getpid();
        printf("Filho (PID %d) está executando o comando 'echo'.\n", pid);

        //executa o comando 'echo' da shell.
        execl("/bin/echo", "echo", "Alô Mundo", (char *)NULL);

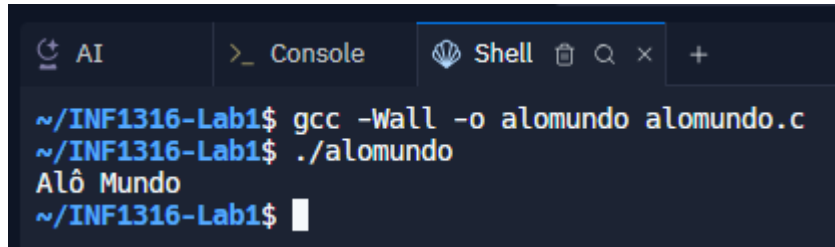
        //caso o execl falhe.
        perror("execl falhou");
        exit(1);
    }

    printf("Pai (PID %d) terminou.\n", mypid);
    return 0;
}

```

- Linha de comando p/ compilação e execução do programa e a saída gerada:

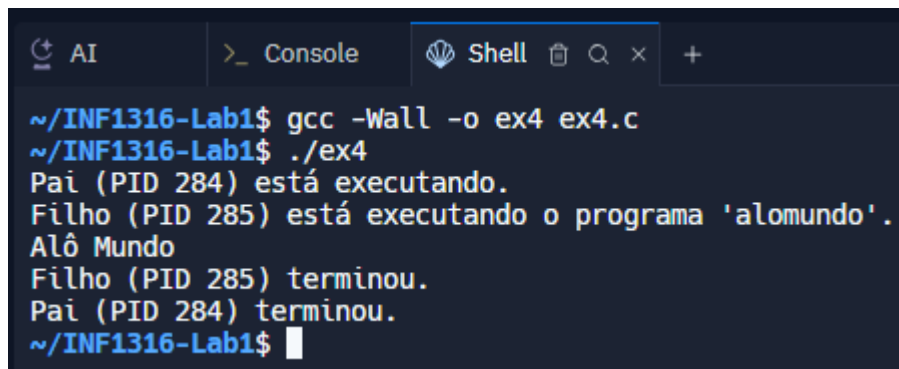
```
~/INF1316-Lab1$ gcc -Wall -o alomundo alomundo.c
~/INF1316-Lab1$ ./alomundo
Alô Mundo
~/INF1316-Lab1$
```

A screenshot of a Replit shell terminal window. The window has a dark background with a light blue border. At the top, there are tabs for 'AI', 'Console', and 'Shell'. The 'Shell' tab is active. The terminal shows the following commands and output:

```
~/INF1316-Lab1$ gcc -Wall -o alomundo alomundo.c
~/INF1316-Lab1$ ./alomundo
Alô Mundo
~/INF1316-Lab1$
```

(tela do shell do replit)

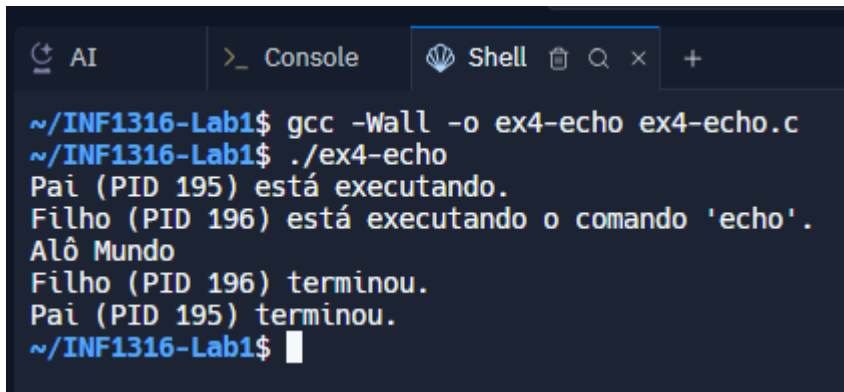
```
~/INF1316-Lab1$ gcc -Wall -o ex4 ex4.c
~/INF1316-Lab1$ ./ex4
Pai (PID 284) está executando.
Filho (PID 285) está executando o programa 'alomundo'.
Alô Mundo
Filho (PID 285) terminou.
Pai (PID 284) terminou.
~/INF1316-Lab1$
```

A screenshot of a Replit shell terminal window, similar to the one above. It shows the compilation and execution of ex4.c. The output includes the parent process (PID 284) running, the child process (PID 285) running 'alomundo', the child process finishing, and the parent process finishing.

```
~/INF1316-Lab1$ gcc -Wall -o ex4 ex4.c
~/INF1316-Lab1$ ./ex4
Pai (PID 284) está executando.
Filho (PID 285) está executando o programa 'alomundo'.
Alô Mundo
Filho (PID 285) terminou.
Pai (PID 284) terminou.
~/INF1316-Lab1$
```

(tela do shell do replit)

```
~/INF1316-Lab1$ gcc -Wall -o ex4-echo ex4-echo.c
~/INF1316-Lab1$ ./ex4-echo
Pai (PID 195) está executando.
Filho (PID 196) está executando o comando 'echo'.
Alô Mundo
Filho (PID 196) terminou.
Pai (PID 195) terminou.
~/INF1316-Lab1$
```



```
~/INF1316-Lab1$ gcc -Wall -o ex4-echo ex4-echo.c
~/INF1316-Lab1$ ./ex4-echo
Pai (PID 195) está executando.
Filho (PID 196) está executando o comando 'echo'.
Alô Mundo
Filho (PID 196) terminou.
Pai (PID 195) terminou.
~/INF1316-Lab1$
```

(tela do shell do replit)

- **Justificativa do resultado obtido:**

Principal:

- O processo filho é criado com 'fork()'.
- O filho usa 'execl()' para executar o programa 'alomundo', que deve ser compilado e estar no mesmo diretório do programa principal. Esse programa imprime "Alô Mundo".
- O pai espera o término do filho com 'waitpid()' e imprime uma mensagem após a conclusão.

Echo:

- O processo filho executa o comando 'echo' da shell, que imprime "Alô Mundo".
- O comando 'execl()' é usado para substituir o código do filho com a execução do comando 'echo'.

Nos dois casos, o processo filho é substituído (é substituído o espaço de endereçamento do processo) pelo novo código após o 'fork()' usando 'execl()'. Se 'execl()' falhar, o código imprime uma mensagem de erro com 'perror()' e o filho termina com 'exit(1)'.