A chamada de sistema fork() - Como criar e Gerenciar Processos

Nesse tutorial, da seção Programação de Sistemas Operacionais, vamos finalmente botar a mão na massa e programar, fazendo códigos com a linguagem C, para criar e gerenciar processos através da função **fork()**, uma importantíssima chamada de sistema (*system calls*), de sistemas operacionais do tipo Unix.

Função fork() - O Que É e Para Que Serve

O **fork** é uma função que é uma chamada de sistema. Ou seja, ela invoca o sistema operacional para fazer alguma tarefa que o usuário não pode.
No caso, o fork é usado para criar um novo processo em sistemas do tipo Unix, e isso só pode ser feito via fork.

Quando criamos um processo por meio do fork, dizemos que esse novo processo é o filho, e processo pai é aquele que usou o fork.

Por exemplo, suponha que você programou um software em C, e nele usou a chamada **fork()**.
Esse programa em C, executando, é o processo pai.

Quando usamos o fork, será criado o processo filho, que será idêntico ao pai, inclusive tendo as mesmas variáveis, registros, descritores de arquivos etc.
Ou seja, o processo filho é uma cópia do pai, exatamente igual.

Porém, é uma cópia, e como tal, depois de criado o processo filho, ele vai ser executado e o que acontece em um processo não ocorre no outro, são processos distintos agora, cada um seguindo seu rumo, onde é possível mudar o valor de uma variável em um e isso não irá alterar o valor desta variável no outro processo, por exemplo.

Como usar a função fork()

Além das bibliotecas que normalmente usamos para programar em C, necessitaremos de duas **sys/types.h** e **unistd.h** do Unix, para podermos trabalhar com a fork.

Para usar a chamada de sistema de criação de processos, simplesmente escrevemos **fork()**, sem passar argumento algum. Fazendo isso, o Sistema Operacional se encarrega do resto, e retorna um número.

Este número é o **pid** (*process identification*, ou identificador de processos), e cada processo tem um valor diferente de pid, é como se fosse o RG, a identificação de cada processo.

Porém, ao armazenar esse retorno da função fork numa variável de nome 'pid' do tipo 'pid_t',

vemos que esse número de pid tem um comportamento especial:
- dentro do processo filho, o **pid tem valor 0**
- dentro do processo pai, o **pid tem o valor do processo filho**
- a fork() retorna um valor negativo, caso tenha ocorrido algum erro.

Assim, para criarmos um programa e diferenciemos o processo pai do filho, basta fazermos um teste com esse valor de retorno da fork().
Primeiro, testamos se a fork() foi bem sucedida.

Em seguida, fazemos **if (pid == 0)**, e isso só é verdade no processo filho.
Ou seja, tudo dentro desse if só vai ser executado pelo processo filho, o pai não entra nessa condicional.

E caso esse teste condicional seja falso, é porque o processo em vigor é o pai.
Então, dentro do **else**, colocamos aquilo que será executado somente no processo pai.

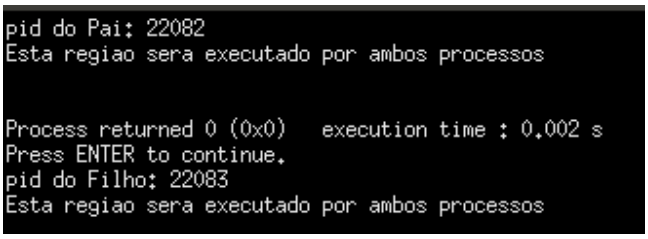
Vamos simplesmente printar uma string, para mostrar na tela o resultado do processo filho e o do processo pai (usamos a função **getpid()** que retorna o pid do processo em execução - assim dá pra ver o real pid do pai e o do filho).

O código para a separação do processo pai do filho é:

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

int main(void)
{
    int i;
    pid_t pid;

    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(1);
    }
    if (pid == 0)
    {
        //O código aqui dentro será executado no processo filho
        printf("pid do Filho: %d\n", getpid());
    }
    else
    {
        //O código neste trecho será executado no processo pai
        printf("pid do Pai: %d\n", getpid());
    }


    printf("Esta regiao sera executada por ambos processos\n\n");
    scanf("%d", &i);
    exit(0);
}
```

É importante colocar algum comando que impeça que nosso programa abra, execute e rapidamente feche, não sendo possível ver os prints. Assim, caso não esteja usando o Code::Blocks, coloque um scanf() no final do programa, para você ver o resultado na tela.

Como o Code Blocks não fecha o programa quando termina, vemos o seguinte resultado:



Resultado do processo pai e do processo filho

Vejamos o que ocorre!
Primeiro, o processo original roda e cai no **else**, pois é o pai, e sua pid é exibida (22082)
Em seguida, exibe a mensagem final "Essa regiao sera executada por ambos processos", então termina seu processo, como é visto na imagem.

Porém, foi criado outro processo com a função **fork()**, então um processo idêntico vai ser executado, e ele é o processo filho, e temos certeza disso pois foi executado o que estava dentro do **if** (pid do filho é 22083). Por fim, o filho mostra a mensagem final.

Como ambos processos criados foram finalizados, agora sim nosso programa terminou!

Memória nos Processos Pai e Filho

Vamos fazer um experimento simples, declarar inicialmente uma variável do tipo **char** e iniciar ela com o valor 'a'.

Então, no processo pai mudamo essa variável para armazenar 'b'.
Vamos exibir, em ambos processos, o valor da variável e seu endereço de memória.

Nosso código fica:

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

int main(void)
{
    int i;
```

```
    char c = 'a';
    pid_t pid;

    if ((pid = fork()) < 0)
    {
        perror("fork");
        exit(1);
    }
    if (pid == 0)
    {
        printf("Caractere e Endereco: %c - %p (filho)\n",c,&c);
    }
    else
    {
        c='b';
        printf("Caractere e Endereco: %c - %p (pai)\n",c,&c);
    }

    scanf("%d", &i);
    exit(0);
}
```

O resultado é o seguinte:

```
Caractere e Endereco: b - 0x7fffe44b81bf (pai)
Caractere e Endereco: a - 0x7fffe44b81bf (filho)
```

Mapa de memória é o mesmo no processo pai e filho

Vamos entender o que ocorre, passo a passo!
Primeiro, declaramos a variável **char** no processo pai.
A fork() é executada e agora temos dois processos.

O pai foi executado (o else), onde antes do printf, mudamos o valor da variável char para 'b'. Assim, a letra b é exibida, bem como seu endereço de memória (endereço da char).

Agora o processo filho ocorreu, e devemos lembrar que ele é uma cópia exata do processo pai. Assim, para o processo filho, o valor da variável char ainda é 'a', pois a mudança de 'a' para 'b' ocorreu no processo pai, e o processo filho não viu isso nem tampouco tem nada a ver com isso.

Para o processo filho, a variável **char** tem aquele valor do início, que é 'a', como é possível ver no resultado no terminal de comando.

Daí, notamos duas coisas: o mapa de memória do processo pai e filho é o mesmo, pois ambas possuem o mesmo endereço de memória da variável char.
O outro fato é: ao mudar o valor de uma variável num processo, isso não alterou o valor no outro.

A função execv() - Filho matando o Pai

A função **execv()** é usada para executar um programa, em ambientes Linux.
Ou seja, ela cria um processo novo, que é o programa que vai executar.

Para ver a **execv** em ação, vamos criar um programa simples, de nome "teste" que simplesmente exibe a PID desse arquivo "teste", que é um executável, cujo código fonte é:

```
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("\nPid do arquivo: %d\n", getpid());
}
```

Agora, nosso programa, que usa a função **execv** para executar o arquivo "teste":

```
#include <stdio.h>

int main(void)
{
    printf("pid do Pai: %d\n", getpid());
    execv("teste", NULL);
    printf("EU TENHO UM SEGREDO PRA CONTAR\n");
}
```

O resultado é:

Usando a função **execv** para criar um processo

Inicialmente, nosso programa exibe sua pid, é o processo pai, que é 11685.
Em seguida, usamos a função **execv** que vai criar um novo processo, um processo filho, que é a execução do programa "teste".

Esse programa "teste" exibe sua pid, que também é 11685, mesmo pid do processo pai.
Por fim, tem um printf "Eu tenho um segredo pra contar", que não é exibido.

Consegue deduzir o que ocorreu?
A função **execv** mata o processo pai, e o filho pega sua PID.
Por isso o último print não apareceu, pois o processo pai foi substituído pelo filho!

A função system() - Filho e Pai, sem crimes

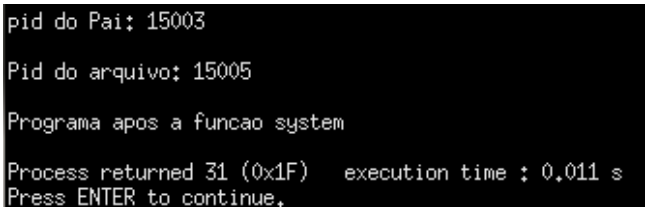
Assim como a função **execv**, a função **system** também executa um programa, na verdade ela simula o terminal, então é como se déssemos um comando no terminal.
A diferença é que aqui a função filho não mata a pai, portanto, não há crimes ;)

Vamos usar a função system para executar o mesmo arquivo "teste", isso é feito no terminal assim **./teste**

Logo, nosso código fica:

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

int main(void)
{
    printf("pid do Pai: %d\n", getpid());
    system("./teste");
    printf("\nPrograma apos a funcao system\n");
}
```

E o resultado da system() é:



Criando processos usando a função system()

Exibimos o pid do processo pai, que é 15003.
Em seguida, usamos a **system** para executar o programa "teste", que será o processo filho de pid 15005.
E, por fim, é exibido o print, depois da função system.

É fácil ver que a função **system** cria um novo processo filho sem matar o pai, e ambos existem, com pids diferentes.

Ambos exemplos, **execv** e **system**, criam novos processos, cada um com suas características. Porém, todas essas funções são implementadas com a chamada de sistema **fork**, pois como dissemos, é só através dela que podemos criar um processo.

No decorrer de nosso estudo sobre Sistemas Operacionais, veremos outras chamadas de sistemas e maneiras de trabalhar com processos.