Insper

16 - `exec` e relações entre processos

Sistemas Hardware-Software - 2019/1

Igor Montagner

Parte 0 - `fork` e `wait` de novo

Vamos começar nossa aula com dois exercícios de revisão.

Exercício: Simule o programa abaixo.

```
pid_t f1, f2;
long var = 20;

f1 = fork();

if (f1 == 0) {
    var *= 2;
}

f2 = fork();
if (f2 == 0) {
    var += 3;
    if (f1 == 0) {
        wait(NULL);
    }
} else {
    var -= 3;
}

printf("Meu var: %ld  f1 %ld f2 %ld, meu pid: %ld\n", var, f1, f2, getpid());
```

Dicas:

- após cada fork dê ao novo processo um `pid`. Isso facilita lembrar quais processos estão em qual parte do programa.
- quantos prints serão feitos?

Exercício: Vamos agora trabalhar no arquivo `fork1.c`. Ele contém três funções `func1, func2` e `func3`. Faça um programa que executa cada uma destas funções em um processo diferente. O processo original deve esperar os três processos terminarem, indicando qual terminou e mostrando o valor de retorno caso ele tenha terminado corretamente e uma mensagem de erro caso tenha dado pau.

Parte 1 - relações entre processos

Quando um processo termina ele retorna um valor de retorno para seu pai. Isto inclui, além dos 8 bits menos significativos do valor retornado pelo main, informações dizendo se o processo terminou normalmente ou se ele foi terminado anormalmente (deu erro ou usuário apertou Ctrl+C).

O quê acontece quando o filho termina antes do pai?

Vamos analisar agora o arquivo abaixo (`fork2.c`).

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

void func1() {
    printf("Acabei rapidinho!\n");
}

int main() {
    pid_t f1, f2, f3;

    printf("pai: pid %d ppid %d\n", getpid(), getppid());

    f1 = fork();
    if (f1 == 0) {
        func1();
        exit(0);
    }

    while (1) {
        ;
    }

    return 0;
}
```

Compile e rode o arquivo. Claramente o pai entra em um loop infinito sem fazer nada.

Exercício: Use o comando `ps` (em outro terminal) para ver o status do processo pai e do processo filho. Anote abaixo o que você encontrou.

Exercício: Pesquise o que é um processo em estado *defunct* (às vezes referido por *zombie*).

Um processo filho fica em estado *zombie* até que seu pai use a chamada `wait` para acessar seu valor de retorno e informações de execução. Logo, o quê acontece se o pai nunca chamar `wait`?

- Se o pai é um serviço e nunca termina o processo filho ocupará recursos indefinidamente. Se vários filhos forem criados assim isto pode se tornar um problema!
- Se o pai termina veremos o que acontece na seção abaixo ;)

Extra: pesquise o quê ocorre quando um sistema não possui mais *pid*s disponíveis.

O quê acontece quando o pai termina antes do filho?

Vamos voltar agora ao arquivo `fork1.c`. Você deve ter notado que a função `func3` é um loop infinito que imprime de 5 em 5 segundos o pid e o ppid de seu processo.

Exercício: Modifique seu `fork1.c` para que ele espere somente 2 filhos acabarem e termine. O quê ocorre com `func3` após o processo original terminar? Algo muda em seus prints?

Quando pai de um processo p termina p é herdado pelo processo `init`, que espera por seu fim e autoriza a destruição os dados relativos ao fim do p. Por esta razão `getppid` muda seu valor após o término do processo original! Este mecanismo funciona também para processos zumbis, que são limpos pelo processo `init` de tempos em tempos.

Parte 2 - carregando novos programas com `exec`

A chamada `execvp` é usada para carregar programas na memória e executá-los. O novo programa é carregado no contexto do processo atual, substituindo-o por completo. Veja um exemplo de uso correto do `execvp` abaixo.

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

int main() {
    char prog[] = "ls";
    // a lista de argumentos sempre começa com o nome do
    // programa e termina com NULL
    char *args[] = {"ls", "-l", "-a", NULL};

    execvp(prog, args);
    printf("Fim do exec!\n");

    return 0;
}
```

Pergunta: por que o programa acima não dá o `printf` abaixo do `execvp` terminar?

Os argumentos passados no `execvp` são passados para o `main` do programa executado via argumentos do `main`. Ao fazer a chamada

```c
char prog[] = "prog1";
char *args[] = {"prog1", "arg1", "arg2", NULL};

execvp(prog, args);
```

O main de `prog1` será chamado com `argc=3` e `argv={"prog1", "arg1", "arg2"}`. O primeiro argumento é sempre o nome do programa chamado. Note que os argumentos são sempre strings e que precisamos convertê-los para o tipo desejado "na mão".

Exercício: Crie um programa `eh_par` que recebe um inteiro como argumento de linha de comando e cujo `main` retorne 1 se o número for par, 0 caso contrário e -1 se ele for negativo.

Dica:

- pesquise para função `atol` para fazer a conversão do argumento de linha de comando para `long`.
- você pode testar seu programa no terminal: basta rodar `eh_par 10` para checar se o número 10 é par. É assim que todo programa de linha de comando recebe parâmetros ;)

Vamos agora juntar `fork`, `wait` e `exec` em um único exercício!

Exercício: Crie um programa que recebe números via `scanf`, executa `eh_par` em um processo filho e usa seu valor de retorno para decidir se o número é par ou não. Seu programa deverá parar de receber números quando `eh_par` retornar -1.

Dicas:

1. comece ambos programas em um arquivo vazio. Tentar adaptar os exemplos pode nos levar a soluções erradas pois não pensamos no contexto geral do programa.
2. teste seu programa `eh_par` antes de fazer a segunda parte.