Estudo sobre Subprocessos Cooperativos

Grupo
Júlio César Machado Bueno
Luiza Diniz e Castro
Roberta Santos Lopes

Sumário

Estudo de comandos

fork()

A função *fork()* é uma primitiva que possibilita a criação de um novo processo em um sistema UNIX.

Este novo processo criado é denominado de “processo filho” ou subprocesso, e possui o mesmo código fonte que o seu processo criador. Embora possua o mesmo código fonte e *uid (User Identifier)* que o processo que o originou, denominado de “processo pai”, durante a sua execução, ele possui atributos diferenciados como o seu PID e é tratado de forma dissociada do seu processo pai pelo Sistema Operacional.

O relacionamento entre o processo pai e processo filho se dá de forma unilateral, onde o pai conhece o PID do seu processo filho enquanto o processo filho não conhece o ID do seu pai, exceto por rotinas de maior privilégio.

A criação de um subprocessos exige que o programador tenha cuidados com o uso de recursos compartilhados como arquivos e variáveis de escopo público. Todas as varáveis possuem valores idênticos até a execução do *fork()*. Isto é, se uma variável instanciada de forma compartilhada (anterior ao *fork()*) for alterada pelo filho, essa alteração se refletirá durante a execução do pai e vice-versa. Outro exemplo de cuidado a ser tomado é o caso onde se o processo pai e processo filho estão em execução e o processo pai é terminado ambos os processos serão terminados. Porém se o processo filho for terminado o processo pai pode ainda continuar.

O funcionamento da função fork() se dá da seguinte forma:

1. Verifica se há lugar na Tabela de Processos;
2. Tenta alocar memória para o filho;
3. Altera mapa de memória e copia para a tabela;
4. Copia imagem do *pai* para o filho;
5. Copia para a tabela de processos a informação do processo pai (PID, prioridades, estado, etc);
6. Atribui um PID ao filho
7. Informa ao *Kernel* e o sistema de arquivos que foi criado um novo processo.

A seguir um exemplo de uso da função *fork();*

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

int main(void){
        int id;
        //Exibe o ID do processo corrente
        printf("Processo Corrente - %i\n\n", getpid());

        id = fork();

        if (id > 0){
                printf("Eu sou o processo pai");
        }elseif( id == 0 ){
                printf("Eu sou o processo filho");
        }else{
                printf("Não foi possível criar o subprocesso");
        }
}
```

exec

A primitiva exec é de fato um gurpo de funções (*execl()*, *execlp()*, *execle()*, *execv()*, *execvp()*) que permitem a execução de um programa externo ao processo corrente, ou seja, lançar de forma independente de um outro processo proveniente do sistema de arquivos. Não representa a criação de um processo, efetivamente, mas sim a substituição do programa de execução, ou seja, do processo.

Na chamada de uma primitiva *exec()*, existe uma sobreposição do segmento de instruções do processo que a chama. Dessa forma, não existe retorno de um *exec()* cuja execução seja correta, pois o endereço de retorno desaparece, ou seja, o processo que chama a primitiva *exec()* morre.

execl()

Neste exemplo, a principal diferença das demais é o número conhecido de parâmetros a serem passados onde o 1º argumento é o caminho para o processo e os seguintes parâmetros são passados ao processo a ser executado. Caso *execl()* não seja executado com sucesso retornará o valor -1;

A seguir um exemplo do uso de *execl()*:

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

int main(void){
        //Executa o comando ls
        execl("/bin/ls", "/bin/ls", NULL);
}
```

wait()

A função *wait()* suspende a execução do processo corrente até o término de todos os seus processos filhos. Caso não haja mais subprocessos sendo executados o processo que executou o *wait()* continua com a sua execução. Caso o processo pai termine antes do processo filho, este processo filho se torna um processo *zumbi (zombie)* ou como recentemente aparece na literatura *defunto (defunct)*. Assim esse subprocesso possui suas instruções e dados terminados mas continuam ocupando espaço na tabela de processos do *kernel* do Sistema Operacional.

A seguir um exemplo de uso de *wait()*:

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

int main(void){
        int id, status;

        d = fork();

        if (id > 0){
                printf("Eu sou o processo pai e a seguir vou aguardar pelo meu filho.");
                wait(&status);
        }elseif( id == 0 ){
                printf("Eu sou o processo filho");
        }else{
                printf("Não foi possível criar o subprocesso");
        }
}
```

exit()

Ao executar o comando *exit()*, se o processo específico possui um processo pai esperando o seu término então um sinal de SIGCHLD é enviado ao processo pai indicando o seu término normal. Caso seu pai não esteja a espera do seu término o processo ser torna zumbi. A função *exit()* possui o seguinte parâmetro:

0 – O processo foi terminado normalmente.

Qualquer inteiro diferente de 0 – O processo foi encerrado com erro.

A seguir um exemplo de uso de *exit()*:

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

int main(void){
        int id;

        id = fork();

        if (id > 0){
                printf("Eu sou o processo pai .");
        }elseif( id == 0 ){
                printf("Eu sou o processo filho e vou terminar a minha execução normalmente.");
                exit(0);
        }else{
                printf("Não foi possível criar o subprocesso");
        }
}
```

getpid()

Retorna como um valor inteiro o PID do processo corrente;

A seguir um exemplo de uso de *getpid()*:

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

int main(void){
        int id;

        id = fork();

        if (id > 0){
                printf("Eu sou o processo pai de ID %i", getpid()) ;
        }elseif( id == 0 ){
                printf("Eu sou o processo filho de ID %i", getpid()) ;
        }else{
                printf("Não foi possível criar o subprocesso");
        }
}
```

getppid()
Retorna como um valor inteiro o PID do processo pai do processo corrente;

A seguir um exemplo de uso de *getppid()*:

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

int main(void){
        int id;

        id = fork();

        if (id > 0){
                printf("Eu sou o processo pai de ID %i", getpid()) ;
        }elseif( id == 0 ){
                printf("Meu processo pai é de ID %i", getppid()) ;
        }else{
                printf("Não foi possível criar o subprocesso");
        }
}
```

Prog1

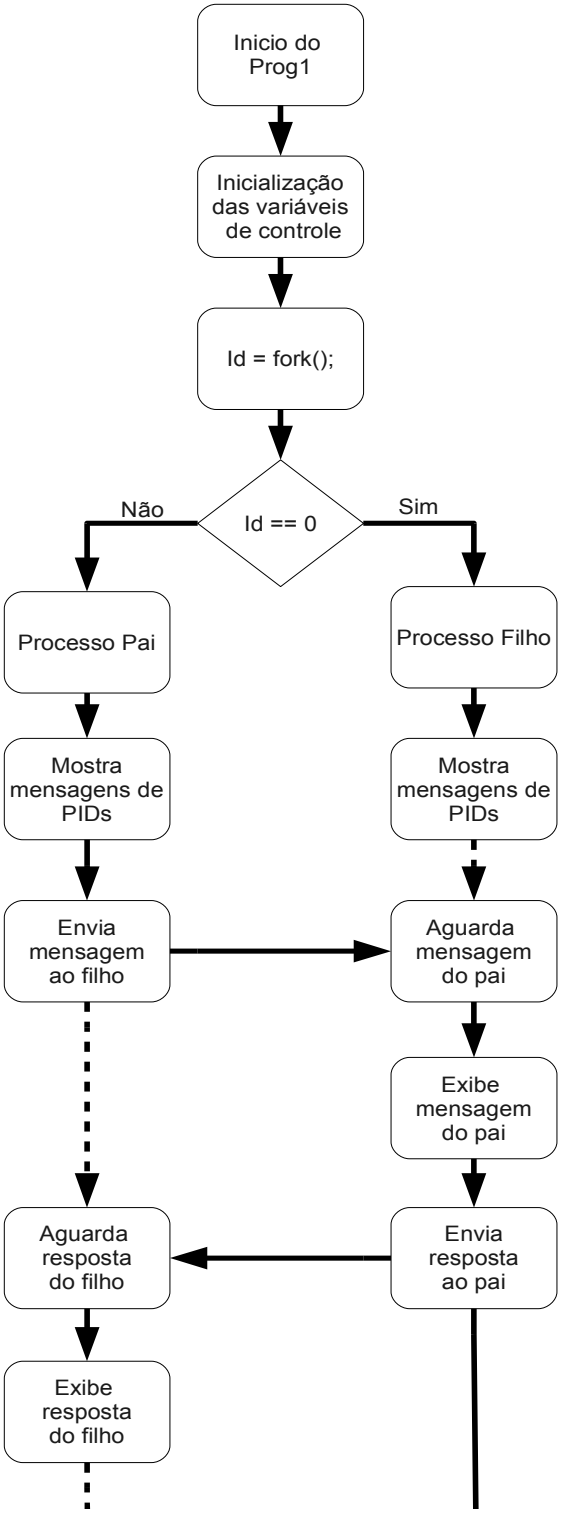
O Prog1 tem como intuito, a exemplificação do uso dos comandos *fork()* e *wait()* associado ao uso de:

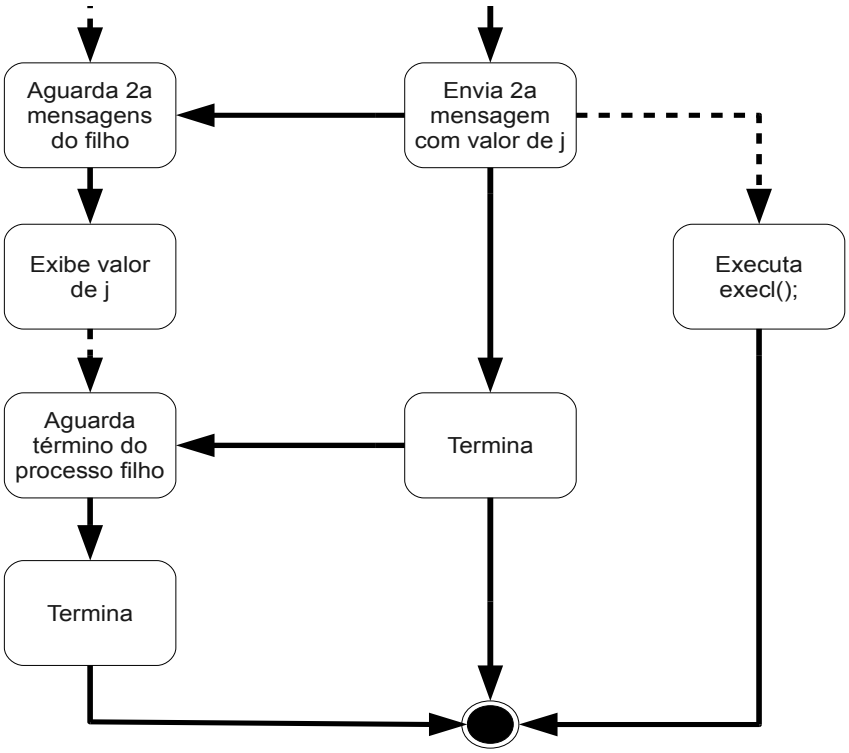
Comunicação inter-processos (através de *Pipes*).

Manipulação de sequência de execução inter-processos (através de *wait()*).

Término e retorno de processos (através de *exit()*).

De acordo com as especificações de trabalho o Prog1 tem o seu curso exemplificado de acordo com o fluxograma abaixo.





Saída do Console

```
Processo Corrente - 14247

Sou o processo Pai de PID 14247 e tenho o processo Filho de PID 14248
Mensagem enviada ao Filho: Olá processo Filho!
Sou o processo de PID 14248 e tenho o Processo Pai de PID 14247
Mensagem enviada pelo processo Pai - Olá processo Filho!
Prog1  Prog1.c    Prog2  Prog2.c    README.txt
Trab1_Descricao_2011_2.pdf    Trabalho Escrito.doc
1a Mensagem enviada pelo processo Filho - Olá processo Pai!
2a Mensagem enviada pelo processo Filho - j=10001
O Processo Filho terminou e o pai também se encerrará.
```

Resposta às perguntas no Programa

As respostas abaixo também estão contidas em forma de comentários no código do Prog1.

***** O que acontece após este comando?

O processo filho executa o comando execl que é criado pelo SO de forma independente. Isso ocorre pois o *ls* não é uma instância de Prog1, diferentemente dos processos filho e pai. Sendo assim, esse comando será executado pelo SO de forma dissociada dos demais, não respeitando nenhuma ordem de execução estabelecida no código do Prog1.

Após a execução de execl os processos voltam a ser executados normalmente.

***** O que pode acontecer se o comando “execl” falhar?

Como o *ls* é um processo independente do Prog1, quando o comando execl falha o Prog1 continua sua execução normalmente.