
**UNIVERSIDADE FEDERAL DO RIO DE JANEIRO – UFRJ**

INSTITUTO DE MATEMÁTICA – IM
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

# ESTUDO SOBRE SUBPROCESSOS COOPERATIVOS

Disciplina: Sistemas Operacionais
Professor: Thomé

| | |
|---|---|
| Júlio César Machado Bueno | 106033507 |
| Luiza Diniz e Castro | 107362705 |
| Roberta Santos Lopes | 107362886 |

# 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 é denominado “processo filho” ou subprocesso, e possui o mesmo código fonte e *uid* (*User Identifier*) que o seu processo criador (“processo pai”). Apesar disso, o processo filho possui atributos diferenciados, como o PID (*Process Identifier),* 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 PID do seu pai, exceto por rotinas de maior privilégio.

A criação de um subprocesso 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. Além disso, outro cuidado a ser tomado é com a finalização dos processos e seus subprocessos. No caso dos processos pai e filho estarem em execução, se o processo pai for finalizado, o filho também o será. Todavia, se o processo filho for finalizado, o processo pai continuará sua execução normalmente.

A primitiva fork() funciona da seguinte maneira:

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 formada por um grupo de funções: *execl()*, *execlp()*, *execle()*, *execv()*, *execvp(),* que permitem a execução de um programa externo ao processo corrente. Em outras palavras, permitem lançar, de forma independente, outro processo proveniente do sistema de arquivos. Não representa a criação de um processo efetivamente, mas sim a substituição do programa em execução.

Na chamada de um *exec*, existe uma sobreposição do segmento de instruções do processo que o chamou pelo segmento de instruções do processo que foi chamado (passado como argumento da função exec). Dessa forma, quando a chamada for bem sucedida, não existirá retorno do exec para o processo que estava em execução, já que o endereço de retorno desaparece quando ele morre. Porém, quando alguma das funções do grupo exec retorna, um erro ocorreu. Sendo assim, existirá um valor de retorno igual a -1.

***execl()***

A principal diferença dessa função para as demais é o número conhecido de parâmetros a serem passados. O primeiro informa ao processo em execução o caminho do processo a ser chamado. Os demais são passados como argumentos para o processo que foi chamado.

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 que todos os seus processos filhos sejam finalizados. Em seguida, o processo pai retorna sua execução do ponto onde parou. Um processo filho se torna um processo zumbi, quando seu processo pai termine antes dele. Dessa forma, ele continua a existir como entrada na tabela de processos do sistema operacional, mesmo que ele não esteja mais ativo para execução.

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;

     id = 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()***

Um processo pai pode estar esperando o término da execução de seu processo filho através do comando wait. O término da execução do processo filho ocorre através de uma chamada ao exit ou quando ele é abortado. Após a execução do comando exit, um status de saída é enviado ao sistema operacional e um sinal de SIGCHLD é enviado ao processo pai, indicando término com sucesso. Dessa forma, o pai volta a executar.
Lembrando que, caso um processo filho não esteja mais associado ao seu processo pai, ele se torna um processo zumbi.

Os parâmetros possíveis para a função exit são zero ou qualquer inteiro diferente de zero. Indicando, respectivamente, que o processo foi finalizado com sucesso ou que 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 o PID do processo corrente em formato de número inteiro.

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 o PID do processo pai do processo corrente em formato de número inteiro.

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

**Funcionamento**

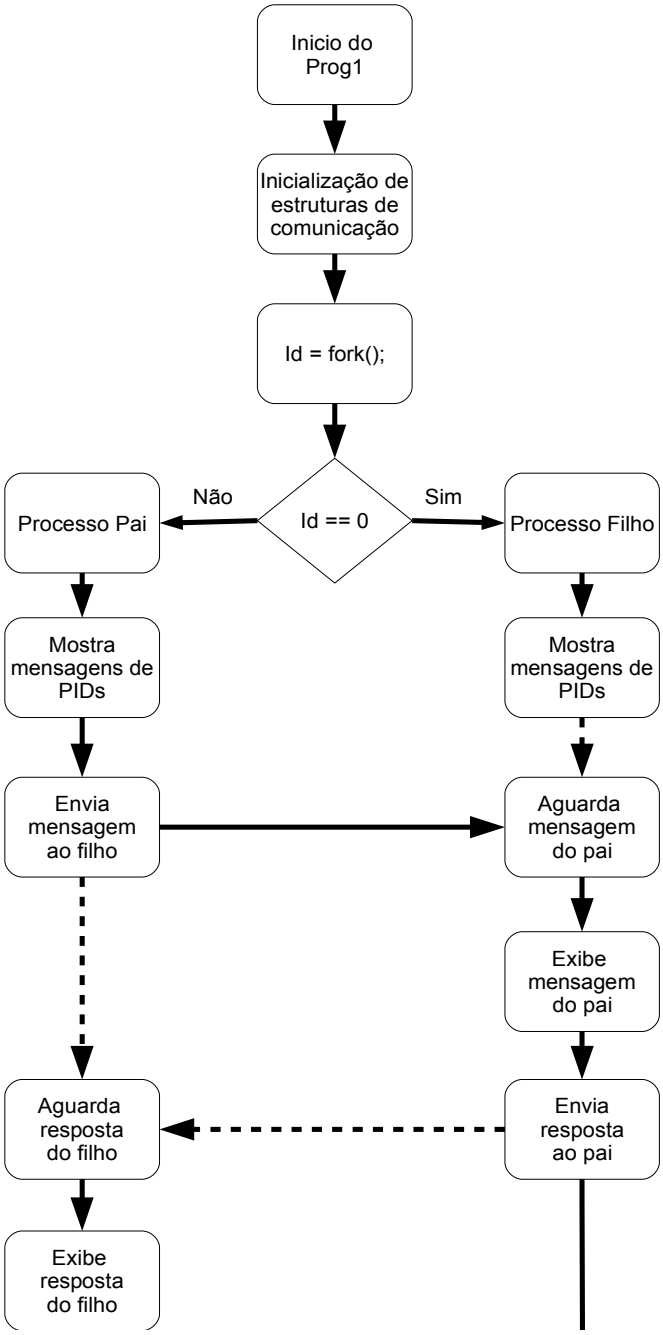
O Prog1.c exemplifica a utilização de comandos como *fork()* e *exec()*, associados ao uso de:

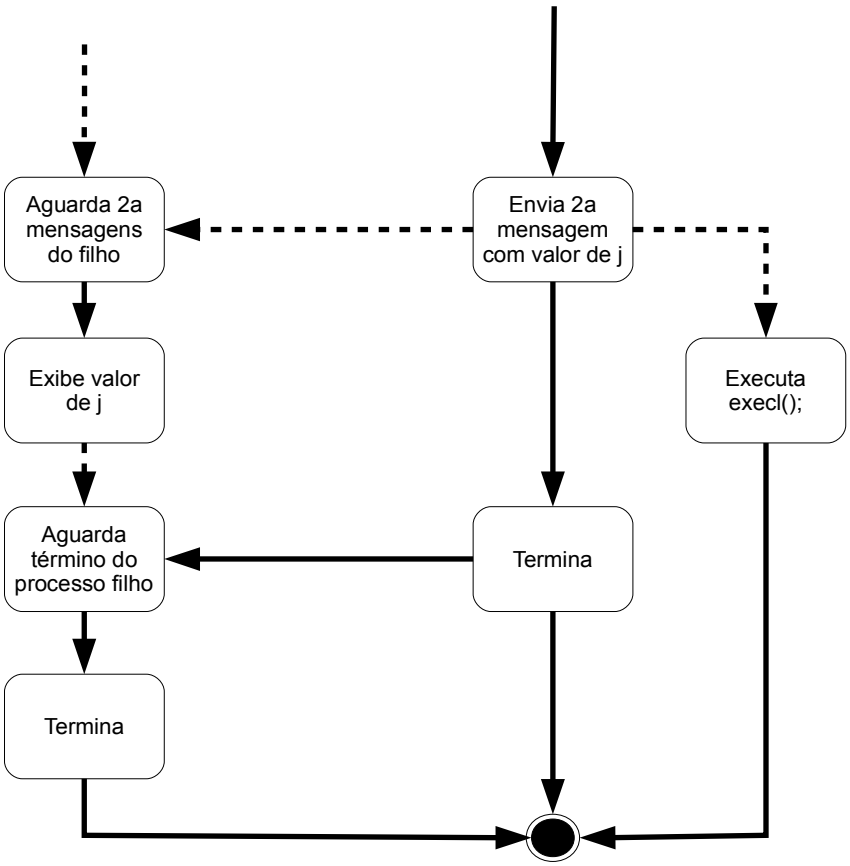
- Comunicação inter-processos (através de *POSIX*).
- Manipulação de sequência de execução inter-processos (através do *wait()*).
- Término e retorno de processos.

É importante ressaltar que o Prog1 pode ser levado a um estado de *Deadlock* pelo compartilhamento do recurso consumível da fila de mensagens. Isso poderá acontecer caso o processo filho seja executado antes do processo pai. Neste caso, o processo filho fica aguardando uma mensagem do processo pai, que nunca chegará, e o processo pai fica aguardando uma mensagem do processo filho.

A fim de evitar essa possibilidade, foi introduzido no código do processo pai uma chamada *wait()* para aguardar o término do processo filho e garantir a seqüência correta de envio de mensagens.

Para melhor entendimento da execução do programa descrito, foi desenvolvido o fluxograma abaixo:






**Saída do Console**

```
Processo Corrente - 14225


Sou o processo Pai de PID 14225 e tenho o processo Filho de PID 14226
Mensagem enviada ao Filho: Olá processo Filho!
Sou o processo de PID 14226 e tenho o Processo Pai de PID 14225
Mensagem enviada pelo pai - Olá processo Filho!
Prog1  Prog1.c   Prog2  Prog2.c   README.txt  Trab1_Descricao_2011_2.pdf
     Trabalho Escrito.doc  Trabalho Escrito PDF.pdf
1a Mensagem enviada pelo filho - Olá processo Pai!
2a Mensagem enviada pelo 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.

*execl("/Bin/ls", "ls", NULL);*

**1) 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. Quando a execução do *execl()* é bem sucedida, o segmento de instruções do processo filho (processo corrente) é substituído pelo segmento do processo que foi chamado pelo *execl()*, portanto o processo filho não continua em execução. Por sua vez, o processo pai identifica que o processo filho não existe mais e continua sua própria execução.

**2) O que pode acontecer se o comando "execl" falhar?**

O comando *execl()* retorna o valor -1, indicando que houve falha na sua execução. Neste caso, o processo filho (processo corrente) retorna sua execução normalmente.

# Prog2

**Funcionamento**

O Prog2.c mostra a criação seqüencial de diversos subprocessos a fim de gerar uma árvore encadeada de processos. Isso ocorre devido a uma iteração responsável por criar subprocessos a partir de outros subprocessos. O número dessas iterações é determinado pela variável m.

Além disso, o Prog2 permite que os processos pais armazenem os PIDs de seus processos filhos e apresentem estes PIDs no momento em que o processo pai inicia a espera pelo término dos mesmos.

Executando o Prog2 inicialmente com o valor m=0 e, modificando incrementalmente o parâmetro m para 1, em seguida para 2 e 3, podemos observar que árvore gerada não é balanceada e o nó associado ao Processo Pai Original possui um número de filhos igual a m+1.

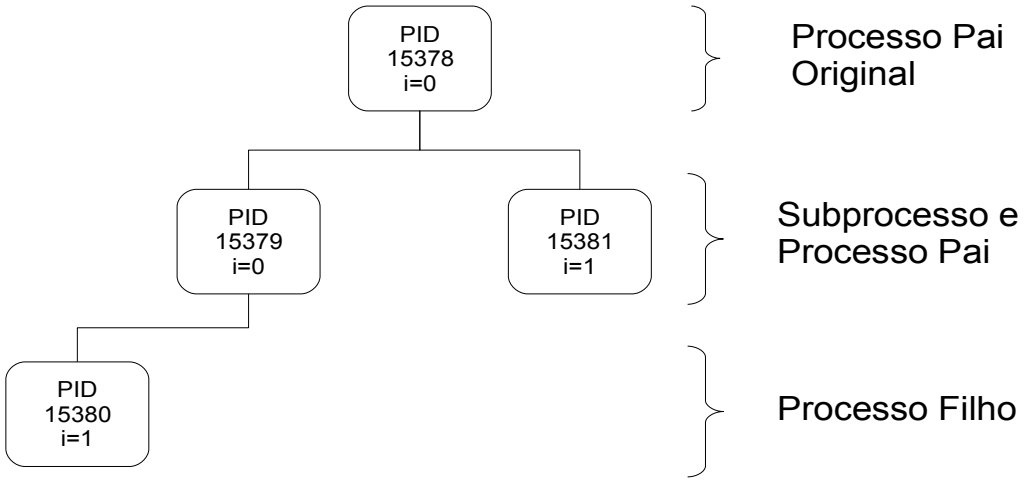
| m | Número de processos |
|---|---|
| 0 | 2 |
| 1 | 4 |
| 2 | 8 |
| 3 | 16 |

Podemos então deduzir empiricamente que o número total de processos gerados pelo Prog2 pode ser dado pela fórmula:

$$\text{nº de processos} = 2^{(m+1)}$$

Desta forma, podemos concluir que para o valor de m=4 teremos 32 processos e para m=10 2048 processos.

A seguir, um exemplo de uma árvore gerada para a variável configurada em m = 1.



**Saída do Console**

```
PID do processo corrente = 15378  |  d1 = 0  |  d2 = 1

PID do processo corrente = 15378  |  d1 = 0  |  d2 = 1  |  m = 1

PID do processo corrente = 15378  |  d1 = 1  |  d2 = 4  |  m = 1  Ramo if

PID do processo corrente = 15378  |  d1 = 1  |  d2 = 4  |  m = 1

PID do processo corrente = 15379  |  d1 = 0  |  d2 = 3  |  m = 1  Ramo else

PID do processo corrente = 15379  |  d1 = 0  |  d2 = 3  |  m = 1

PID do processo corrente = 15379  |  d1 = 2  |  d2 = 9  |  m = 1  Ramo if

PID do processo corrente = 15379

PID do processo corrente = 15379 e número de filhos = 1

O processo de PID = 15379 está esperando os seguintes filhos: 15380

PID do processo corrente = 15378  |  d1 = 3  |  d2 = 13  |  m = 1  Ramo if

PID do processo corrente = 15378

PID do processo corrente = 15380  |  d1 = -1  |  d2 = 12  |  m = 1  Ramo else

PID do processo corrente = 15378 e número de filhos = 2

O processo de PID = 15378 está esperando os seguintes filhos: 15379 15381

PID do processo corrente = 15381  |  d1 = 0  |  d2 = 12  |  m = 1  Ramo else

PID do processo corrente = 15378 e número de filhos = 2

O processo de PID = 15378 está esperando os seguintes filhos: 15379 15381
```

**Resposta às perguntas no Programa**

Todas as perguntas foram respondidas como comentários no próprio código do Prog2.

**Verifique e apresente suas conclusões sobre a ordem em que os processos são ativados e a ordem em que ganham a CPU.**

Apesar dos processos serem criados a partir de um único processo pai original, a ordem em que eles ganham a CPU e outros recursos depende exclusivamente do sistema operacional. Isso significa que após a criação de um processo filho, a partir da função *fork()*, tanto o processo pai quanto o processo filho podem ganhar a CPU, não havendo prioridades. Porém, a ordem de criação é respeitada e assim o valor do PID do processo pai sempre será menor que o do seu processo filho. A utilização do comando *wait()*, garante, durante a execução do processo pai, que ele aguarde pelo término de todos seus processos filhos.

# Prog3

**Especificação geral**

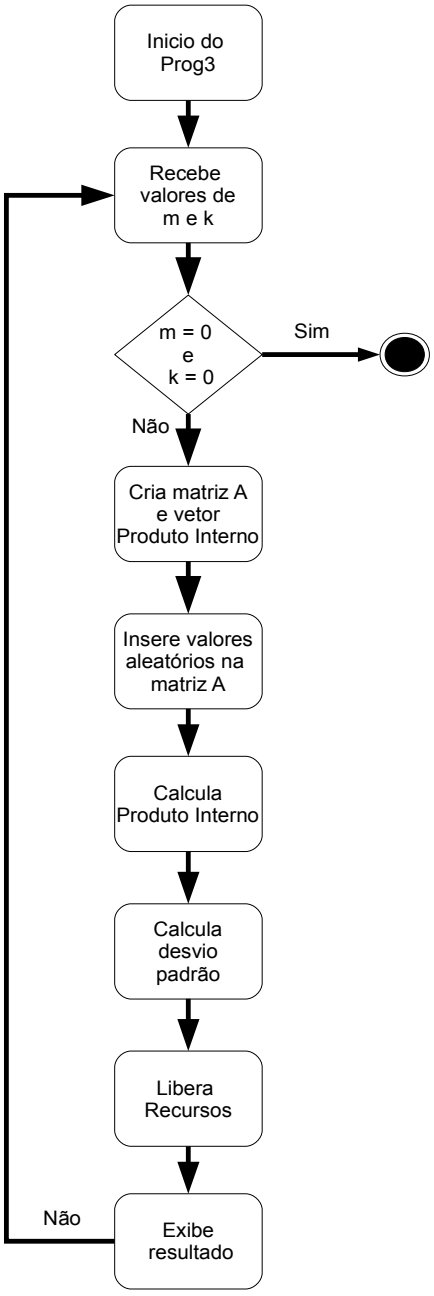
O Prog3 foi construído atendendo às especificações de processamento do Produto Interno de uma linha da matriz de dimensões m x k, definidos pelo usuário. Esse produto interno é dado pela fórmula:

$$PI_i = \sum_{j=1}^{k} A_{i,j} * A_{j,i}$$

**Versão a**

**Estrutura e Análise**

A versão 1 do Prog3 (ou Prog3a) segue o requerimento de realizar todo o processamento de forma seqüencial (sem nenhum paralelismo explícito). A seguir, temos um fluxograma que exemplifica o fluxo de processamento:



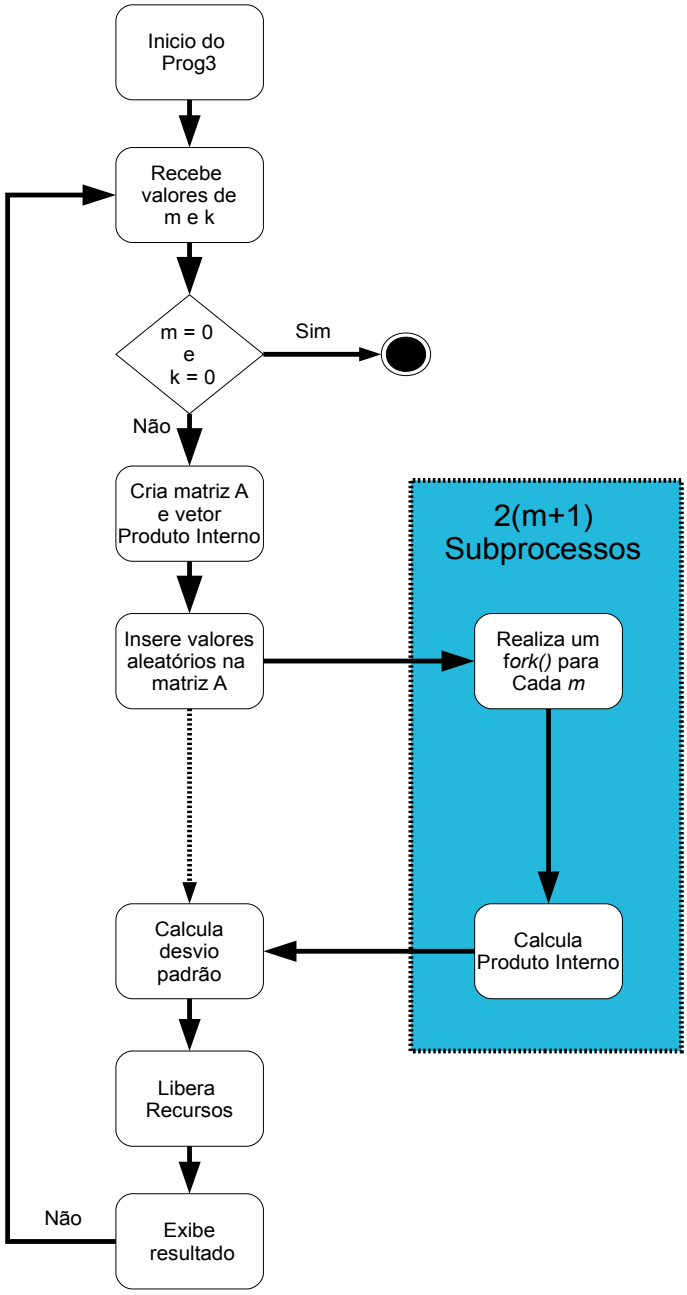
Dado que o Prog3a não possui nenhuma técnica de processamento paralelo, podemos esperar que o seu desempenho seja caracterizado pela sensibilidade aos valores de *m* e *k*, portanto a fim de estabelecermos uma notação de comparação, podemos dizer que o algoritmo possui a complexidade de $O(n^2)$.

**Versão b**

**Estrutura e Análise**

A versão b do Prog3 tem como base a versão do Prog3a, onde é feito uma tentativa de melhoria de performance e aproveitamento de recursos fazendo com que para cada cálculo do Produto Interno, seja criado um processo a fim de calcular a determinada linha *m* da matriz A gerada.

A seguir o fluxograma do programa apresentado:




Como o Prog3b apresenta um enorme número de subprocessos, podemos observar que esta tática para melhorar a performance da implementação, na verdade, piora os resultados obtidos uma vez que o seguinte cenário se observa:

- Há um grande *overhead* por parte do Sistema Operacional na criação deste subprocessos, uma vez que o SO precisa alocar recursos como *time-slice*, sendo assim prejudicial a performance da tarefa.

- Como há muitos processos concorrentes, há também um grande número de trocas de contexto e isso impacta diretamente na performance do sistema.

Desta forma podemos concluir que a implementação é altamente sensível a aumentos no valor de *m* e por fim não se mostrou proveitosa, necessitanto assim de uma outra abordagem.

**Versão c**

**Estrutura de Análise**

Conforme pedido, a Versão c também chamada de Prog3c apresenta uma solução alternativa à Versão b, também visando uma melhoria na performance e no aproveitamento dos recursos computacionais disponíveis.

A abordagem optada foi criar apenas dois processos (Processo Pai e Processo Filho), dos quais cada um, individualmente, será responsável por um segmento da matriz A. Ou seja, a primeira metade dos Produtos Internos seria calculado pelo Processo Pai, e por consequência a segunda metade de A é calculada pelo Processo Filho.

Esta abordagem se mostra bastante eficiente em CPUs com múltiplos núcleos e possui as seguintes vantagens:

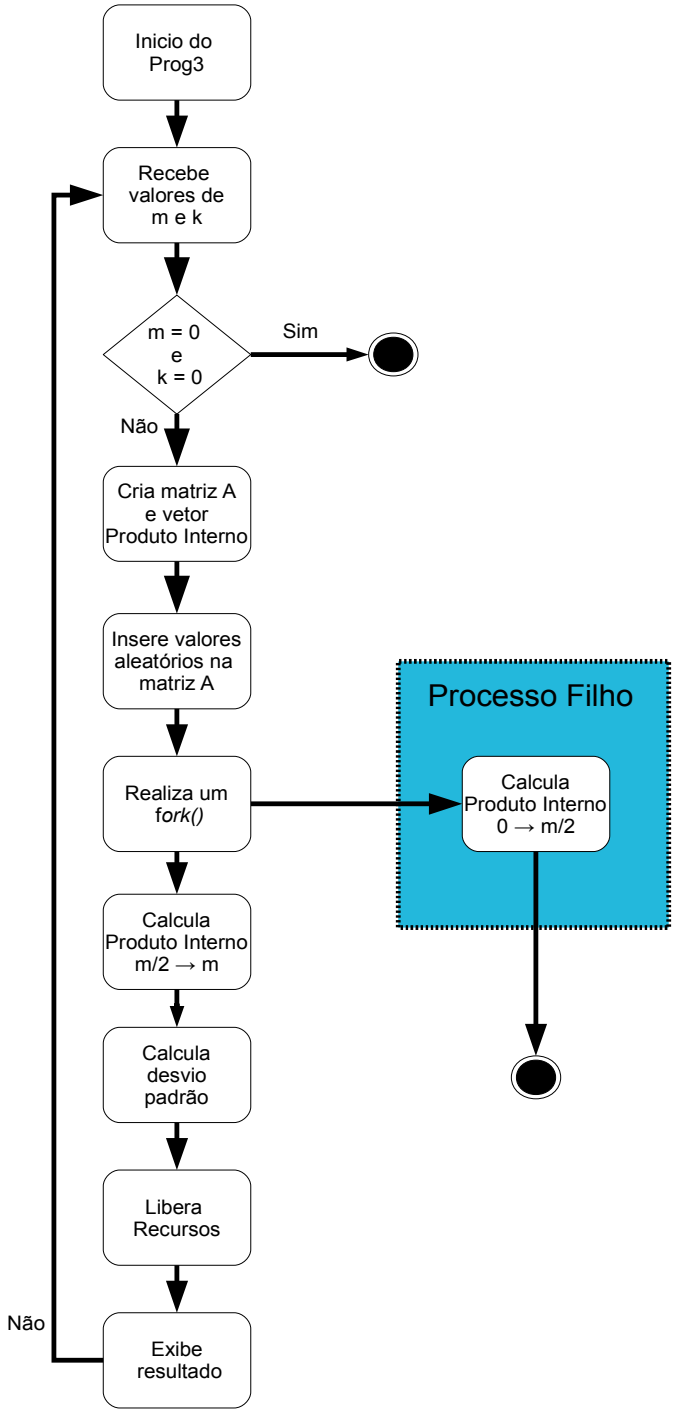
- Não gera *overhead* ao SO uma vez que só há uma criação de processos.
- Não há um aumento significativo de trocas de contexto entre os processos.
- Dado que a CPU possui múltiplos núcleos, não há uma concorrência de fato entre os subprocessos.

Desta forma obtemos uma melhoria em relação ao programa original Prog3a sem os efeitos negativos da implementação de Prog3b.

***Shared Memory***

Tanto na implementação da Versão b quanto na Versão c, foram utilizadas técnicas de compartilhamento de memória (*shared memory*) para que os valores calculados em cada subprocesso específico, fossem compartilhados com os demais, obtendo assim a cooperação desejada. Esta técnica possui a vantagem de fazer com que o acesso aos recursos compartilhados, neste caso, valores de variáveis, seja de forma muito eficiente e direta. É importante lembrar entretanto, que este técnica utilizado possui diversas implementações e restrições dependentes diretamente do SO em questão, e que portanto são difíceis de serem contornadas em todos os casos.

A seguir, temos o fluxo onde o Processo Filho possui sua execução independente do Processo Pai e tem como única função o cálculo da parcela de *m* linhas de A.




**Maquinário utilizado no desenvolvimento e testes**

As implementações e testes foram realizados nas máquinas do LCI-UFRJ, que apresentam a seguinte configuração:

| |
|---|
| Intel Core 2 Duo E7300 2,66GHz<br>2 GB Memória RAM<br>HD 40 GB<br>Sistema Operacional: Gentoo World |

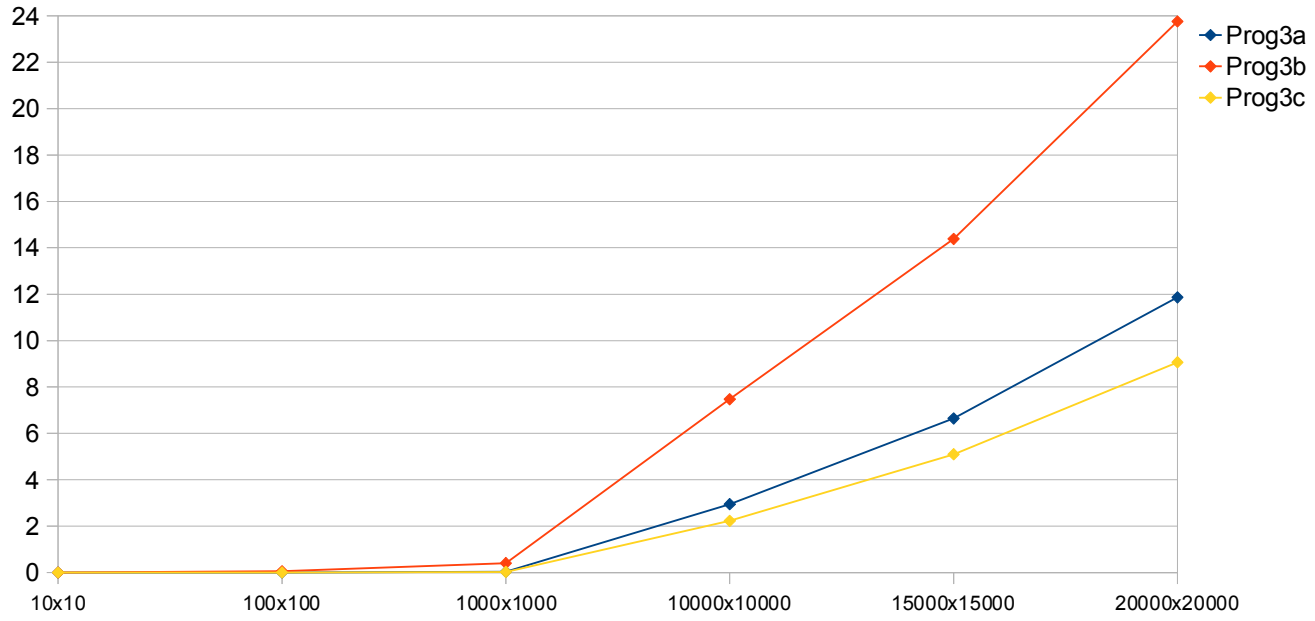
A seguir, temos os valores dos testes de performance realizados comparativamente entre os processos. Para fim de comparações equivalentes, utilizamos valores de *m* e *k* iguais e seus incrementos de forma a deixar claro a natureza de cada programa.

Os resultados obtidos através da execução dos programas estão listados na tabela abaixo:

| **Dimensão da Matriz** *m* x *k* | **Versão 1** Tempo (segundos) | **Versão 2** Tempo (segundos) | **Versão 3** Tempo (segundos) |
|---|---|---|---|
| 10 X 10 | 0,000 | 0,001 | 0,000 |
| 50 X 50 | 0,001 | 0,024 | 0,004 |
| 100 X 100 | 0,004 | 0,053 | 0,004 |
| 500 X 500 | 0,011 | 0,198 | 0,009 |
| 1000 X 1000 | 0,033 | 0,408 | 0,028 |
| 5000 X 5000 | 0,747 | 2,755 | 0,574 |
| 10000 X 10000 | 2,947 | 7,472 | 2,227 |
| 15000 X 15000 | 6,644 | 14,382 | 5,099 |
| 20000 X 20000 | 11,864 | 23,761 | 9,068 |

**Comparação de desempenho entre as versões**

O gráfico a seguir mostra os mesmo valores da tabela anterior para que possa ser visível a diferença de performance entre cada programa, assim como o seu comportamento diante da variação da entrada.




É importante perceber que para valores de *m* e *k* menores que 1000 a diferença de performance entre os programas é irrelevante.