UNIVERSIDADE FEDERAL DO RIO DE JANEIRO – UFRJ

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

ESTUDO SOBRE THREADS

Disciplina: Sistemas Operacionais
Professor: Thomé

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

Sumário

Estudo de Comandos

pthread_create()

A função *pthread_create()* é uma função que possibilita a criação de uma nova *thread* a partir de parâmetros configurados na inicialização das estruturas *pthread_attr_t* e *pthread_t* utilizando a implementação POSIX *thread* presente no cabeçalho *pthread.h*.

Então, como o primeiro passo para a criação da thread, devemos criar a variável do tipo *pthread_t*, em seguida configurar os parâmetros que definem a criação e execução da *thread* a ser criada. A *thread* criada em C executa uma função definida no código. Dentro deste código, variáveis locais são acessíveis apenas a *thread* com exceção dos recursos compartilhados. Ao contrário da criação de um subprocesso, onde o subprocesso executa todo o código do processo pai posterior a parte em que foi chamada a criação do subprocesso, a *thread* executa somente o código definido pela função especificada.

Para a execução da função *pthread_create* é necessário a passagem dos argumentos:

- Variável do tipo *pthread_t*
- Variável do tipo *pthread_attr_t* (*NULL* se não há execução de *pthread_attr_init()*)
- Função a ser executada como um *thread* distinta
- Ponteiro para a variável passada como argumento para a função

A função *pthread_create()* retorna o valor 0 (zero) se a *thread* foi criada com sucesso. Caso contrário, retorna o número do erro associado a falha da criação da *thread*. A seguir um exemplo de uso da função *pthread_create():*

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

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   exit(0);
}
```

pthread_join()

Ao executar a função *pthread_join()*, uma *thread* fica suspensa enquanto outra *thread*, passada como parâmetro não é terminada ou cancelada. Ou seja, a *thread* "pai" fica aguardando o término ou o cancelamento da *thread* "filho". *pthread_join()* tem então a mesma funcionalidade que *wait()* sendo então responsável pela sincronização entre as *threads*.

pthread_join() necessita dos seguintes de um argumentos:

- *pthread_t* utilizado na função *pthread_create()*.
- Valor de retorno da t*hread* (*thread_return*), Usualmente *NULL*.

A seguir um exemplo da execução de *pthread_join()*:

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

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_exit()

A função *pthread_exit()* termina a execução da *thread* corrente. Caso a *thread* "pai" tenha executado o comando *pthread_join()* para esta *thread* "filho" em questão, *pthread_exit()* pode retornar um valor que será disponível à *thread* "pai". Como padrão, retorna-se o valor *NULL*.

O valores das variáveis locais são liberados na chamada de *pthread_exit()*. Já recursos compartilhados continuam disponíveis no sistema mesmo que a sua *thread* foi terminada.

A seguir o exemplo do comnado *pthread_exit()*:

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

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Cria de fato a thread
   retorno = pthread_create(&thread, NULL, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_attr_init()

Inicializa atributos padrões para uma *thread* que será criada. É importante para a configuração dos atributos de execução da *thread* de forma flexível.

pthread_attr_init() recebe como um parâmtro um endereço de memória para uma variável do tipo *pthread_attr_t*. É possível iniciar uma nova *thread* com *pthread_create* sem inicializar atributos para a mesma. Nesse caso, o SO configura valores padrões para a nova *thread*. A seguir uma tabela que informa os valores padrões do SO para a criação de uma *thread*.

Atributo	Padrão
Detach state	PTHREAD_CREATE_JOINABLE
Scope	PTHREAD_SCOPE_PROCESS
Inherit scheduler	PTHREAD_INHERIT_SCHED
Scheduling policy	SHCED_OHTER
Scheduling order	0
Guard size	4096 bytes
Stack address	#Address
Stak size	#Address size in bytes

No exemplo a seguir, como não há modificação dos atributos, a execução de *pthread_create()* somente com o atributo inicializado como parâmetro é a mesma que execução com o argumento *NULL* como parâmetro:

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

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("Mensagem exibida pela Thread criada!");
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   pthread_attr_t atributosThread;
   int retorno;

   //Inicializa os atributos da thread
   pthread_attr_init(&atributosThread);

   //Cria de fato a thread
   retorno = pthread_create(&thread, &atributosThreads, print, NULL);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

pthread_attr_setscope()

Configura o atributo *SCOPE* da *thread* a ser iniciada. Desta forma configura o nível de execução da nova *thread*, se será no nível *kernel* ou no nível usuário. *pthread_attr_setscope()* faz parte da família de funções *pthread_attr_set*()* que configura os parâmetros da *thread* a ser criada. Ela recebe os seguintes itens como parâmetro.

- Endereço de memória para a variável do tipo *thread_attr_t*
- Constante do SO que definem o nível de execução podendo ser:
  - PTHREAD_SCOPE_PROCESS – execução em nível usuário, padrão do SO.
  - PTHREAD_SCOPE_SYSTEM – execução em nível *kernel*.

Configuração do atributo *SCOPE* através de pthread_attr_setscope*()* para a criação de uma *thread* de nível *kernel*:

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

//Inclui o cabeçalho para a criação da thread
#include <pthread.h>

int print(void *arg){
   printf("%s", arg);
   pthread_exit(NULL);
}

int main(void){
   //Cria os valores da thread
   pthread_t thread;
   pthread_attr_t atributosThread;
   int retorno;

   //Define o argumento a ser passado
   char string[256] = "Mensagem exibida pela Thread criada!";

   //Inicializa os atributos da thread
   pthread_attr_init(&atributosThread);

   //Configura a thread como thread do nível kernel
   pthread_attr_setscope&atributosThread, PTHREAD_SCOPE_SYSTEM);

   //Cria de fato a thread
   retorno = pthread_create(&thread, &atributosThreads, print, (void*) &string);

   //Verifica se houve um erro ao criar a thread
   if(retorno != 0){
      printf("Erro número %i ao criar a thread", retorno);
      exit(1);
   }

   //Aguarda o retorno da thread criada
   pthread_join(&thread, NULL);

   exit(0);
}
```

Prog1

Funcionamento

O Prog1 utiliza a criação de *threads* para o cálculo do Produto Interno descrito na especificação do trabalho utilizando os comandos pthread_create*(),* pthread_join*(), pthread_exit()* associados ao uso de:

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

Configuração da máquina utilizada no desenvolvimento e testes

As implementações e testes foram realizados nas máquinas do LCI-UFRJ onde, de acordo com os testes especificados, 1 dos núcleos foi habilitado ou não e que apresentam a seguinte configuração:

Intel Core 2 Duo E7300 2,66GHz (2 núcleos) 2 GB Memória RAM HD 40 GB Sistema Operacional: Linux Gentoo World

Comparação de desempenho entre as versões

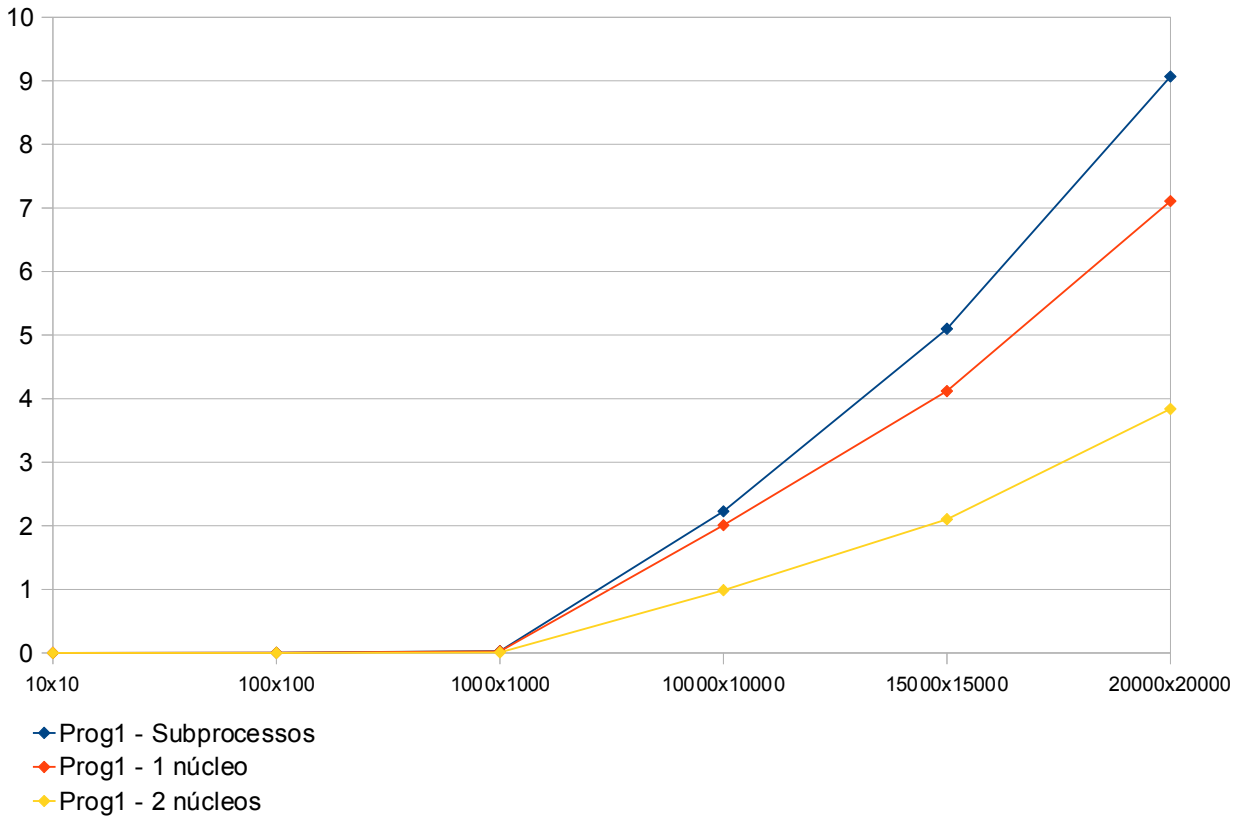
A seguir, temos os valores dos testes de performance realizados comparativamente entre os processos. Para fim de comparações equivalentes, utilizamos valores de *m* e *k* iguais e seus incrementos de forma a deixar claro a natureza de cada versão do programa. É importante perceber que para valores de *m* e *k* menores que 1000 a diferença de performance entre os programas é irrelevante. Foi utilizado a versão 3 do Prog1 do trabalho anterior como base comparativa, dado que ela apresenta a melhor implementação feita que utiliza subprocessos.

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

Dimensão da Matriz *m* x *k*	**Versão 1** Subprocessos Tempo (segundos)	**Versão 2** 1 núcleo habilitado Tempo (segundos)	**Versão 3** 2 núcleos habilitados Tempo (segundos)
10 X 10	0,000	0,000	0,000
50 X 50	0,004	0,004	0,000
100 X 100	0,004	0,004	0,001
500 X 500	0,009	0,007	0,003
1000 X 1000	0,028	0,025	0,009
5000 X 5000	0,574	0,414	0,314
10000 X 10000	2,227	2,009	0,987
15000 X 15000	5,099	4,119	2,103
20000 X 20000	9,068	7,108	3,837

Análise dos resultados e conclusões

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 de *m* e *k*.



Desta forma podemos então concluir que o uso de *threads* se mostra mais eficiente que o uso de subprocessos em máquinas com 1 núcleo e mais claramente em máquinas que possuem 2 ou mais núcleos. Isso comprova o que se é pensado em teoria com relação a criação de *threads* ser menos onerosa que a de subprocessos, gerando menos *overhead* já que a estrutura de PCB não é copiada a cada nova criação. Também fica mais claro que a alternância entre *threads* é mais rápida que entre subprocessos.

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

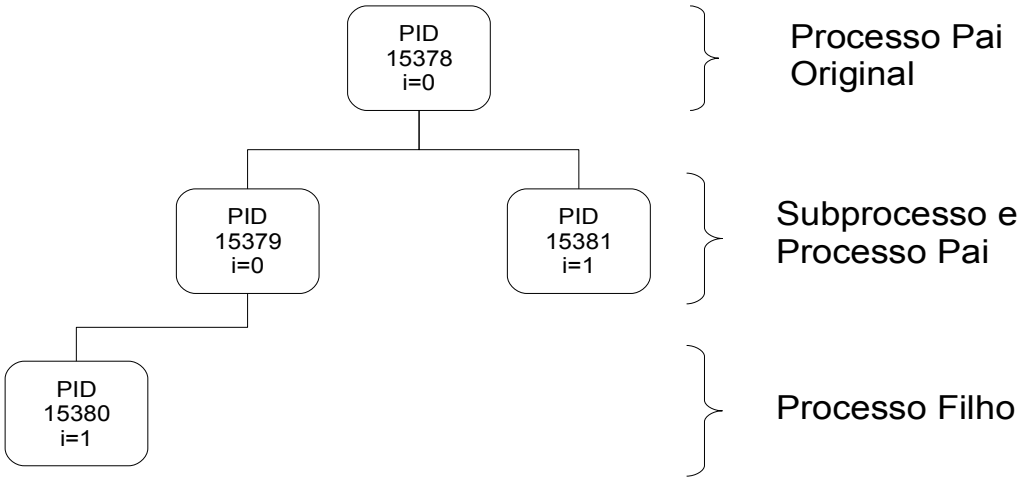
1	4
2	8
3	16

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

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

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

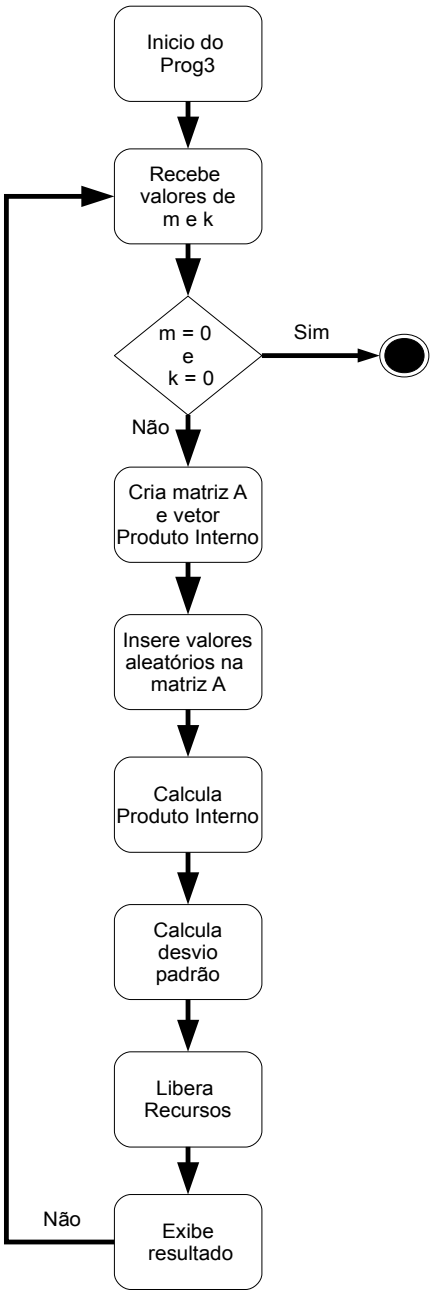
O Prog3 foi construído atendendo às especificações de processamento do Produto Interno de uma linha da matriz de dimensões m x k, definidos pelo usuário. Esse produto interno é dado pela fórmula:

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

Versão 1

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:



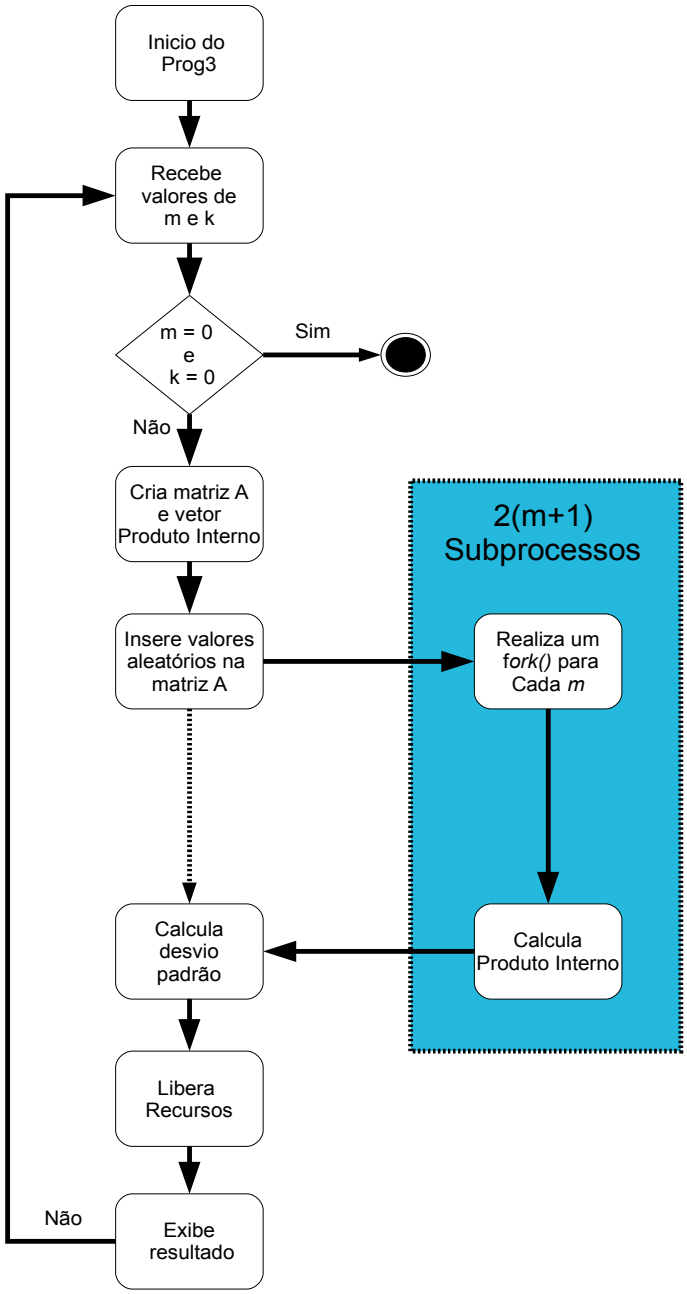
Dado que o Prog3a não possui nenhuma técnica de processamento paralelo, podemos esperar que o seu desempenho seja caracterizado pela sensibilidade aos valores de *m* e *k. P*ortanto, a fim de estabelecermos uma notação de comparação, podemos dizer que o algoritmo possui a complexidade de $O(n^2)$.

Versão 2

Estrutura e Análise

A versão 2 do Prog3 (ou Prog3b) tem como base a versão do Prog3a. Nela, é feita uma tentativa de melhoria de performance e aproveitamento de recursos. Para isso, a cada cálculo do Produto Interno foi criado um processo para calcular uma determinada linha *m* da matriz A gerada.

A seguir o fluxograma do programa apresentado:



Devido ao elevado número de subprocessos gerados no Prog3b, observou-se que a tática adotada para melhorar a performance da implementação anterior (Prog3a), na realidade piora os resultados obtidos. Tal fato ocorre devido aos seguintes fatores:

- O Sistema Operacional precisa alocar recursos, como o time-slice, para todos os subprocessos criados. Isso prejudica a performance da tarefa, devido ao elevado overhead do SO. Em outras palavras, o Sistema Operacional gasta mais tempo nas rotinas de controle dos subprocessos criados do que na execução da tarefa.

- Como há muitos processos concorrentes, há também um grande número de trocas de contexto.

Desta forma podemos concluir que a implementação é altamente sensível a aumentos no valor de *m*. Sendo assim, a versão Prog3b não se mostrou proveitosa, necessitanto de uma outra abordagem, descrita a seguir.

Versão 3

Estrutura e Análise

Conforme solicitado, a versão 3 do Prog3 (ou Prog3c) apresenta uma solução alternativa à versão 2, também visando uma melhoria na performance e no aproveitamento dos recursos computacionais disponíveis.

Sabendo-se que a carga de teste foi realizada em uma máquina com CPU de dois núcleos, a abordagem adotada gerou apenas dois processos: Processo Pai e Processo Filho. Cada um, individualmente, responsável por um segmento da matriz A. Ou seja, a primeira metade dos Produtos Internos é calculada pelo Processo Pai e a segunda metade pelo Processo Filho.

Caso a CPU tivesse mais núcleos, a divisão poderia ser feita em mais processos. Logo, para esta implementação, a quantidade de processos criados é igual a quantidade de núcleos da CPU.

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

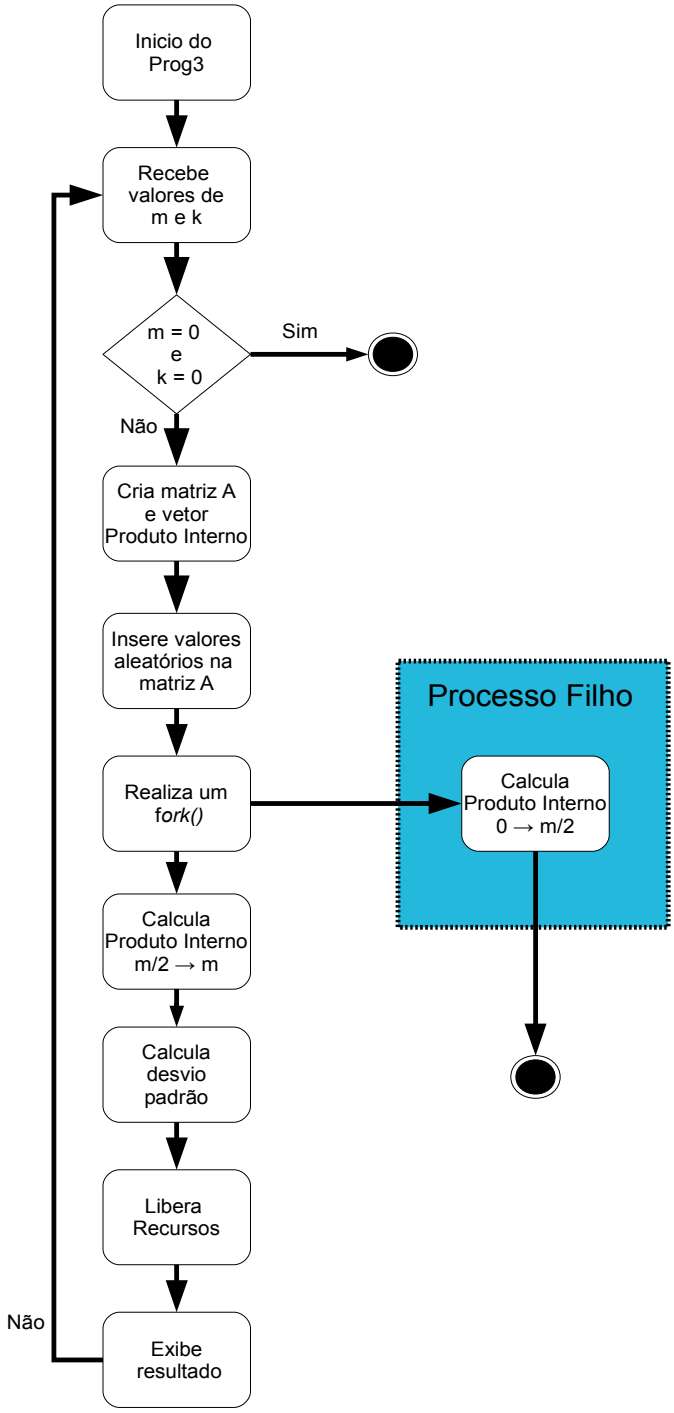
- Não gera *overhead* ao SO uma vez que só há a criação de um único processo (no caso específico da CPU com dois núcleos - Prog3c).
- Não há um aumento significativo de trocas de contexto entre processos.
- Não há concorrência entre os subprocessos.

Desta forma, obtém-se uma melhoria em relação ao programa Prog3a, sem os efeitos negativos da implementação de Prog3b.

Shared Memory

Tanto na implementação da Versão 2 quanto na Versão 3, foram utilizadas técnicas de compartilhamento de memória (*shared memory*) para que os valores calculados em cada subprocesso fossem compartilhados com os demais, obtendo-se assim a cooperação desejada. O *shared memory* possui a vantagem de fazer com que o acesso aos recursos compartilhados (neste caso, valores de variáveis) seja realizado de forma eficiente e direta. É importante lembrar, entretanto, que existem diversas implementações e restrições que dependem diretamente do SO que são difíceis de ser contornadas.

A seguir, tem-se 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.



Apêndice

A seguir disponibilizamos os códigos fonte de todos os programas utilizados no estudo.

Prog1.c

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>
```

```
#include <sys/wait.h>
#include <sys/stat.h>
#include <mqueue.h>

#define MSG_SIZE 256
#define MAX_MSG_SIZE 10000
#define MSGQOBJ_NAME "/IPC_CHANNEL1"

int main(void){
      int   status, id, j;

      //Cria as estruturas de IPC e o canal de comunicação POSIX
      mqd_t msgq_id;
      unsigned int sender;
      char msgcontent[MSG_SIZE];
      char msgcontentRCV[MAX_MSG_SIZE];
      struct mq_attr msgq_attr;
      int msgsz;

      msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR | O_CREAT | O_EXCL, S_IRWXU |
S_IRWXG, NULL);
      if (msgq_id < 0) {
            msgq_id = mq_open(MSGQOBJ_NAME, O_RDWR);
            if(msgq_id < 0){
                  perror("mq_open");
                  exit(1);
            }
      }

      //Insira um comando para pegar o PID do processo corrente e mostre na tela
da console.
```

```
    printf("Processo Corrente - %i\n\n", getpid());


    id = fork();


    if (id != 0){
        //Faça com que o processo pai execute este trecho de código
        //Mostre na console o PID do processo pai e do processo filho
        printf("Sou o processo Pai de PID %i e tenho o processo Filho de PID
%i\n", getpid(), id);



        //Monte uma mensagem e a envie para o processo filho
        strcpy(msgcontent, "Olá processo Filho!");
        mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, 10);



        //Mostre na tela o texto da mensagem enviada
        printf("Mensagem enviada ao Filho: %s\n", msgcontent);



        //Aguarde a resposta do processo filho
        wait(&status);
        msgsz = mq_receive(msgq_id, msgcontentRCV, MAX_MSG_SIZE, &sender);
        if (msgsz == -1) {
            perror("mq_receive()");
            exit(1);
        }

        //Mostre na tela o texto recebido do processo filho
        printf("\n1a Mensagem enviada pelo filho - %s\n", msgcontentRCV);
```

```
            //Aguarde mensagem do filho e mostre o texto recebido
            msgsz = mq_receive(msgq_id, msgcontentRCV, MAX_MSG_SIZE, &sender);
            if (msgsz == -1) {
                  perror("mq_receive()");
                  exit(1);
            }
            printf("2a Mensagem enviada pelo filho - %s\n", msgcontentRCV);



            //Aguarde o término do processo filho
            wait(&status);
            mq_close(msgq_id);



            //Informe na tela que o filho terminou e que o processo pai também
vai encerrar
            printf("O Processo Filho terminou e o pai também se encerrará.\n");
            exit(0);
      }else{
            //Faça com que o processo filho execute este trecho de código
            //Mostre na tela o PID do processo corrente e do processo pai
            printf("Sou o processo de PID %i e tenho o Processo Pai de PID %i\n",
getpid(), getppid());



            //Aguarde a mensagem do processo pai e ao receber mostre o texto na
tela
            mq_getattr(msgq_id, &msgq_attr);
            msgsz = mq_receive(msgq_id, msgcontentRCV, MAX_MSG_SIZE, &sender);
            if (msgsz == -1) {
                  perror("mq_receive()");
```

```
            exit(1);
        }
        printf("Mensagem enviada pelo pai - %s\n", msgcontentRCV);


        //Envie uma mensagem resposta ao pai
        strcpy(msgcontent, "Olá processo Pai!");
        mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, 10);


        //Execute o comando "for" abaixo
        for (j = 0; j <= 10000; j++);


        //Envie mensagem ao processo pai com o valor final de "j"
        sprintf(msgcontent, "j=%i", j);
        mq_send(msgq_id, msgcontent, strlen(msgcontent)+1, 10);


        //Execute o comando abaixo e responda às perguntas
        printf("\n");
        execl("/bin/ls", "ls", NULL);

        /*
```

==========

Responda: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.

Responda: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.

```
          ------------------------------------------------------------------------
----------------------------------------------

          */



          mq_close(msgq_id);

          exit(0);

     }


}
```

Prog2.c

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <sys/wait.h>

#include <unistd.h>


#define m 1


int i, j, k, id, d1, d2, status;

int nFilhos = 0;

int meuPID = -1;

char pid[16];

char pid_list[512];


int main(void) {


            //inicialize as variáveis d1 e d2 com valores distintos;

            d1 = 0;

            d2 = 1;




            printf("\n");

            //mostre o PID do processo corrente e os valores de d1 e d2 na
tela da console

            printf("PID do processo corrente = %i    |    d1 = %i    |    d2 =
%i\n\n", getpid(), d1, d2);




            /*
```

```
================================================================================
=====================================

            Responda: Quais processos executarão este trecho do código?

                  Somente o processo pai original ("raiz da árvore de
processos") executará este trecho.

            ------------------------------------------------------------------
-------------------------------------------------

            */



            j = 0;



            for (i = 0; i <= m; i++){

               //mostre na tela da console, a cada passagem, os seguintes
valores: PID do processo corrente; “i”, “d1”, “d2” e “m”

               printf("PID do processo corrente = %i     |     d1 = %i     |
d2 = %i    |   m = %i\n\n", getpid(), d1, d2, m,i);



               /*



================================================================================
==================================

               Responda: Quais processos executam este trecho do código?

               Todos os processos executam este trecho já que neste trecho o
fork() ainda não foi realizado.

               ------------------------------------------------------------------
-------------------------------------------------

               */

               id = fork();



               //Verifica validade da atualização

               if(meuPID != getpid()){

                     meuPID = getpid();

                     nFilhos = 0;

                     //Limpa o meu vetor
```

```
                    memset(pid_list, 0, sizeof(pid_list));

               }



               if (id){

                    //altere os valores de d1 e d2 de diferentes maneiras como
exemplificado abaixo

                    d1 = d1 + i + 1;

                    d2 = d2 + d1 * 3;


                    //  mostre  na  tela  da  console,  a  cada  passagem,  os
seguintes valores:

                    //  PID  do  processo  corrente,  "i",  "d1",  "d2",  "m"  e
informe estar no ramo “then” do “if”

                    printf("PID do processo corrente = %i    |    d1 = %i    |
d2 = %i    |   m = %i  Ramo if\n\n", getpid(), d1, d2, m, i,j);


                    //Contabilizando o número de filhos

                    nFilhos++;


                    //Armazenando os processos filhos criados

                    sprintf(pid, "%i ", id);

                    strcat(pid_list, pid);



                    /*
```

==========

Responda: Quais processos executam este trecho do código?

Todos os processos que possuem algum subprocesso associado a ele, ou seja, todos os processos pai.

```
                    */

                }else{

                    //altere os valores de d1 e d2 de diferentes maneiras e
também diferente do usado no trecho "then"

                    d1 = d1 - i;

                    d2 = (d2 - d1) * 3;



                    //execute o comando de atualização de "j" abaixo

                    j = i + 1;



                    //mostre na tela da console, a cada passagem, os seguintes
valores: PID do processo corrente; "i", "d1", "d2", "m" e informe
                estar no ramo "else" do "if"

                    printf("PID do processo corrente = %i    |    d1 = %i    |
d2 = %i    |   m = %i  Ramo else\n\n", getpid(), d1, d2, m, i,j);



                    /*


================================================================================
==================================

                    Responda: Quais processos executam este trecho do código?

                    Este trecho de código é executado por todos os processos
que possuem um pai, ou seja, por todos os processos exceto

                    o processo pai original.

                    ------------------------------------------------------------
-----------------------------------------------------

                    */

                }



            }

            /*


================================================================================
==================================

            Responda: Quais processos executam este trecho do código?
```

```
            Todos os processos executam este trecho.

            ------------------------------------------------------------------
-----------------------------------------

            */



            if (id != 0) {

               //mostre na console o PID do processo corrente e verifique
quais processos executam este trecho do código

               printf("PID do processo corrente = %i\n", getpid());



               for (i = j; i <= m; i++){

                    /*


==========================================================================
==================================

                    Responda: Explique o papel da variável "j"

                    A variável "j" contabiliza a altura da árvore genealógica
a partir dos processos que são filho e pai ao mesmo tempo.

                    Desta forma, o processo pai original e os procesos folhas
(processos filhos que não possuem filhos) não são considerados.



                    Responda: Verifique se o comando "for" está correto de
forma que cada processo pai aguarde pelo término de todos seus

                    processos filhos

                    O comando for(i = j; i == m; i++) está incorreto porque
ele se refere somente ao último pai.

                    O correto seria for(i = j; i<=m; i++), garantindo que cada
processo pai tenha conhecimento de todos os seus filhos e aguarde

                    pelo término deles.

                    ------------------------------------------------------------
-----------------------------------------------------

                    */



                    //mostre na console o PID do processo corrente e o número
de filhos que ele aguardou ou está aguardando
```

```
                    printf("PID do processo corrente = %i e número de filhos =
%i", getpid(), nFilhos);


                    //Mostra os processos que estão sendo esperados no momento

                    printf("\nO processo de PID = %i está esperando os
seguintes filhos: %s\n\n", getpid(), pid_list);




                    wait(&status);


                    if (status == 0){


                         /*


==========================================================================
==================================

                         Responda: o que ocorre quando este trecho é
executado?

                         Os processos filhos referentes ao processo corrente
terminaram a sua execução com sucesso.

                         ----------------------------------------------------
------------------------------------------------------------

                         */

                    }else{

                         /*


==========================================================================
==================================

                         Responda: o que ocorre quando este trecho é
executado?

                         Os processos filhos referentes ao processo corrente
ainda não terminaram sua execução ou terminaram com falhas.

                         ----------------------------------------------------
------------------------------------------------------------

                         */
```

```
                        }
                    }
                }

                exit(0);
}
```

Prog3a.c

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

#define INF 0x33333333

int **aloca_matriz(int m, int k) {
      //ponteiro para a matriz e variável de iteração
      int **v, i;
```

```
      //Veririfica os parâmetros
      if(m < 1 || k < 1 ){
            printf ("\n\nErro: Valores de m e k inválidos!\n\n");
            exit(1);
      }

      //Aloca a linha da matriz
      v = (int **) calloc (m, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
      }

      //Aloca as colunas
      for( i = 0; i < m; i++ ){
            v[i] = (int*) calloc (k, sizeof(int));
            if(v[i] == NULL){
                  printf ("\n\nErro: Memória insuficiente!\n\n");
                  exit(1);
            }
      }

      //Retorna a matriz
      return (v);
}



int **free_matriz(int m, int k, int **v){
```

```
      int  i;
      if(v == NULL){
            exit(1);
      }

      //Verifica parâmetros
      if(m < 1 || k < 1){
            printf("\n\nErro: Parâmetro invalido!\n\n");
            return(v);
      }

      //Libera as linhas da matriz
      for(i=0; i<m; i++){
            free(v[i]);
      }

      //Libera a matriz
      free(v);
      return (NULL);
}


int *aloca_vetor(int m){
      //ponteiro do vetor
      int *v;

      //Veririfica o parâmetro
      if(m < 1){
            printf ("\n\nErro: Parâmetro invalido!\n\n");
            exit(1);
      }
```

```
	//Aloca a linha da matriz
	v = (int *) calloc (m+1, sizeof(int *));
	if(v == NULL){
		printf ("\n\nErro: Memória insuficiente!\n\n");
		exit(1);
	}

	//Retorna o vetor
	return (v);
}


int main(void){
	int **matriz, *produtoInterno, i, j, menor_i, maior_i, k, m, menor, maior,
somatorio;
	double soma, soma_desvio, desvio_padrao, tempo_execucao;
	srand((unsigned)time(NULL));
	struct timeb inicio_execucao, fim_execucao;


	//Recebe os valores iniciais de m e k
	printf("Defina o número de linhas  -> m = ");
	scanf("%i", &m);
	printf("Defina o número de colunas -> k = ");
	scanf("%i", &k);


	while( m!= 0 && k!=0 ){
		//Inicia a contagem do tempo de execução
		ftime(&inicio_execucao);
```

```
//Aloca a matriz e vetor de Produto Interno
printf("\nMontando a matriz... ");
matriz = aloca_matriz(m, k);
produtoInterno = aloca_vetor(m);
printf("Concluído!\n\n");


//Inicializa outros valores da iteração
menor = INF;
maior = -INF;
menor_i = 0;
maior_i = 0;
soma = 0;
soma_desvio = 0;
desvio_padrao = 0;


//Gera sobre a matriz
printf("Inserindo valores na Matriz... ");
for(i=0; i<m; i++){
      for(j=0; j<k; j++){
            //gera o número aleatório e armazena na matriz
            matriz[i][j] = (rand()%201)-100;
      }
}
printf("Concluído!\n\n");
```

```
printf("Calculando a Produto Interno...");
fflush(stdout);
//Calcula o somatório
for(i=0; i<m; i++){
      somatorio = 0;

      for(j=0; j<k; j++){
            //Realiza o produto interno
            somatorio += matriz[i][j] * matriz[i][j];
      }

      //Armazena o PI(i) e soma para cálculo de média
      produtoInterno[i] = somatorio;
      soma += produtoInterno[i];
}
printf(" Concluído!\n");


//Calcula o desvio padrão
for(i=0; i<m; i++){
      soma_desvio += pow(produtoInterno[i]-(soma/m), 2);

      //Detecta o maior e menor
      if(produtoInterno[i] <= menor){
            menor = produtoInterno[i];
            menor_i = i+1;
      }
      if(produtoInterno[i] >= maior){
            maior = produtoInterno[i];
            maior_i = i+1;
      }
```

```
            }

            desvio_padrao = sqrt(soma_desvio/m);




            //Calcula o tempo de execução

            ftime(&fim_execucao);

            tempo_execucao                  =                  (((fim_execucao.time-
inicio_execucao.time)*1000.0+fim_execucao.millitm)-
inicio_execucao.millitm)/1000.0;




            //Libera a matriz

            free_matriz(m, k, matriz);

            free(produtoInterno);




            //Exibe os valores resultantes

            printf("---------------------------Valores
Aferidos----------------------------\n");

            printf("Menor valor = %i (m=%i) e Maior valor = %i (m=%i)\n", menor,
menor_i, maior, maior_i);

            printf("Desvio Padrão = %f\n", desvio_padrao);

            printf("Tempo de execução = %.3f segundos\n", tempo_execucao);



printf("-----------------------------------------------------------------------\n
");



            //Recebe os valores de m e k para nova iteração

            printf("Montando a matriz da nova iteração\n\n");

            printf("Defina o número de linhas  -> m = ");

            scanf("%i", &m);

            printf("Defina o número de colunas -> k = ");

            scanf("%i", &k);
```

```
	}


	exit(0);

}
```

Prog3b.c

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <math.h>

#include <sys/timeb.h>
```

```
#include <sys/wait.h>


#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>


#define INF 0x33333333


struct shared{
      double *variancia;
      int   shmid,   *produtoInterno,   *produtoInterno_compartilhado,   *menor_i,
*maior_i, *menor, *maior;
      key_t key;
};


int **aloca_matriz(int m, int k) {
      //ponteiro para a matriz e variável de iteração
      int **v, i;


      //Veririfica os parâmetros
      if(m < 1 || k < 1 ){
            printf ("\n\nErro: Valores de m e k inválidos!\n\n");
            exit(1);
      }


      //Aloca a linha da matriz
      v = (int **) calloc (m, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
```

```
	}

	//Aloca as colunas
	for( i = 0; i < m; i++ ){
		v[i] = (int*) calloc (k, sizeof(int));
		if (v[i] == NULL){
			printf ("\n\nErro: Memória insuficiente!\n\n");
			exit(1);
		}
	}

	//Retorna a matriz
	return (v);
}



int **free_matriz(int m, int k, int **v){
	int  i;
	if(v == NULL){
		exit(1);
	}

	//Verifica parâmetros
	if(m < 1 || k < 1){
		printf ("\n\nErro: Parâmetro invalido!\n\n");
		return (v);
	}

	//Libera as linhas da matriz
	for(i=0; i<m; i++){
```

```
            free (v[i]);
      }

      //Libera a matriz
      free (v);
      return (NULL);
}


int *aloca_vetor(int m){
      //ponteiro do vetor
      int *v;

      //Veririfica o parâmetro
      if(m < 1){
            printf ("\n\nErro: Parâmetro invalido!\n\n");
            exit(1);
      }

      //Aloca a linha da matriz
      v = (int *) calloc (m+1, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
      }

      //Retorna o vetor
      return (v);
}
```

```
int max(int a, int b){
      if(a >= b){return a;}
      else{return b;}
}

int main(void){
      int i, j, m, k, somatorio, **matriz, *produtoInterno, *pids, id, status;
      double soma_desvio, desvio_padrao, tempo_execucao;
      int shmid[10];
      struct timeb inicio_execucao, fim_execucao;

      //Cria estrutura de compartilhamento de memória
      struct shared compartilhado;
      srand((unsigned)time(NULL));


      //Recebe os valores iniciais de m e k
      printf("Defina o número de linhas  -> m = ");
      scanf("%i", &m);
      printf("Defina o número de colunas -> k = ");
      scanf("%i", &k);


      while( m!= 0 && k!=0 ){
            //Inicia a contagem do tempo de execução
            ftime(&inicio_execucao);


            //Aloca a matriz Principal, vetor de Produto Interno e de PIDs
            printf("\nMontando a matriz... ");
```

```
            matriz = aloca_matriz(m, k);
            compartilhado.produtoInterno = aloca_vetor(m);
            pids = aloca_vetor(m);
            printf("Concluído!\n\n");



            //Cria o compartilhamento de memória
            for(i=0; i<6; i++){
                  if(i == 0){
                        if( (shmid[i] = shmget(getpid(), max(m, k)*sizeof(int),
IPC_CREAT | SHM_W | SHM_R)) < 0 ){
                              if((shmid[i] = shmget((unsigned)time(NULL), max(m,
k)*sizeof(int), IPC_CREAT | SHM_W | SHM_R)) < 0){
                                    perror("shmget 1");
                                    exit(1);
                              }
                        }
                  }
                  else if(i > 0 && i < 5){
                        if((shmid[i] = shmget(getpid()+i, sizeof(int), IPC_CREAT |
SHM_W | SHM_R)) < 0){
                              perror("shmget 2");
                              exit(1);
                        }
                  }
                  else{
                        if((shmid[i] = shmget(getpid()+5, sizeof(double),
IPC_CREAT | SHM_W | SHM_R)) < 0){
                              perror("shmget 3");
                              exit(1);
                        }
                  }
```

```
        }

        if((compartilhado.produtoInterno_compartilhado    =    shmat(shmid[0],
NULL, 0)) == (int *) -1){

              perror("shmat produtointerno");

              exit(1);

        }

        if((compartilhado.menor = shmat(shmid[1] , NULL, 0)) == (int *) -1){

              perror("shmat menor");

              exit(1);

        }

        if((compartilhado.maior = shmat(shmid[2], NULL, 0)) == (int *) -1){

              perror("shmat maior");

              exit(1);

        }

        if((compartilhado.menor_i = shmat(shmid[3], NULL, 0)) == (int *) -1){

              perror("shmat menor_i");

              exit(1);

        }

        if((compartilhado.maior_i = shmat(shmid[4], NULL, 0)) == (int *) -1){

              perror("shmat maior i");

              exit(1);

        }

        if((compartilhado.variancia = shmat(shmid[5], NULL, 0)) == (double *)
-1){

              perror("shmat variancia");

              exit(1);

        }




        //Inicializa outros valores da iteração

        *compartilhado.menor = INF;
```

```
*compartilhado.maior = -INF;
*compartilhado.variancia = 0;
soma_desvio = 0;
desvio_padrao = 0;



//Insere na matriz
printf("Inserindo valores na Matriz... ");
for(i=0; i<m; i++){
      for(j=0; j<k; j++){
            //gera o número aleatório e armazena na matriz
            matriz[i][j] = (rand()%201)-100;
      }
}
printf("Concluído!\n\n");



//Calcula o somatório
printf("Calculando o Produto Interno... ");
fflush(stdout);
for(i=0; i<m; i++){
      somatorio = 0;
      pids[i] = fork();

      if(pids[i] == 0){
            for(j=0; j<k; j++){
                  //Realiza o produto interno
                  somatorio += matriz[i][j] * matriz[i][j];
            }
```

```
                    //Armazena o PI(i) e soma para cálculo de média para o
desvio padrão

                    compartilhado.produtoInterno_compartilhado[i] = somatorio;

                    *compartilhado.variancia                                    +=
compartilhado.produtoInterno_compartilhado[i];

                    exit(0);

               }else{

                    wait(&status);

               }

          }

          printf("Concluído!\n\n");




          //Calcula o desvio padrão

          for(i=0; i<m; i++){

               soma_desvio                                                    +=
pow(compartilhado.produtoInterno_compartilhado[i]-(*compartilhado.variancia/m),
2);



               //Detecta o maior e menor Produto Interno

               if(compartilhado.produtoInterno_compartilhado[i]              <=
*compartilhado.menor){

                    *compartilhado.menor                                       =
compartilhado.produtoInterno_compartilhado[i];

                    *compartilhado.menor_i = i+1;

               }

               if(compartilhado.produtoInterno_compartilhado[i]              >=
*compartilhado.maior){

                    *compartilhado.maior                                       =
compartilhado.produtoInterno_compartilhado[i];

                    *compartilhado.maior_i = i+1;

               }

          }

          desvio_padrao = sqrt(soma_desvio/m);
```

```
            //Calcula o tempo de execução

            ftime(&fim_execucao);

            tempo_execucao                  =                  (((fim_execucao.time-
inicio_execucao.time)*1000.0+fim_execucao.millitm)-
inicio_execucao.millitm)/1000.0;




            //Libera a matriz

            free_matriz(m, k, matriz);




            //Exibe os valores resultantes

            printf("---------------------------Valores
Aferidos---------------------------\n");

            printf("Menor  valor  =  %i  (i=%i)  e  Maior  valor  =  %i  (i=%i)\n",
*compartilhado.menor,       *compartilhado.menor_i,       *compartilhado.maior,
*compartilhado.maior_i);

            printf("Desvio Padrão = %f\n", desvio_padrao);

            printf("Tempo de execução = %.3f segundos\n", tempo_execucao);



printf("-----------------------------------------------------------------------\n
");




            //Recebe os valores de m e k para nova iteração

            printf("Montando a matriz da nova iteração\n\n");

            printf("Defina o número de linhas  -> m = ");

            scanf("%i", &m);

            printf("Defina o número de colunas -> k = ");

            scanf("%i", &k);

      }
```

```
	exit(0);
}
```

Prog3c.c

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

#include <sys/wait.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define INF 0x33333333

struct shared{
	double *variancia;
	int shmid, *produtoInterno, *produtoInterno_compartilhado, *menor_i, *maior_i, *menor, *maior;
	key_t key;
};

int **aloca_matriz(int m, int k) {
	//ponteiro para a matriz e variável de iteração
	int **v, i;
```

```
      //Veririfica os parâmetros
      if(m < 1 || k < 1 ){
            printf ("\n\nErro: Valores de m e k inválidos!\n\n");
            exit(1);
      }

      //Aloca a linha da matriz
      v = (int **) calloc (m, sizeof(int *));
      if(v == NULL){
            printf ("\n\nErro: Memória insuficiente!\n\n");
            exit(1);
      }

      //Aloca as colunas
      for( i = 0; i < m; i++ ){
            v[i] = (int*) calloc (k, sizeof(int));
            if (v[i] == NULL){
                  printf ("\n\nErro: Memória insuficiente!\n\n");
                  exit(1);
            }
      }

      //Retorna a matriz
      return (v);
}



int **free_matriz(int m, int k, int **v){
```

```
      int  i;
      if(v == NULL){
            exit(1);
      }

      //Verifica parâmetros
      if(m < 1 || k < 1){
            printf ("\n\nErro: Parâmetro invalido!\n\n");
            return (v);
      }

      //Libera as linhas da matriz
      for(i=0; i<m; i++){
            free (v[i]);
      }

      //Libera a matriz
      free (v);
      return (NULL);
}


int *aloca_vetor(int m){
      //ponteiro do vetor
      int *v;

      //Veririfica o parâmetro
      if(m < 1){
            printf ("\n\nErro: Parâmetro invalido!\n\n");
            exit(1);
      }
```

```
	//Aloca a linha da matriz
	v = (int *) calloc (m+1, sizeof(int *));
	if(v == NULL){
		printf ("\n\nErro: Memória insuficiente!\n\n");
		exit(1);
	}

	//Retorna o vetor
	return (v);
}


int max(int a, int b){
	if(a >= b){return a;}
	else{return b;}
}
int main(void){
	int i, j, m, k, somatorio, **matriz, *produtoInterno, *pids, id, status;
	double soma_desvio, desvio_padrao, tempo_execucao;
	int shmid[10];
	struct timeb inicio_execucao, fim_execucao;

	//Cria estrutura de compartilhamento de memória
	struct shared compartilhado;
	srand((unsigned)time(NULL));


	//Recebe os valores iniciais de m e k
	printf("Defina o número de linhas  -> m = ");
```

```
	scanf("%i", &m);
	printf("Defina o número de colunas -> k = ");
	scanf("%i", &k);



	while( m!= 0 && k!=0 ){
		//Inicia a contagem do tempo de execução
		ftime(&inicio_execucao);



		//Aloca a matriz Principal, vetor de Produto Interno e de PIDs
		printf("\nMontando a matriz... ");
		matriz = aloca_matriz(m, k);
		compartilhado.produtoInterno = aloca_vetor(m);
		pids = aloca_vetor(m);
		printf("Concluído!\n\n");



		//Cria o compartilhamento de memória
		for(i=0; i<6; i++){
			if(i == 0){
				if( (shmid[i] = shmget(getpid(), max(m, k)*sizeof(int),
IPC_CREAT | SHM_W | SHM_R)) < 0 ){
					if((shmid[i] = shmget((unsigned)time(NULL), max(m,
k)*sizeof(int), IPC_CREAT | SHM_W | SHM_R)) < 0){
						perror("shmget 1");
						exit(1);
					}
				}
			}
			else if(i > 0 && i < 5){
				if((shmid[i] = shmget(getpid()+i, sizeof(int), IPC_CREAT |
```

```
SHM_W | SHM_R)) < 0){
                              perror("shmget 2");
                              exit(1);
                         }
                    }
                    else{
                         if((shmid[i]    =    shmget(getpid()+5,    sizeof(double),
IPC_CREAT | SHM_W | SHM_R)) < 0){
                              perror("shmget 3");
                              exit(1);
                         }
                    }
               }
               if((compartilhado.produtoInterno_compartilhado    =    shmat(shmid[0],
NULL, 0)) == (int *) -1){
                    perror("shmat produtointerno");
                    exit(1);
               }
               if((compartilhado.menor = shmat(shmid[1] , NULL, 0)) == (int *) -1){
                    perror("shmat menor");
                    exit(1);
               }
               if((compartilhado.maior = shmat(shmid[2], NULL, 0)) == (int *) -1){
                    perror("shmat maior");
                    exit(1);
               }
               if((compartilhado.menor_i = shmat(shmid[3], NULL, 0)) == (int *) -1){
                    perror("shmat menor_i");
                    exit(1);
               }
               if((compartilhado.maior_i = shmat(shmid[4], NULL, 0)) == (int *) -1){
                    perror("shmat maior i");
```

```
		exit(1);
	}
	if((compartilhado.variancia = shmat(shmid[5], NULL, 0)) == (double *)
-1){
		perror("shmat variancia");
		exit(1);
	}



	//Inicializa outros valores da iteração
	*compartilhado.menor = INF;
	*compartilhado.maior = -INF;
	*compartilhado.variancia = 0;
	soma_desvio = 0;
	desvio_padrao = 0;



	//Insere na matriz
	printf("Inserindo valores na Matriz... ");
	for(i=0; i<m; i++){
		for(j=0; j<k; j++){
			//gera o número aleatório e armazena na matriz
			matriz[i][j] = (rand()%201)-100;
		}
	}
	printf("Concluído!\n\n");



	//Calcula o somatório
	printf("Calculando o Produto Interno... ");
```

```
            fflush(stdout);


            id = fork();
            if( id == 0){
                  for(i=0; i<ceil(m/2); i++){
                        somatorio = 0;


                        if(id == 0){
                              for(j=0; j<k; j++){
                                    //Realiza o produto interno
                                    somatorio += matriz[i][j] * matriz[i][j];
                              }


                              //Armazena o PI(i) e soma para cálculo de média para
o desvio padrão
                              compartilhado.produtoInterno_compartilhado[i]       =
somatorio;
                              *compartilhado.variancia                           +=
compartilhado.produtoInterno_compartilhado[i];
                              exit(0);
                        }
                  }
            }else{
                  for(i=ceil(m/2); i<m; i++){
                        somatorio = 0;


                        if(id == 0){
                              for(j=0; j<k; j++){
                                    //Realiza o produto interno
                                    somatorio += matriz[i][j] * matriz[i][j];
                              }
```

```
                        //Armazena o PI(i) e soma para cálculo de média para
o desvio padrão

                        compartilhado.produtoInterno_compartilhado[i]       =
somatorio;

                        *compartilhado.variancia                            +=
compartilhado.produtoInterno_compartilhado[i];

                    }

                }

                wait(&status);

            }

            printf("Concluído!\n\n");




            //Calcula o desvio padrão

            for(i=0; i<m; i++){

                soma_desvio                                                 +=
pow(compartilhado.produtoInterno_compartilhado[i]-(*compartilhado.variancia/m),
2);



                //Detecta o maior e menor

                if(compartilhado.produtoInterno_compartilhado[i]            <=
*compartilhado.menor){

                    *compartilhado.menor                                     =
compartilhado.produtoInterno_compartilhado[i];

                    *compartilhado.menor_i = i+1;

                }

                if(compartilhado.produtoInterno_compartilhado[i]            >=
*compartilhado.maior){

                    *compartilhado.maior                                     =
compartilhado.produtoInterno_compartilhado[i];

                    *compartilhado.maior_i = i+1;

                }

            }

            desvio_padrao = sqrt(soma_desvio/m);
```

```
            //Calcula o tempo de execução

            ftime(&fim_execucao);

            tempo_execucao                  =                  (((fim_execucao.time-
inicio_execucao.time)*1000.0+fim_execucao.millitm)-
inicio_execucao.millitm)/1000.0;




            //Libera a matriz

            free_matriz(m, k, matriz);




            //Exibe os valores resultantes

            printf("---------------------------Valores
Aferidos---------------------------\n");

            printf("Menor  valor  =  %i  (i=%i)  e  Maior  valor  =  %i  (i=%i)\n",
*compartilhado.menor,       *compartilhado.menor_i,       *compartilhado.maior,
*compartilhado.maior_i);

            printf("Desvio Padrão = %f\n", desvio_padrao);

            printf("Tempo de execução = %.3f segundos\n", tempo_execucao);



printf("-----------------------------------------------------------------------\n
");




            //Recebe os valores de m e k para nova iteração

            printf("Montando a matriz da nova iteração\n\n");

            printf("Defina o número de linhas  -> m = ");

            scanf("%i", &m);

            printf("Defina o número de colunas -> k = ");

            scanf("%i", &k);

      }
```

```
    exit(0);

}
```

Método de execução

Os programas citados no estudo devem ser compilados e executados em um ambiente UNIX compatível através dos seguintes comandos:

1) Prog 1

gcc -o Prog1 Prog1.c **-lrt**

./Prog1

2) Prog 2

gcc -o Prog2 Prog2.c

./Prog2

3) Prog 3 - versão 1

gcc -o Prog3a Prog3a.c **-lm**

./Prog3a

3) Prog 3 - versão 2

gcc -o Prog3b Prog3b.c **-lm**

./Prog3b

3) Prog 3 - versão 3

gcc -o Prog3c Prog3c.c **-lm**

./Prog3c