



**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.....	3
pthread_create().....	3
pthread_join().....	4
pthread_exit().....	5
pthread_attr_init().....	5
pthread_attr_setscope().....	7
Prog1.....	8
Funcionamento.....	8
Configuração da máquina utilizada no desenvolvimento e testes.....	8
Comparação de desempenho entre as versões.....	8
Análise dos resultados e conclusões.....	9
Prog2.....	10
Especificações gerais.....	10
Versão 1.....	11
Estrutura e Análise.....	11
Versão 2.....	12
Estrutura e Análise.....	12
Interface e Operação.....	12
Apêndice.....	16
Prog1.c.....	16
Método de execução.....	20

# 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*.

Cada *thread* criada na linguagem C executa uma função definida no código. As variáveis locais são acessíveis por *thread*. No caso dos recursos compartilhados, todas as *threads* tem acesso. Ao contrário da criação de um subprocesso, onde o subprocesso executa todo o código do processo pai a partir do momento em que foi criado (através do comando *fork()*), a *thread* executa somente o código definido pela função especificada na sua criação.

Para a execução da função *pthread\_create* é necessário a passagem dos argumentos:

- Variável do tipo *pthread\_t*, que identifica a *thread*
- Variável do tipo *pthread\_attr\_t*, que identifica os atributos da *thread*
- (essa variável tem o valor *NULL* se não há execução de *pthread\_attr\_init()*, que inicializa os atributos da *thread*)
- Função a ser executada pela *thread*
- 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()*:

```
#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* “filha”. Análogo aos conceitos de subprocessos, a função *pthread\_join()* tem a mesma funcionalidade que o *wait()*. Dessa forma, o *pthread\_join* é responsável pela sincronização entre as *threads*, para os casos em que a continuação da execução de uma *thread* “pai” depende do término da execução de uma *thread* “filha”.

Argumentos necessários:

- *pthread\_t* utilizado na função *pthread\_create()*.
- *thread\_return*, que é o valor de retorno da *thread*, usualmente *NULL*.

A seguir um exemplo da execução de *pthread\_join()*:

```
#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 comando *pthread\_exit()*:

```
#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. Esta função é importante para a que se possa configurar os atributos de execução da *thread* de forma flexível.

*pthread\_attr\_init()* recebe como parâmetro um endereço de memória para uma variável do tipo *pthread\_attr\_t* (que identifica a *thread*). 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*. Também pode-se destruir os atributos de uma thread através da função *pthread\_attr\_destroy()*.

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

```
#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, &atributosThread, 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. A *pthread\_attr\_setscope()* faz parte da família de funções *pthread\_attr\_set\*()* que configura os parâmetros da *thread* a ser criada. Em casos de sucesso, a *pthread\_attr\_setscope* retorna zero, caso contrário retorna o número correspondente ao erro. Ela recebe os seguintes itens como parâmetro.

- Endereço de memória para a variável do tipo *thread\_attr\_t*
- Constante do SO que define o nível de execução da nova *thread* :
  - PTHREAD\_SCOPE\_PROCESS – para execução em nível usuário, padrão do SO.
  - PTHREAD\_SCOPE\_SYSTEM – para 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, &atributosThread, 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

Foram realizados dois testes nas máquinas do LCI-UFRJ. De acordo com a especificação do trabalho, o primeiro teste utilizou apenas um dos núcleos do processador e o segundo teste utilizou os dois núcleos disponíveis.

Configuração da máquina onde o teste foi realizado:

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

Utilizando-se a máquina descrita anteriormente, foi avaliada a performance dos seguintes programas que calculam o produto interno de uma matriz: Prog1.c (versão utilizando *threads*) e Prog3c.c (versão utilizando subprocessos – a escolha desta versão foi devido ao seu melhor desempenho dentre as versões apresentadas no trabalho 1).

A escolha dos parâmetros de entrada para os testes ocorreu de forma a garantir uma comparação válida, considerando a natureza de cada programa.

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

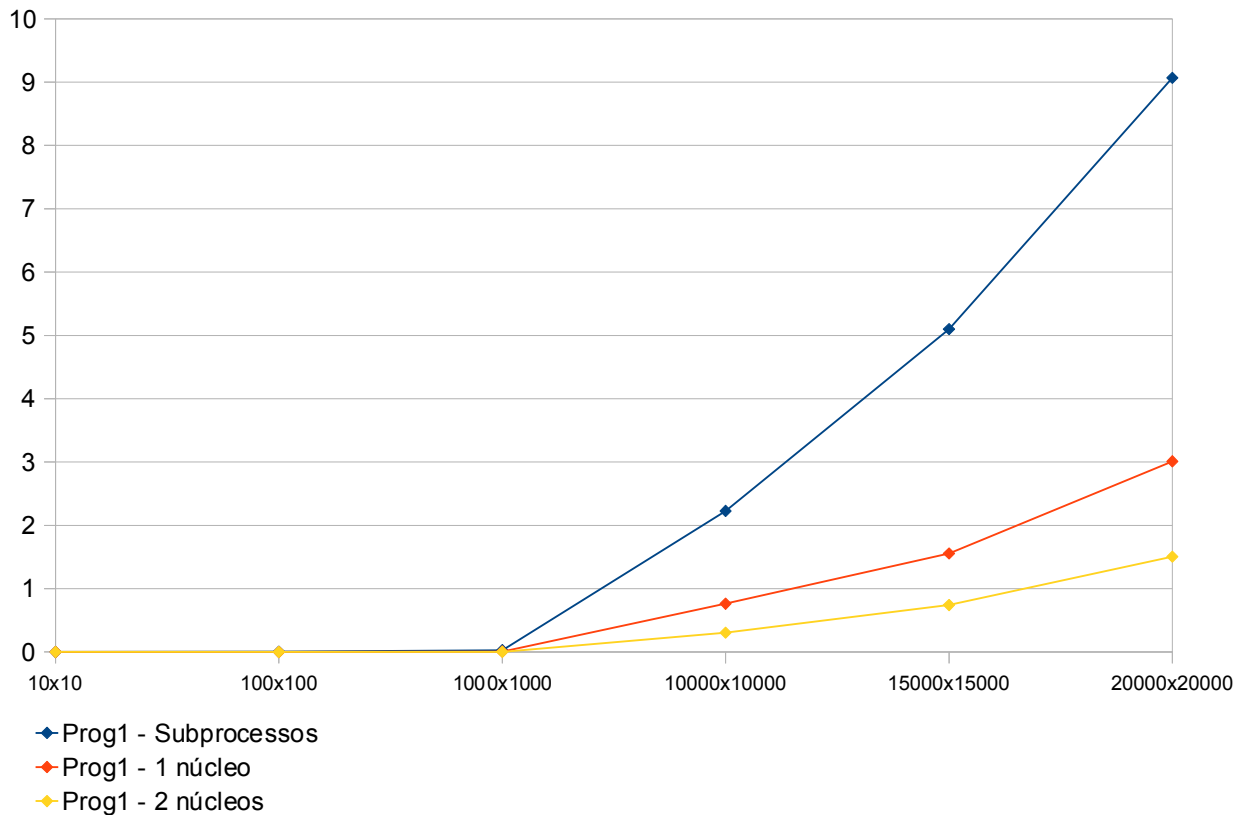
Dimensão da Matriz <i>m x k</i>	Versão 1 Subprocessos Tempo (segundos)	Versão 2 1 núcleo habilitado Tempo (segundos)	Versão 3 2 núcleos habilitados Tempo (segundos)
10 X 10	0,000	0,000	0,000
50 X 50	0,004	0,000	0,000
100 X 100	0,004	0,000	0,000
500 X 500	0,009	0,002	0,001
1000 X 1000	0,028	0,006	0,003
5000 X 5000	0,574	0,181	0,079
10000 X 10000	2,227	0,763	0,304
15000 X 15000	5,099	1,557	0,741
20000 X 20000	9,068	3,012	1,506

Para todas as versões, os testes realizados para entradas com valores de *m* e *k* menores que 1000, a diferença de performance entre os programas mostrou-se irrelevante.



### Análise dos resultados e conclusões

Baseado nos dados apresentados na tabela anterior, foi gerado o gráfico abaixo para facilitar a análise comparativa do desempenho dos programas, diante da variação dos valores de  $m$  e  $k$ .



A partir da análise gráfica, conclui-se que o uso de *threads* é mais eficiente que o uso de subprocessos em máquinas com apenas um núcleo e ainda mais eficiente em máquinas com dois núcleos.

Na comparação entre as versões do programa que utilizam *threads*, observa-se que quando se tem um processador com dois núcleos, as duas *threads* podem entrar em execução paralelamente. Dessa forma, em teoria, o tempo de execução do programa é reduzido em aproximadamente metade do tempo de quando ele é rodado em uma máquina com processador de um núcleo.

Associado a este fato, é importante considerar que a criação de *threads* é menos onerosa que a criação de subprocessos. Visto que a estrutura do PCB (Process Control Block) não é copiada a cada nova criação de *threads*, ela gera menos *overhead* os subprocessos.

# Prog2

## Especificações gerais

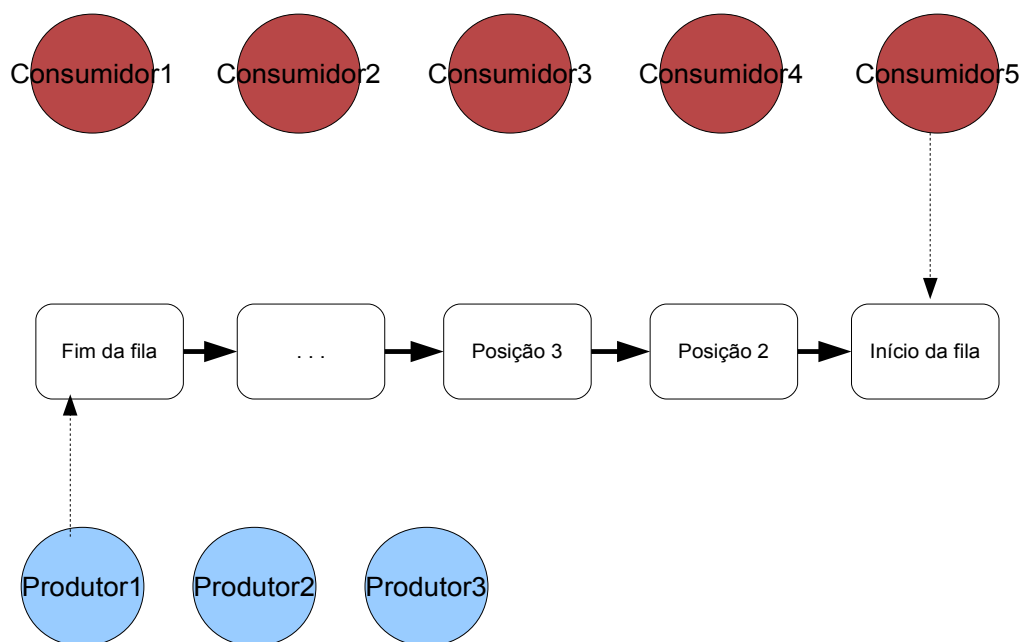
O Prog2 foi criado para ser uma simulação de um problema conhecido na literatura como Sistema Produtor/Consumidor. Neste problema temos  $m$  entidades chamadas de Produtor e  $n$  entidades chamadas de Consumidor. Também temos no sistema uma terceira entidade chamada de Recurso ou Produto. Nesse sistema, cada Produtor é responsável por produzir de forma independente das demais entidades, uma unidade do Recurso em tempo constante e finito. Da mesma forma, um Consumidor deve consumir uma unidade do Recurso também em tempo constante e finito.

A dinâmica deste sistema se dá pela forma em que os Produtores se organizam para produzir; como os Recursos são armazenados e disponibilizados aos Consumidores; e como estes se dispõem para consumir os Recursos.

Dadas as especificações do trabalho temos a seguinte dinâmica a ser implementada:

- 3 Produtores e 5 Consumidores.
- Produção de 1000 Recursos no total.
- Recursos dispostos em um fila (*buffer*) com 50 posições.
- Produtores inserem no final da fila.
- Consumidores consomem no início da fila.
- Ambiente assíncrono.

Para ilustrar em formas gerais como essa dinâmica se, dá podemos esboçar a seguinte figura:

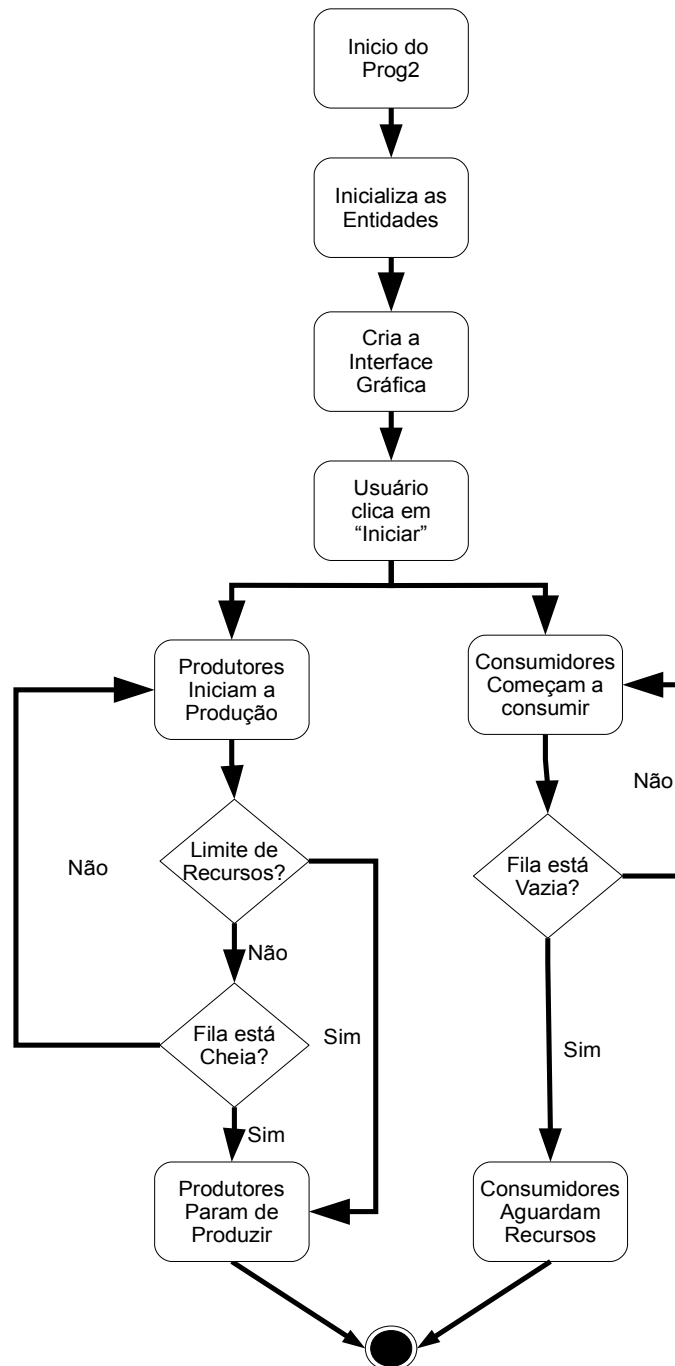


A especificação do Prog2 também requeria que a simulação contasse com uma interface gráfica que exibisse os valores relacionados a implementação. Desta forma, diferente dos demais programas, foi escolhida a linguagem **Java** por possuir uma fácil implementação de Interface Gráfica além de compatibilidade entre os diversos SO.

## Versão A

### Estrutura e Análise

A versão A do Prog2 determina a execução da simulação através do fluxo de execução abaixo. Porém, é importante ressaltar que, dado um ambiente assíncrono, este fluxo tem como objetivo apenas exemplificar uma execução.



Quando as atividades do sistema Produtor/Consumidor são iniciadas pelo usuário, os produtores e consumidores disputam fatias de tempo no processador. De acordo com a especificação do trabalho, os Produtores tem a mesma prioridade de acesso ao processador do que os Consumidores. Nesta versão, os consumidores não disputam os recursos entre si, apenas seguem a fila circular.

## **Versão B**

### **Estrutura e Análise**

A versão B do Prog2 pode ser ilustrada através do mesmo fluxo de execução da versão A, já que este não representa a ordem de acesso dos Consumidores à fila de recursos.

Entretanto, nesta versão, como os Consumidores disputam os Recursos entre si, o que pode ocorrer uma variação considerável do número de Recursos consumidos por cada Consumidor. Os Produtores também devem ter prioridade maior que os Consumidores no acesso a Região Crítica, acarretando em alguns períodos onde são produzidos diversos Recursos sem que os Consumidores tem acesso aos mesmos.

### **Interface e Operação**

Esta interface exibe o status do processamento de cada um dos Produtores e Consumidores. Além disso, durante a execução do programa, é exibido um registro em forma de *log* que informa todas as ações do simulador, assim como dos recursos a cada produção ou consumo dos mesmos. Dessa forma:

- Quando o produtor produz um recurso, é inserida um registro indicando:
  - Número de identificação do Recurso produzido. Afim de simplificação, cada novo Recurso tem o seu identificador igual a sua posição de produção.
  - Total de recursos já produzidos até o momento.
  - Total de recursos disponíveis para consumo.
- Quando o Consumidor consome um Recurso, é inserido um novo registro indicando:
  - O número de identificação do Recurso consumido.
  - Total de recursos ainda disponíveis para consumo.

Também é exibido um painel informativo contendo valores dos Semáforos utilizados para controlar o acesso a Região Crítica do sistema. Nesse caso, temos um Semáforo utilizado pelos Consumidores e outro pelos Produtores. Os valores podem ser 0 ou 1 para os Produtores e, dependendo da versão selecionada, 0 ou 1 ou, valores inteiros entre 0 e 5 para o Consumidores.

É possível selecionar entre os dois tipos de dinâmicas, de acordo com a proposta da alteração da especificação no item b através de um *checkbox* denominado “Versão B”. Não selecionado o *checkbox*, a dinâmica é a descrita no item a. Selecionando o *checkbox* a dinâmica é a descrita pelo item b. Durante a execução, não é possível alternar entre as dinâmicas.

A seguir temos um quadro que mostra os itens descritos, assim como a sua disposição na interface. É importante lembrar que pequenas variações de layout podem decorrer em diferentes SOs.

Programa 2 – Trabalho 2 – SO

Simulação do Sistema Produtor/Consumidor

Produtor 1

Status:

Produtos:

Produtor 2

Status:

Produtos:

Produtor 3

Status:

Produtos:

Consumidor 1

Status:

Consumidos:

Consumidor 2

Status:

Consumidos:

Consumidor 3

Status:

Consumidos:

Consumidor 4

Status:

Consumidos:

Consumidor 5

Status:

Consumidos:

Legenda

Ativo – Entidade ativa na Região Crítica

Sleeping – Entidade aguardando acesso a Região Crítica

Waiting – Entidade em Região Crítica aguardando Recurso

Terminou – Entidade terminou a sua função

Semáforos

Semáforo Produtor:

Semáforo Consumidor:

☐ Versão B

Iniciar

Mostramos a seguir imagens de iterações e seus resultados.  
Iteração em execução da Versão A:

Programa 2 – Trabalho 2 – SO

Simulação do Sistema Produtor/Consumidor

Produtor 1

Status: Sleeping

Produtos: 244

Produtor 2

Status: Ativa

Produtos: 243

Produtor 3

Status: Sleeping

Produtos: 243

- Consumidor 2

+ Produtor 2

- Consumidor 3

+ Produtor 1

- Consumidor 4

+ Produtor 3

- Consumidor 5

+ Produtor 1

- Consumidor 1

+ Produtor 3

- Consumidor 2

+ Produtor 2

-> Recurso consumido: 722

-> Recurso produzido: 724

-> Recurso consumido: 723

-> Recurso produzido: 725

-> Recurso consumido: 724

-> Recurso produzido: 726

-> Recurso consumido: 725

-> Recurso produzido: 727

-> Recurso consumido: 726

-> Recurso produzido: 728

-> Recurso consumido: 727

-> Recurso produzido: 729

-> Recurso consumido: 728

-> Recurso produzido: 730

Total já produzido: 724

Total já produzido: 725

Total já produzido: 726

Total já produzido: 727

Total já produzido: 728

Total já produzido: 729

Total já produzido: 730

Recurso disponível no momento: 1

Recurso disponível no momento: 2

Recurso disponível no momento: 1

Recurso disponível no momento: 2

Recurso disponível no momento: 1

Recurso disponível no momento: 2

Recurso disponível no momento: 3

Recurso disponível no momento: 2

Recurso disponível no momento: 3

Recurso disponível no momento: 2

Recurso disponível no momento: 3

Recurso disponível no momento: 2

Recurso disponível no momento: 3

Consumidor 1

Status: Sleeping

Consumidos: 146

Consumidor 2

Status: Sleeping

Consumidos: 146

Consumidor 3

Status: Sleeping

Consumidos: 145

Consumidor 4

Status: Sleeping

Consumidos: 145

Consumidor 5

Status: Sleeping

Consumidos: 145

Legenda

Ativo – Entidade ativa na Região Crítica

Sleeping – Entidade aguardando acesso a Região Crítica

Waiting – Entidade em Região Crítica aguardando Recurso

Terminou – Entidade terminou a sua função

Semáforos

Semáforo Produtor: 0

Semáforo Consumidor: 0

☐ Versão B

Executando...

Término da execução da Versão A:

Programa 2 – Trabalho 2 – SO

## Simulação do Sistema Produtor/Consumidor

**Produtor 1**

Status: Terminou

Produzidos: 334

**Produtor 2**

Status: Terminou

Produzidos: 332

**Produtor 3**

Status: Terminou

Produzidos: 334

+ Consumidor 4 - Consumidor 5 + Produtor 1 - Consumidor 1 + Produtor 2 - Consumidor 2 + Produtor 3 - Consumidor 3 + Produtor 1 - Consumidor 4 - Consumidor 5 X Consumidor 1	-> Recurso consumido: 994 -> Recurso produzido: 996 -> Recurso consumido: 995 -> Recurso produzido: 997 -> Recurso consumido: 996 -> Recurso produzido: 998 -> Recurso consumido: 997 -> Recurso produzido: 999 -> Recurso consumido: 998 -> Recurso produzido: 1000 -> Recurso consumido: 999 -> Recurso consumido: 1000 -> Simulação termina porque não serão mais produzidos recursos	Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 0
--	--	--

**Consumidor 1**

Status: Terminou

Consumidos: 200

**Consumidor 2**

Status: Terminou

Consumidos: 200

**Consumidor 3**

Status: Terminou

Consumidos: 200

**Consumidor 4**

Status: Terminou

Consumidos: 200

**Consumidor 5**

Status: Terminou

Consumidos: 200

**Legenda**

Ativo – Entidade ativa na Região Crítica

Sleeping – Entidade aguardando acesso a Região Crítica

Waiting – Entidade em Região Crítica aguardando Recurso

Terminou – Entidade terminou a sua função

**Semáforos**

Semáforo Produtor: 1      Semáforo Consumidor: 1

☐ Versão B      Iniciar

Iteração em execução da Versão B:

Programa 2 – Trabalho 2 – SO

## Simulação do Sistema Produtor/Consumidor

**Produtor 1**

Status: Sleeping

Produzidos: 109

**Produtor 2**

Status: Ativa

Produzidos: 110

**Produtor 3**

Status: Sleeping

Produzidos: 109

- Consumidor 3 - Consumidor 5 + Produtor 2 - Consumidor 4 + Produtor 3 + Produtor 1 - Consumidor 5 - Consumidor 3 ! Consumidor 2 -> Esperando recurso ser produzido... ! Consumidor 1 -> Esperando recurso ser produzido... ! Consumidor 3 -> Esperando recurso ser produzido... ! Consumidor 5 -> Esperando recurso ser produzido... + Produtor 2	-> Recurso consumido: 323 -> Recurso consumido: 324 -> Recurso produzido: 325 -> Recurso consumido: 325 -> Recurso produzido: 326 -> Recurso produzido: 327 -> Recurso consumido: 326 -> Recurso consumido: 327 -> Recurso produzido: 328	Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 0 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 0 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 2 Recursos disponíveis no momento: 1 Recursos disponíveis no momento: 0 Recursos disponíveis no momento: 1
--	---	--

**Consumidor 1**

Status: Waiting

Consumidos: 64

**Consumidor 2**

Status: Waiting

Consumidos: 74

**Consumidor 3**

Status: Waiting

Consumidos: 62

**Consumidor 4**

Status: Sleeping

Consumidos: 68

**Consumidor 5**

Status: Waiting

Consumidos: 59

**Legenda**

Ativo – Entidade ativa na Região Crítica

Sleeping – Entidade aguardando acesso a Região Crítica

Waiting – Entidade em Região Crítica aguardando Recurso

Terminou – Entidade terminou a sua função

**Semáforos**

Semáforo Produtor: 0      Semáforo Consumidor: 1

☒ Versão B      Executando...

## Término da execução da Versão B:

Programa 2 – Trabalho 2 – SO

### Simulação do Sistema Produtor/Consumidor

**Produtor 1**  
Status: Terminou  
Produzidos: 332

**Produtor 2**  
Status: Terminou  
Produzidos: 333

**Produtor 3**  
Status: Terminou  
Produzidos: 335

! Consumidor 2 -> Esperando recurso ser produzido...

! Consumidor 1 -> Esperando recurso ser produzido...

! Consumidor 3 -> Esperando recurso ser produzido...

! Consumidor 4 -> Esperando recurso ser produzido...

+ Produtor 1	-> Recurso produzido: 998	Total já produzido: 998	Recursos disponíveis no momento: 1
+ Produtor 2	-> Recurso produzido: 999	Total já produzido: 999	Recursos disponíveis no momento: 2
- Consumidor 5	-> Recurso consumido: 998		Recursos disponíveis no momento: 1
+ Produtor 3	-> Recurso produzido: 1000	Total já produzido: 1000	Recursos disponíveis no momento: 2
- Consumidor 5	-> Recurso consumido: 999		Recursos disponíveis no momento: 1
- Consumidor 1	-> Recurso consumido: 1000		Recursos disponíveis no momento: 0
X Consumidor 3	-> Simulação termina porque não serão mais produzidos recursos		
X Consumidor 4	-> Simulação termina porque não serão mais produzidos recursos		
X Consumidor 2	-> Simulação termina porque não serão mais produzidos recursos		

**Consumidor 1**  
Status: Terminou  
Consumidos: 198

**Consumidor 2**  
Status: Terminou  
Consumidos: 215

**Consumidor 3**  
Status: Terminou  
Consumidos: 193

**Consumidor 4**  
Status: Terminou  
Consumidos: 204

**Consumidor 5**  
Status: Terminou  
Consumidos: 190

**Legenda**  
Ativo – Entidade ativa na Região Crítica  
Sleeping – Entidade aguardando acesso a Região Crítica  
Waiting – Entidade em Região Crítica aguardando Recurso  
Terminou – Entidade terminou a sua função

**Semáforos**  
Semáforo Produtor: 1      Semáforo Consumidor: 5

☒ Versão B      **Iniciar**

# Apêndice

A seguir disponibilizamos o código fonte do Prog1 utilizado no estudo.

## Prog1.c

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

#include <pthread.h>

#define INF 0x33333333
#define THREADS 2

int **matriz, *produtoInterno;
int i, j, k, m, menor, maior, menor_i, maior_i, menor_matriz, maior_matriz,
menor_matriz_i, maior_matriz_i, menor_matriz_j, maior_matriz_j;
int parte[THREADS], returnThread[THREADS];
double soma, soma_desvio, desvio_padrao, tempo_execucao;
struct timeb inicio_execucao, fim_execucao;
pthread_t thread[THREADS];
pthread_attr_t atributosThread;

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);
}

void *calculaProdutoInterno(void *arg){
    int i, j, *pos, inicio, fim, somatorio;

    pos = (int *) arg;
    inicio = (*pos - 1)*floor(m/THREADS);
    fim = inicio + floor(m/THREADS) ;

    //Detecta se a thread é a última. Nesse caso precisa calcular também o resto
da divisão
    if( m-fim < THREADS ){
        fim = m;
    }

    //Calcula o somatório
    for(i=inicio; i<fim; 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("\nThread %i terminou de calcular as linhas entre %i e %i", *pos,
inicio+1, fim);

    //Termina a thread
    pthread_exit(NULL);
}

int main(void){

    //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);

    //Inicia o parâmetro da contagem de tempo
    srand((unsigned)time(NULL));

    //Configura os atributos da Thread
    pthread_attr_init(&atributosThread);
    pthread_attr_setdetachstate(&atributosThread, PTHREAD_CREATE_JOINABLE);
    //No caso, tipo Kernel
    pthread_attr_setscope(&atributosThread, PTHREAD_SCOPE_SYSTEM);

    while( m!= 0 && k!=0 ){
        //Inicializa outros valores da iteração
        menor = INF;
        maior = -INF;
        menor_matriz = INF;
        maior_matriz = -INF;
        menor_i = 0;
        maior_i = 0;
        menor_matriz_i = 0;
        maior_matriz_i = 0;
        menor_matriz_j = 0;
        maior_matriz_j = 0;
        soma = 0;
        soma_desvio = 0;
        desvio_padrao = 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");

        //Gera 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;

                //Detecta o maior e menor na matriz
                if(matriz[i][j] <= menor_matriz){
                    menor_matriz = matriz[i][j];
                    menor_matriz_i = i+1;
                    menor_matriz_j = j+1;
                }
            }
        }
    }
}

```

```

        }
        if(matriz[i][j] >= maior_matriz){
            maior_matriz = matriz[i][j];
            maior_matriz_i = i+1;
            maior_matriz_j = j+1;
        }
    }
}
printf("Concluído!\n\n");

printf("Calculando a Produto Interno...");
//Neste programa estamos calculando o tempo necessário para realizar o
produto interno utilizando threads.
//pega o tempo inicial da execução
ftime(&inicio_execucao);

//Loop que cria as threads
for(i=0; i<THREADS; i++){

    //Determina o posicionamento da Thread na matriz
    parte[i] = i+1;

    //Inicializa as threads passando o parâmetro da sua posição
    returnThread[i] = pthread_create(&(thread[i]), &atributosThread,
calculaProdutoInterno, (void*) &(parte[i]));

    //Verifica se houve um erro ao criar a Thread
    if(returnThread[i] > 0){
        printf("Não foi possível criar a Thread para o segmento i=
%i", i);
        exit(1);
    }

}

//Aguarda o término das threads
for(i=0; i<THREADS; i++){
    pthread_join(thread[i], NULL);
}
//pega o tempo final da execução
ftime(&fim_execucao);
printf("Concluído!\n\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 as linhas da matriz
        free_matriz(m, k, matriz);
        free(produtoInterno);

        //Exibe os valores resultantes
        printf("-----Valores
Aferidos-----\n");
        printf("Menor valor na matriz = %i (i=%i, j=%i) e Maior valor na matriz
= %i (i=%i, j=%i)\n", menor_matriz, menor_matriz_i, menor_matriz_j, maior_matriz,
maior_matriz_i, maior_matriz_j);
        printf("Menor valor Produto Interno = %i (m=%i) e Maior valor Produto
Interno = %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\n");

        //Recebe os valores de m e k para nova iteração
        printf("Nova iteração\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

O Prog1 deve ser compilado e executado em um ambiente UNIX compatível através dos seguintes comandos:

A) Prog 1 - Este programa utiliza threads

```
gcc -o Prog1 Prog1.c -pthread -lm
```

```
./Prog1
```

B) Prog 1 - Este programa utiliza subprocessos

```
gcc -o Prog3c Prog3c.c -lm
```

```
./Prog3c
```

O Prog2 deve ser compilado e executado em qualquer ambiente que possua uma máquina virtual Java 1.5 ou superior.

```
eclipse-3.5 /opt/sun-jdk-1.6.0.26/bin/java
```

```
-Criar um Novo Projeto
```

```
-Adicionar os arquivos do diretório ./Prog2
```

```
java -jar ./Prog2.jar
```