

# Trabalho Final - Sistemas de Computação

Rômulo Augusto Vieira Costa<sup>1</sup>, Rodrigo Dracxler<sup>1</sup>

<sup>1</sup>Instituto de Computação – Universidade Federal Fluminense (UFF)  
R. Passo da Pátria, 156 - São Domingos, 24210-240  
Niterói – RJ – Brasil

romulo-vieira96@yahoo.com.br, dracxler@gmail.com

**Resumo.** *A maioria dos sistemas operacionais modernos permitem a programação concorrente, onde dois ou mais processos são executados de forma simultânea. Apesar disso evitar ao máximo a ociosidade da CPU, alguns problemas podem acontecer, como dois ou mais processos tentarem acessar ao mesmo tempo os mesmos recursos, impactando no desempenho do programa. Um exemplo onde isso fica claro são nas atividades do tipo Produtor-Consumidor. Para sanar este problema e exercitar os conceitos vistos em sala de aula, o presente trabalho apresenta a implementação de um semáforo para leitura e consumo de matrizes em threads compartilhadas. São expostas as tecnologias, bibliotecas e constantes utilizadas para completar esta tarefa, bem como dicas para compilação e execução do código. Ao fim de tudo, conclusões resumidas são fornecidas ao leitor.*

## 1. Introdução

Quando um programa é executado, ele deve rodar como parte de algum processo. Esse processo, como todos os outros, é caracterizado por um estado e um espaço de endereço por meio do qual o programa e os dados podem ser acessados. O estado inclui no mínimo o contador de programa, uma palavra de estado de programa, um ponteiro de pilha e os registradores gerais [Tanenbaum 2013].

A maioria dos sistemas operacionais modernos permite que processos sejam criados e encerrados dinamicamente. Para tirar total proveito dessa característica e obter processamento paralelo, é preciso uma chamada de sistema para criar um novo processo. Essa chamada de sistema pode apenas fazer um clone do processo chamado, ou permitir ao chamador a criação de um processo que especifique seu estado inicial, incluindo seu programa, dados e endereço de início [Silberschatz et al. 2015].

Em alguns casos, o processo criador (pai) mantém controle parcial, ou até mesmo total, do processo criado (filho). Com essa finalidade, existem instruções virtuais para permitir que um processo-pai interrompa, reinicie, examine e encerre seus filhos. Dessa forma, surgiu a programação concorrente, baseada na execução de forma colaborativa de múltiplos processos ou *threads* que trabalham em uma mesma atividade [Tanenbaum 2013, Oliveira et al. 2002].

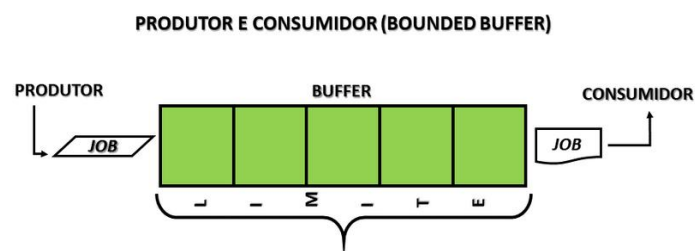
Em sistemas multiprogramados com apenas um processador, os processos se alternam no uso da CPU (Central Processing Unit) e dos demais recursos de acordo com critérios de escalonamento de processos que são estabelecidos pelo sistema operacional. Em sistemas com múltiplos processadores, a possibilidade de paralelismo é real, permitindo que várias tarefas sejam executadas em simultâneo, sendo cada uma delas executada em um processador diferente.

É natural que processos de uma aplicação concorrente compartilhem recursos. Entretanto, tal compartilhamento acarreta em problemas que podem comprometer a execução da aplicação. Por este motivo, é necessário que os processos sejam sincronizados pelo sistema operacional com o objetivo de garantir o funcionamento correto dos programas. A **sincronização** contém os mecanismos que garantem esta comunicação, e ela é fundamental para garantir a confiabilidade na execução de aplicações concorrentes.

### 1.1. Produtor-Consumidor

Um desafio clássico na ciência da computação que envolve sincronização é o Produtor-Consumidor, também conhecido como Bounded Buffer Problem (BBP). Nele, duas ou mais *threads* concorrentes têm acesso a um mesmo recurso do programa que está sendo executado, o que implica na possibilidade de conflitos entre estas *threads*, que disputam o controle da CPU.

No caso do Produtor-Consumidor, o recurso disputado é um *buffer* de memória de tamanho limitado/fixo. O produtor será o responsável por enfileirar novos *jobs* ou dados nesse *buffer*, ao passo que o consumidor assume a responsabilidade de extrair esses dados, conforme ilustrado na Figura 1<sup>1</sup>.



**Figura 1. Estrutura básica do Produtor-Consumidor.**

Entretanto, surge um desafio neste tipo de comunicação: e se tanto o produtor quanto o consumidor conseguirem acessar as variáveis deste processo de forma concorrente?

### 1.2. Semáforos

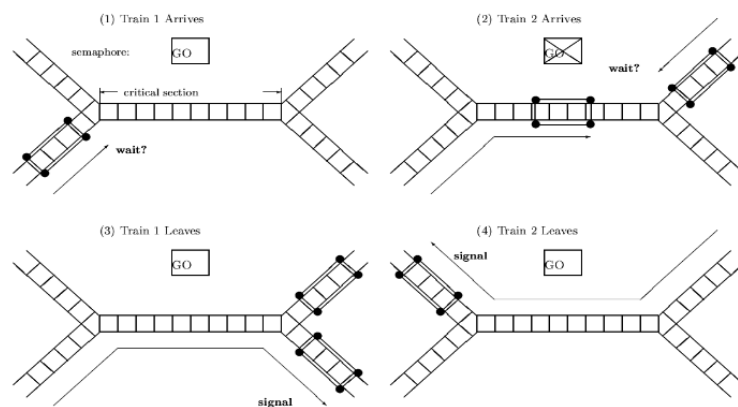
Uma alternativa para atacar este problema é utilizar a metodologia dos semáforos<sup>2</sup>. Tal técnica foi criada em 1965, por E.W. Dijkstra, que criou tal mecanismo inspirado na sincronização que ocorre entre sinais de trens (conforme Figura 2).

Um semáforo *S* é uma variável inteira que, exceto na inicialização, é acessada apenas por meio de duas operações atômicas: `wait ( )` e `signal ( )`. A operação `wait ( )` era originalmente denominada *P* (do holandês *proberen*, que significa “testar”); e `signal ( )`, originalmente denominada *V* (de *verhogen*, que significa “incrementar”).

Os sistemas operacionais costumam fazer a distinção entre semáforos de contagem e binários. O valor de um semáforo de contagem pode variar dentro de um domínio irrestrito. O valor de um semáforo binário pode variar somente entre 0 e 1. Assim, os

<sup>1</sup><https://deinfo.uepg.br/alunos/2021/SO/producerconsumer.java/>

<sup>2</sup><https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html>



**Figura 2. Funcionamento de um semáforo de trens que inspirou esta metodologia. Nela, um processo (trem) só pode avançar quando outro processo já finalizou.**

semáforos binários comportam-se de maneira semelhante aos *locks mutex*. Na verdade, em sistemas que não fornecem tal funcionalidade, os semáforos binários podem ser usados em seu lugar para oferecer exclusão mútua.

Semáforos de contagem podem ser usados para controlar o acesso a determinado recurso composto por um número finito de instâncias. O semáforo é inicializado com o número de recursos disponíveis. Cada processo que deseja usar um recurso executa uma operação `wait ( )` no semáforo (decrementando a contagem). Quando um processo libera um recurso, ele executa uma operação `signal ( )` (incrementando a contagem). Quando a contagem do semáforo chega a 0, todos os recursos estão sendo usados. Depois disso, processos que queiram usar um recurso ficarão bloqueados até a contagem se tornar maior do que 0.

Diante da robustez de tal solução, o presente trabalho apresenta uma implementação prática de um semáforo para a seguinte cadeia de execução para uma lista de matrizes:  $P \rightarrow CP1 \rightarrow CP2 \rightarrow CP3 \rightarrow C$ . A comunicação entre os pares de classes de *threads* deve ser feita por estruturas compartilhadas (um vetor de nome `shared`, sendo `shared[0]` usado por `P` e `CP1`, `shared[1]` por `CP1` e `CP2`, `shared[2]` usado por `CP2` e `CP3` e `shared[3]` usado por `CP3` e `C`). O objetivo desta tarefa é aplicar de forma prática os conceitos de sincronização e semáforos estudados em sala de aula.

O restante do artigo está dividido da seguinte maneira. Seção 2 apresenta as ferramentas e técnicas utilizadas para a conclusão deste trabalho, enquanto a Seção 3 mostra o passo a passo para compilação e execução do código. Finalmente, conclusões resumidas são apresentadas na Seção 4.

## 2. Ferramentas e técnicas utilizadas

O trabalho apresenta a implementação de um semáforo para lidar com a aplicação Produtor-Consumidor. Para isso, são utilizadas as seguintes ferramentas:

- **Linux Ubuntu Pop\_Os 21.04:** distribuição Linux gratuita de código aberto, baseada no Ubuntu, e conta com um desktop GNOME personalizado. O Pop\_Os foi desenvolvido principalmente para ser fornecido juntamente com os computadores

personalizados produzidos pela System76, mas também pode ser baixado e instalado na maioria dos computadores. A arte do sistema (temas, ícones, logos, etc.) tem como base o movimento artístico pop art dos anos 50. É considerada uma distribuição fácil de configurar para jogos, principalmente devido ao seu suporte de GPU (Graphics Processing Unit) discreta. Além disso, fornece criptografia de disco por padrão, Pop Shell (um organizador de janelas simplificado), bem como perfis de gerenciamento de energia integrados que incluem os modos "Economia de energia", "Balanceado" e "Alta performance";

- **Linguagem C:** linguagem de programação compilada de propósito geral, estruturada, imperativa e procedural, criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix (originalmente escrito em Assembly). É uma das linguagens de programação mais populares e tem influenciado muitas outras, como por exemplo o Java e mais notavelmente C++;
- **OpenMP:** conjunto de diretivas de compilador assim como uma API (Application Programming Interface) para programas escritos em C ou C++, proporcionando suporte à programação paralela em ambientes de memória compartilhada.

## 2.1. Arquivos

Conforme solicitado nas diretrizes do trabalho, é necessária a entrada de um arquivo de texto chamado **"entrada.in"** que deve conter referências para 50 outros arquivos, cada um deles contendo duas matrizes de números decimais e dimensão 10x10. As matrizes foram geradas de forma automática através do código disponível no arquivo intitulado **"gera\_matriz.ipynb"**. A formatação da matriz pode ser vista na Figura 3, seguindo os moldes também exigidos para o trabalho.

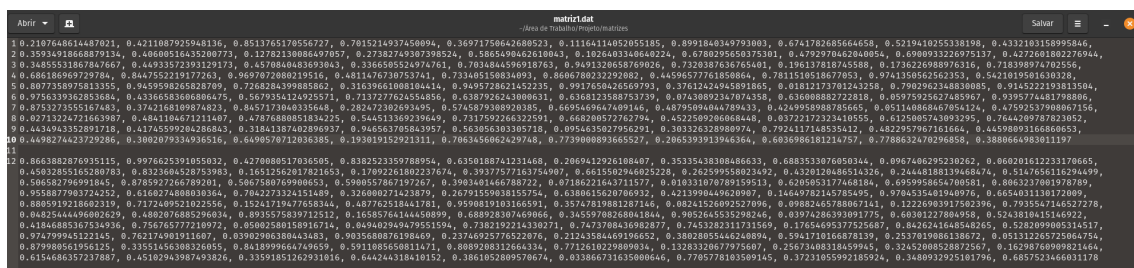


Figura 3. Exemplo de um arquivo de matriz.

Fonte: O Autor

O arquivo de saída gerado pela *thread* produtora recebeu o nome de **"saida.out"** e foi formatado de modo que cada linha do arquivo corresponda a uma linha da matriz.

## Por que utilizar dados .dat e não .txt?

Geralmente, as aplicações na linguagem C utilizam arquivos de entrada com extensão .dat, que não indica quais tipos de dados estão salvos no arquivo, tampouco oferece alguma especificação sobre eles. Seu uso é comum para lidar com serviços que exigem diferentes tipos de dados e/ou dados binários, enquanto que do .txt espera-se texto simples. De modo a fazer essa distinção e seguir o padrão, os autores optaram por criar os arquivos de matrizes na extensão .dat.

## 2.2. Estrutura de Dados

As principais bibliotecas utilizadas neste trabalho são exibidas na Tabela 1<sup>3</sup>, enquanto as constantes são expostas na Tabela 2<sup>4</sup>.

**Tabela 1. Principais bibliotecas utilizadas no projeto.**

Biblioteca	Utilidade
stdio.h	permite implementar subrotinas relativas às operações de entrada/saída
string.h	permite a manipulação de cadeias de caracteres e regiões de memória
stdlib.h	permite alocação de memória, controle de processos e conversões
pthread.h	padrão para usar threads em sistemas POSIX
semaphore.h	permite o uso de semáforos
stdbool.h	permite manipular variáveis lógicas
syslog.h	define constantes simbólicas, zeros ou mais que formam o openlog()
unistd.h	fornece acesso à API do sistema operacional POSIX
sys_types.h	define os tipos de dados usados no código-fonte do sistema
omp.h	usado para chamar a API do OpenMP

**Fonte: O Autor.**

**Tabela 2. Constantes utilizadas no projeto.**

Constante	Valor	Utilização
SHARED_SIZE	4	Memória compartilhada
BUFF_SIZE	5	Tamanho do <i>Buffer</i> em cada memória compartilhada
NP	1	Número de <i>threads</i> produtoras
NCP1	5	Número de <i>threads</i> Produtoras-Consumidoras 1
NCP2	4	Número de <i>threads</i> Produtoras-Consumidoras 2
NCP3	3	Número de <i>threads</i> Produtoras-Consumidoras 3
NC	1	Número de <i>threads</i> consumidoras
NITERS	50	Número de itens produzidos/consumidos
NLINES	10	Número de linhas das matrizes
NCOLS	10	Número de colunas das matrizes

**Fonte: O Autor.**

## 3. Compilação e Execução

Para a execução do programa, abra um terminal dentro do diretório contendo os arquivos das matrizes e execute:

**gcc -o trabalho trabalho.c -pthread**

Isso irá gerar um arquivo executável. Ele pode ser construído a partir do seguinte comando:

**./trabalho**

O terminal irá exibir a execução do produtor consumidor, conforme visto na Figura 4.

<sup>3</sup>As bibliotecas syslog.h, unistd.h, sys\_types.h e omp.h foram utilizadas para os exercícios extras.

<sup>4</sup>NLINES e NCOLS foram utilizadas nos exercícios extras

Figura 4. Exemplo do fluxo de execução do semáforo implementado no projeto.

Fonte: O autor

### 3.1. Exercícios Extras

Alguns exercícios extras foram propostos para incrementar novas funcionalidades à aplicação. A seguir, são listadas estas novas e atividades e como elas foram realizadas.

**Exercício 1 - Transformar a aplicação em um *daemon*, isto é, uma aplicação que cria um processo filho, se torna filho de outro processo na árvore do SO para que possa ser executado em *background*, mude permissões de acesso e dê um “kill” no processo-pai. Fazer syslog de todo o sistema, inclusive no início e fim de cada estágio, informando sempre o nome do arquivo de origem que está sendo trabalhado.**

O seguinte código foi utilizado para realização dessa tarefa. Na sequência, um arquivo **.syslog** foi gerado no diretório **root**. A partir dele, os autores extraíram os *logs* referentes à execução do programa.

```
1 // Transformando o processo em Daemon
```

```

2 FILE *fp= NULL;
3 pid_t process_id = 0;
4 pid_t sid = 0;
5
6 // Criando processo-filho
7 process_id = fork();
8 // Tratativa de erro no fork()
9 if (process_id < 0)
10 {
11 printf("fork failed!\n");
12 // Retorna c digo de erro no status de sa da
13 exit(1);
14 }
15 // Processo-pai. Deve passar por um "kill"
16 if (process_id > 0)
17 {
18 printf("process_id of child process %d \n", process_id);
19 // Caso condi o seja satisfeita , retorna status de sucesso na sa da
20 exit(0);
21 }
22
23 //unmask the file mode
24 umask(0);
25
26 // Inicia nova sess o
27 sid = setsid();
28 if(sid < 0)
29 {
30 // Retorna c digo de erro
31 exit(1);
32 }
33
34 // Muda para diret rio ra z
35 chdir("/");
36 syslog(LOG.NOTICE, "Log Info %d\n", 10);
37
38 // Encerra stdin. stdout e stderr
39 close(STDIN_FILENO);
40 close(STDOUT_FILENO);
41 close(STDERR_FILENO);
42
43 // Abre arquivo de log no modo escrita
44 fp = fopen("Log.txt", "w+");
45 while (1)
46 {
47 // N o bloqueia as trocas de contexto. Processo fica em sleep()
48 sleep(1);
49 fprintf(fp, "Logging info...\n");
50 fflush(fp);
51
52 // Implementa e chama a fun o principal no Daemon
53 }
54 fclose(fp);
55 closelog();
56 openlog("PROJETO", LOG_PID, LOG_DAEMON);
57 return (0);

```

### Exercício 3 - Fazer com que cada thread do tipo CP1 calcule a multiplicação em paralelo, com OpenMP, em 2 sub-threads.

O seguinte código foi utilizado para esta atividade. Para visualizar o resultado da multiplicação, basta executar um **print** no ID da *thread*, número da linha e número da coluna.

```
1 // Multiplica o em paralelo de cada thread CP1 – extra 3
2 void multiply(double A[NLINES][NCOLS], double B[NLINES][NCOLS], double
   C[NLINES][NCOLS])
3 {
4
5     int i, tid; // i = número de threads; tid = identificador de
      threads
6     #pragma omp parallel num_threads(2) default(none) private(i, tid)
      shared(A, B, C) // construtor do OpenMP que lida com duas threads
7     // Default = 1 usula que define que o tipo de todas as variáveis
      envolvidas na região paralela deve ser declarado explicitamente (
      sobre-se define o de que, por omissão, as variáveis
      são consideradas compartilhadas)
8     // Shared = 1 usula que define sobre as variáveis definidas em
      listas compartilhadas por todos os threads ficando
      responsabilidade do programador garantir o seu correto manuseamento
9
10    {
11        // // Obtem o identificador de thread
12        // omp_get_thread_num() = Função básica do OpenMP que
      retorna o identificador do thread corrente.
13        tid = omp_get_thread_num();
14        int start_line = tid * (NLINES / 2);
15        int end_line = start_line + (NLINES / 2);
16        int j, k; // declarando variáveis a serem usadas nas matrizes
17        #pragma omp parallel for // #pragma omp for divides loop
      iterations between the spawned threads.
18        for (i = start_line; i < end_line; i++) {
19            for (j = 0; j < NCOLS; j++) {
20                C[i][j] = 0;
21                for (k = 0; k < NCOLS; k++) {
22                    C[i][j] += A[i][k] * B[k][j]; // Multiplica matriz
      A * matriz B e salva em matriz C
23                }
24            }
25        }
26    }
27 }
```

### Exercício 4 - Fazer com que cada thread do tipo CP1 calcule a multiplicação em paralelo com OpenMP, em 2 sub-threads.

O código a seguir ilustra tal atividade. Lembrando que todos eles encontram-se ao longo do arquivo **extras.c** e eles devem ser executados no mesmo diretório com os arquivos das matrizes, exatamente como na parte principal do trabalho.

```
1 // Calcula soma dos vetores em paralelo para cada thread do CP2 – Extra
   4
```



```

2 void sum_cols(double matrix[NLINES][NCOLS], double vector_sums[NCOLS])
3 {
4     int i, j, tid; // declarando i, j e tid para esta fun o
5     #pragma omp parallel num_threads(2) private(i, j, tid) shared(
6     matrix, vector_sums) // The private variable is updated after the
7     end of the parallel construct.
8     {
9         tid = omp_get_thread_num();
10        int start_col = tid * (NCOLS / 2);
11        int end_col = start_col + (NCOLS / 2);
12
13        #pragma omp parallel for
14        for (j = start_col; j < end_col; j++) {
15            double sum = 0;
16            for (i = 0; i < NLINES; i++) {
17                sum += matrix[i][j]; // sum = sum + matriz
18            }
19            vector_sums[j] = sum; // atribuindo resultado de sum a
20            vari vel indicada
21        }
22    }
23 }

```

## 4. Conclusões

O presente trabalho apresentou a implementação prática dos semáforos para solucionar o problema de sincronização em atividades Produtoras-Consumidoras, de modo a explorar os conceitos vistos em sala de aula.

Diante dos códigos fornecidos pelo professor e da vasta bibliografia sobre o tema disponível em livros, artigos e fóruns de discussão, os problemas que surgiram durante o desenvolvimento do trabalho puderam ser contornados.

Algumas atividades extras mostraram-se especialmente desafiadoras e o grupo lamenta a falta de tempo hábil para tentar resolvê-las. Mas de qualquer forma, o trabalho cumpriu seu propósito.

## Referências

- Oliveira, R., Carissimi, A., and Toscani, S. (2002). Sistemas operacionais como programas concorrentes. *Segunda Escola Regional de Alto Desempenho*, 1:1 – 39.
- Silberschatz, A., Galvin, P., and Gagne, G. (2015). *Fundamentos de Sistemas Operacionais*. LTC, Nova Yor, EUA.
- Tanenbaum, A. (2013). *Organização Estruturada de Computadores*. Pearson Universidade - Brasil, São Paulo, Brasil.