

Solução para o Problema do Produtor-Consumidor

Ricardo Mendes do Nascimento

Universidade Regional Integrada do Alto Uruguai e das Missões (URI)
Santo Ângelo – RS – Brasil

ricna.net@hotmail.com

***Resumo.** Este artigo tem como objetivo apresentar a solução implementada para o problema produtor-consumidor. Serão explicados sucintamente, e conforme surgir necessidade, aqueles conceitos essenciais para a compreensão da solução apresentada no artigo.*

1. Introdução ao Problema

Produtor – Consumidor é um problema clássico em Sistemas Operacionais que busca exemplificar de forma clara situações de impasses que ocorrem no gerenciamento de processos de um SO, e a solução para estes.

Partindo do princípio de que cada processo executado em um SO tem a possibilidade de utilizar determinados recursos oferecidos, e que quando um recurso R está sendo utilizado por um processo P nenhum outro processo poderá utilizar este mesmo recurso, encontramos um problema no momento em que um processo Z tenta utilizar o recurso R que já está em uso.

O sistema operacional deve ter a capacidade de identificar esta situação e não permitir que o processo Z tenha permissão de acesso ao recurso R, pois esta ação acarretaria em consequências indesejáveis. Por outro lado o processo Z precisa acessar o recurso R para que possa continuar, este impasse o leva a um estado de *espera ociosa*.

A *espera ociosa* não é desejável, pois faz com que o processo Z continue em execução através de um laço ocioso, consumindo processamento e sendo *escalonado* (escolhido) pelo SO, até que consiga utilizar o recurso R e continuar sua execução normalmente. Cada vez que o processo Z é *escalonado*, ele tenta utilizar o recurso R, e se não consegue consome o tempo e processamento do CPU no seu laço ocioso.

O problema do Produtor-Consumidor apresenta justamente de forma exemplificada esta situação como base para estudo de diferentes soluções.

Neste problema podemos de forma análoga imaginar um armazém, com capacidade para N fardos de alfafa, e dois indivíduos: o Produtor, que produz fardos e os coloca no armazém, e o Consumidor que retira os fardos do armazém e em seguida os consome (deve ser um cavalo). Sendo o armazém um recurso, e os indivíduos processos que

disputam o mesmo recurso, temos a possibilidade de desenvolver algoritmos para resolver o problema deles, já que eles nunca poderão se encontrar dentro do armazém.

Existem diversas soluções para este problema, algumas eficientes, outras um pouco duvidosas ou instáveis, porém todas são interessantes e merecedoras de análise quando se busca solucionar este problema.

2. Solucionado o Problema com Semáforos

Para solucionar o problema do Produtor-Consumidor, optei por utilizar uma solução semelhante à apresentada por Tanenbaum no livro *Sistemas Operacionais Modernos*. Após explicar todos os conceitos necessários, e outras soluções para o mesmo problema, Tanenbaum apresenta uma solução utilizando os *semáforos* propostos por Dijkstra. O algoritmo, apresentado em ANSI C, foi a base para a implementação do software e solução do problema.

2.1. Fim da espera ociosa

Nesta solução é eliminado problema de *espera ociosa*, descrito anteriormente. Através do conceito de *dormir* e *acordar* garante-se que nunca um processo ficará consumindo processamento à toa, ou seja, se quiser utilizar um recurso e não conseguir este processo irá dormir, e só será acordado quando o recurso que ele deseja utilizar ficar disponível novamente. Este controle é adquirido através das operações *down* e *up*, que são executadas pelos processos e alteram o valor do *semáforo binário* responsável pelo recurso em questão.

No nosso caso precisamos de três semáforos, sendo que dois deles são *semáforos contadores* e o terceiro é um *semáforo binário*.

2.2. Semáforos Contadores

Semáforos são variáveis do tipo inteiro que podem assumir valores maiores ou iguais a zero. Um *semáforo contador* representa simplesmente a alocação de determinado recurso, fornecendo a estes uma maneira de conversação que os possibilita sincronizar suas atividades.

Os semáforos *full* e *empty* têm o mesmo propósito, de controlar a sincronização entre os processos. São eles que garantem que o Produtor não irá produzir quando o buffer estiver cheio e que o Consumidor não irá consumir quando o buffer estiver vazio.

Especificamente o semáforo *full* é utilizado como um *semáforo contador* que guarda o número de caixas ocupadas com fardos no armazém. Além disso, ele é responsável por colocar o Consumidor para *dormir*, quando este executar a operação *down* (também proposta por Dijkstra).

A operação *down* consiste somente em decrementar um semáforo qualquer (binário ou contador). O processo que executar esta operação irá verificar o valor do semáforo no qual quer aplicar o *down*. Se o valor do semáforo for maior que zero, o processo irá

decrementá-lo, caso contrário este processo irá *dormir* até que seja acordado por outro processo que execute a operação *up* no mesmo semáforo.

Tendo *empty* o mesmo propósito essencial do *full*, obviamente este será responsável por contar o número de caixas desocupadas no armazém, e conseqüentemente colocar o Produtor para *dormir* até que o Consumidor execute a operação *up* neste semáforo.

Na operação *up* o processo incrementa o semáforo e verifica se existe outro processo dormindo neste semáforo, se existir ele será acordado.

2.3. O Semáforo Binário

A capacidade do sistema operacional de garantir que somente um processo possa utilizar determinado recurso em determinado instante de tempo é denominada *exclusão mútua*. Logo, o correto gerenciamento da *exclusão mútua* garante que nunca dois processos estarão em *regiões críticas* (considerada parte do processo que acessa um recurso, ou o próprio recurso) conflitantes.

O *mutex* (mutual exclusion) é simplesmente o semáforo responsável pela gerência da *exclusão mútua* entre os processos em nesta solução. Este tipo de semáforo é conhecido também por *semáforo binário*, que faz nada mais nada menos que avisar para os processos se determinado recurso já está sendo utilizado. Ele deve ser inicializado com o valor 1 (um), informando que nenhum processo está utilizando o recurso que este semáforo está gerenciando. Quando um processo executar um *down* neste semáforo, o seu valor será alterado para 0 (zero), informando que existe um processo utilizando o recurso (ou está em sua região crítica). Quando o processo sair da *região crítica* o valor será novamente incrementado através da operação *up* e o recurso estará disponível para outro processo.

3. Implementação

A implementação foi feita com a linguagem Java através da IDE Netbeans. Buscando utilizar os recursos da orientação a objeto sem descaracterizar a solução clássica utilizando semáforos, foram criados atributos de classes com os mesmos nomes dos semáforos clássicos: *full*, *empty* e *mutex*

Os *semáforos contadores* foram colocados como atributos de uma classe denominada Armazém e o semáforo binário na classe Sistema, que simboliza o SO, responsável pela gerência da exclusão mútua e escalonamento.

A classe Armazém simboliza o buffer limitado, o recurso que será utilizado pelos processos representados pelas instâncias das classes: Produtor e Consumidor. Tanto Produtor como Consumidor são compostos em determinados instantes de tempo por um objeto Fardo, que também compõe o Armazém.

No diagrama de classes abaixo é possível visualizar de forma clara a estrutura do software, já incluindo a classe responsável pela visualização dos atributos da classe Sistema, que simboliza o SO.

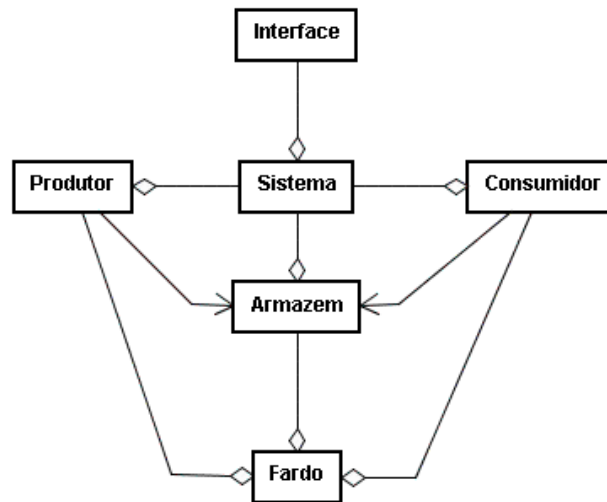


Figura 1. Diagrama de Classes Simplificado

3.1. Classe Armazém

Optou-se por colocar nesta classe os métodos que representam as operações *down* e *up* nos *semáforos contadores* que também encontram-se nesta classe, como mostra a figura abaixo. No algoritmo ANSI C, a função *down* receberia o endereço do semáforo em questão, que por sua vez seria uma variável global. Nesta solução em Java foi dada a classe Armazém a responsabilidade de controlar estes semáforos.

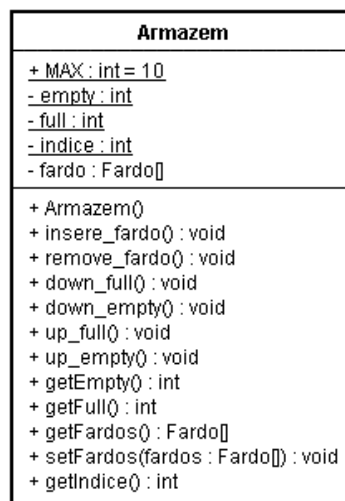


Figura 2. Classe Armazem

Esta figura mostra todos atributos e métodos da classe armazém, assim podemos ver a relação com a classe Fardo através do vetor *fardo*, e os métodos *down_full*, *up_full*, *down_empty* e *up_empty* que garantem a sincronização entre os processos.

3.2. Classe Sistema

Aqui podemos ver a classe principal do software. Nela são escalonados os processos através do método *escalona* e controlada a exclusão mútua através do atributo (semáforo) *mutex*. O método *roda* deveria disparar um loop infinito que executaria incessantemente o método *interrompe* que conforme o resultado dispararia o escalonador, através do método *escalona*. Porém como foi criada uma interface gráfica para visualização dos dados, optou-se por deixar o loop infinito na interface gráfica, que chama o método *Sistema.roda()* após cada atualização gráfica, conservando desta maneira a independência da classe Sistema em relação a interface gráfica.



Figura 3. Classe Sistema

3.3. Classes: Produtor e Consumidor

A classe Produtor possui atributos para controlar todas ações executadas pela instância criada durante a execução do software, entre elas: quantidade de vezes que foi escalonado, que foi interrompido durante a execução, que produziu fardos para estocar, que estocou fardos no armazém e que teve que ficar dormindo. Além disso possui uma seqüência de passos definidos pelas constantes que assumem valores de 1 a 6. Cada passo que executa através do método *anda* é alterado o valor do atributo *proximo_passo*, que guarda o local onde este deverá recomeçar sua execução na próxima vez que for escalonado ou continuar caso não seja interrompido pelo SO.

Os passos são exatamente os mesmos apresentados no livro de Tanenbaum. O passo ENTRAR_NO_ARMAZEM executa o *down* no *mutex* e conseqüentemente SAIR_DO_ARMAZEM executa o *up* no *mutex*. Nesses dois passos o atributo

enter_region do produtor é alterado, simbolizando sua entrada na região crítica e sua saída da mesma.

De forma extremamente similar, embora com algumas funcionalidades inversas, foi implementada a classe Consumidor, logo sua apresentação não é necessária de forma individual.

Ambas possuem também um atributo *armazém* que objetiva o acesso à região crítica dos processos. Todavia não possui o aspecto de agregação e sim de navegação entre estas classes, pois somente um objeto Armazém é instanciado pelo objeto *sistema* instanciado pela classe Interface, e é repassado por referência para o objeto *produtor* e para o objeto *consumidor* que são atributos da classe Sistema. Isso agrega realismo ao software no que se refere ao gerenciamento da exclusão, já que os dois podem acessar um mesmo objeto alocado na memória física em que o software estiver rodando, e se acessarem sem sincronismo o objeto *armazém* (que é compartilhado entre eles) valores inconsistentes seriam gerados e obtidos pelos processos, acarretando numa visão poluída na interface gráfica.

Já o atributo *meu_fardo* é instanciado individualmente por cada um destes objetos. No caso do *produtor* no passo PRODUZIR_FARDO, e no caso do *consumidor* no passo PEGAR_FARDO. Este atributo tem o objetivo de simbolizar a existência ou não de um item (Fardo) com o processo.

Produtor	Consumidor
- produzidos : int - estocados : int - nr_interrupcoes : int - dormindo : boolean - enter_region : boolean - proximo_passo : int - dormidas : int - escalonado : int - PRODUZIR_FARDO : int = 1 - DOWN_EMPTY : int = 2 - ENTRAR_NO_ARMAZEM : int = 3 - ESTOCAR_FARDO : int = 4 - SAIR_DO_ARMAZEM : int = 5 - UP_FULL : int = 6 - armazem : Armazem - meu_fardo : Fardo	- consumidos : int - retirados : int - nr_interrupcoes : int - dormindo : boolean - enter_region : boolean - proximo_passo : int - dormidas : int - escalonado : int - DOWN_FULL : int = 1 - ENTRAR_NO_ARMAZEM : int = 2 - PEGAR_FARDO : int = 3 - SAIR_DO_ARMAZEM : int = 4 - UP_EMPTY : int = 5 - CONSUMIR_FARDO : int = 6 - armazem : Armazem - meu_fardo : Fardo
+ Produtor() + anda() : void + acordado() : boolean + dormindo() : boolean + dormir() : void + acordar() : void + getProduzidos() : int + setProduzidos(produzidos : int) : void + getNr_interrupcoes() : int + setNr_interrupcoes(nr_interrupcoes : int) : void + getArmazem() : Armazem + setArmazem(armazem : Armazem) : void + getEscalonado() : int - produz_fardo() : void - larga_fardo() : void + inEnter_region() : boolean + getEstocados() : int + getMeu_fardo() : Fardo + getDormidas() : int + getStrProximo_passo() : String	+ Consumidor() + anda() : void + acordado() : boolean + dormindo() : boolean + dormir() : void + acordar() : void + getNr_interrupcoes() : int + setNr_interrupcoes(nr_interrupcoes : int) : void + getArmazem() : Armazem + setArmazem(armazem : Armazem) : void - pega_fardo() : void - consome_fardo() : void + inEnter_region() : boolean + getConsumidos() : int + setConsumidos(consumidos : int) : void + getRetirados() : int + getEscalonado() : int + getMeu_fardo() : Fardo + getDormidas() : int + getStrProximo_passo() : String

Figura 4. Classe Produtor e Classe Consumidor.

3.4. Outras classes

A classe Fardo é somente uma maneira de simbolizar um item, sem utilizar tipos de dados existentes. Esta classe não possui nenhum atributo ou método.

A classe Interface possui diversos atributos do pacote Swing com objetivo somente de mostrar os atributos do atributo *sistema*.

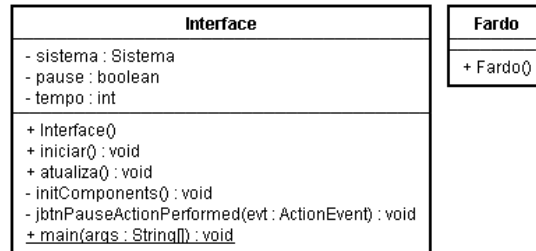


Figura 4. Classe Interface e Classe Fardo.

4. Considerações Finais

Através da implementação foi possível verificar e validar a solução clássica utilizando semáforos. Nesta é eliminado o problema da espera ociosa e eficientemente gerenciada a exclusão mútua, além de favorecer o desempenho do processamento através da sincronização entre os processos.

Durante a implementação surgiu incerteza em relação a qual variável utilizar para manipulação dos índices do buffer. Inicialmente foi utilizado o semáforo **full**, mas desta forma não era possível obter consistência no processamento, já que os semáforos contadores nesta solução são alterados fora da região crítica, permitindo que a soma de **empty** e **full** possa assumir valores de N-2 a N sendo N o tamanho do buffer. Aparentemente seria um erro na solução, porém foi verificado que a função destes semáforos é somente controlar a sincronização entre os processos, fazendo-os dormir ou acordar quando preciso. Portanto é necessário utilizar uma outra variável inteira para determinar qual índice do buffer deverá ser manipulado.

Logo, através desta implementação foi possível compreender a real funcionalidade dos semáforos, e como, de maneira simples, é possível resolver problemas de extrema importância computacional.

5. Referência

Tanembaum, Andrew S.

Sistemas operacionais modernos / Andrew S. Tanembaum ; tradução Ronaldo A.L. Gonçalves, Luís A. Consularo; revisão técnica Regina Borges de Araújo. 2 ed. São Paulo : Prentice Hall, 2003.

Notas de aula de Sistemas Operacionais I - Prof. Dr. Bráulio Mello.