Sistemas Operacionais

Aula 09

Sincronização e Comunicação entre Processos – SCC5854

Capítulo 7 (MACHADO; MAIA, 2014)

Prof. Dr. Jonathan Ramos

jonathan@unir.br

Departamento Acadêmico de Ciências de Computação – DACC

Núcleo de Tecnologia – NT

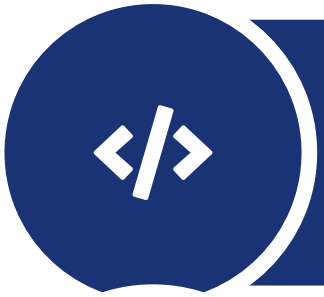
29/11/2022

Sumário

;

Sumário

Introdução



Aplicações concorrentes:
Podem rodar com 1 ou mais *cores* usando processos e *threads*.

Em sistemas MultiCore:
Há maior paralelismo na execução de instruções, o que estende as vantagens que a programação concorrente proporciona.

Porém:
Processos compartilham recursos do sistema, o que pode ocasionar situações indesejáveis e comprometer a execução!

Soluções:
Os processos precisam ter execução sincronizadas com mecanismos do SO: semáforos e monitores.

Sumário

Aplicações Concorrentes

Frequentemente os processos precisam se comunicar entre si:

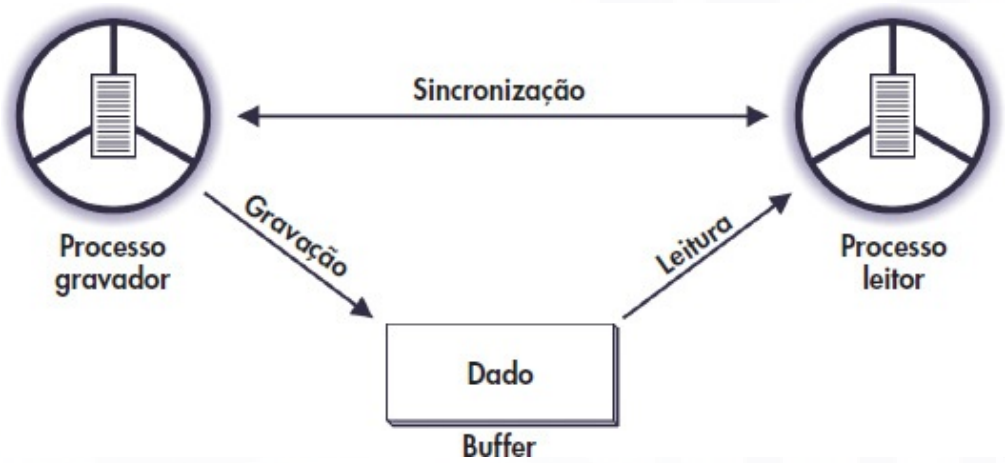


Figura: **Sincronização e comunicação entre processos: exemplo em que dois processos compartilham um *buffer*.** (1) **Um processo só poderá gravar dados no *buffer* caso este não esteja cheio**. (2) **Um processo só poderá ler dados armazenados do *buffer* caso exista dado para ser lido**.

Aplicações Concorrentes

Frequentemente os processos precisam se comunicar entre si:

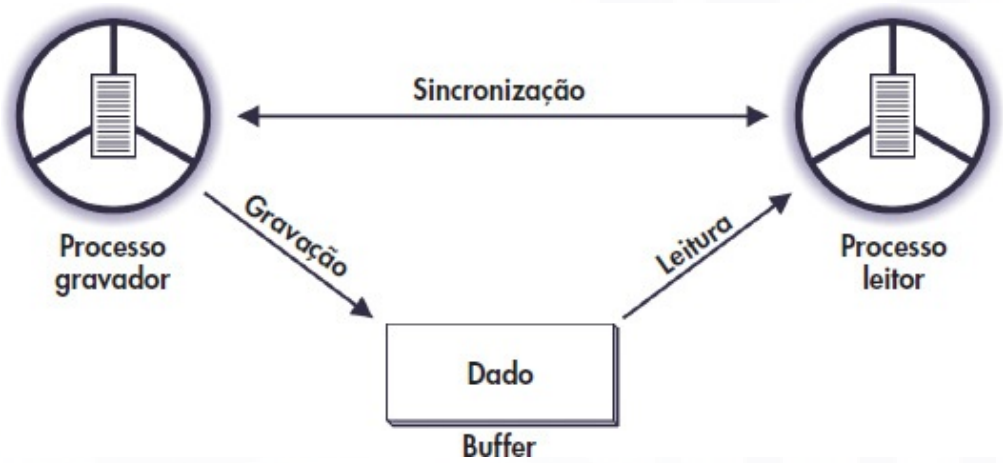


Figura: **Sincronização e comunicação entre processos: exemplo em que dois processos compartilham um *buffer*.** (1) **Um processo só poderá gravar dados no *buffer* caso este não esteja cheio**. (2) **Um processo só poderá ler dados armazenados do *buffer* caso exista dado para ser lido**.

Em ambas as situações:

Os processos deverão aguardar até que o *buffer* esteja pronto para as operações, seja de **gravação** ou de **leitura**.

Aplicações Concorrentes

Frequentemente os processos precisam se comunicar entre si:

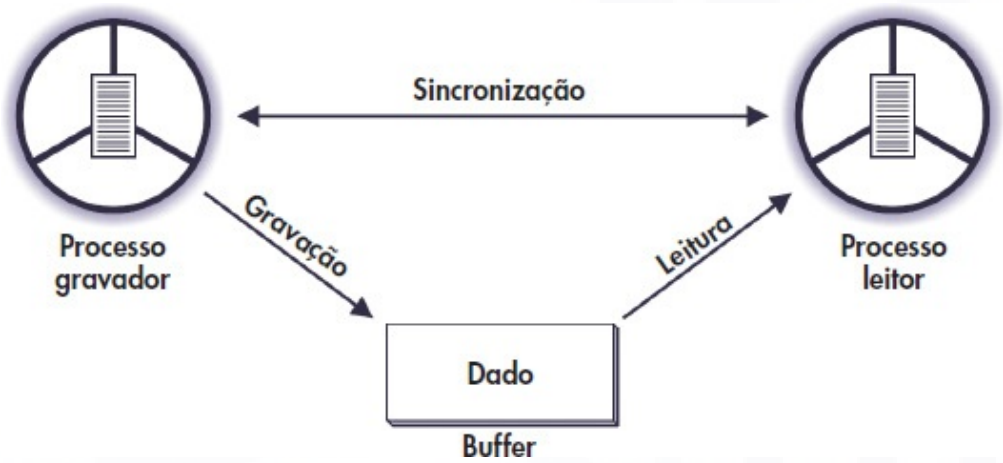


Figura: **Sincronização e comunicação entre processos: exemplo em que dois processos compartilham um *buffer*.** (1) **Um processo só poderá gravar dados no *buffer* caso este não esteja cheio**. (2) **Um processo só poderá ler dados armazenados do *buffer* caso exista dado para ser lido**.

Em ambas as situações:

Os processos deverão aguardar até que o *buffer* esteja pronto para as operações, seja de **gravação** ou de **leitura**.

Mecanismos de sincronização:

É o que **garante a comunicação entre processos concorrentes** e o acesso a **recursos compartilhados**

Aplicações Concorrentes

Frequentemente os processos precisam se comunicar entre si:

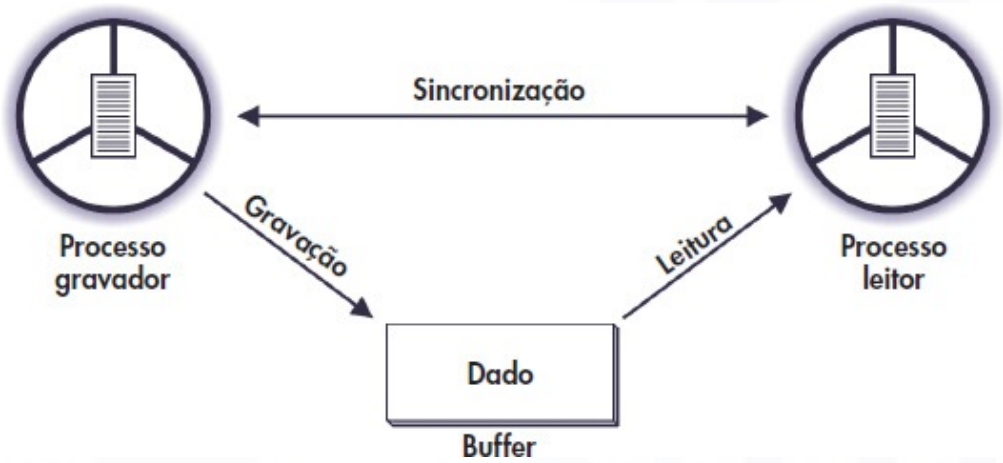


Figura: **Sincronização e comunicação entre processos: exemplo em que dois processos compartilham um *buffer*.** (1) **Um processo só poderá gravar dados no *buffer* caso este não esteja cheio**. (2) **Um processo só poderá ler dados armazenados do *buffer* caso exista dado para ser lido**.

Em ambas as situações:

Os processos deverão aguardar até que o *buffer* esteja pronto para as operações, seja de **gravação** ou de **leitura**.

Mecanismos de sincronização:

É o que **garante a comunicação entre processos concorrentes** e o acesso a **recursos compartilhados**

Termos equivalentes neste capítulo:

Os termos **subprocesso** e ***thread*** têm o mesmo significado que **processo**.

Sumário

Especificação de Concorrência em Programas

Existem várias notações utilizadas para especificar a concorrência em programas: **Uma das implementações mais claras e simples de expressar concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND (Dijkstra, 1965a).**

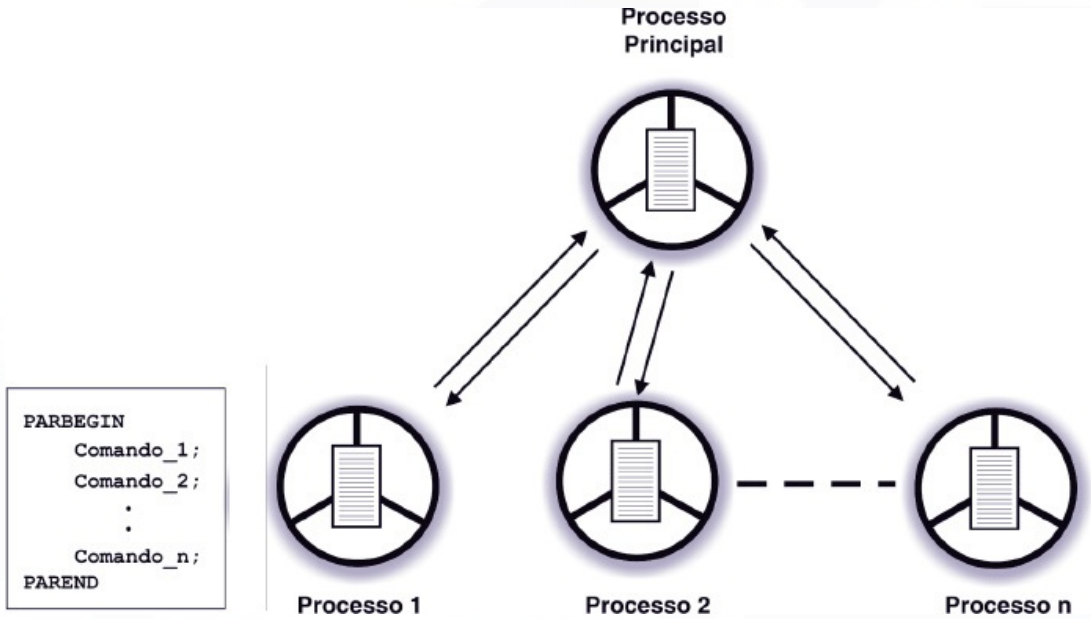


Figura: Concorrência em programas.

PARBEGIN

Especifica que **a sequência de comandos** seja **executada concorrentemente** em uma **ordem imprevisível**

PAREND

Define **um ponto de sincronização**, onde **o processamento só continuará** quando todos os processos **já tiverem terminado suas execuções**

Especificação de Concorrência em Programas

Exemplo:

$$X := \textbf{SQRT(1024)} + \textbf{(35.4 * 0.23)} - \textbf{(302 / 7)} \quad (1)$$

```
1  PROGRAM Expressao;
2  VAR X, Temp1, Temp2, Temp3 : REAL;
3  BEGIN
4  PARBEGIN
5  Temp1 = SQRT (1024);
6  Temp2 = 35.4 * 0.23);
7  Temp3 = (302 / 7)} ;
8  PAREND;
9  X = Temp1 + Temp2      Temp3;
10 WRITELN ('x = ', X);
11 END
```

O cálculo final de X só pode ser realizado quando todas as variáveis dentro da estrutura tiverem sido calculadas.

Sumário

Problemas de Compartilhamento de Recursos: Problema 1

Problema 1:

Compartilhamento de um arquivo em disco: **ContaCorrente** que atualiza o saldo bancário de um cliente após um lançamento de débito ou crédito no arquivo de contas-correntes **ArqContas**

ArqContas

São armazenados os saldos de todos os correntistas do banco

```
1  PROGRAM ContaCorrente;
2  .
3  .
4  READ (ArqContas, RegCliente);
5  READLN (ValorDepRet);
6  RegCliente.Saldo = RegCliente.
       Saldo + ValorDepRet;
7  WRITE (ArqContas, RegCliente);
8  .
9  .
10 END
```

O programa:

1. lê o registro do cliente no arquivo (RegCliente);
2. lê o valor a ser depositado ou retirado (ValorDepRet);
3. atualiza o saldo no arquivo de contas.

Considerando **processos concorrentes pertencentes a dois funcionários** do banco que **atualizam o saldo de um mesmo cliente simultaneamente**, a situação de compartilhamento do recurso pode ser analisada no exemplo da Tabela a seguir

Problemas de Compartilhamento de Recursos: Problema 1

Caixa	Instrução	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

Tabela: Problema de Concorrência I: O processo do primeiro funcionário (Caixa 1) **lê o registro do cliente e soma ao campo Saldo o valor do lançamento de débito**. **Antes de gravar o novo saldo no arquivo, o processo do segundo funcionário (Caixa 2) lê o registro do mesmo cliente**, **que está sendo atualizado**, para realizar outro lançamento, desta vez de crédito.

Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

Problemas de Compartilhamento de Recursos: Problema 2

Dois processos (A e B) executam um comando de atribuição

- **Inicialmente, a variável X possui o valor 2.**
- **O Processo A soma 1 à variável X;**
- **o Processo B diminui 1 da mesma variável que está sendo compartilhada**

Processo A

$X = X + 1;$

Processo B

$X = X \ \ 1;$

No final, o valor de X fica 2?

Problemas de Compartilhamento de Recursos: Problema 2

Dois processos (A e B) executam um comando de atribuição

- **Inicialmente, a variável X possui o valor 2.**
- **O Processo A soma 1 à variável X;**
- **o Processo B diminui 1 da mesma variável que está sendo compartilhada**

Processo A

$X = X + 1;$

Processo B

$X = X \ \ 1;$

No final, o valor de X fica 2?

Nem Sempre!

Os comandos de atribuição, em uma linguagem de alto nível, podem ser **decompostos em comandos mais elementares**:

Processo A	Processo B
LOAD x, R_a	LOAD x,R_b
ADD1,R_a	SUB1,R_b
STORE R_a,x	STORE R_b,x

Se o **Processo A carrega x**, **é interrompido**, **o Processo B carrega x, executa subtração e armazena o valor 1 em x**, em seguida, **o Processo A retoma processamento e salva o valor 3**.

Problemas de Compartilhamento de Recursos: Problema 2

Veremos passo a passo:

Proc.	Instrução	X	R_a	R_b
A	LOAD X, R_a	2	2	*
A	ADD 1, R_a	2	3	*
B	LOAD X, R_b	2	*	2
B	SUB 1, R_b	2	*	1
A	STORE R_a,X	3	3	*
B	STORE R_b,X	1	*	1

Tabela: Problema de Concorrência II.

1. O **Processo A carrega x no Registrador A e adiciona 1 no valor do Registrador A** (3);
2. O processo A **é interrompido**;
3. **O Processo B carrega x no Registrador B, executa subtração de 1 no Registrador B (1)**;
4. **O Processo A retoma processamento e salva o valor do Registrador A em X (3)**.
5. **O Processo B retoma processamento e salva o valor do Registrador B em X (1)**.

Problemas de Compartilhamento de Recursos

Analisando os dois exemplos apresentados:

Concluímos:

Em qualquer situação, onde dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esses tipos de problemas, conhecidos como condições de corrida (*race conditions*).

Sumário

Exclusão Mútua

Evita que dois ou mais processos usem o mesmo recurso simultaneamente:

Quando um processo **estiver usando um recurso**, **todos os demais processos concorrentes devem esperar para usar o recurso**. **Exemplo da vida real?**

Exclusão Mútua

Evita que dois ou mais processos usem o mesmo recurso simultaneamente:

Quando um processo **estiver usando um recurso**, **todos os demais processos concorrentes devem esperar para usar o recurso**. **Exemplo da vida real?**

Região crítica (*critical region*):

Este conceito **só se aplica quando os processos estiverem usando o recurso compartilhado**, as demais partes do programa não dependentes do recurso podem executar normalmente.

Exclusão Mútua

Evita que dois ou mais processos usem o mesmo recurso simultaneamente:

Quando um processo **estiver usando um recurso**, **todos os demais processos concorrentes devem esperar para usar o recurso**. **Exemplo da vida real?**

Região crítica (*critical region*):

Este conceito **só se aplica quando os processos estiverem usando o recurso compartilhado**, as demais partes do programa não dependentes do recurso podem executar normalmente.

O ideal é

Evitar que dois processos **entrem em suas regiões críticas** ao mesmo tempo, **evitando problemas decorrentes do compartilhamento**

Ainda assim, algumas situações indesejadas podem acontecer...

Exclusão Mútua: Problema 1

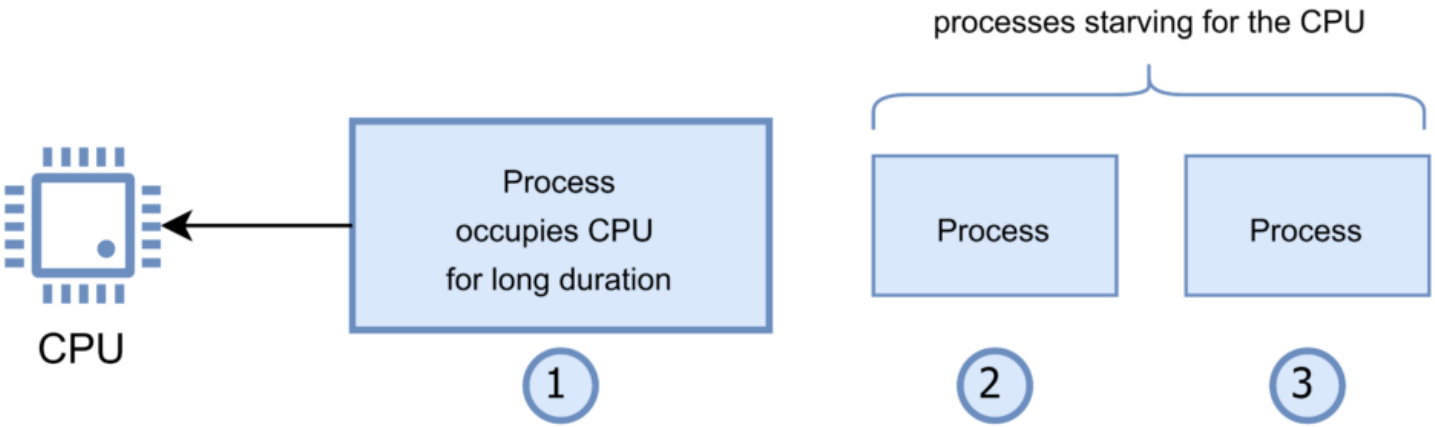


Figura: *Starvation*.

Starvation ou espera indefinida pela região crítica:

Um processo nunca consegue executar sua **região crítica** e, consequentemente, **acessar o recurso compartilhado:**

- O SO possui um **sistema de prioridades** para definir a **prioridade de acesso ao recurso**;
- **um processo em baixíssima prioridade pode ficar esperando indefinidamente**.
- **Fila de espera? Resolve? Consegue pensar em algum problema que poderia acontecer?**

Exclusão Mútua: Problema 2

Um processo diz que tá usando o recurso mas não está

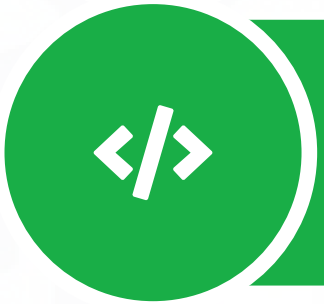
Por exemplo, a garrafinha ou celular no equipamento da academia ou uma bíblia no banco da igreja (marcando lugar):

- No caso de esta situação ocorrer, **um recurso estaria livre, porém alocado a um processo**:
- Com isso, **vários processos estariam sendo impedidos de utilizar o recurso**, reduzindo o grau de compartilhamento.

Exclusão Mútua: Problema 2

Ver Seções 7.5.1 e 7.5.2:

7.5.1 – Soluções de Hardware:
1) Desabilitação de interrupções; 2) Instrução test-and-set;



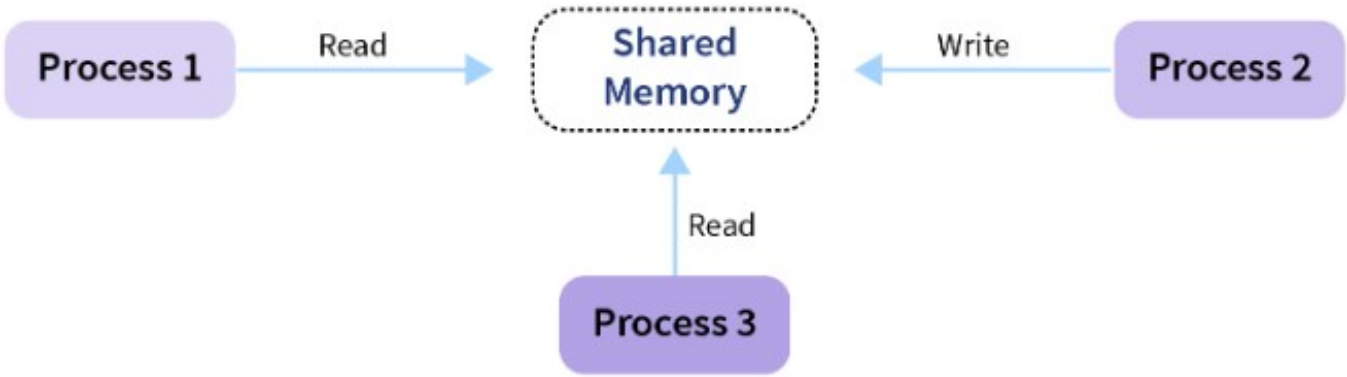
7.5.2 – Soluções de Software:
1) Algoritmos 1, 2, 3, 4, 5, de Dekker, e de Peterson; 2) Algoritmo para exclusão mútua entre N processos

Sumário

Sincronização Condicional

Sincronização condicional

O acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso.



Processos produtores

Geram informações: **Escrita**.

Processos consumidores

Usam as informações: **Leitura**

Os processos envolvidos devem estar sincronizados a uma variável de condição. (Um processo não pode tentar gravar dados em um *buffer* cheio ou realizar uma leitura em um *buffer* vazio)

Sincronização Condicional

Problema do produtor/consumidor: *bounded buffer*

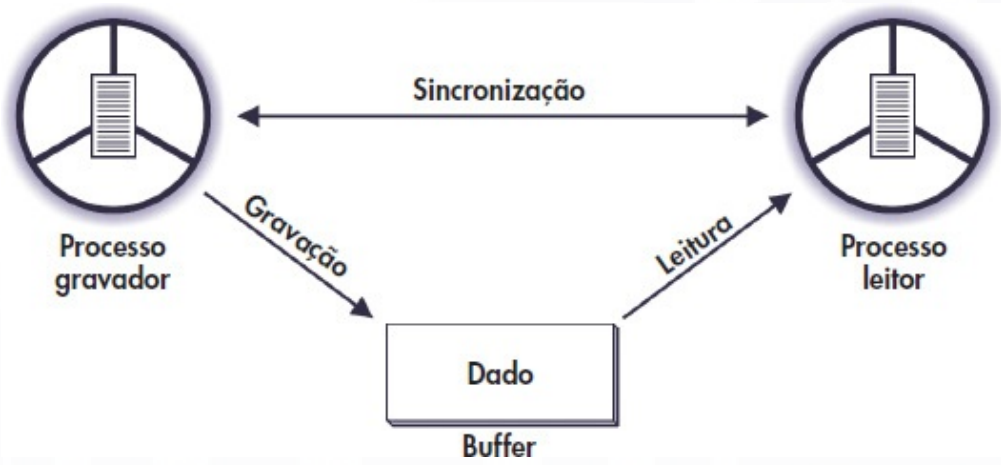


Figura: Bounded Buffer.

Definimos:

- **TamBuff**: Tamanho total;
- **Cont**: Controla o acesso.

Leitura

- Somente quando **Cont** > 0.
- Se **Cont** $= 0$, deve aguardar uma gravação para ler.

Gravação

- Somente quando **Cont** $<$ **TamBuff**.
- Se **Cont** $=$ **TamBuff**, deve aguardar uma leitura para gravar.

Sincronização Condicional

Problema do produtor/consumidor: *bounded buffer*

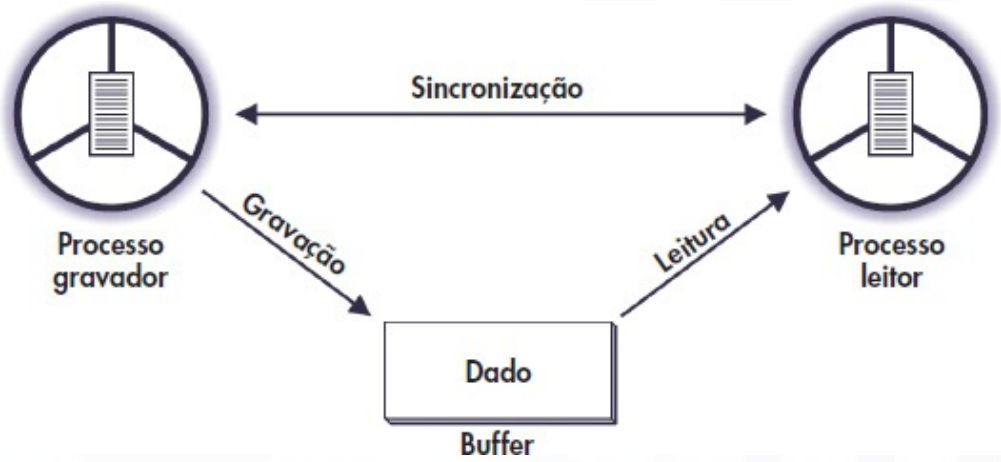


Figura: Bounded Buffer.

Problema: a variável Cont pode ser alterada de forma assíncrona, gerando inconsistência.

Solução: Mecanismos de sincronização semáforos e monitores

Definimos:

- **TamBuff**: Tamanho total;
- **Cont**: Controla o acesso.

Leitura

- Somente quando **Cont** > 0.
- Se **Cont** $= 0$, deve aguardar uma gravação para ler.

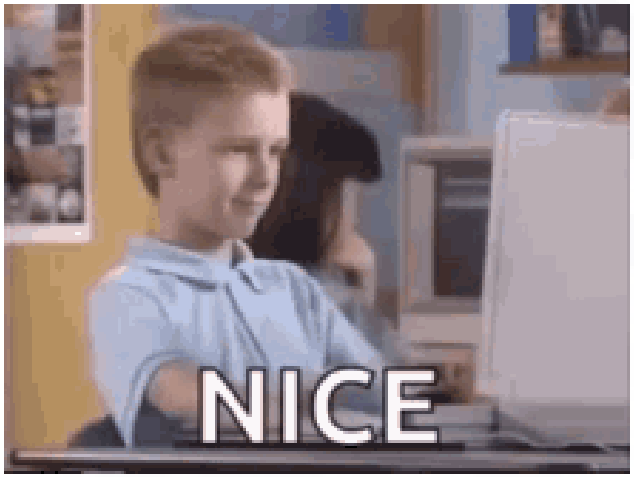
Gravação

- Somente quando **Cont** $<$ **TamBuff**.
- Se **Cont** $=$ **TamBuff**, deve aguardar uma leitura para gravar.

Sumário

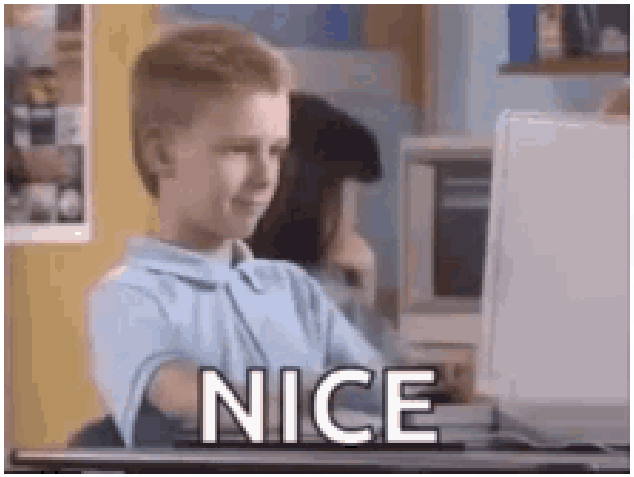
Semáforos: Proposto por Dijkstra (DIJKSTRA, 1965)

Permite a exclusão mútua e a sincronização condicional entre processos



Semáforos: Proposto por Dijkstra (DIJKSTRA, 1965)

Permite a exclusão mútua e a sincronização condicional entre processos

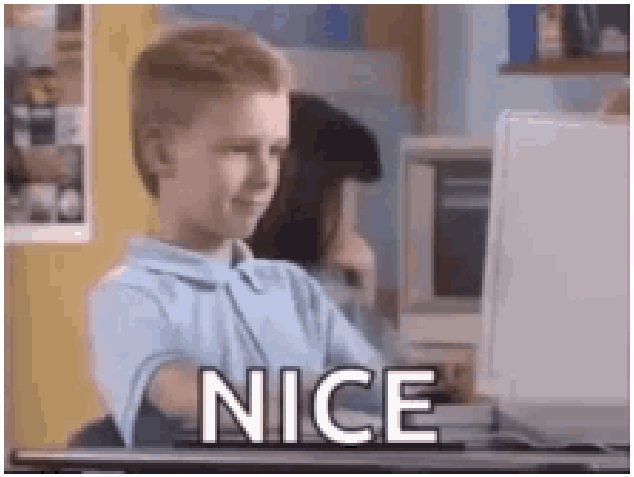


Um semáforo:

- É uma variável **inteira, não negativa**, que só pode ser manipulada por duas instruções:
  - **UP**: incrementa 1 ao valor do semáforo;
  - **DOWN**: decrementa 1
- São **indivisíveis**, ou seja, a execução destas instruções **não pode ser interrompida**

Semáforos: Proposto por Dijkstra (DIJKSTRA, 1965)

Permite a exclusão mútua e a sincronização condicional entre processos



Um semáforo:

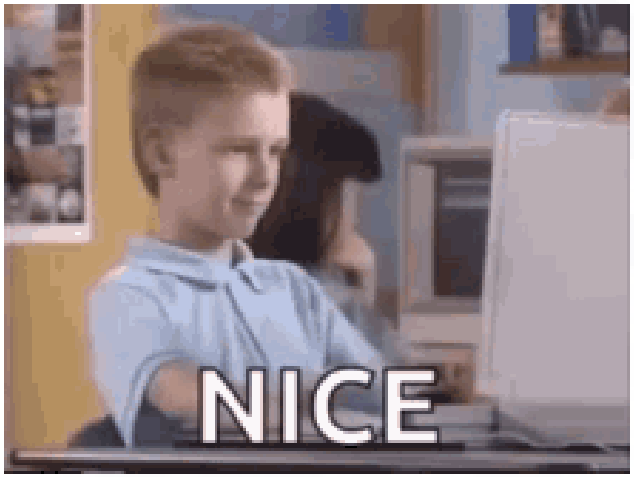
- É uma variável **inteira, não negativa**, que só pode ser manipulada por duas instruções:
  - **UP**: incrementa 1 ao valor do semáforo;
  - **DOWN**: decrementa 1
- São **indivisíveis**, ou seja, a execução destas instruções **não pode ser interrompida**

DOWN em 0:

Valores negativos não podem ser atribuídos a um semáforo: a instrução DOWN em 0 faz com que o processo entre no estado de espera

Semáforos: Proposto por Dijkstra (DIJKSTRA, 1965)

Permite a exclusão mútua e a sincronização condicional entre processos



Um semáforo:

- É uma variável **inteira, não negativa**, que só pode ser manipulada por duas instruções:
  - **UP**: incrementa 1 ao valor do semáforo;
  - **DOWN**: decrementa 1
- São **indivisíveis**, ou seja, a execução destas instruções **não pode ser interrompida**

DOWN em 0:

Valores negativos não podem ser atribuídos a um semáforo: a instrução DOWN em 0 faz com que o processo entre no estado de espera

Classificação dos semáforos:

- **Binário**: *mutexes* (*mutual exclusion semaphores*), **só podem ter os valores 0 e 1**;
- **Contadores**: podem ter **qualquer valor inteiro positivo**, além do 0.

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:

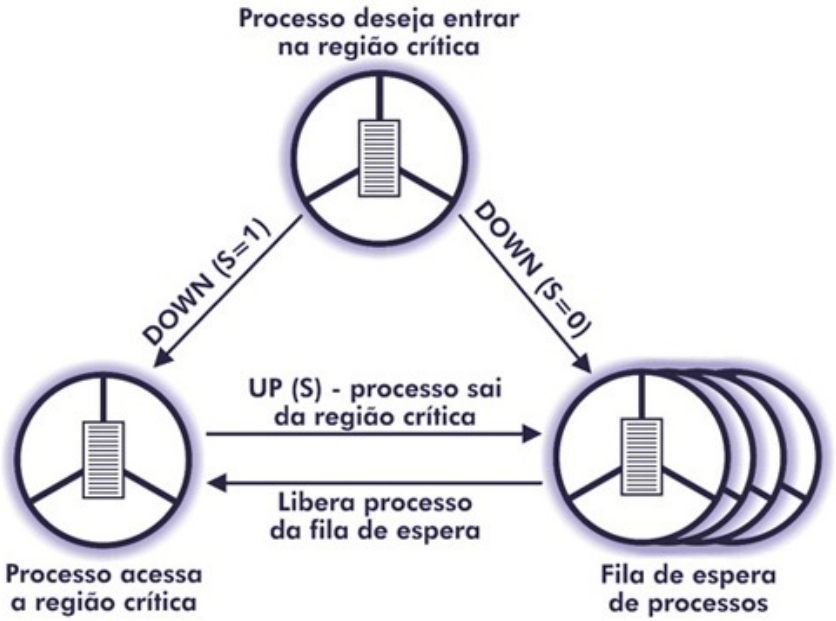
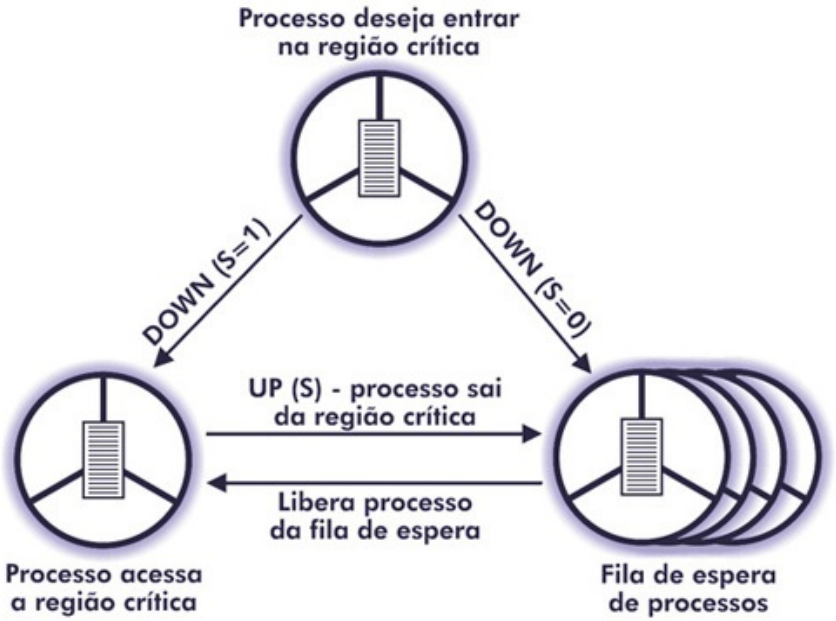


Figura: *Uso do semáforo binário na exclusão mútua.*

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:



Vantagem:

Não ocorrência da **espera ocupada**

Figura: *Uso do semáforo binário na exclusão mútua.*

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:

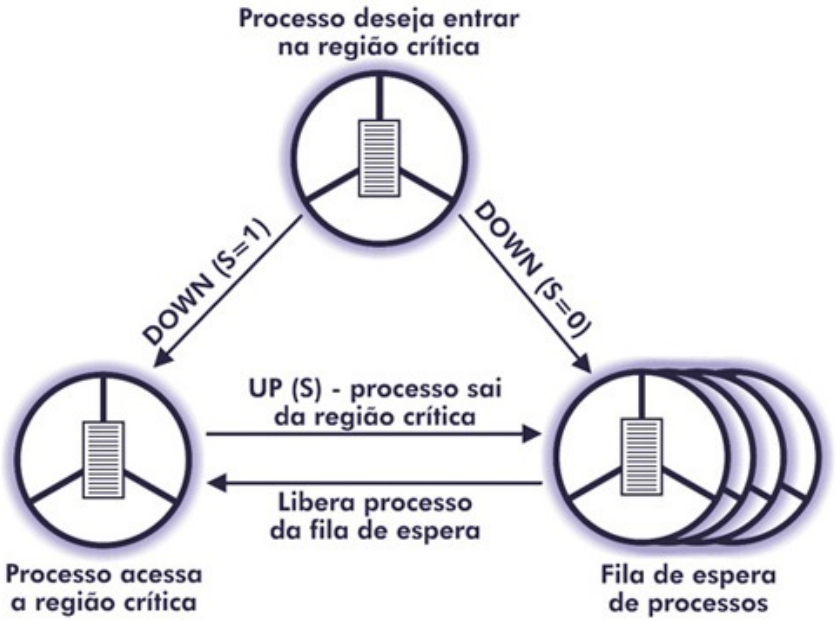


Figura: *Uso do semáforo binário na exclusão mútua.*

Vantagem:

Não ocorrência da **espera ocupada**

DOWN e UP:

Funcionam como protocolos de entrada e saída para que **o processo entre e saia da sua região crítica**.

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:

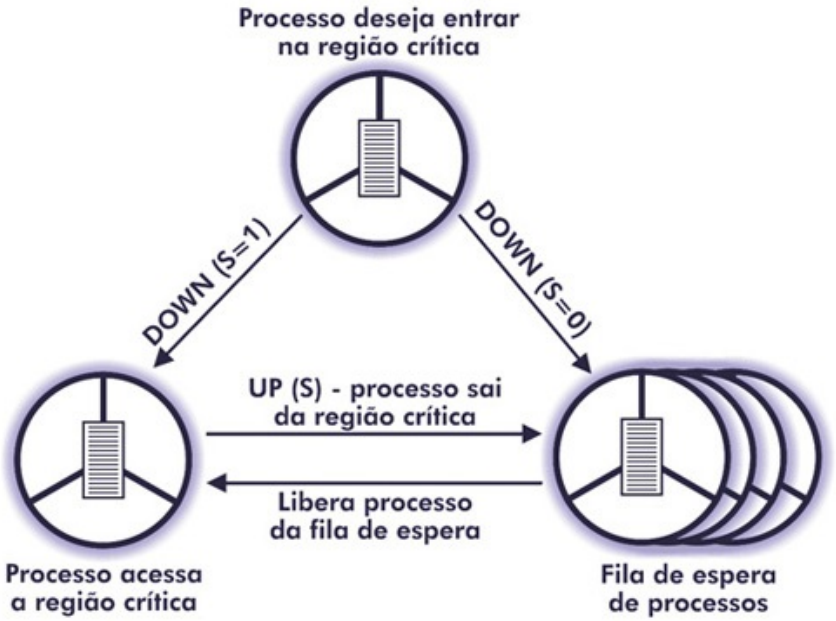


Figura: *Uso do semáforo binário na exclusão mútua.*

Semáforo:

Fica associado a um recurso compartilhado:

- 1: nenhum processo está utilizando o recurso;
- 0: o recurso está em uso.

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:

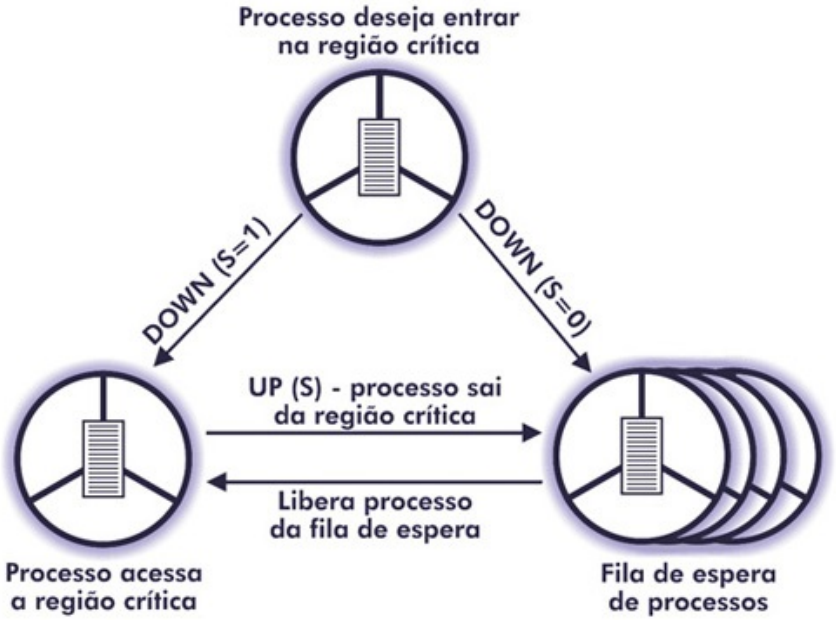


Figura: *Uso do semáforo binário na exclusão mútua.*

Quando vai entrar na região crítica:

Um processo executa uma instrução DOWN:

- Semáforo = 1, este valor é decrementado, e o processo que solicitou a operação pode executar as instruções da sua região crítica;
- Semáforo = 0, o processo fica impedido do acesso, permanecendo em estado de espera e, não gerando *overhead* no processador

Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado:

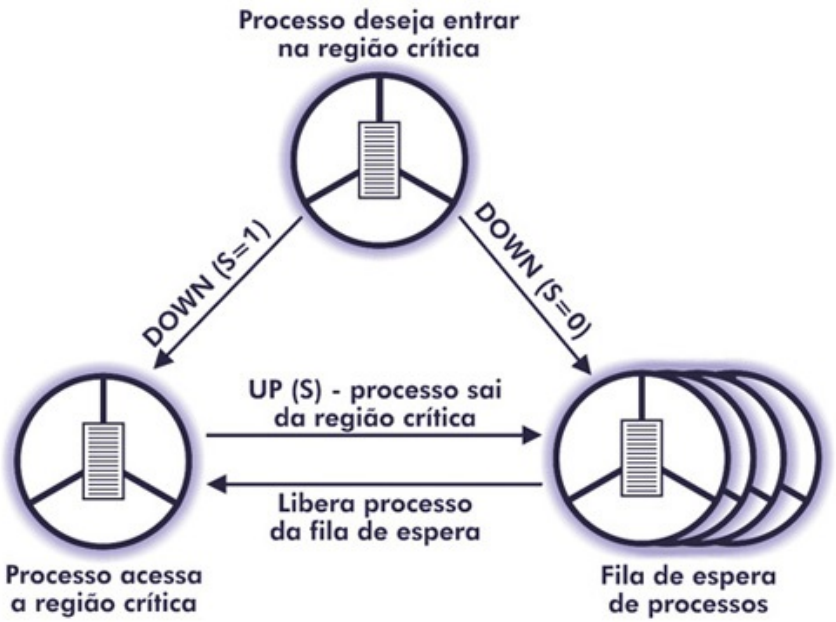


Figura: *Uso do semáforo binário na exclusão mútua.*

Quando vai entrar na região crítica:

Um processo executa uma instrução DOWN:

- Semáforo = 1, este valor é decrementado, e o processo que solicitou a operação pode executar as instruções da sua região crítica;
- Semáforo = 0, o processo fica impedido do acesso, permanecendo em estado de espera e, não gerando *overhead* no processador

Ao sair da região crítica:

Executa uma instrução UP e libera o acesso ao recurso;

- Se um ou mais processos estiverem esperando pelo uso do recurso (operações DOWN pendentes), o SO selecionará um processo na fila de espera associada ao recurso e alterará o seu estado para pronto

Sincronização Condicional Utilizando Semáforos

Vamos exemplificar com um a sincronização de uma operação de E/S:
PPara o exemplo do ProdutoConsumidor:

Pode ser aplicado o mesmo conceito anterior

Porém, com vários semáforos:

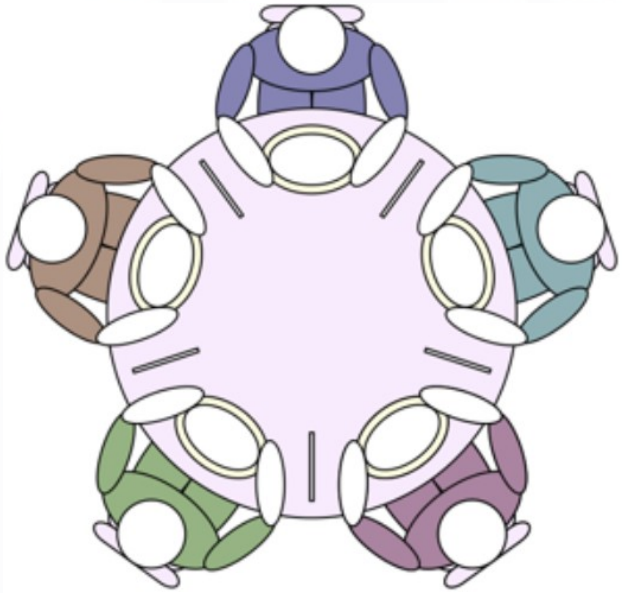
1. um binário para a exclusão mútua;
2. dois semáforos contadores para a sincronização condicional.

Ver código exemplo da Seção 7.7.2.

Problema dos Filósofos: exemplo clássico de sincronização de processos

Contexto

1. **Cinco filósofos estão sentados em uma mesa** redonda para jantar;
2. Cada filósofo tem um prato **com espaguete à sua frente**;
3. **Cada prato possui um garfo** para pegar o espaguete;
4. O espaguete está muito escorregadio e, para que um filósofo consiga comer, **será necessário utilizar dois garfos**.



Você é capaz de propor um algoritmo que implemente cada filósofo de modo que ele execute as tarefas de comer e pensar sem nunca ficar travado?

Figura: Jantar dos filósofos.

Problema dos Filósofos: exemplo clássico de sincronização de processos

Contexto

1. **Cinco filósofos estão sentados em uma mesa** redonda para jantar;
2. Cada filósofo tem um prato **com espaguete à sua frente**;
3. **Cada prato possui um garfo** para pegar o espaguete;
4. O espaguete está muito escorregadio e, para que um filósofo consiga comer, **será necessário utilizar dois garfos**.

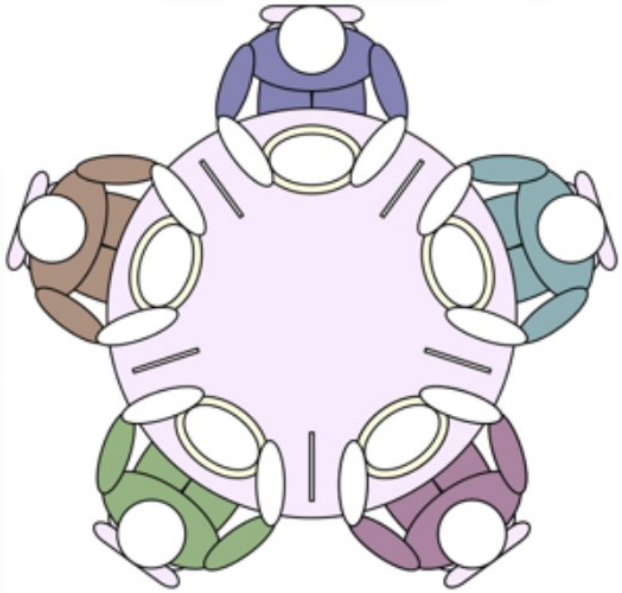


Figura: Jantar dos filósofos.

Você é capaz de propor um algoritmo que implemente cada filósofo de modo que ele execute as tarefas de comer e pensar sem nunca ficar travado?

Deadlock

Se todos os filósofos estiverem segurando apenas um garfo cada um, nenhum filosófo conseguirá comer.

Problema dos Filósofos: exemplo clássico de sincronização de processos

Existem várias soluções para resolver o problema dos filósofos sem *deadlock*:

1. Permitir que apenas quatro filósofos sentem à mesa simultaneamente;
2. Permitir que um filósofo pegue um garfo apenas se o outro estiver disponível;
3. Permitir que um filósofo ímpar pegue primeiro o seu garfo da esquerda e depois o da direita, enquanto um filósofo par pegue o garfo da direita e, em seguida, o da esquerda.

Problema do Barbeiro: outro exemplo clássico de sincronização de processos

Contexto:

1. um barbeiro **recebe clientes para cortar o cabelo**;
2. Na barbearia **há uma cadeira de barbeiro e apenas cinco cadeiras para clientes** esperarem;
3. Quando um cliente chega, **caso o barbeiro esteja trabalhando ele se senta**, **se houver cadeira vazia**, **ou vai embora, se todas as cadeiras estiverem ocupadas**;
4. No caso de o barbeiro **não ter nenhum cliente para atender**, **ele se senta na cadeira e dorme** até que um novo cliente apareça.

Figura: Problema do barbeiro.

Problema do Barbeiro: outro exemplo clássico de sincronização de processos

Contexto:

1. um barbeiro **recebe clientes para cortar o cabelo**;
2. Na barbearia **há uma cadeira de barbeiro e apenas cinco cadeiras para clientes** esperarem;
3. Quando um cliente chega, **caso o barbeiro esteja trabalhando ele se senta**, **se houver cadeira vazia**, **ou vai embora, se todas as cadeiras estiverem ocupadas**;
4. No caso de o barbeiro **não ter nenhum cliente para atender**, **ele se senta na cadeira e dorme** até que um novo cliente apareça.

Figura: Problema do barbeiro.

Solução:

Ver seção 7.7.4 do livro.

Sumário

Monitores

São mecanismos de sincronização de alto nível e simples:

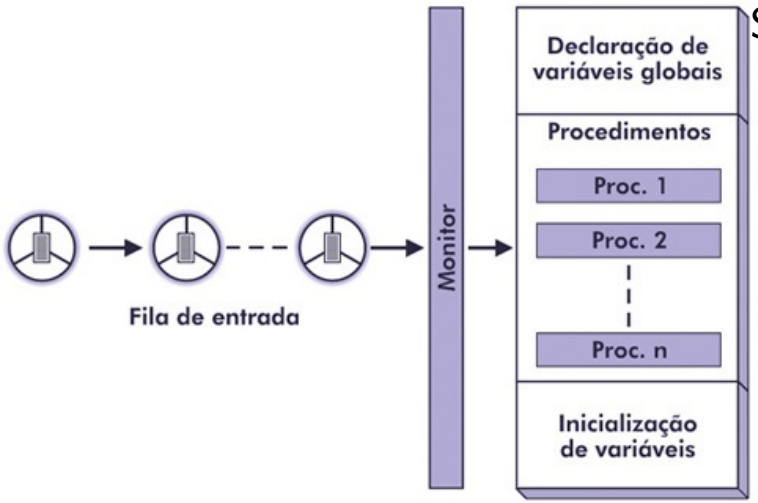


Figura: *Estrutura do monitor.*

Semáforos: Exige do desenvolvedor bastante cuidado, pois

Monitores:

- são considerados mecanismos de alto nível;
- estruturados em função de serem implementados pelo compilador;
- o desenvolvimento de programas concorrentes fica mais fácil;
- as chances de erro são menores.

É considerado um mecanismo estruturado, enquanto semáforos são considerados não estruturados.

Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de monitores.

Monitores

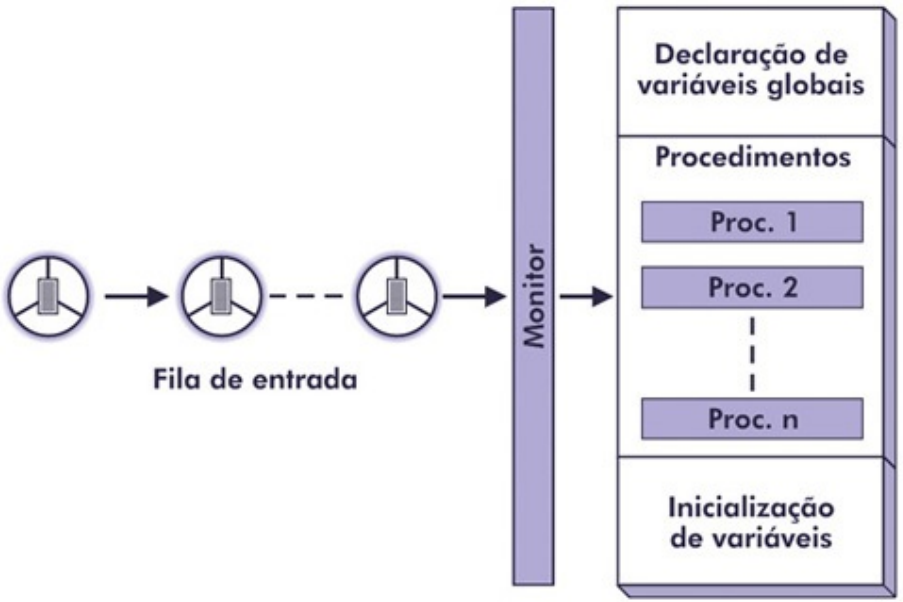


Figura: *Estrutura do monitor: é formado por procedimentos e variáveis encapsulados dentro de um módulo.*

Parte importante:

A implementação automática da exclusão mútua entre os procedimentos declarados:

- somente um processo pode estar executando um dos procedimentos do monitor em um determinado instante

Chamada a recurso compartilhado:

o monitor verifica se já existe outro processo executando algum procedimento do monitor;

- Caso exista, o processo ficará aguardando a sua vez em uma fila de entrada;

Variáveis globais:

São visíveis apenas aos procedimentos da sua estrutura:

- inacessíveis fora do contexto do monitor.

Inicialização das variáveis:

é realizada por um bloco de comandos do monitor:

- Executado apenas uma vez, na ativação do programa onde está declarado o monitor.

Exclusão Mútua Utilizando Monitores

Ver seção 8.8.1:

```
1  PROGRAM Monitor_1;
2      MONITOR Regiao_Critica;
3      VAR X : INTEGER;
4  PROCEDURE Soma;
5      BEGIN
6          X := X + 1;
7      END;
8          PROCEDURE Diminui;
9      BEGIN
10         X := X      1;
11     END;
12     BEGIN
13         X := 0;
14     END;
15 BEGIN
16     PARBEGIN
17         Regiao_Critica.Soma;
18         Regiao_Critica.Diminui;
19     PAREND;
20 END.
```

Sincronização Condicional Utilizando Monitores

Ver seção 8.8.2:

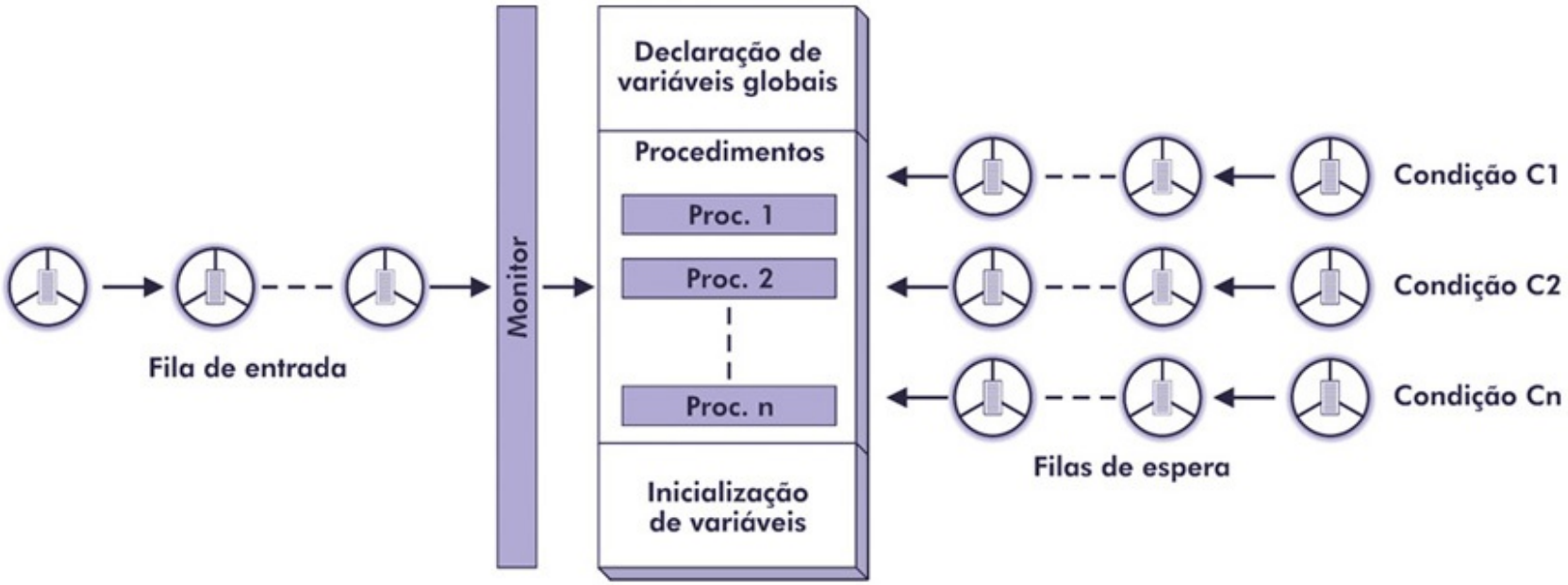


Figura: *Estrutura do monitor com variáveis de condição.*

Sincronização Condicional Utilizando Monitores

Ver seção 8.8.2:

```
1  MONITOR Condicional;
2  VAR Cheio, Vazio : (* Variaveis especiais de condicao *);
3  PROCEDURE Produz;
4  BEGIN
5  IF (Cont = TamBuf) THEN WAIT (Cheio);
6  .
7  .
8  IF (Cont = 1) THEN SIGNAL (Vazio);
9  END;
10 PROCEDURE Consome;
11 BEGIN
12 IF (Cont = 0) THEN WAIT (Vazio);
13 .
14 .
15 IF (Cont = TamBuf      1) THEN SIGNAL (Cheio);
16 END;
17 BEGIN
18 END;
```

Sumário

Deadlock

***Deadlock* é a situação em que um processo aguarda por um recurso que nunca estará disponível:**

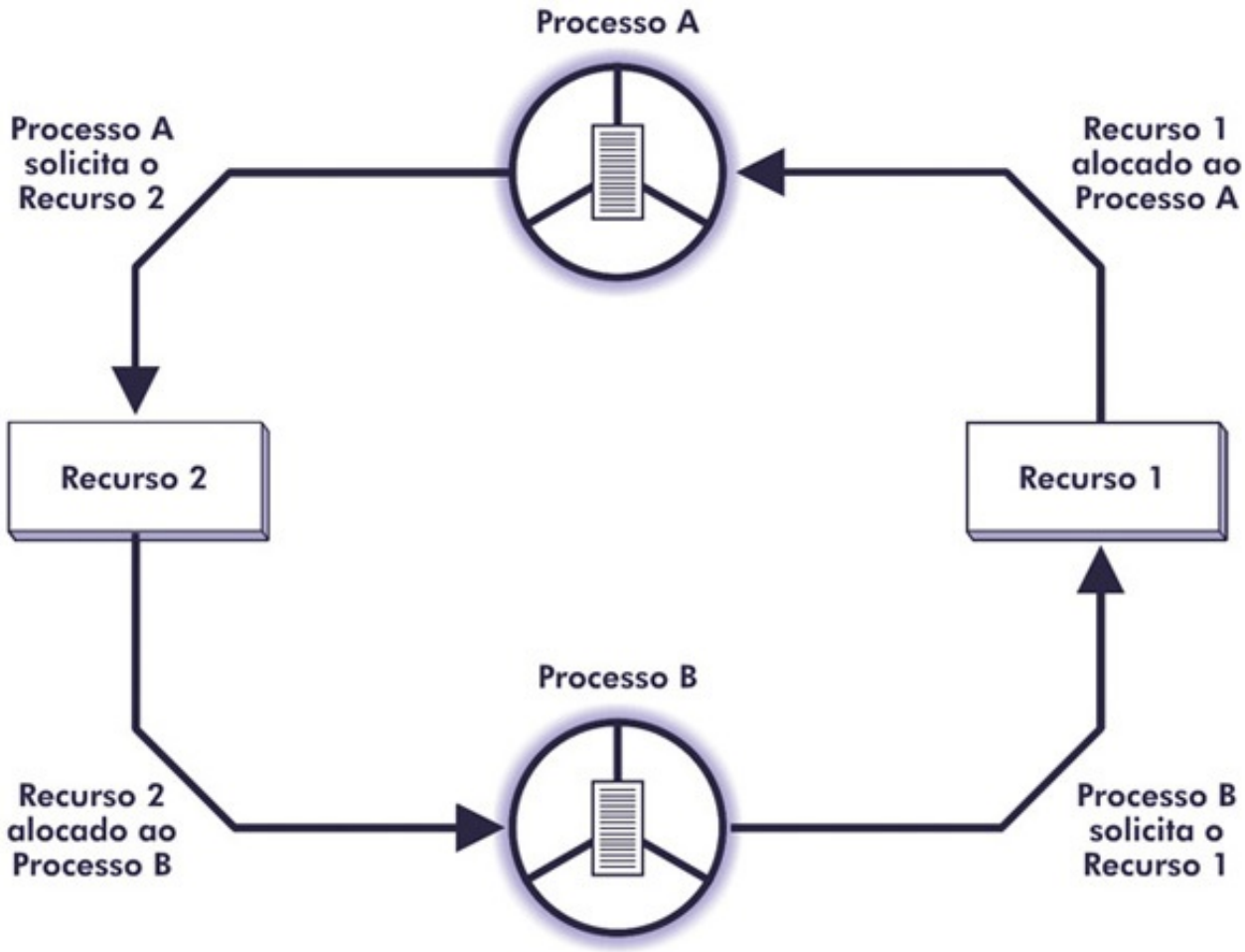


Figura: *Espera circular*.

Deadlock

Para que ocorra a situação de *deadlock*, quatro condições são necessárias simultaneamente:

1. **Exclusão mútua**: cada recurso **só pode estar alocado a um único processo** em um determinado instante;
2. **Espera por recurso**: um processo, além dos recursos já alocados, **pode estar esperando por outros recursos**;
3. **Não preempção**: um recurso **não pode ser liberado de um processo** só porque outros processos desejam o mesmo recurso;
4. **Espera circular**: um processo pode **ter de esperar por um recurso** alocado a outro processo, e viceversa.

Prevenção de *Deadlock*

Garantir que nenhuma das 4 situações anteriores aconteça:

Figura: *Exemplo de deadlock no trânsito.*

O que na prática não é possível...

Detecção do *Deadlock*

Os SOs:

Devem manter estruturas de dados que **identificam cada recurso do sistema**:

- **o processo que o está alocando** e os **processos que estão à espera da liberação do recurso**

Os algoritmos que implementam esse mecanismo:

Geralmente, **verificam a existência da espera circular**:

- **Percorrendo toda a estrutura** sempre que **um processo solicita um recurso** e **ele não pode ser imediatamente garantido**.

Correção do *Deadlock*

Após a detecção do *deadlock*:

O SO pode:

1. Eliminar um ou mais processos envolvidos no *deadlock* e desalocar os recursos já garantidos por eles

Correção do *Deadlock*

Após a detecção do *deadlock*:

O SO pode:

1. Eliminar um ou mais processos envolvidos no *deadlock* e desalocar os recursos já garantidos por eles

Porém:

A eliminação dos processos envolvidos no *deadlock* e, consequentemente, a liberação de seus recursos podem não ser simples:

- Os processos eliminados não têm como ser recuperados.

Correção do *Deadlock*

Após a detecção do *deadlock*:

O SO pode:

1. Eliminar um ou mais processos envolvidos no *deadlock* e desalocar os recursos já garantidos por eles

Porém:

A eliminação dos processos envolvidos no *deadlock* e, consequentemente, a liberação de seus recursos podem não ser simples:

- Os processos eliminados não têm como ser recuperados.

Como escolher qual processo eliminar então?

- De forma aleatório
- Levando em consideração algum critério de prioridade.

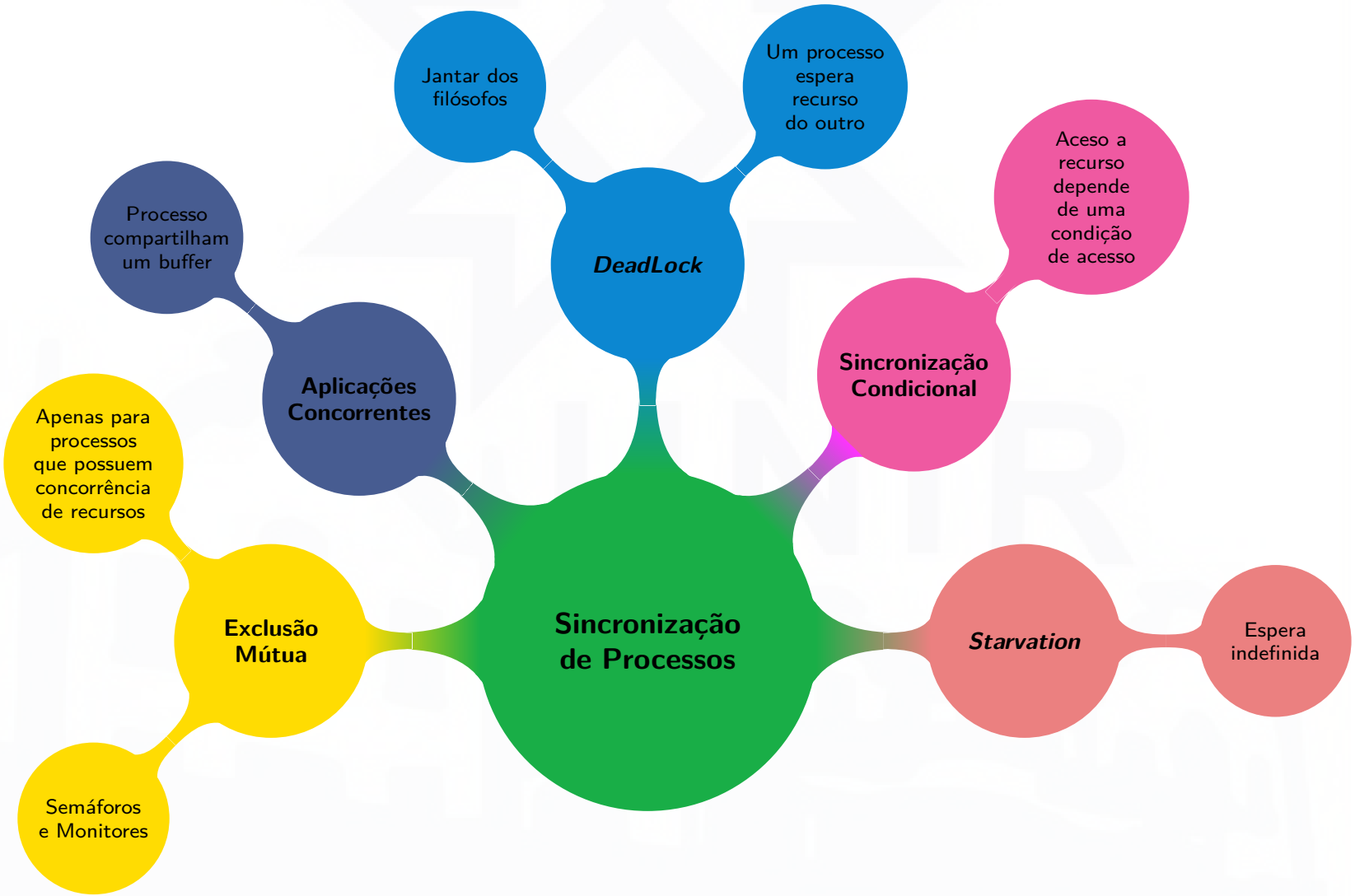
Correção do *Deadlock*

Solução menos drástica:
Suspender um processo, liberar seus recursos e depois de resolvido o *deadlock*, retornar seu processamento. Este procedimento é conhecido com *rollback*.

Contudo...
Além do *overhead* gerado, é muito difícil de ser implementado, por ser bastante dependente da aplicação que está sendo processada

Conclusão

MindMap de Sincronização de Processos



Sumário

Exercícios I

1. Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.
2. Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?
3. O que é exclusão mútua e como é implementada?
4. Como seria possível resolver os problemas decorrentes do compartilhamento da matriz, apresentado anteriormente, utilizando o conceito de exclusão mútua?
5. O que é *starvation* e como podemos solucionar esse problema?
6. O que é espera ocupada e qual o seu problema?
7. Explique o que é sincronização condicional e dê um exemplo de sua utilização.
8. Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.

Exercícios II

9. Apresente uma solução para o problema dos filósofos que permita que os cinco pensadores sentem à mesa, porém evite a ocorrência de *starvation* e *deadlock*.
10. Explique o que são monitores e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
11. Qual a vantagem da forma assíncrona de comunicação entre processos e como esta pode ser implementada?
12. O que é *deadlock*, quais as condições para obtê-lo e quais as soluções possíveis?

Referências I

DIJKSTRA, E. W. Solution of a problem in concurrent programming control. **Comm. ACM**, v. 8, n. 9, p. 569, 1965. Disponível em: <http://doi.acm.org/10.1145/365559.365617>.

MACHADO, F. B.; MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 5a. ed. [S.l.]: Ed. LTC, 2014.

FIM!

jonathan@unir.br