

Sistemas Operacionais

Sincronização

Lesandro Ponciano

2025

Objetivos da Aula

- Apresentar o conceito de sincronização de processos
- Discutir seção crítica
 - Condição de corrida
 - Problemas
 - Soluções do problema usando software e hardware
- Examinar problemas clássicos de sincronização de processo

Cooperação entre Processos

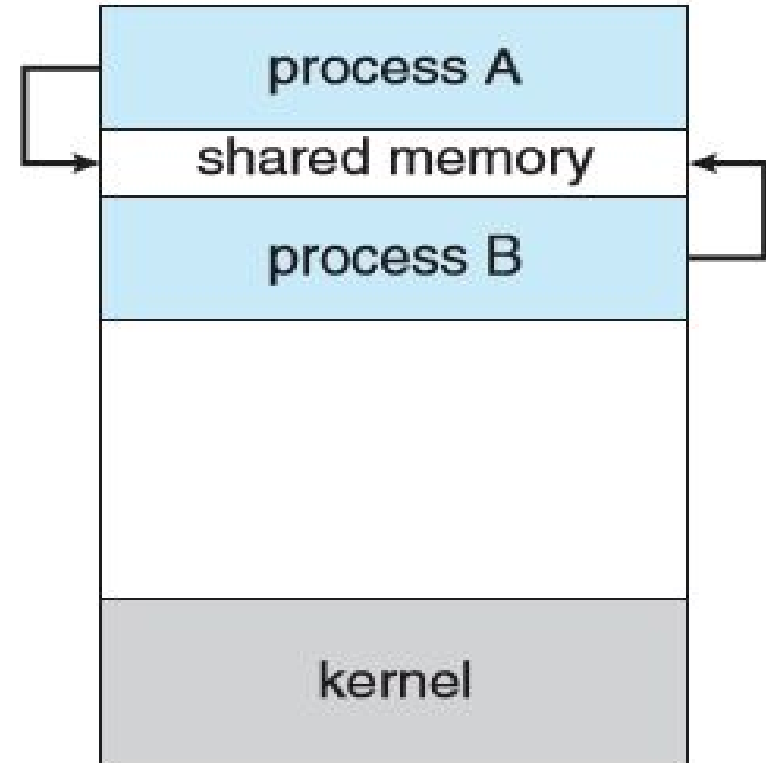
- Razões para processos cooperarem
 - Compartilhamento de informações
 - Velocidade de processamento
 - Modularidade
 - Conveniência

Comunicação Interprocessos (IPC)

- Comunicação inter processos (*Inter-Process Communication*, IPC)
 - Mecanismo que permite que processos troquem dados e informações
- Dois modelos
 - Memória compartilhada
 - Troca de mensagem

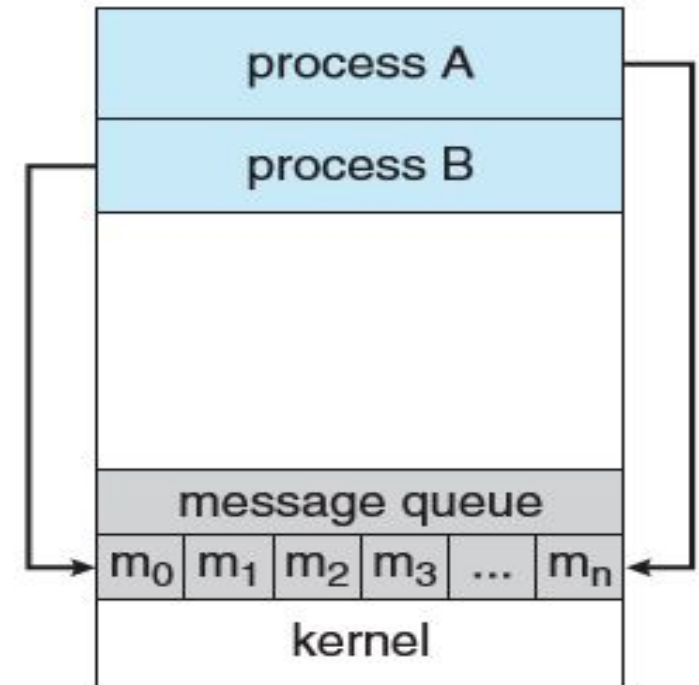
Memória Compartilhada

- Processos devem combinar
 - Um espaço de memória que os dois têm acesso
 - Não escreverem no mesmo espaço simultaneamente



Troca de Mensagens

- Se os processos *A* e *B* querem se comunicar, devem enviar e receber mensagens entre si
 - um link de comunicação deve ser implementado entre eles
- Tipos de comunicação
 - Direta e indireta
 - Síncrona ou assíncrona
 - Armazenamento em *buffer* automático ou explícito



Comunicação

- Comunicação **direta**
 - Simetria, há identificação no envio e no recebimento
 - Um link é criado entre os processos
 - Só existe um link entre eles
- Comunicação **indireta**
 - Os processos se comunicam com o uso de caixas postais (ou portas)
 - Só existe o *link* se existir caixa postal compartilhada
 - Podem existir vários *links*, cada um com uma caixa postal

Comunicação

- Comunicação **Síncrona**
 - O processo emissor é bloqueado até a mensagem ser recebida pelo processo receptor ou pela caixa postal
 - O receptor é bloqueado até a mensagem ficar disponível
- Comunicação **Assíncrona**
 - O processo emissor envia a mensagem e retorna à operação
 - O processo receptor recupera uma mensagem válida ou uma mensagem nula

Buffer Automático ou Explícito

- Buffers são filas temporárias nas quais as mensagens são armazenadas
- Buffer explícito
 - A fila de mensagens tem tamanho zero
 - O emissor deve ser bloqueado até o receptor receber a mensagem

Buffer Automático

- Capacidade limitada
 - A fila de mensagens tem capacidade limitada
 - Se a fila não estiver cheia quando uma mensagem for enviada, a mensagem é inserida na fila e o emissor poderá continuar a execução sem esperar
 - Se a fila estiver cheia, o processo emissor deverá esperar bloqueado
- Capacidade ilimitada
 - Tamanho da fila é potencialmente infinito
 - O emissor nunca é bloqueado

O Problema da Sincronização

- Processos (e *threads*) podem executar concorrentemente
 - Podem ser interrompidos a qualquer momento, completando parcialmente a execução
- Acesso simultâneo a dados compartilhados pode resultar em **inconsistência de dados**
- Manter a consistência dos dados requer mecanismos para garantir a **execução ordenada** de processos cooperativos

Seção Crítica

- Também chamada região crítica
- Parte do código de um programa onde é realizado o acesso do recurso compartilhado

do {

Entrada

Seção crítica

Saída

Restante

} while (true);

Condição de Corrida (Disputa)

- Ocorrer quando dois ou mais processos têm acesso compartilhado a dados e um processo consegue alterar o conteúdo dos dados sem que o outro perceba
- A condição de corrida leva a um estado errôneo/inconsistente

Condição de Corrida: Exemplo 1

- Os processos **A** e **B** têm acesso a uma mesma posição de memória (in) correspondente à fila de impressão
- Os processos **A** e **B** tentam alterar o valor da seguinte forma
 - 1) O processo **A** lê a variável “in” (in = 7) que indica a próxima entrada livre. Então acontece uma troca de contexto.
 - 2) O processo **B** lê a variável “in” (in=7) e grava o arquivo “procB.txt” a ser impresso na posição 7 da fila de impressão. Então acontece uma troca de contexto.
 - 3) O processo **A** grava o arquivo “procA.txt” a ser impresso na posição 7 da fila de impressão.
 - 4) O arquivo “procB.txt” nunca será impresso.

Condição de Corrida: Exemplo 2

- Dois processos (produtor e consumidor) compartilham um buffer de tamanho fixo
 - O produtor produz (insere) informação no buffer
 - O consumidor consome (remove) informação do buffer
- Suponha buffers usando contadores
 - Contador inteiro `counter` mantém o número de buffers cheios
- `counter` é
 - inicialmente definido como 0 (`counter=0`)
 - incrementado pelo produtor depois que ele produz um novo buffer
 - decrementado pelo consumidor depois de ele consome um buffer

Produtor

```
while (true) {  
  
    while (counter == BUFFER_SIZE) ;  
        /* não faça nada */  
  
    /* Produz um item */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```


Consumidor

```
while (true) {  
  
    while (counter == 0) ;  
    /* não faça nada */  
  
    /* consome um item */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

Análise da Condição de Corrida

No baixo nível, ao ser executado no processador

counter++ é

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

counter-- é

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Suponha counter igual a 5 e os seguintes passos:

S0: producer executa	register1 = counter	{register1 = 5}
S1: producer executa	register1 = register1 + 1	{register1 = 6}
S2: consumer executa	register2 = counter	{register2 = 5}
S3: consumer executa	register2 = register2 - 1	{register2 = 4}
S4: producer executa	counter = register1	{counter = 6}
S5: consumer executa	counter = register2	{counter = 4}

Evitar Condição de Corrida

- Condições para que dois processos possam concorrer sem ocorrer condições de corrida
 - Nunca dois ou mais processos podem estar simultaneamente dentro de suas regiões críticas correspondentes
 - Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs
 - Nenhum processo que esteja rodando fora de sua região crítica pode bloquear a execução de outro processo
 - Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica
 - Pode sofrer de inanição ou *starvation*, quando um processo nunca consegue executar a sua seção crítica

Exclusão Mútua

- É um mecanismo para evitar que dois processos acessem um mesmo recurso (variável) ao mesmo tempo
 - Quando um processo estiver acessando uma região crítica, nenhum outro poderá fazê-lo
 - Há soluções em hardware e em software
- Solução em hardware para exclusão mútua
 - Desabilitação de Interrupções: inibe-se qualquer interrupção enquanto o processo estiver na região crítica
 - Problema: um processo usuário pode nunca mais habilitar as interrupções e ficar com a CPU por tempo indeterminado

Exclusão Mútua em Software

- Solução em software para exclusão mútua
 - Variáveis de Impedimento: utilização de uma variável compartilhada para sinalizar se um recurso está sendo utilizado ou não
 - O processo verifica o valor da variável (0) antes de entrar na região crítica e muda seu estado após ter entrado (1)
 - Outros processos são impedidos de entrar em suas regiões críticas antes que o primeiro saia e retorne o valor a zero

Solução de Peterson

- Exclusão mútua para dois processos no nível de usuário
 - “Se dois processos chegam quase ao mesmo tempo na região crítica, o que chegou por último, cede a vez para o primeiro”
- Dinâmica
 - Antes de utilizar as variáveis compartilhadas, cada processo deve chamar a função **entrar_regiao** informando o seu id
 - Faz com que o processo espere até que seja seguro, antes de entrar em sua região crítica
 - Ao sair da região crítica, o processo deve chamar a função **sair_regiao** informando o seu id
 - Liberar processo que estiver esperando

Instrução TSL (*Test and Set Lock*)

- Depende do hardware do processador implementar a função TSL
- Transfere o valor de uma posição de memória para um registrador e escreve um valor não nulo neste endereço
 - Operação de transferência e armazenamento são indivisíveis
 - O processador bloqueia o acesso à memória a qualquer outro processador ou processo
- Variável compartilhada flag controla o acesso à função TSL
 - Quando flag = 0, qualquer processo pode fazê-la igual a 1 através da função TSL e então ler ou escrever na memória compartilhada
 - O processo deve voltar flag a 0 ao sair da região crítica

Exemplo de Instrução TSL

- A instrução TSL possui o seguinte formato e semântica
 - Test-and-Set(X,Y)
 - Quando executada, o valor lógico da variável Y é copiado para X, sendo atribuído à variável Y o valor lógico verdadeiro

- Exemplo

```
Pode = TRUE;  
WHILE (Pode) DO  
    Test-and-Set (Pode,Bloqueio)  
Região_Crítica;  
Bloqueio = FALSE;
```

Enquanto o Bloqueio é TRUE
significa que outro processo já
chegou primeiro

Semáforos

- O valor de um semáforo indica quantos processos podem ter acesso a um recurso compartilhado
 - Proposto por Dijkstra em 1965
- Chamada **Down** ou **Wait**
 - verifica se o valor é maior que 0; se for, decrementa o valor do semáforo e o processo continua a executar. Se for 0, o processo que chamou Down é posto para dormir numa fila de espera
- Chamada **Up** ou **Signal**
 - incrementa o valor do semáforo e verifica se há processos dormindo. Se houver, escolhe um deles e o acorda. Então, o processo escolhido acorda e dá um Down(decrementa semáforo)

Funções de Semáforos

- Definição da função `wait()` ou `down()`

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

As operações sobre semáforos são atômicas.
Ou seja, desabilita-se as interrupções do processador

- Definição da função `signal()` ou `up()`

```
signal(S) {  
    S++;  
}
```

Espera Ocupada

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Espera ocupada, também é chamada de “espera ativa” ou “espera ociosa”, consiste no processo verificar uma condição repetidamente até que a mesma seja satisfeita

Características dos Semáforos

- Semáforos devem ser implementados como chamadas de sistema
- No caso de sistemas com mais de um processador, utilizam-se variáveis de travamento TSL para garantir que somente um deles escreva num semáforo em um instante de tempo
- As chamadas ao semáforo dependem do programador e possibilitam erros

Monitores

- Recurso provido pela linguagem de programação
- O programador define a região do monitor e apenas um processo pode estar por vez dentro de um monitor
- Cabe ao compilador implementar a exclusão mútua na região

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

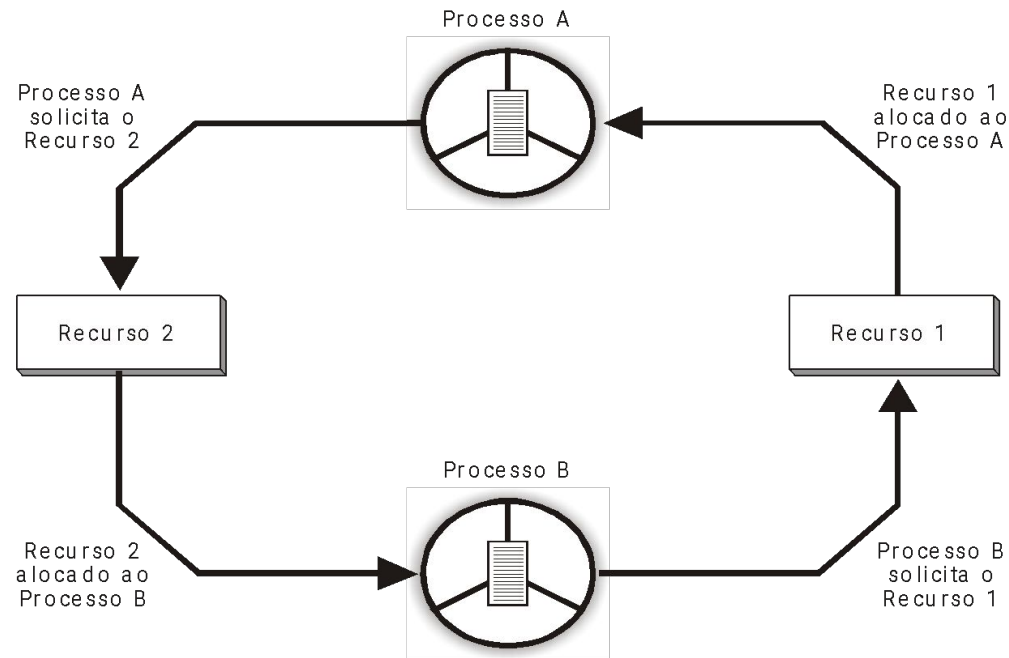
    Initialization code (...) { ... }
}
```

Starvation and Deadlock

- Duas situações comuns quando se trabalha com sincronização
 - *Starvation (inanição)*
 - *Deadlock (impasse)*

Deadlocks

- Em condições normais, para utilizar um recurso o processo precisa
 - Solicitar
 - Usar
 - Liberar




Formalismo de Deadlock

- Um conjunto de processos está em *deadlock* quando cada processo do conjunto está esperando um evento que só pode ser causado por outro processo do conjunto
 - Eventos são de requisição e liberação de recursos
- Um processo aguarda por um recurso que nunca será liberado pois tal recurso está com outro processo que também está em *deadlock*

Exemplo de Deadlock

- O processo 1 precisa do recurso 2 e o processo 2 precisa do recurso 1, então existe um *deadlock*

<pre>void processo1(void) { wait(&recurso1); wait(&recurso2); usar_recurso1(); signal(&recurso2); signal(&recurso1); }</pre>	<pre>void processo2(void) { wait(&recurso2); wait(&recurso1); usar_recurso2(); signal(&recurso1); signal(&recurso2); }</pre>
--	--



Condições

Necessárias para Deadlock

- Quatro condições simultâneas:
 - 1) **Exclusão Mútua**: cada recurso só pode estar alocado a um único processo em um determinado instante
 - 2) **Posse e espera**: 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 um outro processo e vice-versa

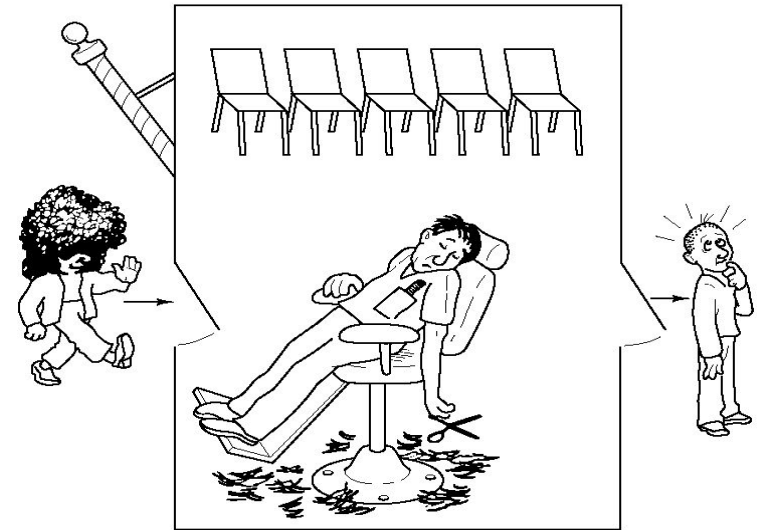
Lidando com Deadlocks

- Três maneiras de lidar com o problema
 - Usar um protocolo para impedir ou evitar que o *deadlock* ocorra
 - Permitir que o *deadlock* ocorra, detectá-lo e se recuperar dele
 - Ignorar o problema e fingir que *deadlocks* nunca correm
 - Solução mais usada, Windows, Unix, etc
 - Cabe ao programador desenvolver aplicações que não entrem em *deadlock*
- *Deadlocks* podem
 - ser evitados pedindo-se que cada processo informe *a priori* os recursos que utilizará
 - ser impedidos atacando-se as condições necessárias

Problemas Clássicos de Sincronização

Barbeiro Sonolento

- Caracterização do problema:
 - Os clientes sentam nas cadeiras disponíveis e esperam sua vez
 - Se não houver cadeiras, o cliente vai embora
 - O barbeiro atende os clientes por ordem de chegada e, não havendo clientes, ele dorme



Barbeiro Sonolento

Instanciado no problema do Buffer Limitado

```
int n;  
semaphore mutex = 1;  
semaphore empty = n;  
semaphore full = 0
```

Exclusão mútua

Número de buffers vazios

Número de buffers cheios

Processo Produtor
(Colocando clientes na fila)

```
do {
```

```
    . . .  
    /* produce an item in next_produced */
```

```
    . . .  
    wait(empty);  
    wait(mutex);
```

} Bloqueia o consumo

```
    . . .  
    /* add next_produced to the buffer */
```

```
    . . .  
    signal(mutex);  
    signal(full);
```

} Libera o consumo

```
} while (true);
```

Barbeiro Sonolento

Instanciado no problema do Buffer Limitado

Processo Consumidor
(Removendo clientes da fila)

```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    /* remove an item from buffer to next_consumed */  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    /* consume the item in next_consumed */  
    . . .  
} while (true);
```

} Bloqueia a produção

} Libera a produção

Leitores-Escritores

- Modela o acesso a bases de dados
 - muitos processos podem ler uma base ao mesmo tempo
 - se um processo estiver escrevendo, nenhum outro pode ter acesso a base, nem mesmo para leitura
- Execução do Algoritmo
 - Primeiro leitor executa um *wait* no semáforo *rw_mutex* (bloqueia os escritores)
 - Último leitor executa um *signal* no semáforo *rw_mutex* (desbloqueia os escritores)
 - Uma variável *read_count* conta o número de leitores ativos
- Esta implementação dá prioridade aos leitores
 - Mesmo que um escritor tente acessar a base e não consiga devido a um leitor, novos leitores podem ir entrando o que atrasa ainda mais a possibilidade do escritor mexer na base

Leitores-Escritores

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
    . . .  
    /* reading is performed */  
    . . .  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
} while (true);
```

Bloqueia os escritores

Desbloqueia os escritores

Leitor

```
do {  
    wait(rw_mutex);  
    . . .  
    /* writing is performed */  
    . . .  
    signal(rw_mutex);  
} while (true);
```

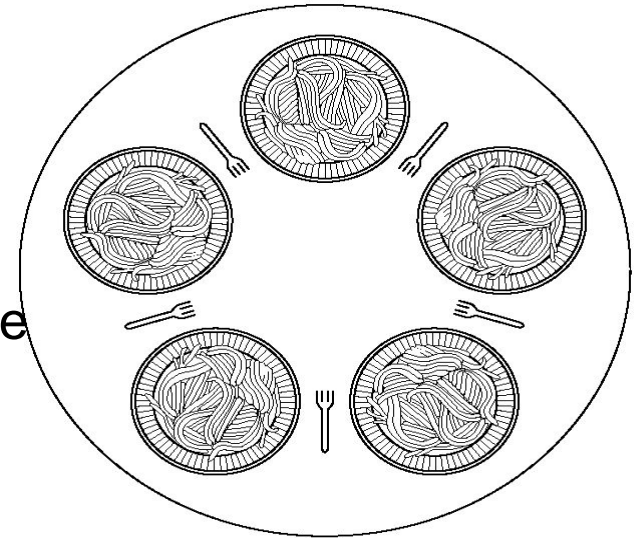
Bloqueia apenas
os outros escritores

Desbloqueia os
outros escritores

Escritor

Jantar dos Filósofos

- Cinco filósofos estão sentados à mesa, e devem alternar suas atividades entre pensar e comer
 - Para comer, cada um deles precisa de dois garfos
 - Quando um filósofo sente fome, ele deve pegar um garfo à esquerda e outro à direita (em qualquer ordem), comer e, quando terminar, devolver os garfos
- Modela o acesso exclusivo a um número limitado de recursos



Jantar dos Filósofos

- Se todos tentarem comer ao mesmo tempo, nenhum come logo existe *deadlock*
- Modificação: após pegar o garfo da esquerda, verificar se o da direita está livre. Se não estiver, devolva o garfo
 - Pode provocar *starvation* (inanição) se houver sincronismo, ou seja, um ou mais filósofos podem ficar sem comer
- Criar uma região crítica protegida com um semáforo binário
 - Evita-se *starvation* e *deadlock*
 - Mas obtém-se paralelismo pobre (apenas um filósofo pode comer de cada vez e não dois)

- Solução usando monitores
 - Não ocorre *deadlock*, mas pode ocorrer *starvation*

Uma solução mais robusta (e complexa) é apresentada em Tanenbaum (2009)

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Atividade de Fixação

1. Cite e explique os modelos de comunicação interprocessos.
2. Conceitue e diferencie seção crítica, condição de corrida, exclusão mútua e espera ocupada.
3. Quais são as 4 condições necessárias para que ocorra deadlock?
4. Como funcionam semáforos e monitores?

Referências

TANENBAUM, Andrew S. Sistemas operacionais modernos. 3. ed. São Paulo: Pearson Prentice Hall, 2009. xvi, 653 p. ISBN 9788576052371

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. Fundamentos de sistemas operacionais: princípios básicos. Rio de Janeiro, RJ: LTC, 2013. xvi, 432 p. ISBN 9788521622055

Sistemas Operacionais

Prof. Dr. Lesandro Ponciano

<https://orcid.org/0000-0002-5724-0094>