

## AULA 07 - Comunicação entre Processos

### Fundamentos

Muitas vezes, podemos ter processos trabalhando juntos na solução de um problema e, portanto, utilizando recursos comuns, como uma posição de memória comum ou um arquivo onde cada processo pode ler e escrever. O que pode acontecer na prática?

### Situação 1

Spool de impressão (fila de documentos a serem impressos)

Variáveis compartilhadas:

- ✂ **in** ✂ aponta para a próxima entrada livre no spool onde o próximo arquivo a ser impresso será armazenado
- ✂ **out** ✂ aponta para o nome do próximo arquivo a ser impresso.

Um processo 1 lê o valor de in (por exemplo 2) e é interrompido. Um outro processo 2 vai usar o spool e também lê o valor de in = 2, envia seu arquivo para impressão e segue com sua execução. Antes de o arquivo ser impresso, o controle retorna para o processo 1 que submete seu arquivo a ser impresso para a posição que ele tinha lido de in (na ocasião 2). Assim, este último arquivo sobrescreve a anterior enviado pelo processo 2.

### Situação 2

Considere 2 trechos de código C que compartilham o acesso a uma determinada variável (**count**)

Processo 1	Processo 2
<pre>while(count == SIZE); //espero para executar count++; buffer[in] = item; in = (in + 1) % SIZE;</pre>	<pre>while(count == 0) ; //espero para executar count--; item = buffer[out]; out = (out + 1) % SIZE;</pre>

Ou ainda, um código em Assembly:

Mov ax, count add ax, 1 mov count, ax	mov ax, count dec ax, 1 mov count, ax
---	---

O que vai ocorrer se houver uma troca de contexto justamente quando o processo 1 vai executar a linha 2 e o processador passa a executar o processo 2?

Essa concorrência entre dois ou mais processos para manipulação de um dado compartilhado, de tal forma que a seqüência do acesso afeta o resultado da execução é chamado de condição de corrida (*race condition*)

## Regiões Críticas

O problema é: como evitar as condições de corrida? A questão chave para evitar problemas em situações onde existe compartilhamento de recursos (*memória*, arquivo, etc) é evitar que mais de um processo acesse o dado compartilhado ao mesmo tempo => geralmente, consideraremos 2 processos apenas.

- ⌘ exclusão mútua de execução: é uma forma encontrada para se ter certeza de que se um processo estiver usando uma variável ou um arquivo compartilhados, os demais serão impedidos de fazer a mesma coisa.
- ⌘ região crítica ou seção crítica: é a parte do programa cujo processamento pode levar à ocorrência de condições de corrida.

Apesar dessa solução aparentar ser suficiente, ela acaba dificultando a cooperação de processos paralelos que utilizem dados compartilhados na solução de um problema. Assim, devemos fazer algumas outras considerações:

1. dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes.
2. nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito do número de processadores do sistema.

3. nenhum processo que esteja rodando fora de sua região crítica pode bloquear a execução de outro processo.
4. nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica.

Estratégias de soluções => consideram exclusão mútua com espera ocupada => quando um processo entra em sua região crítica, nenhum outro processo pode entrar em sua região crítica correspondente.

## **Exclusão Mútua com Espera Ociosa**

Existem algumas formas de tentarmos obter a exclusão mútua entre processos em relação a uma região compartilhada. A seguir descrevemos algumas estratégias de solução.

### **1. Inibição das Interrupções**

É a solução mais simples => cada processo deve *inibir as interrupções* logo após entrar em uma região crítica, habilitando-as imediatamente antes de sair dela. Dessa forma evitamos as interrupções de relógio e conseqüentemente os problemas gerados pelo escalonamento de processos citados anteriormente.

Qual o problema desta abordagem?

Dar a processos usuários o poder de desabilitar interrupções => uso indevido causaria problemas.

Portanto: esta estratégia não é apropriada para garantir a exclusão mútua.

### **2. Variáveis de Travamento ou Impedimentos (*lock variables*)**

Nesta estratégia existe uma variável compartilhada, denominada de variável de travamento (**lock**), cujo valor inicial é 0 (zero). Ao desejar entrar em uma região crítica, o processo primeiro deve verificar o valor da variável de travamento. Se  $lock = 0$ , o processo altera seu valor para 1 e entra na região crítica. Por outro lado, se a variável  $lock = 1$ , o processo está impedido de entrar na

região crítica e, portanto, deve esperar até que lock volte para 0, liberando o acesso a RC (*Região Crítica*).

Qual é o problema dessa estratégia?

Pode acontecer de um processo P1 ler o valor de lock e ela está igual a 0 e a seguir ele é interrompido, antes de alterar o valor de lock para 1. Dai o processo escalonado (P2) lê o valor de lock = 0 e muda para 1, podendo entrar em sua RC. É interrompido e a execução volta para P1 que também coloca lock = 1 e entra na sua RC. Nesse instante, temos P1 e P2 na RC !!!

### 3. Escrita Alternada (*Alternância Obrigatória*)

Uma variável, no código abaixo representada por **vez**, estabelece de quem é a vez de entrar na região crítica. Inicialmente vez = 0 e o Processo 0 inicia a execução entrando na RC. Por outro lado, P1 tenta entrar na RC e verifica que vez != 1 ficando bloqueado checando constantemente o valor da variável compartilhada aguardando ela passar para 0 (zero) e com isso ganhar o acesso a RC. Este tempo e processamento gasto por P1 testando continuamente o valor da variável vez é chamado de *espera ocupada* (*busy waiting*). Esta espera acaba desperdiçando tempo de CPU e deveria ser evitada.

Ao sair da RC o Processo 0 altera o valor de vez para 1 e libera o acesso à RC para o Processo 1.

```
while (TRUE) {
    while (vez != 0) /* espera */
        regioao_critica ();
    vez = 1;
    regioao_ao_critica ();
}
```

(a) Processo 0

```
while (TRUE) {
    while (vez != 1) /* espera */
        regioao_critica ();
    vez = 0;
    regioao_ao_critica ();
}
```

(b) Processo 1

**Pergunta:** Qual o problema que existe nesta estratégia?

**Resposta:** Um processo pode ser impedido de entrar na RC mesmo com o outro estando fora da RC.

**Imagine a seguinte situação:** Suponha que o Processo 0 executou sua região crítica e atribui à variável **vez** o valor 1, para permitir que o Processo 1 execute sua região crítica. Agora, a execução é alternada para o Processo 1 que executa muito rapidamente a região crítica, coloca a variável **vez** em 0 inicia a sua região não crítica. Neste ponto temos ambos os processos executando suas regiões não críticas com **vez=0**. O Processo 0 executa o laço rapidamente, saindo da região crítica e colocando **vez** em 1. Alterna-se a execução para o Processo 1 (que ainda está executando sua região não-crítica). Como a região não-crítica do Processo 1 é muito extensa, altera-se novamente para o Processo 0 que agora não consegue entrar em sua região crítica, pois a variável **vez** está com valor 1.

Portanto: esta solução não é adequada quando um processo é muito mais lento que o outro. Além disso, não podemos fazer nenhuma consideração sobre a velocidade de execução de cada processo. Assim, existe uma obrigatoriedade dos processos alternarem a entrada em suas RCs.

#### 4. Solução de Perterson

Em 1981, G. L. Peterson, descobriu uma técnica de se obter exclusão mútua sem a obrigatoriedade da alternância citada anteriormente.

Funcionamento: antes de entrar na sua região crítica cada processo deve chamar *enter\_region*, com seu número de processo, 0 ou 1, como parâmetro. Imagine que o Processo 0 chame a função *enter\_region* para manifestar seu interesse em entrar na RC, setando *interested[process]=TRUE* e coloca a variável *turn=0*. Nesse instante o outro processo (P1) não está interessado em entrar na sua RC, portanto, o procedimento retorna imediatamente. Caso P1 chame *enter\_region* ele ficará bloqueado até que *interested[0]=FALSE*. Tal situação acontecerá somente quando P0 chamar *leave\_region*.

Problema da solução: Apenas a espera ocupada.

**IDÉIA:**

```
# include "prototypes.h"
# define FALSE 0
# define TRUE 1
# define N      2

int turn;

int interested[N];

void enter_region (int process)
{
    int other;
    other = 1 - process;
    interested [process] = TRUE; /*MOSTRA INTERESSE PELA RC*/
    turn = process;
    while (turn == process && interested[other] == TRUE)
}

void leave_region (int process)
{
    interested[process] = FALSE; /*SAÍDA DA RC*/
}
```

## Bloqueio e Desbloqueio de Processos: primitivas SLEEP/WAKEUP

Apesar da solução de Peterson ser correta, ela utiliza o esquema de espera ocupada em sua implementação => quando um processo não pode entrar na região crítica ele é posto para executar um *loop* até que seja permitido entrar na região.

Portanto, devemos procurar um meio de bloquear a execução de determinado processo, em vez de ocasionar uma espera ocupada. Uma forma simples de resolver este problema é através do uso das primitivas: SLEEP e WAKEUP.

1. **SLEEP:** é uma chamada de sistema que bloqueia o processo que chamou => suspende a execução do processo até que outro processo o acorde.
2. **WAKEUP:** utiliza como parâmetro o *id* do processo a ser acordado.

## Problema do Produtor-Consumidor ou do Buffer Limitado

É um exemplo para ilustrar o uso das primitivas citadas anteriormente. Neste problema, dois processos compartilham um buffer de tamanho fixo. Um dos processos, denominado produtor, coloca informação no buffer e, o outro, conhecido como consumidor, retira informação do buffer.

Qual é o problema? O problema acontece quando o produtor deseja colocar dados no buffer e ele está cheio.

Qual a solução? Colocar o produtor para dormir e só acordá-lo quando o consumidor tiver retirado um ou mais itens do buffer. O mesmo acontece se o consumidor deseja retirar um item do buffer e ele já está vazio.

### CÓDIGO DO PROBLEMA:

```
# define N      100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE){
        item = produce_item( );
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer); /*Buffer estava vazio? Se sim,
consumidor esta dormindo*/
    }
}

void consumer(void)
{
    int item;

    while (TRUE){
        if (count == 0) sleep ( );
        item = remove_item( );
        count = count - 1;
        if (count == N -1) wakeup (producer); /*Buffer estava cheio? Se
sim, logo produtor esta dormindo*/
        consume_item (item);
    }
}
```

Imagine a situação: em um determinado momento o buffer está vazio e o consumidor acabou de checar que count é 0 (zero). Antes de ir dormir, o escalonador decide começar executar o produtor. O produtor insere um novo item no buffer, incrementa count e infere que count era 0 (zero) e que, portanto, o consumidor deveria estar dormindo. Logo o produtor chama wakeup para acordar o consumidor. Pelo fato do consumidor não está dormindo realmente, o sinal é perdido. Agora o consumidor volta a usar a CPU e terá o valor de count = 0 (valor que estava atribuído antes de sua interrupção) e dormirá. Com o tempo o produtor preencherá todo o buffer e também ira dormir. Com isso, ambos dormirão para sempre!

Todo problema aconteceu porque o sinal de acordar foi perdido. Solução é adicionar um bit para captar estes sinais (*bit de espera pelo sinal de acordar*). Este bit é setado quando um sinal de acordar é enviado. Assim, se um processo tenta dormir e o sinal de acordar tá setado ele permanecerá acordado e desligará o bit, resolvendo o problema.

## **Exercícios**

1. O que é uma condição de corrida? Dê um exemplo prático do dia-a-dia que ilustra o conceito apresentado (exemplo não computacional!).
2. A solução de espera ociosa usando a variável vez (escrita alternada) funciona quando dois processos estão executando em uma máquina multiprocessada de memória compartilhada?
3. O que é a exclusão mútua? Como podemos implementar esse conceito numa programação usando a linguagem C e Java?
4. Qual o problema da solução que desabilita as interrupções para implementar a exclusão mútua?
5. Qual o problema da espera ocupada (*busy waiting*)?

## ***Bibliografia Básica***

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. LTC, 2ª Edição, São Paulo: Prentice-Hall, 2003. (Cap. 2)