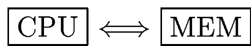
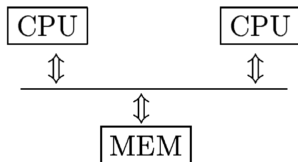
Programação Concorrente – Aula 2

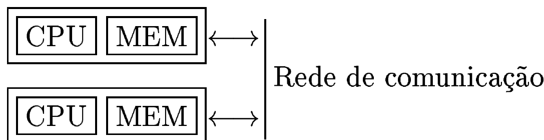
Gilmar Gimenes Rodrigues

1 Threads

- Apontador de instruções (program counter).
- Registradores de trabalho da CPU.
- Pilha de execução.

1. Single CPU:


2. Multi-CPU (shared mem.)


3. Multi-CPU (distribuida)


Em 1 e 2 pode haver compartilhamento de variáveis.
Em 3 há troca de mensagens.

2 Ações Atômicas: Indivisíveis

Nenhum resultado intermediário da ação é visível para outras threads.

Hipóteses:

1. As instruções de leitura e escrita de tipos básicos (`char`, `int`, ...) na memória são ações indivisíeis.
2. Valores são manipulados carregando-os da memória para registradores, operando sobre registradores e depois guardando os resultados na memória.
3. Cada thread tem seu próprio conjunto de registradores.
   (a) Se houver uma CPU por thread, então cada thread será a "dona" do conjunto de registradores de trabalho da sua CPU.

(b) Se várias threads rodarem na mesma CPU, então sempre que houver uma troca da thread em execução nessa CPU os registradores de trabalho deverão ter seus valores salvos na pilha da thread que deixa a CPU e restaurados da pilha da thread que ganha a CPU.

4. Variáveis locais e resultados intermediários de avaliação de expressões são guardadas em registradores ou na pilha de execução da thread.

2.1 Problema da não-atomicidade:

```
int x = 0;   // Valor inicial de x

thread_soma_1:                          thread soma_2:
        x = x + 1                               x = x + 2
```

Cada uma dessas threads tem três instruções:

```
thread_soma_1:                          thread soma_2:
        MOV reg, x                              MOV reg, x
        ADD reg, 1                              ADD reg, 2
        MOV x, reg                              MOV x, reg
```

Possíveis resultados: 1, 2, 3

Exemplo de possível entrelaçamento (interleaving):

```
thread_soma_1:                          thread soma_2:
        MOV reg, x
        ADD reg, 1
                                                MOV reg, x
                                                ADD reg, 2
                                                MOV x, reg
        MOV x, reg
```

Valor final de x: 1

Outro possível entrelaçamento:

```
thread_soma_1:                          thread soma_2:
        MOV reg, x
        ADD reg, 1
                                                MOV reg, x
                                                ADD reg, 2
        MOV x, reg
                                                MOV x, reg
```

Valor final de x: 2

Não queremos que a “corretude” dependa do caso (do escalonamento que acabar acontecendo).

2.2 Sincronização:

Evitar interleavings indesejáveis

3 Exclusão mútua

Várias threads com a seguinte estrutura:

```
protocolo de entrada
região crítica
protocolo de saída
região critica
```

Vamos assumir que essa estrutura básica ocorre dentro de um laço perpétuo:

```
for( ; ; ) {
    protocolo de entrada;
    região crítica;
    protocolo de saída;
    região não crítica;
}
```

O **problema da região crítica** consiste em projetar protocolos de entrada e saída que satisfaçam as seguintes propriedades:

1. **Exclusão mútua:** No máximo uma thread pode estar na sua região crítica num instante.
2. **Ausência de deadlock:** Se duas ou mais threads estão querendo entrar na região crítica, apenas uma delas consegue.
3. **Sem atrasos desnecessários:** Se apenas uma thread está querendo entrar, nada a impedirá de entrar já.
4. **Garantia de entrada:** Se uma thread está querendo entrar, em algum momento ela conseguirá.

3.1 Solução em sistemas single-CPU

Desabilitar interrupções de hardware:

```
x = x + 1          x = x + 2

CLI                CLI
MOV reg, x         MOV reg, x
ADD reg, 1         ADD reg, 2
MOV x, reg         MOV x, reg
STI                STI
```

Obs: `CLI` desabilita e `STI` habilita as interrupções de hardware.

Problemas:

- Só funciona para máquinas com uma só CPU.

- Só serve para regiões críticas muito curtas, pois interrupções não devem ficar desabilitados por muito tempo.
- Instruções que desabilitam interrupções requerem privilégio (não rodam em modo usuário)

Essa solução só serve para:

- Quem escreve o S.O. de uma máquina single-CPU
- Quem trabalha com programação em "baixo nível" (sistemas embutidos, DOS, ...)

3.2 Soluções sem ajuda do hardware

- Dekker (~1965)
- Peterson(1981): Algoritmo "Tie Breaker" (Desempatador)

Tentativa de resolver o problema da região crítica para 2 threads:

Usar uma variável "vez" funcionaria?

```
int vez = 0; /* variável compartilhada entre as 2 threads   */
             /* se vez for 0 ninguém está na região crítica */
             /* se vez for 1 a thread 1 está na r.c.        */
             /* se vez for 2 a thread 2 está na r.c.        */
```

```
thread_1()                              thread_2()
{                                       {
    /* espera a thread_2 */                 /* espera a thread_1 */
    /* sair da r.c.      */                 /* sair da r.c.      */
    while (vez == 2)                        while (vez == 1)
        ;                                       ;
    vez = 1;                                vez = 2;
    regiao_critica();                       regiao_critica();
    vez = 0;                                vez = 0;
    regiao_nao_critica();                   regiao_nao_critica();
}                                       }
```

Essa mesma tentativa em linguagem de montagem:

```
L1:
    MOV reg, vez
    CMP reg, 2    (ou 1)
    JE L1
    MOV reg, 1
    MOV vez, reg
    região crítica
    MOV reg, 0
    MOV vez, reg
```

A "solução" acima não funciona!!! (Por que?)