

① 10ª Aula

Sincronização em Sistemas Distribuídos

→ Tempo lógico

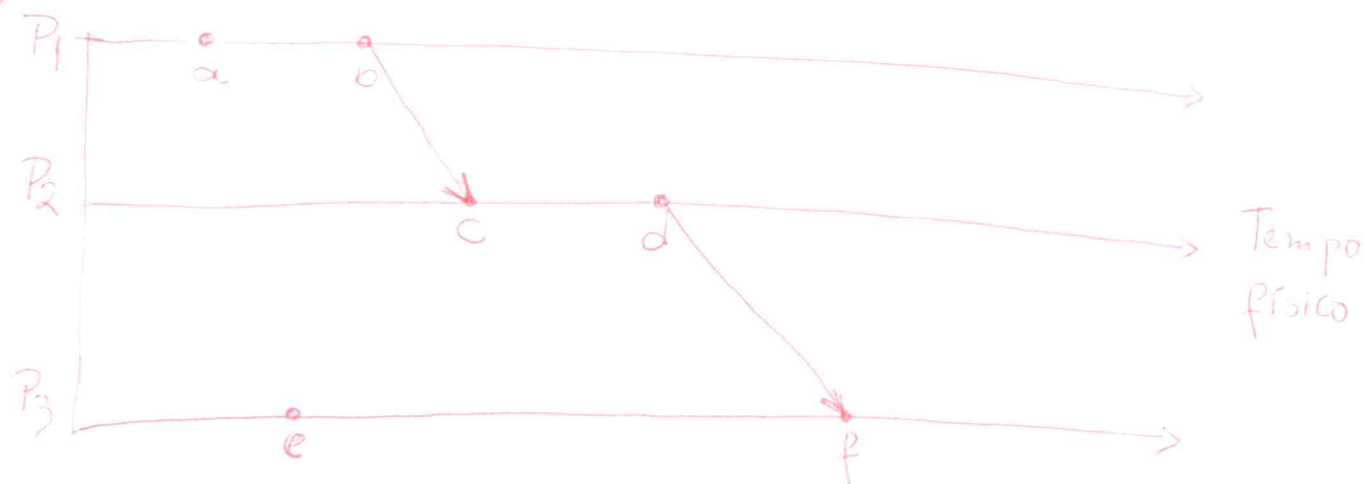
- * Vistos de um processo, os eventos são ordenados pelo tempo do relógio local
- * Não é possível sincronizar os relógios físicos perfeitamente, logo não tem como utilizá-los para determinar a ordem de ocorrência de 2 eventos em um Sistema Distribuído

②

Solução para o problema:

- Utilizar esquema baseado em causalidade (happened-before = "acontece antes")
- $a \rightarrow b$, se e somente se, a ocorre antes de b
- Em vez de sincronizar os relógios físicos ordenamos os eventos

③



$a \rightarrow b$ (em P_1)

$c \rightarrow d$ (em P_2)

$b \rightarrow c$ por causa de m_1

$d \rightarrow f$ por causa de m_2

$a // e$ = eventos concorrentes

④

Dois situações:

(1) Se os eventos e_1 e e_2 acontecem no mesmo processo, então $e_1 \rightarrow e_2$ (ordem observada pelo processo)

(2) Quando um processo envia uma mensagem m para outro processo, o send acontece antes do receive ($\text{send}(m) \rightarrow \text{receive}(m)$)

Note que a relação \rightarrow é transitiva

⑤ Relógio lógico de Lamport (1978)

- Mecanismo numérico para capturar e representar a relação → (acontece antes)
- Cada processo P_i tem um relógio lógico (contador) designado por L_i
- Não está ligado a um relógio físico

⑥ Regras:

timestamps (marcas de tempo) são colocados nos eventos de acordo com as regras

- Evento local: L_i é incrementado antes de colocá-lo como timestamp em um evento
- Send: Quando P_i envia m , ele copia em m o valor do seu L_i
- Recebe: Quando P_j recebe uma mensagem com timestamp t , ele efetua o cálculo
$$L_j = \max \{L_j, t\} + 1$$

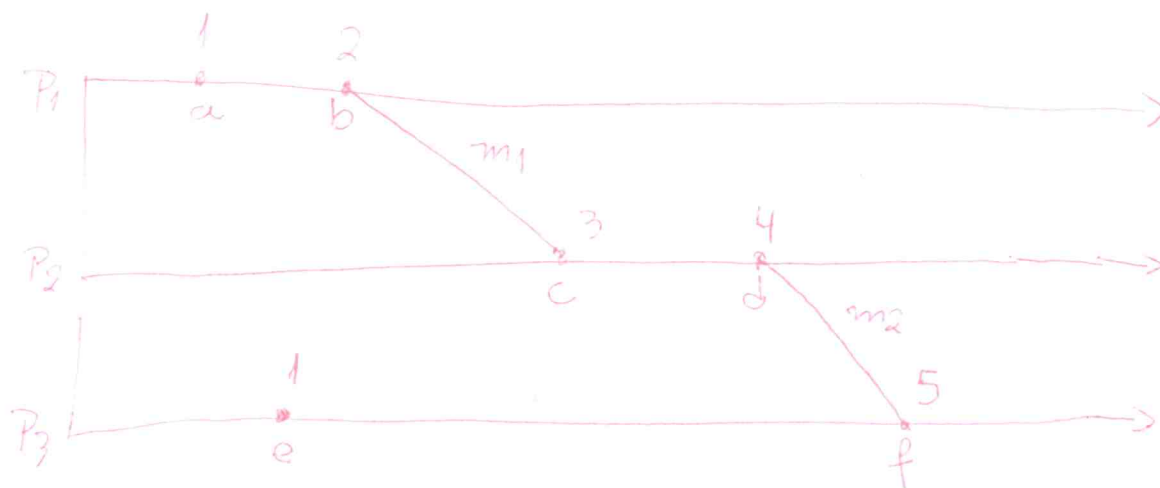
⑦

Exemplo

→ L_1, L_2, L_3 são inicializados com 0 (zero)

→ No envio de m_1 , P_1 manda junto $L_1 = 2$

→ P_2 recebe m_1 e calcula $L_2 = \max\{0, 2\} + 1$



⑧

Note que:

$$a \rightarrow b \Rightarrow L(a) < L(b)$$

$$L(a) < L(b) \not\Rightarrow a \rightarrow b$$

No exemplo anterior: $\underbrace{L(e)}_1 < \underbrace{L(b)}_2$, porém $e \parallel b$ (não existe relação de causalidade)

Como fazer para ter $a \rightarrow b \Leftrightarrow L(a) < L(b)$?

Resposta: Relógios vetoriais

⑨ Relógios vetoriais (1988)

- Existem N processos
- Cada processo P_i possui seu relógio vetorial V_i
- Cada relógio vetorial V_i tem N posições, i.e., $V_i[N]$
- $V_i[i]$ contém o número de eventos que ocorreram no próprio P_i
- $V_i[j]$, com $i \neq j$, contém o número de eventos que ocorreram em P_j e potencialmente afetaram P_i

⑩ Regras:

- Inicialização: $V_i[j] = 0$ para $j = 1$ até N
- Evento local: Antes de colocar timestamp em um evento, o processo calcula $V_i[i] = V_i[i] + 1$
- Send: P_i acrescenta V_i em toda mensagem que ele envia
- Receive: Quando P_i recebe uma mensagem com timestamp t , ele calcula $V_i[j] = \max\{V_i[j], t[j]\}$ para todo $j \neq i$

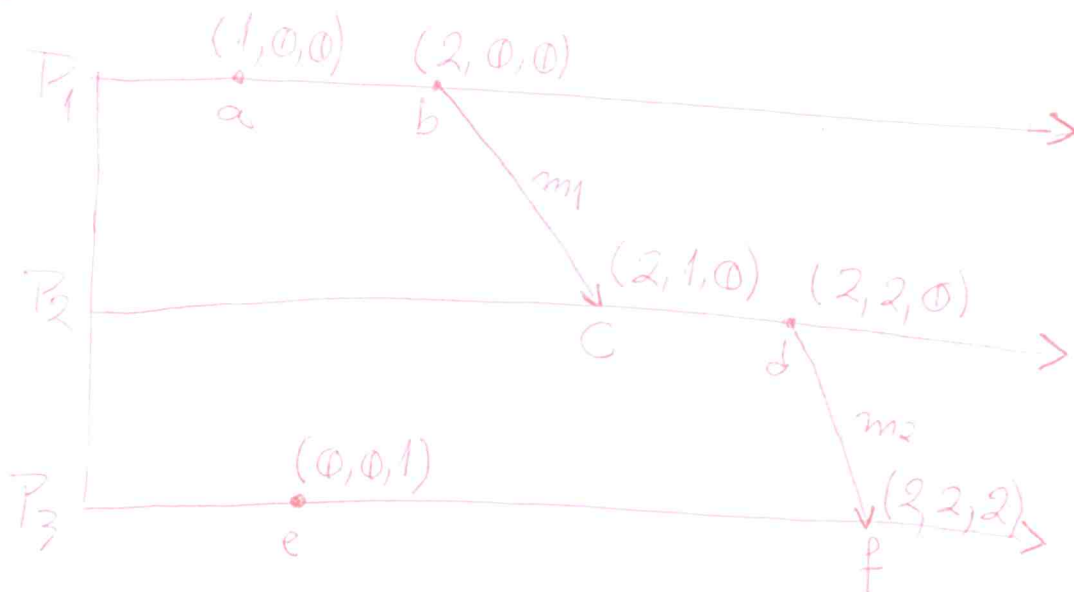
11) Exemplo

P_1 : $a(1,0,0)$; $b(2,0,0)$ envia $(2,0,0)$ de coroa m_1 mensagem m_1

P_2 : o processo recebe $(2,0,0)$ e calcula $\max\{(0,0,0), (2,0,0)\} = (2,0,0)$ adicionando 1 ao seu próprio relógio produzindo $c(2,1,0)$
 Note que, o evento c "sabe" que ocorreram 2 eventos no processo P_1 antes da ocorrência do evento c em P_2

P_3 : análogo!

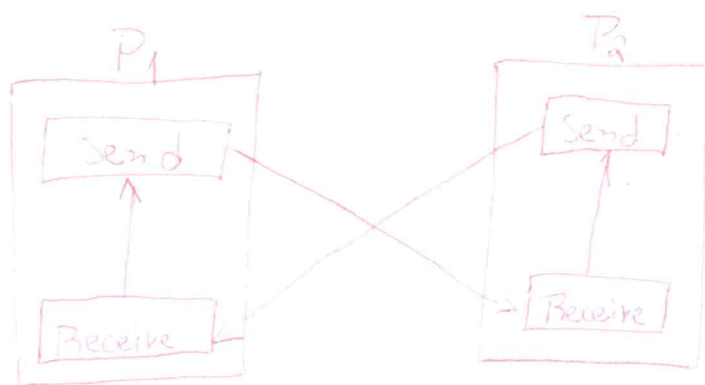
12)



⑬ Programação com MPI

- Comunicação ponto-a-ponto: P_i troca mensagem com P_j (2 participantes na comunicação)
- MPI_Send e MPI_Recv: Rotinas ponto-a-ponto
- MPI_Send e MPI_Recv é bloqueante: Não deixam o programa seguir em frente enquanto não obtiverem confirmação do recebimento da mensagem.

⑭ Problema:

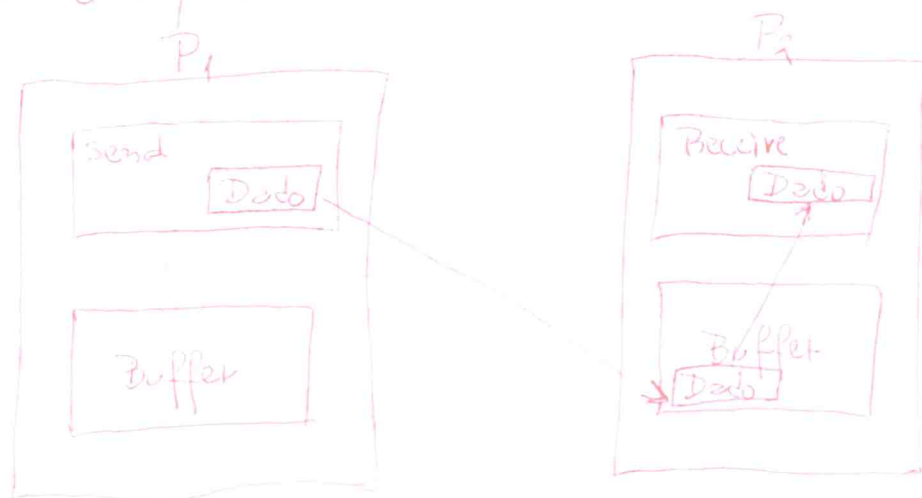


Com Send e Recebe bloqueante existe o problema de "deadlock"

Implemente com MPI e verifique! se isso acontece mesmo!

15) Buffering

Normalmente as implementações de MPI possuem um buffer para o MPI-Send não ser bloqueante.



16) Mensagens não-bloqueantes

- Uma rotina não-bloqueante não espera pelos eventos de confirmação para prosseguir
- MPI_Isend e MPI_Irecv: Rotinas não-bloqueantes
- Impacto no desempenho