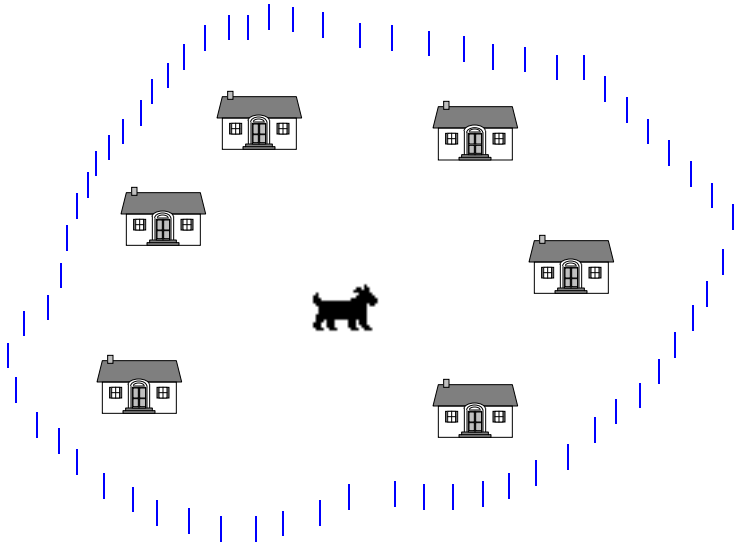
Exclusão Mútua em Sistemas Distribuídos

Recurso deve ser utilizado por apenas um processo de cada vez, com garantia de

- justiça
- ausência de deadlock
- ausência de livelock



Premissas:

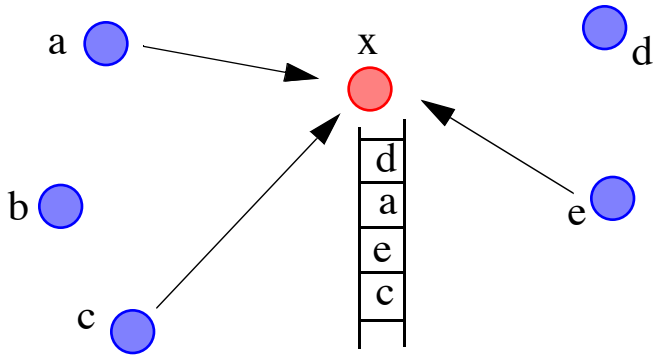
- processos não falham durante acesso a recurso (região crítica)
- processos terminam de usar recurso em tempo finito

Algoritmo de Lamport78

1[a] solução inteiramente distribuída: Lamport

- Time, clocks and the ordering of events in a distributed system, CACM, jul78.

Base do Algoritmo: fila de pedidos centralizada



x recebe todos os pedidos, informa nó que tem prioridade, espera liberação.

Como distribuir a fila de pedidos?

- Todos os nós devem receber todos os pedidos
- Pedidos são ordenados, todos os nós devem ordenar a fila da mesma maneira

Estabelecendo a ordem dos eventos: Relógios Lógicos

A cada evento de uma execução E é atribuído um valor de relógio lógico, que é um elemento de um conjunto ordenado T.

- Exemplo de T: inteiros não negativos.
- Tempos lógicos não precisam ter qualquer relação com tempo real

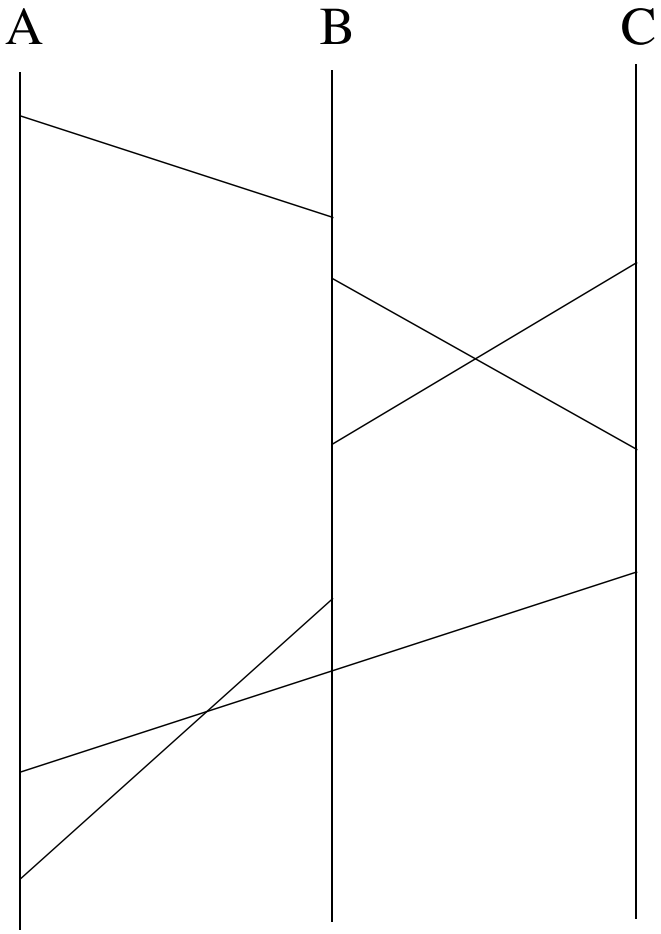
Uma atribuição de relógios lógicos para uma execução E é uma atribuição de um valor de tempo lógico para cada evento, de maneira consistente com todas as possíveis dependências entre eventos de E:

- Nenhum par de eventos pode ter o mesmo valor de tempo lógico
- Tempos lógicos em cada processo são estritamente crescentes, de acordo com a sua ocorrência em E.
- O tempo lógico de cada evento Send é estritamente menor que o evento Receive correspondente.
- Para qualquer valor de $t \in T$, existe um número finito de eventos aos quais são atribuídos valores menores que t.

Ordem lógica "parece" ordem de tempo real para qualquer processo.

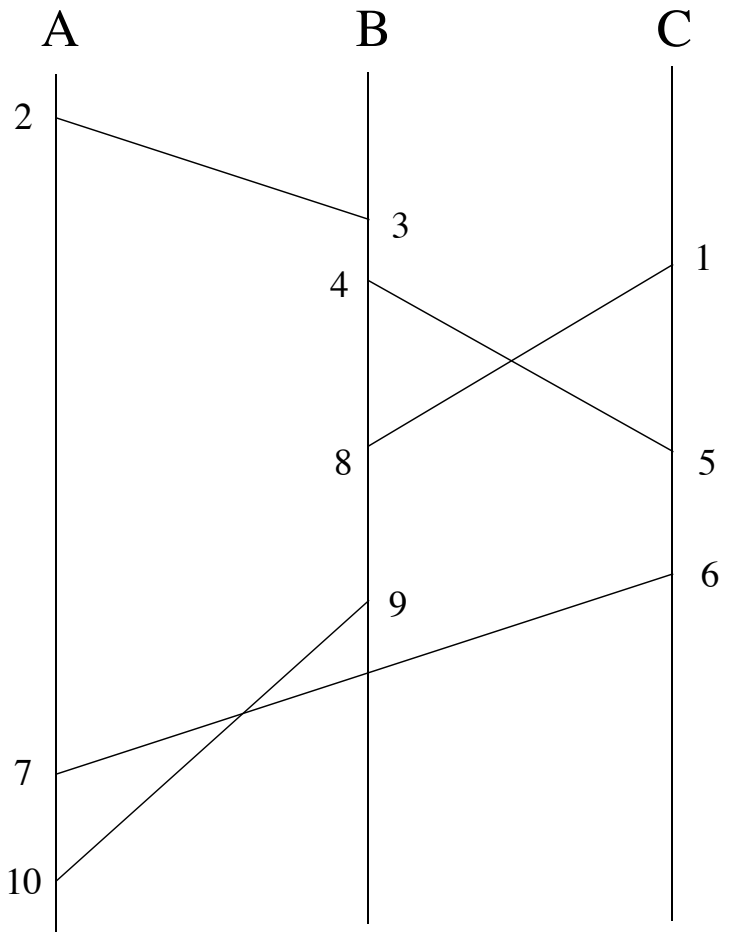
Exemplo

Diagrama de execução



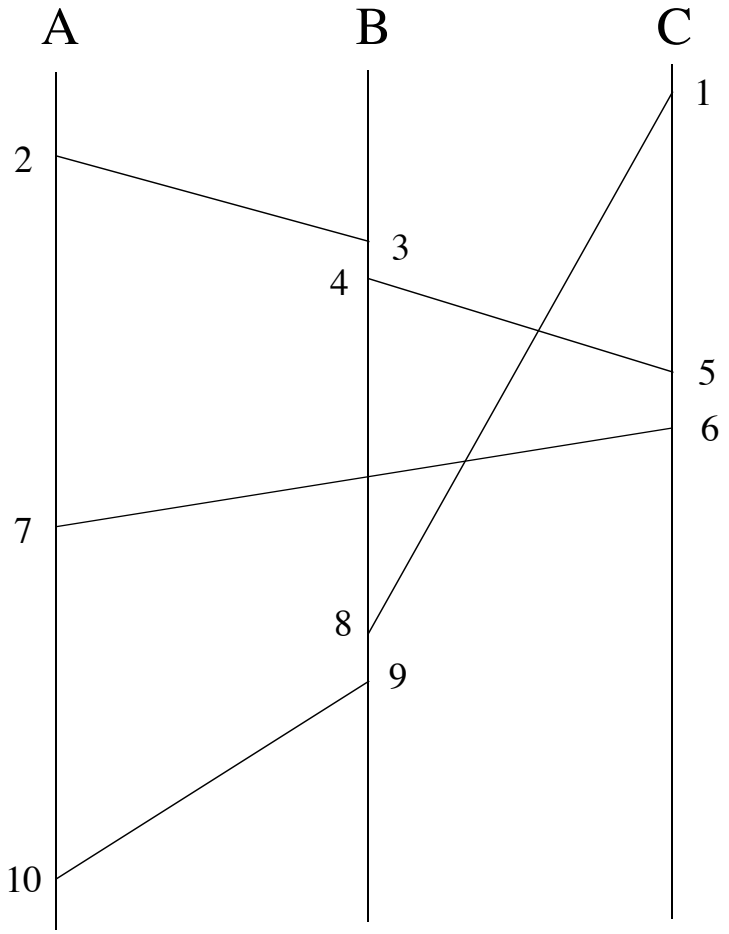
Exemplo

Diagrama de execução



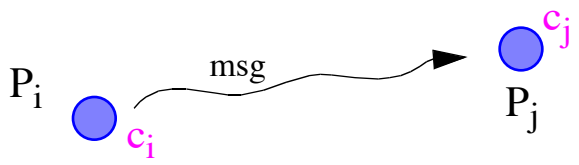
Exemplo

Diagrama de execução (reordenado)



Técnicas para implementação de Tempo Lógico

Lamport: avanço do relógio



Relógio lógico em cada nó c_i: 0.. + ∞

- Entre dois eventos sucessivos, P_i incrementa relógio lógico c_i
- Quando mensagem é enviada de P_i para P_j:
  - P_i carimba mensagem com valor do relógio local e transmite (msg, c_i, i)
  - quando Pj recebe mensagem, acerta relógio local se este está atrasado:

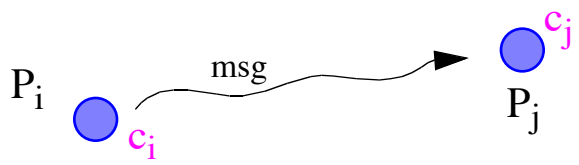
$$\textbf{if } (c_j < c)\ c_j = c_j;$$

e então data o recebimento da mensagem com

$$++\ c_j\ ;$$

Timestamps

Ordenação Causal



Evento X que aconteceu no nó P_i e é datado $c_i(X)$ ocorreu antes do evento Y aconteceu no nó P_j e é datado $c_j(Y)$ se e somente se

- $c_i(X) < c_j(X)$ ou
- $c_i(X) == c_j(X)$ && $P_i < P_j$

Timestamp

c_i	i

Algoritmo de Filas Distribuídas (Lamport78)

- 3 tipos de mensagens:
  - req: P_i envia quando deseja acesso
  - ack: P_j envia em resposta a req
  - rel: P_i envia quando termina acesso

- Fila é um vetor de mensagens q[j] que a cada instante contém uma mensagem recebida de P_j. Inicialmente q[j] == (rel, 0, j)
- Quando mensagem é broadcast por P_i, é colocada também na fila q[i] (como se processo enviasse mensagem para si mesmo)
- P_i ganha acesso quando seu pedido estiver na frente da fila e não houver mensagem anterior ainda em transito

Algoritmo de Filas Distribuídas (Lamport78)

```
inf c=0;
message q[N];
```

```
forall j ≠ i
     send (req, c, i) to j;
q[i] ← (req, c, i); c ++;
wait forall j ≠ i
          timestamp( q[i] ) < timestamp ( q[j] );
< acesso exclusivo >
broadcast ( rel, c, i);
c ++;
```

```
when receive (msg, k, j) {
     c = max (c, k) + 1;
     switch (msg) {
          case req:
               q[j] ← (req, k, j);
               send (ack, c, i) to j;
               break;
          case rel:
               q[j] ← (rel, k, j);
               break;
          case ack:
               if (q[j].typ ≠ req) q[j] ← (ack, k, j);
               break;
     }
}
```

Algoritmo de Filas Distribuídas (Lamport78)

Prova de correção:

- Pedidos são feitos com ordem total
- P_i decide que tem direito de acessar recurso $\Rightarrow$ não existe mensagem de tipo req transmitida antes de sua própria (P_i espera até que tenha recebido uma mensagem recente de todos os outros processos). Nota: P_i pode economizar enviar ack a P_j se P_i já enviou req mas não recebeu ainda a mensagem rel correspondente.

  Ordem total garante justiça e ausência de deadlock

- Processo só retira mensagem da fila dos outros processos quando termina acesso. Enquanto isto, processos não têm acesso devido à ordem total. Portanto, algoritmo garante exclusão mútua.

Número de Mensagens: 3 (N – 1), onde N é o número total de processos participantes.