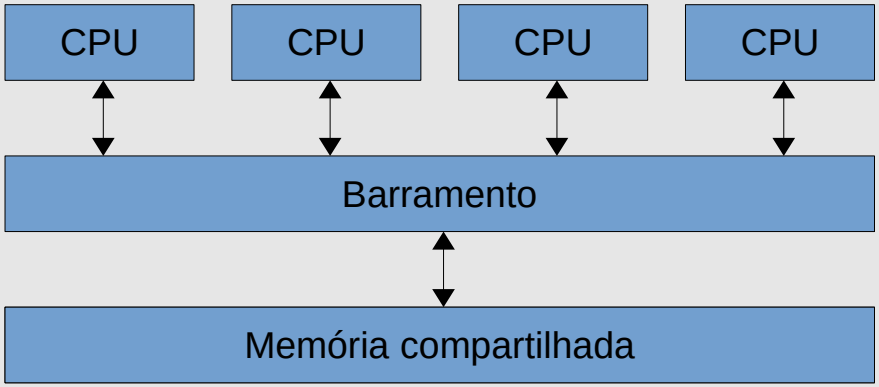
INF 310 – Programação Concorrente e Distribuída

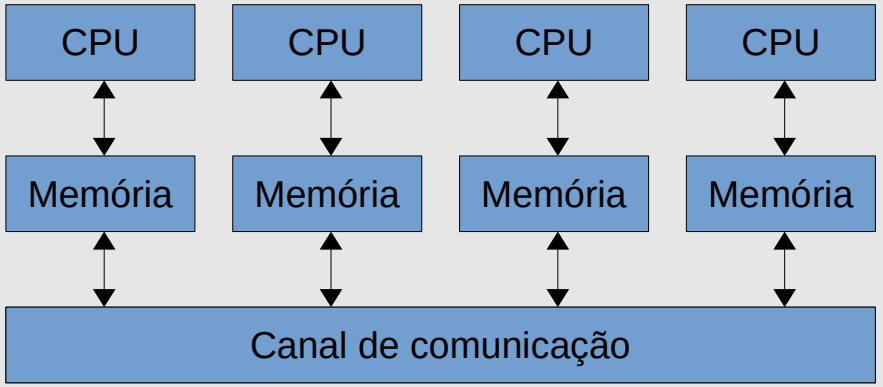
Comunicação via troca de mensagens

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Introdução

- Programação concorrente surgiu junto com a multiprogramação em SO de máquinas convencionais
  - Memória compartilhada
  - Mutex, barreiras, condições...
- Em determinadas situações não existe memória compartilhada
  - Soluções vistas até aqui não se aplicam



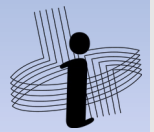


Operações *send* e *receive*

- Denominação genérica das operações de envio (*send*) e recepção (*receive*) de mensagens
- Forma geral
  - *send* (*p*, *msg*) – envia a mensagem *msg* para o processo *p*
  - *receive* (*q*, *msg*) – recebe uma mensagem do processo *q* e a armazena em *msg*

Operações *send* e *receive*

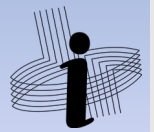
- Possíveis semânticas para as operações *send* e *receive*
  - *send* deve bloquear o remetente até a mensagem ser aceita pelo destinatário?
  - *receive* deve bloquear o receptor caso não haja mensagem a ser recebida?
  - *send* deve identificar um destinatário específico?
  - *receive* deve especificar um remetente específico?
- As 2 primeiras questões estão relacionadas com o sincronismo ou assincronismo da troca de mensagens
  - Quando a primitiva de comunicação bloqueia o processo ela é dita bloqueante
- As 2 últimas questões estão relacionadas com a nomeação explícita ou implícita
  - Quando os processos são nomeados diz-se que a nomeação é explícita. Caso contrário é implícita.



Operações *send* e *receive*

Send	Bloqueante	Não-bloqueante
Nomeação explícita	Envia *msg* para destinatário *d* e espera até mensagem ser aceita	Envia *msg* para destinatário *d* e prossegue execução
Nomeação implícita	Difunde *msg* e espera até mensagem ser aceita	Difunde *msg* e prossegue execução

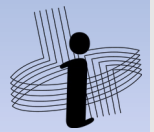
Receive	Bloqueante	Não-bloqueante
Nomeação explícita	Espera mensagem do remetente *r*	Se tem mensagem do remetente *r*, então recebe. Caso contrário, prossegue
Nomeação implícita	Espera mensagem de qualquer remetente	Se tem alguma mensagem, então recebe. Caso contrário, prossegue



UFV

Operações *send* e *receive*

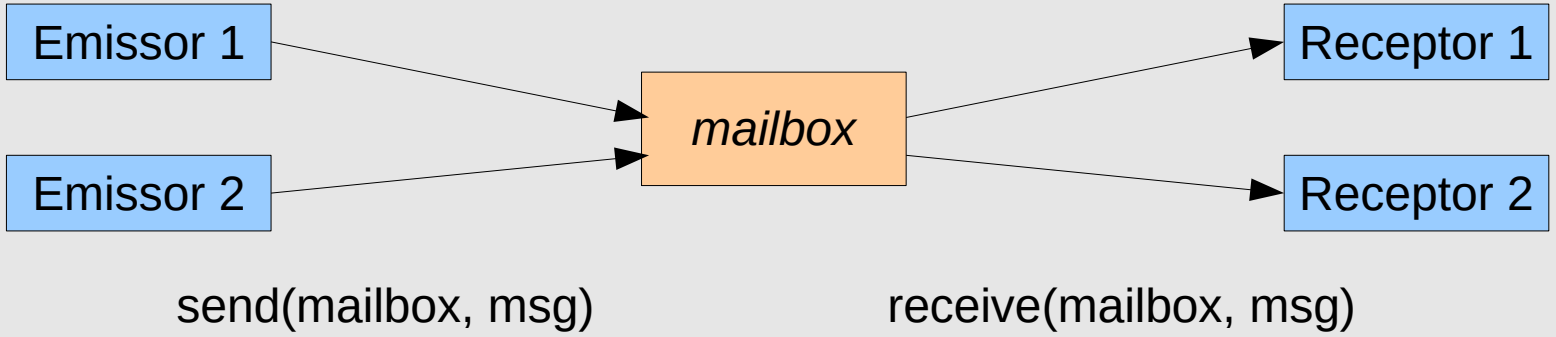
- Operações bloqueantes funcionam como um mecanismo de sincronização
- Operações não bloqueantes requerem o uso de buffers (filas) para armazenar mensagens
- Em geral, o *receive* é implementado como bloqueante.
- A nomeação implícita permite implementação do não-determinismo na comunicação
  - nomeação explícita não implica necessariamente em determinismo (é possível o uso de *receive* não bloqueante para testar vários remetentes)



UFV

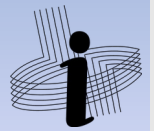
Caixas postais

- Meio termo entre as nomeações implícita e explícita
- Receptor/emissor indica explicitamente a caixa postal ao invés do emissor/receptor
- As caixas postais tornam visíveis as filas de mensagens
- Uma caixa postal pode ser compartilhada por vários processos, tanto para enviar quanto para receber mensagens



Comunicação síncrona

- Forma de comunicação importante e eficiente
- Usa *send* e *receive* bloqueante
  - processos atingem pontos bem definidos para depois prosseguirem em conjunto
- Dispensa o uso de buffers (ou filas) para as mensagens
- Exemplo: produtor-consumidor com buffer limitado

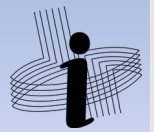


UFV

Produtor-consumidor

```
Process producer;
   msg : integer;
   loop
      “produz msg”
      send(consumer, msg)
   endloop;

Process consumer;
   msg : integer;
   loop
      receive(producer, msg)
      “consome msg”
   endloop;
```

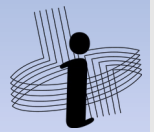


UFV

Produtor-consumidor

```
Process producer;
   msg : integer;
   loop
      “produz msg”
      send(buffer_manager, msg)
   endloop;

Process consumer;
   msg, ready : integer;
   loop
      send(buffer_manager, ready)
      receive(buffer_manager, msg)
      “consome msg”
   endloop;
```

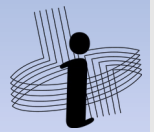


UFV

Produtor-consumidor

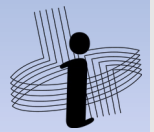
```
Process buffer_manager;
   place : array[0..9] of integer;
   in, out : integer initial 0;
   count : integer initial 0;
   msg : integer;

   loop
      if count>0 and count<10 {
         receive(any, msg)          //recebe de qq processo
         if any = producer {
            place[in] := msg;
            in := (in+1) mod 10;
            count := count + 1;
         }
         else {
            msg := place[out];
            send(consumer, msg);
            out := (out+1) mod 10;
            count := count - 1;
         }
...
```



Produtor-consumidor

```
...
   else if count = 0 {          //buffer vazio
      receive(producer, msg);
      place[in] := msg;
      in := (in+1) mod 10;
      count := count + 1;
   }
   else {
      receive(consumer, msg);  //buffer cheio
      msg := place[out];
      send(consumer, msg);
      out := (out+1) mod 10;
      count := count - 1;
   }
endloop;
```



UFV