# SISTEMAS OPERACIONAIS

## AULA 9 – CONCORRÊNCIA E SINCRONIZAÇÃO DE PROCESSOS, PARTE 2

Prof.ª Sandra Cossul, Ma.

# REVISANDO...

- Uma **solução** para o problema da **região crítica** estará **correta** quando apresentar as seguintes propriedades:
  - Existir exclusividade mútua entre os processos;
  - Um processo não é impedido de entrar na região crítica se nenhum outro estiver executando a sua região crítica;
  - Nenhum processo pode ficar esperando para sempre para entrar na região crítica
  - Não depender da velocidade relativa da execução dos processos, nem da quantidade de processadores (cores) existentes.

# REVISANDO...

- **Soluções incompletas** para o problema da região crítica podem causar:
  - **Starvation** (postergação indefinida – um processo está preso tentando entrar na região crítica e nunca consegue por ser sempre preterido em benefício de outros processos)
  - **Deadlock** – quando dois ou mais processos estão à espera de um evento que nunca vai acontecer

# TIPOS DE SOLUÇÕES PARA OBTER EXCLUSÃO MÚTUA

- **Abordagem por Software**
  - Algoritmos de Dekker e Peterson
  - Os processos executam um determinado algoritmo na entrada e outro na saída da região crítica
- **Abordagem por Hardware**
  - Desabilitação de interrupções e uso de instruções privilegiadas
- **Abordagem alternativa: Suporte do SO ou de uma Linguagem de Programação**
  - monitores, semáforos e transmissão de mensagens

# TIPOS DE SOLUÇÕES PARA OBTER EXCLUSÃO MÚTUA

- Soluções com **abordagem por software** não são muito empregadas na prática devido a:
  - **Complexidade,** o que dificulta a depuração dos programas
  - **Busy-waiting,** quando um processo aguarda em um laço a sinalização de um evento (fica utilizando o CPU sem realizar nenhum processamento útil)

- **Busy-waiting** também é uma possibilidade na abordagem por hardware no uso de instruções privilegiadas.

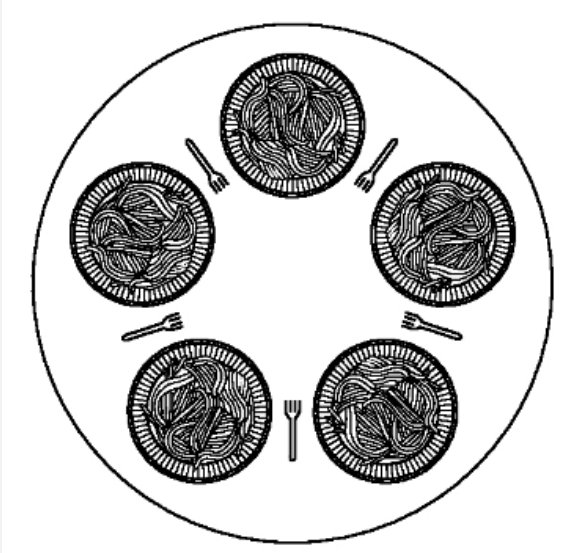
# PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO E COMUNICAÇÃO DE PROCESSOS

- Problemas que representam situações comuns entre os processos relacionados ao compartilhamento de recursos

- É interessante relatar os problemas e testar diferentes soluções para resolvê-los

# PROBLEMAS DE SINCRONIZAÇÃO DE PROCESSOS

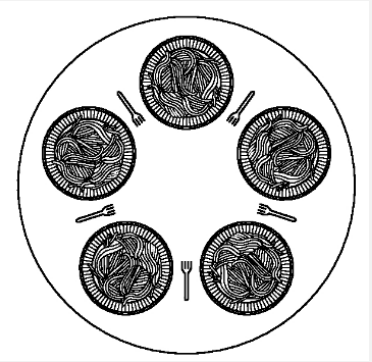
- **Jantar dos filósofos**
  - 5 filósofos estão sentados em uma mesa
  - Cada um com um prato de macarrão a sua frente
  - Entre cada dois filósofos existe um garfo
  - Os filósofos podem estar em um de dois estados possíveis:
    - Meditando ou comendo
  - Para comer, o filósofo precisa de dois garfos
  - Os filósofos não conversam entre si nem podem observar os estados uns dos outros



# PROBLEMAS DE SINCRONIZAÇÃO DE PROCESSOS

- **Jantar dos filósofos**
  - Esse problema representa uma grande classe de **problemas de sincronização** entre **vários processos** e **vários recursos** sem usar um coordenador central

  - Resolver o problema do jantar dos filósofos consiste em encontrar uma forma de **coordenar suas ações de maneira que todos os filósofos consigam meditar e comer.**



# SEMÁFOROS

- **Mecanismo de sincronização entre processos**
- Criado pelo matemático holandes E.W. Dijkstra (1965)
- Apesar de antigo, o semáforo continua sendo o mecanismo de sincronização mais utilizado na construção de aplicações concorrentes, sendo usado de forma explícita ou como base na construção de mecanismos de coordenação mais abstratos, como os monitores.

# SEMÁFOROS

- Tipo abstrato de dado:
  - **valor inteiro** (contador) → S.valor
  - **fila de processos** (inicialmente vazia) → S.fila

- Somente permite duas operações atômicas:
  - **Decrementar/Testar (P)**
  - **Incrementar (V)**

# SEMÁFOROS

- **Operação P (testar/decrementar)**
  - Valor inteiro é decrementado
  - Se valor negativo, processo é bloqueado e inserido no fim da fila desse semáforo

```
P(S):
      S.valor = S.valor - 1;
      Se S.valor < 0
            Então bloqueia o processo, insere em S.fila
```

- **Operação V (incrementar)**
  - Valor inteiro é incrementado
  - Se existir algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado.

```
V(S):
      S.valor = S.valor + 1;
      Se S.fila não está vazia
            Então retira processo P de S.fila, acorda P
```

# SEMÁFOROS

- As **operações P e V são atômicas**, ou seja, não são interrompidas no meio da execução.
  - Evitar condições de disputa sobre as variáveis internas do semáforo e proteger sua integridade

- **Para cada recurso compartilhado**:
  - Criar um semáforo inicializado com o valor 1 (para que somente uma tarefa consiga entrar na seção crítica de cada vez)
  - Todo processo, antes de acessar o recurso, deve executar um P(S)
  - Ao sair da região crítica, o processo deve executar V(S)

# SEMÁFOROS

- **Resolvem o problema da exclusão mútua**
- **Eficiência -** as tarefas que aguardam o semáforos são suspensas e não consomem CPU; quando o semáforo é liberado, somente a primeira tarefa da fila de semáforos é acordada.

- **Justiça -** a fila de tarefas do semáforo obedece uma política FIFO, garantindo que as tarefas receberão o semáforo na ordem das solicitações.

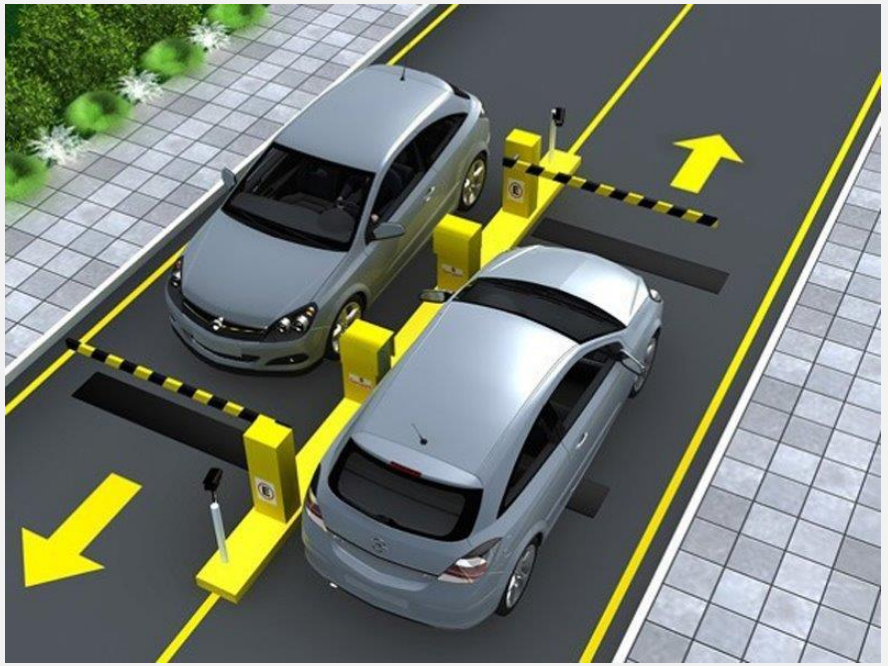
- **Independência -** somente as tarefas que solicitaram o semáforo através da operação decrementa(S) são consideradas na decisão de quem irá obtê-lo.

# SEMÁFOROS

- Além de controlar a exclusão mútua no acesso a seções críticas…
- **O contador interno do semáforo funciona como um contador de recursos:**
  - Se positivo, indica quantas instâncias daquele recurso estão disponíveis
  - Se negativo, indica quantas tarefas estão aguardando aquele recurso

# SEMÁFOROS – EXEMPLO DE USO

- **Controle de vagas em um estacionamento controlado por cancelas**



# SEMÁFOROS – EXEMPLO DE USO

- **Controle de vagas em um estacionamento controlado por cancelas**
- Valor inicial do semáforo S representa o número total de vagas no estacionamento (E)

- Quando um carro deseja entrar no E, ele solicita uma vaga usando decrementa (S)
  - Enquanto o semáforo for positivo, não haverão bloqueios, pois há vagas livres

- Caso não existam mais vagas livres, o carro ficará aguardando no semáforo até que uma vaga seja liberada, o que ocorre quando outro carro sair do E e invocar incrementa (S).

# SEMÁFOROS – EXEMPLO DE USO

```
1  init (vagas, 100) ;          // estacionamento tem 100 vagas
2
3  // cancela de entrada invoca esta operacao para cada carro
4  void obtem_vaga()
5  {
6     down (vagas) ;            // solicita uma vaga
7  }
8
9  // cancela de saída invoca esta operacao para cada carro
10 void libera_vaga ()
11 {
12    up (vagas) ;              // libera uma vaga
13 }
```

# MUTEX

- Versão simplificada de semáforos
- Um variável tipo **mutex** (mutual exclusion) pode assumir apenas os valores: **livre (1) e ocupado (0).**
- Somente duas operações são permitidas sobre o mutex:
  - **Lock –** usada pelo processo para solicitar acesso à região crítica (entrada);
  - **Unlock –** permite ao processo informar que não deseja mais usar a região crítica (saída).

- Na implementação **mutex**, o processo que fez o lock obrigatoriamente deve fazer o unlock.

# MONITORES

- Ao usar **semáforos** ou **mutexes,** o programador precisa identificar explicitamente os pontos de sincronização necessários em seu programa.
- Essa abordagem é eficaz para problemas pequenos e de problemas de sincronização simples, mas se torna **inviável e suscetível a erros em sistemas mais complexos**

- Por exemplo, se o programador esquecer de liberar um semáforo previamente alocado, o programa pode entrar em um impasse.
- Por outro lado, se ele esquecer de requisitar um semáforo, a exclusão mútua sobre um recurso pode ser violada.

# MONITORES

- Foi proposta uma técnica de alto nível: **monitores** pelos cientistas da computação Per Brinch Hansed e Charles Hoare em 1972

- **Um monitor é uma estrutura de sincronização que requisita e libera a seção crítica associada a um recurso de forma transparente, sem que o programador tenha de se preocupar com isso.**

# MONITORES - ELEMENTOS

- **um recurso compartilhado**, visto como um conjunto de variáveis internas ao monitor.
- **conjunto de procedimentos e funções,** que permitem o acesso a essas variáveis
- **Um mutex ou semáforo,** para controle de exclusão mútua; cada procedimento de acesso ao recurso deve obter o mutex antes de iniciar e liberá-lo ao concluir
- **Um invariante,** sobre o estado interno do recurso

# MONITORES

- **É um conjunto especial de procedimentos, variáveis e estruturas de dados, agrupados em um módulo especial ou pacote**

- Os processos podem chamar qualquer procedimento do monitor, mas não podem acessar suas variáveis internas diretamente

# MONITORES

- **Requerimento**:
  - somente um processo de cada vez pode executar alguma rotina no monitor
  - portanto, um segundo processo ficará bloqueado caso chame qualquer uma das rotinas públicas de um monitor que já está sendo executado por algum outro processo

- **Variáveis de condição –** sincronização de controle
  - **Wait()**
  - **Signal()**

# MONITORES

- **Wait**
  - Bloqueia o processo chamador, independente de qualquer outro fator
  - Usada quando o processo entra no monitor e verifica que precisará esperar algum evento para que possa prosseguir  (ex.: quer inserir no buffer, mas este está cheio)

- **Signal**
  - Libera um processo previamente bloqueado via wait
  - Usado quando o evento que o processo bloqueado esperava ocorreu (ex.: dado foi retirado do buffer e não está mais cheio)

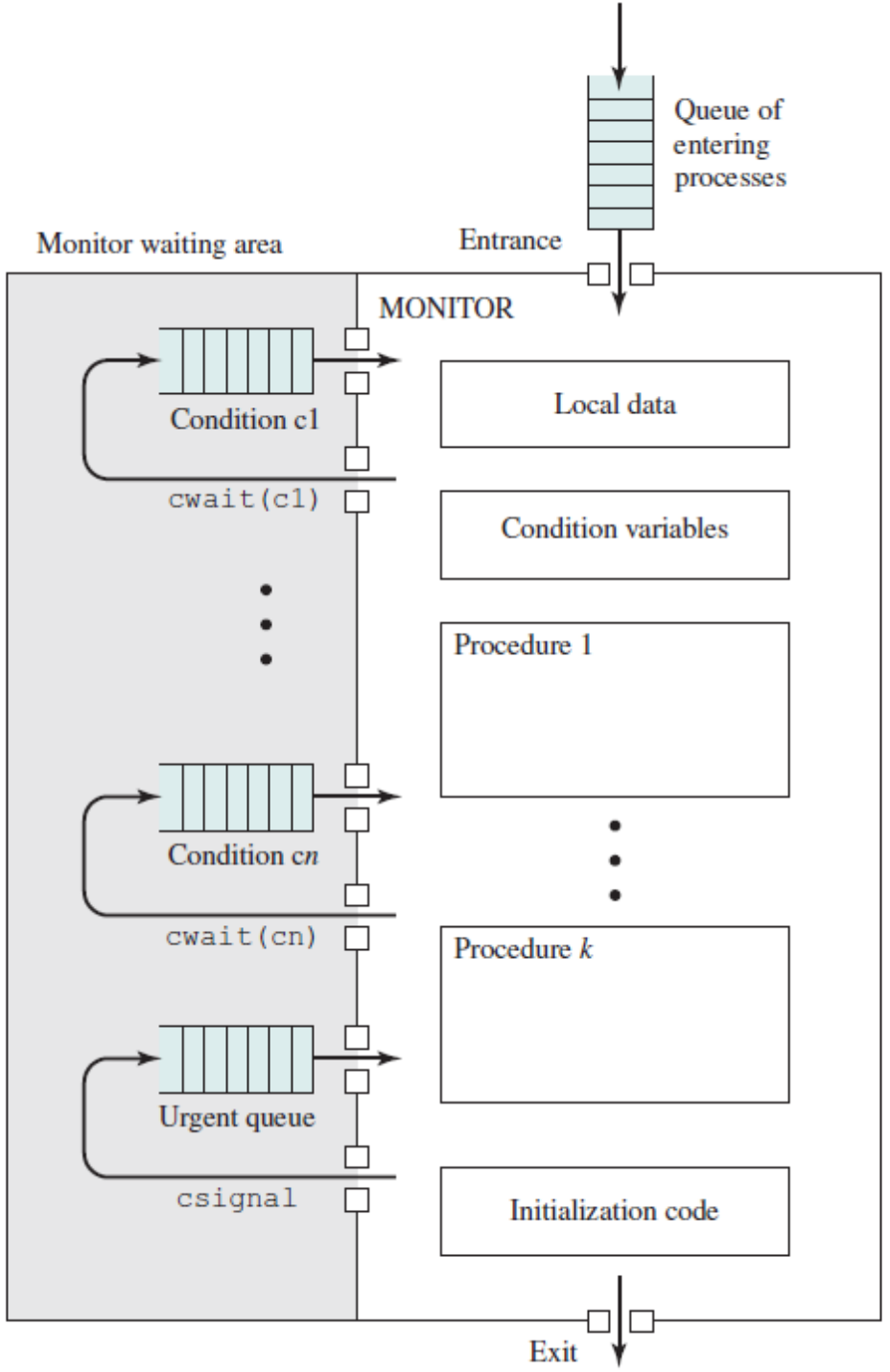
# MONITORES

- Com a restrição de apenas um processo executar por vez no monitor, ele atende e **resolve o problema de exclusão mútua**

- As variáveis de dados no monitor podem ser acessadas por apenas um processo de cada vez

- Assim, um **recurso compartilhado** pode ser **protegido** colocando isso em um **monitor**

# ESTRUTURA DE UM MONITOR



# TRANSMISSÃO DE MENSAGENS

# TRANSMISSÃO DE MENSAGENS

- Na interação entre processos, dois requerimentos devem ser atendidos:
  - **Sincronização** → para garantir exclusão mútua
  - **Comunicação** → processos cooperantes precisam trocar informação

- **Transmissão de mensagens**

# TRANSMISSÃO DE MENSAGENS

- Funciona com duas funções implementadas pelo SO:
  - **Send** (*destino, mensagem*) – envia uma mensagem para um destino qualquer
  - **Receive** (*fonte, mensagem*) – recebe uma mensagem de uma fonte

- A troca de mensagens entre dois processos precisa de sincronização, sendo possível três possíveis implementações:
  - **Bloqueio de envio, bloqueio de recebimento**
  - **Envio não bloqueado, bloqueio de recebimento**
  - **Envio não bloqueado, recebimento não bloqueado**

# TRANSMISSÃO DE MENSAGENS

- **Bloqueio de envio, bloqueio de recebimento**
  - Processos remetente e destinatário bloqueados até que a mensagem seja recebida
- **Envio não bloqueado, bloqueio de recebimento**
  - Processo remetente continua a executar e o destinatário bloqueia até a mensagem requisitada chegar
  - Permite um processo enviar uma ou mais mensagens para vários destinatários de forma rápida (ex.: servidor de processos)
- **Envio não bloqueado, recebimento não bloqueado**
  - Nenhum processo precisa esperar

# TRANSMISSÃO DE MENSAGENS

- Formas de endereçamento das mensagens:
  - **endereçamento direto –** mensagem é enviada explicitamente a um processo em particular (processo remetente deve conhecer o identificador do processo destinatário)
  - **endereçamento indireto –** mensagens são enviadas para estruturas de dados compartilhadas que guardam as mensagens temporariamente (conceito de **caixa postal**).
    - as mensagens são retiradas da caixa postal por outro processo

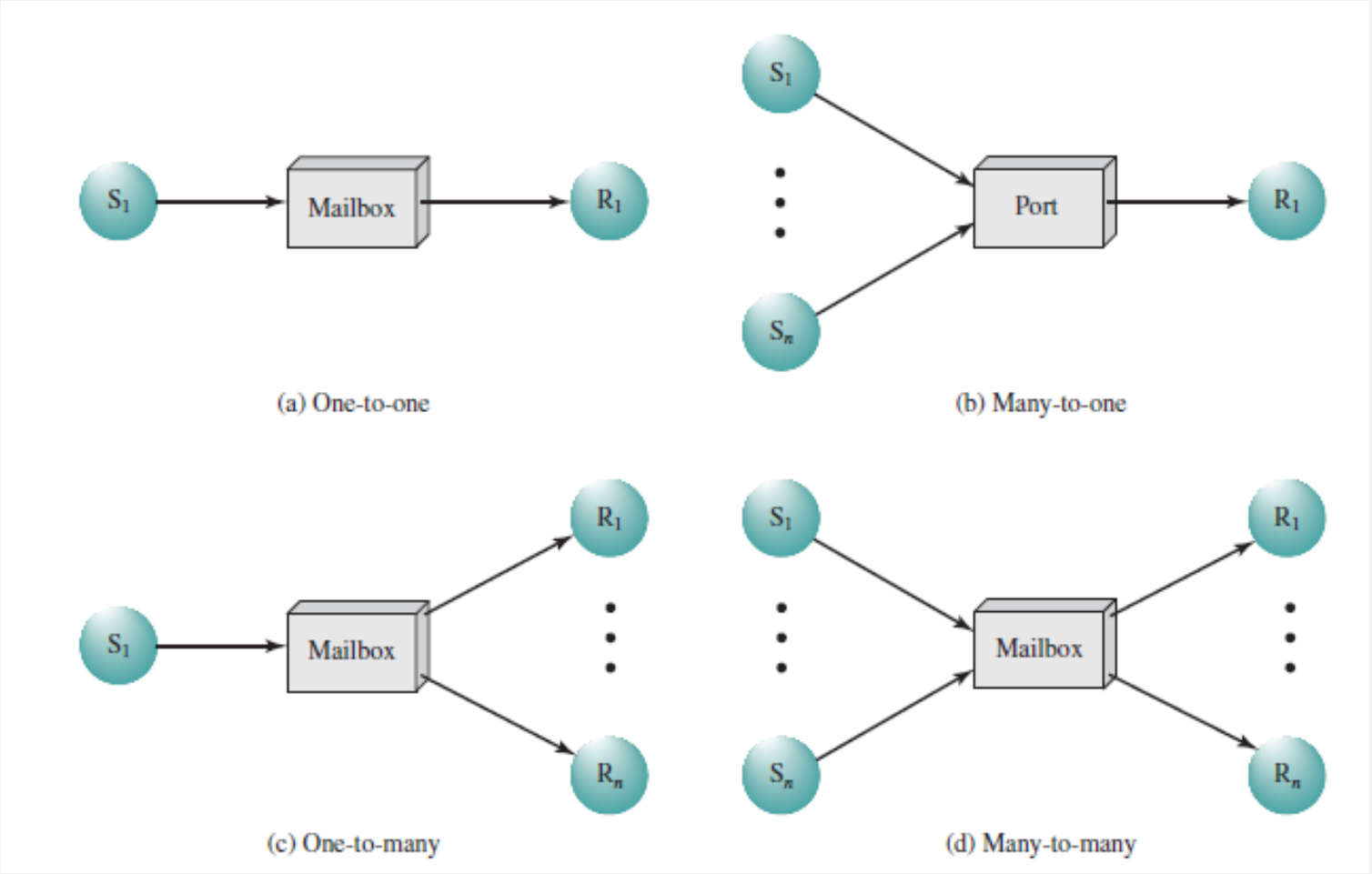
# TRANSMISSÃO DE MENSAGENS

- Uma vantagem de usar endereçamento indireto é a flexibilidade no uso das mensagens, permitindo uma comunicação de **N para N processos**

- **um-para-um:** comunicação privada entre 2 processos
- **muitos-para-um:** múltiplos processos remetentes enviam para um destinatário
- **um-para-muitos:** um processo remetente e múltiplos destinatários (broadcast)
- **muitos-para-muitos:** múltiplos processos servidores provêm serviços para múltiplos clientes

# TRANSMISSÃO DE MENSAGENS

Comunicação privada entre 2 processos!



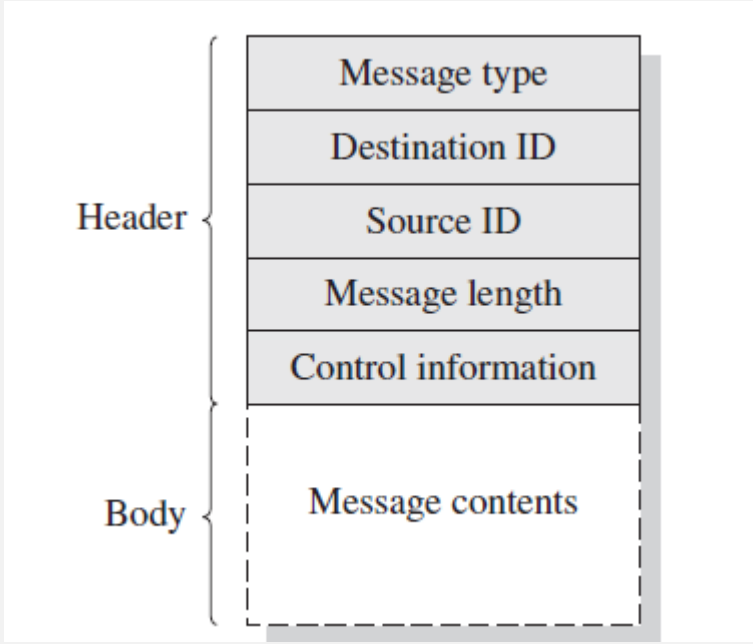

Comunicação cliente-servidor: 1 processo provê serviço para outros processos.

Aplicações em que uma mensagem é transmitida a vários processos.

Permite múltiplos processos de servidores proverem serviços concorrentes a múltiplos clientes.

# TRANSMISSÃO DE MENSAGENS

- **Formato da mensagem**
- O formato é variável
- Pode ser uma sequência de mensagens independentes
- Pode ser um fluxo sequencial e contínuo de dados



# TRANSMISSÃO DE MENSAGENS

- **Regras de enfileiramento**
- O mais simples é FIFO
- Pode não ser suficiente se algumas mensagens são mais importantes que as outras

- Uma alternativa é especificar uma prioridade na mensagem

# TRANSMISSÃO DE MENSAGENS

- **Problemas:**
  - Erros de comunicação como extravio de mensagens
    - Confirmações ACK
    - Relógios para timeouts
    - Retransmissões

- É responsabilidade dos protocolos de comunicação prover mecanismos para a recuperação de falhas simples.

# PRÓXIMA AULA

- Deadlock
- Escalonamento de processos

# BIBLIOGRAFIA


- Tanenbaum, A. S. **Sistemas Operacionais Modernos.** Pearson Prentice Hall. 3rd Ed., 2009.
- Silberschatz, A; Galvin, P. B.; Gagne G.; **Fundamentos de Sistemas Operacionais.** LTC. 9th Ed., 2015.
- Stallings, W.; **Operating Systems: Internals and Design Principles**. Prentice Hall. 5th Ed., 2005.
- Oliveira, Rômulo, S. et al. **Sistemas Operacionais** - V11 - UFRGS. Disponível em: Minha Biblioteca, Grupo A, 2010.
- *baseado nos slides da Prof.ª Roberta Gomes (UFES) e do Prof. Felippe Fernandes da Silva