DCA-108 Sistemas Operacionais

Luiz Affonso Guedes
www.dca.ufrn.br/~affonso
affonso@dca.ufrn.br

Capítulo 3

Programação Concorrente

Conteúdo

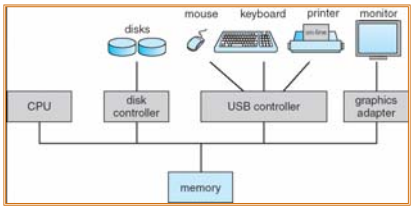
- Caracterização e escopo da programação concorrente.
- Abstrações e Paradigmas em Programação Concorrente
  - Tarefas, região crítica, sincronização, comunicação.
- Redes de Petri como ferramenta de modelagem de sistemas concorrentes.
- Propriedades de sistemas concorrentes
  - Exclusão mútua, Starvation e DeadLock
- Primitivas de Programação Concorrente
  - Semáforos, monitores
  - Memória compartilhada e troca de mensagens
- Problemas clássicos em programação concorrente
  - Produtor-consumidor
  - Leitores e escritores
  - Jantar dos filósofos

Objetivos

- Apresentar os principais conceitos e paradigmas associados com programação concorrente.
- Apresentar Redes de Petri como uma ferramenta de modelagem de sistemas concorrentes.
- Associar os paradigmas e problemas de programação concorrente com o escopo dos Sistemas Operacionais

Recordando

- Cenário Atual dos Sistemas Operacionais
  - Uma ou mais CPUs, controladores de devices conectedos via uma barramento comum, acessando memórias compartilhadas.
  - Execução concorrente de CPUs e devices competindo por recursos.



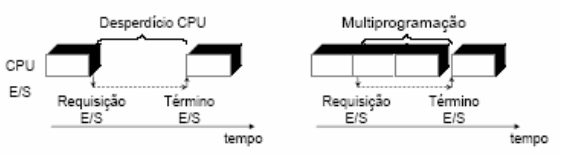
Recordando

Objetivos de sistema operacional

- Executar programas de forma conveniente para o usuário.
- Gerenciar os recursos de software e hardware como um todo.
- Utilizar os recursos de hardware de forma eficiente e segura.

Recordando

- Para se construir SO eficientes, há a necessidade Multiprogramação!



Conseqüências da Multiprogramação

- Necessidade de controle e sincronização dos diversos programas.
- Necessidade de se criar conceitos e abstração novas
  - Modelagem
  - Implementação
- Necessidade de se estudar os paradigmas da Programação Concorrente!

Conceitos de Programação

- Programação Seqüencial:
  - Programa com apenas um fluxo de execução.
- Programação Concorrente:
  - Possui dois ou mais fluxos de execução seqüenciais, que podem ser executados concorrentemente no tempo.
  - Necessidade de comunicação para troca de informação e sincronização.
    - Aumento da eficiência e da complexidade

Paralelismo X Concorrência

- Paralelismo real só ocorre em máquinas multiprocessadas.
- Paralelismo aparente (concorrência) é um mecanismo de se executar "simultaneamente" M programas em N Processadores, quando M > N.
  - N = 1, caso particular de monoprocessamento.
- Programação Distribuída é programação concorrente ou paralela????
  - E o modelo cliente-servidor?

Programação Concorrente

- Paradigma de programação que possibilita a implementação computacional de vários programas seqüenciais, que executam "simultaneamente" trocando informações e disputando recursos comuns.
- Programas cooperantes
  - Quando podem afetar ou ser afetados entre si.

Motivação para o Uso da Programação Concorrente

- Aumento do desempenho
  - Possibilidade de se implementar multiprogramação.
- Possibilidade de desenvolvimento de aplicações que possuem paralelismo intrínseco.
  - SO Modernos, por exemplo.
  - Sistema de Automação Industrial.

Desvantagens da Programação Concorrente

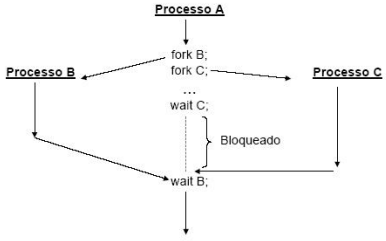
- Programas mais complexos
  - Pois há, agora, vários fluxos de programas sendo executados concorrentemente.
  - Esses fluxos podem interferir uns nos outros.
- Execução não determinística
  - Na programação seqüencial, para um dado conjunto de entrada, o programa irá apresentar o mesmo conjunto de saída.
  - Em programação concorrente, isto não é necessariamente verdade.

Especificação das Tarefas

- Quantas tarefas concorrentes haverá no sistema?
- O quê cada uma fará?
- Como elas irão cooperar entre si?
  - Comunicação entre tarefas.
- Quais recursos elas irão disputar?
  - Necessidade de mecanismo de controle de acesso a recursos.
  - Memória, cpu, devices, etc.
- Qual é a ordem que elas devem executar?
  - Sincronismo entre tarefas.

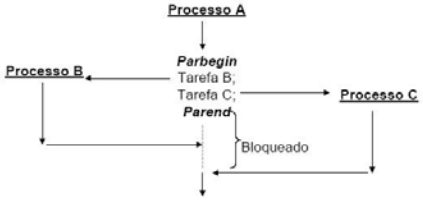
Primitivas de Paralelismo

- fork/joint (fork/wait)



Primitivas de Paralelismo

- parbegin/parend



Problema do Compartilhamento de Recursos

- Programação concorrente implica em compartilhamento de recursos
- Como manter o estado (dados) de cada tarefa (fluxo) coerente e correto mesmo quando diversos fluxos se interagem?
- Como garantir o acesso a um determinado recurso a todas as tarefas que necessitarem dele.
  - Uso de CPU, por exemplo.

Problema de Condição de Corrida

- Ocorre quando duas ou mais tarefas manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são efetuados.
  - Há a necessidade de um mecanismo de controle, senão o resultado pode ser imprevisível.

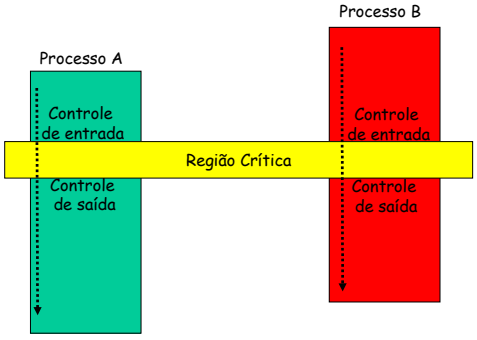
Problema de Exclusão Mútua

- Ocorre quando duas ou mais tarefas necessitam de recursos de forma exclusiva:
  - CPU, impressora, dados, etc.
  - → esse recurso é modelado como uma região crítica.

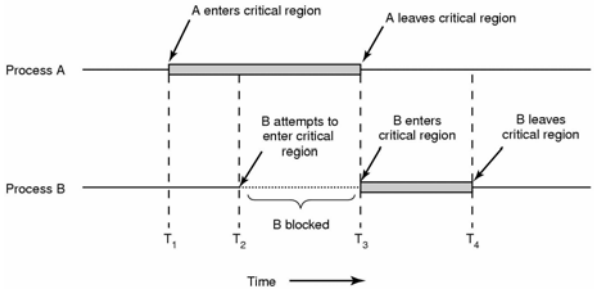
Requisitos Básicos

- Eliminar problemas de corridas
- Criar um protocolo que as diversas tarefas possam cooperar sem afetar a consistência dos dados
- Controle de acesso a regiões críticas
  - Implementação de mecanismos de exclusão mútua.

Exclusão Mútua – Idéia Básica



Exclusão Mútua – Idéia Básica



Propriedades da Região Crítica

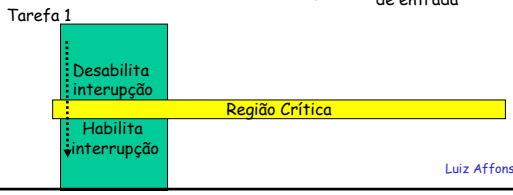
- Regra 1 – Exclusão Mútua
  - Duas ou mais tarefas não podem estar simultaneamente numa mesma região crítica.
- Regra 2 - Progressão
  - Nenhuma tarefas fora da região crítica pode bloquear a execução de uma outra tarefa.
- Regra 3 – Espera Limitada (Starvation)
  - Nenhuma tarefa deve esperara infinitamente para entrar em uma região crítica.
- Regra 4 –
  - Não fazer considerações sobre o número de processadores e nem sobre suas velocidades relativas.

Implementação Iniciais de Exclusão Mútua

- Desativação de Interrupção
- Uso de variáveis especiais de lock
- Alternância de execução

Desabilitação de Interrupção

- Não há troca de tarefas com a ocorrência de interrupções de tempo ou de eventos externos.
- Desvantagens:
  - Uma tarefa pode dominar os recursos.
  - Não funciona em máquinas multiprocessadas, pois apenas a CPU que realiza a instrução é afetada (violação da regra 4)



Variável do Tipo Lock

- Criação de uma variável especial compartilhada que pode assumir dois valores:
  - Zero→ livre
  - 1 → ocupado
- Desvantagem:
  - Apresenta condição de corrida.

```
.
.
.
while (lock == 1);
lock = 1;
Regiao_critica();
lock = 0;
.
.
.
```

Espera ocupada

Se houver uma interrupção nesse ponto?

Alternância

- As tarefas se intercalam no acesso da região crítica.
- Desvantagem
  - Teste contínuo do valor da variável
    - Desperdício do processador: espera ocupada
  - Se as tarefas não necessitares utilizar a região crítica com a mesma freqüência.

Tarefa 2

```
while(true){
   while(vez != 0);
    Regiao_critica();
    vez = 1;
    Regiao_nao_critica();
}
```

Tarefa 2

```
while(true){
   while(vez != 1);
    Regiao_critica();
    vez = 0;
    Regiao_nao_critica();
}
```


Mecanismos Modernos de Exclusão Mútua

- Abordagens Iniciais – Algorítmicas
  - Combinações de variáveis do tipo lock e alternância (Dekker 965, Peterson 1981)
  - Ineficientes
- Abordagens Modernas
  - Primitivas implementadas na linguagem e suportadas pelo SO.
    - Mais eficientes.
    - Não há espera ocupada.
  - Mutex, Semáforo e Monitores

Mutex

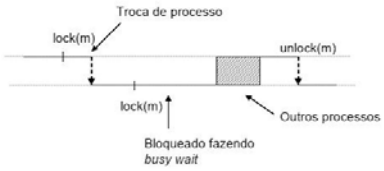
- Variável compartilhada para controle de acesso a região crítica.
- CPU são projetadas levando-se em conta a possibilidade do uso de múltiplos processos.
- Inclusão de duas instruções assembly para leitura e escrita de posições de memória de forma atômica.

```
...
lock(flag);
 Regiao_critica();
 unlock(flag);
 Regiao_nao_critica();
...
```

Operações atômicas

Mutex

- Implementação com espera ocupada (busy waiting)
  - Inversão de prioridade
  - Solução ineficiente



Mutex

- Implementação Bloqueante
  - Evita a espera ocupada
    - Ao acessar um flag ocupado lock(flag), o processo é bloqueado.
    - O processo só é desbloqueado quando o flag é desocupado.
  - Implementação de duas primitivas
    - Sleep → bloqueia um processo a espera de uma sinalização.
    - Wakeup → sinaliza a liberação de um processo.

Semárofos

- Mecanismo proposto por Dijkstra (1965)
- Semáforo é um tipo abstrato de dados:
  - Um valor inteiro não negativo
  - Uma fila FIFO de processos
- Há apenas duas operações atômicas:
  - P(Proberem, Down, Testar)
  - V(Verhogen, Up, Incrementar)

Semáforos

- Operações Atômicas V(s) e P(s), sobre um semáforo s.
  - Semáforo binário: s só assume 0 ou 1.
  - Semáforo contador: s >= 0.

Primitivas P e V

```
P(s): s.valor = s.valor - 1
      Se s.valor < 0 {
        Bloqueia processo (sleep);
        Insere processo em S.fila;
      }
```

```
V(s): s.valor = s.valor + 1
      Se S.valor <=0 {
        Retira processo de S.fila;
        Acorda processo (wakeup);
      }
```


Usando Semáforo para Exclusão Mútua

A iniciação do semáforo s é efetuada em um dos processos

Processo 1

```
...
s = 1;
...
P(s);
Regiao_critica();
V(s);
Regiao_nao_critica();
...
```

Processo 2

```
...

...
P(s);
Regiao_critica();
V(s);
Regiao_nao_critica();
...
```


Semáforo X Mutex

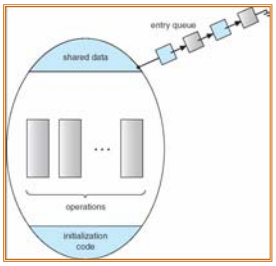
- Mutex
  - As operações lock() e unlock() devem ser executadas necessariamente pelo mesmo processo.
- Semáforos
  - As primitivas V(s) e P(s) podem ser executadas por processos diferentes
  - S pode assumir valor maior que 1.
  - → Gerência de recursos
  - → Mais geral que mutex

Monitores

- Primitiva de alto nível para sincronização de processo.
- Bastante adequado para programação orientada a objetos.
- Somente um processo pode estar ativo dentro de um monitor de cada.

Sintaxe de um Monitor

- Um monitor agrupar várias funções mutuamente excludentes.




```
monitor monitor name
{
  // shared variable declarations
  initialization code ( . . . ) {
    . . .
  }
  public P1 ( . . . ) {
    . . .
  }
  public P2 ( . . . ) {
    . . .
  }
      .
      .
      .
  public Pn ( . . . ) {
    . . .
  }
}
```


Monitores X Semáforos

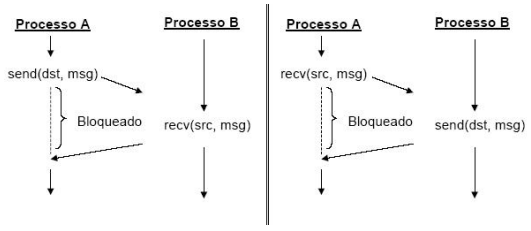
- Monitores permitem estruturar melhor os programas
- Pode-se implementar Monitores através de Semáforos e vice-versa.
- Java inicialmente só implementava monitores.
  - Atualmente também possui Semáforos

Troca de Mensagem

- Primitivas do tipo mutex, semáforos e monitores são baseadas no compartilhamento de variáveis
  - Necessidade do compartilhamento de memória
  - Sistemas distribuídos não existe memória comum
- Necessidade de outro paradigma de programação
  - Troca de mensagens
- Primitivas
  - Send(mensagem) e Receive(mensagem)
  - RPC (Remote Procedure Call)

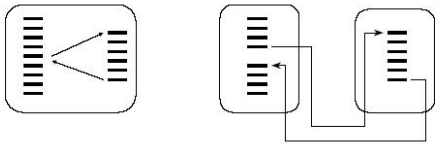
Primitivas Send e Receive

- Primitivas bloqueante ou não bloqueante
- Exemplos
  - Sockets, MPI, etc.



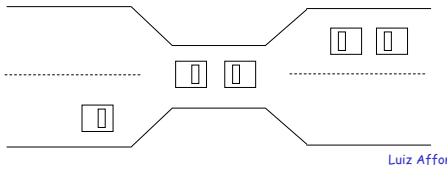
Remote Procedure Call (RPC)

- Primitiva baseada no paradigma de linguagens procedurais.



DeadLock

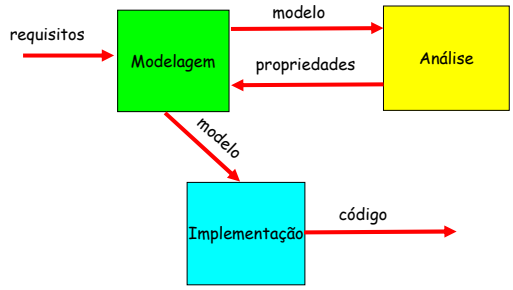
- Travamento perpétuo:
  - Problema inerente em sistemas concorrentes.
- Situação na qual um, ou mais processos, fica eternamente impedido de prosseguir sua execução devido ao fato de cada um estar aguardando acesso a recursos já alocados por outro processo.



Condições para Haver Deadlock

1. Exclusão mútua
   - Todo recurso ou está disponível ou está atribuído a um processo.
2. Segura/espera
   - Os processos que detém um recurso podem solicitar novos recursos.
3. Recurso não-preempitível
   - Um recurso concedido não pode ser retirado de um processo por outro.
4. Espera Circular
   - Existência de um ciclo de 2 ou mais processos, cada uma esperando por um recurso já adquirido (em uso) pelo próximo processo no ciclo.

Ciclo de Desenvolvimento



Ciclo de Desenvolvimento

- Programas Concorrentes são mais complexos do que Programas Seqüenciais.
  - Problema de deadlock
    - Travamento perpétuo
  - Problema de starvations
    - Atraso por tempo indeterminado
  - Existência de sincronismo e comunicação entre tarefas.
- Necessidade de ferramentas de Modelagem e Análise de Sistemas Concorrentes;
  - Redes de Petri é uma dessas ferramentas!

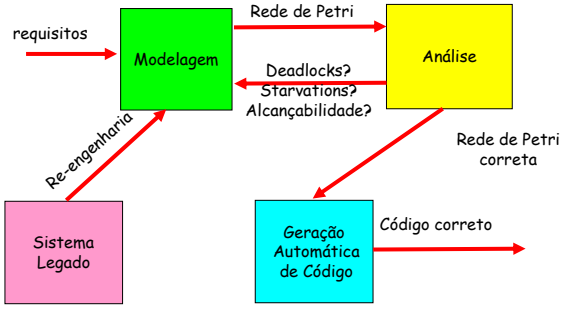
Redes de Petri

- Rede de Petri é uma ferramenta matematicamente bem formulada, voltada à modelagem e à análise de sistemas a eventos discretos.
  - Permite avaliar propriedades estruturais do sistema modelado.
  - Sistemas a eventos discretos: sistemas de computação e sistemas de manufatura.
- Redes de Petri possuem representações gráfica e algébrica.

Redes de Petri - Histórico

- Proposta por Carl Adam Petri em 1962, em sua tese de doutorado
  - Kommunikation mit Automaten
  - Comunicação assíncrona entre componentes de sistemas de computação.
- Extensão de Autômatos Finitos
- Na década de 1970, pesquisadores do MIT mostraram que Redes de Petri poderiam ser aplicadas para modelar e analisar sistemas com componentes concorrentes.

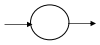
Redes de Petri – Ciclo de Desenvolvimento



Rede de Petri – Comportamento Básico

- Rede de Petri é um grafo orientado.
- O grafo modela as pré-condições e pós-condições dos eventos que podem ocorrer no sistema.
- Eventos mudam a configuração do sistema.
- Uma configuração representa o estado do sistema.

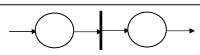
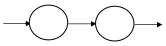
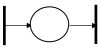
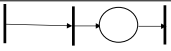
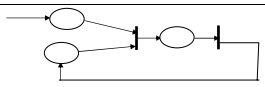
Redes de Petri – Elementos Básicos

Símbolo	Componente	Significado
	Lugar	Um estado, um recurso. Vértice do grafo
	Transição	Eventos instantâneos. Vértice do grafo
	Arco	Ligam lugares a transições e vice-versa. Aresta do grafo.
	Marcação	Condição, quantificação de recursos (habitam nos lugares)

Redes de Petri – Regras Básicas

1. Arcos só podem ligar lugares a transições e vice-versa.
2. O disparo de uma transição muda a configuração de marcação da Rede.
3. ...
4. ...

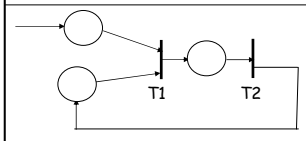
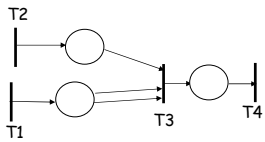
Redes de Petri – Regras Básicas

Rede de Petri	Correta?
	
	
	
	
	

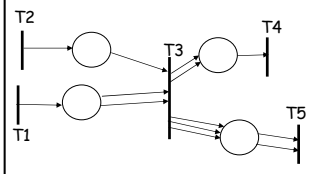
Redes de Petri – Regras Básicas

1. ...
2. ...
3. Uma transição está habilitada a disparar se todos os lugares que levam a ela (lugares de entrada) têm marcações em quantidade igual ou maior aos seus respectivos arcos de ligação.
4. O disparo de uma transição é instantâneo e provoca a retirada de marcações dos lugares de entrada (em quantidade igual a cardinalidade de seus arcos) e coloca marcações nos lugares de saída (em quantidade igual a cardinalidade de seus arcos).

Redes de Petri – Regras Básicas

Rede de Petri	Seqüência de Transições
	
	

Redes de Petri – Regras Básicas

Rede de Petri	Seqüência de Transições
	

Redes de Petri – Exemplos de Uso

- Estado de uma máquina
- Estado de um processo
- Máquina de fazer
- Sanduíche
- Modelagem de trânsito

Redes de Petri – Exemplo

- Modelagem da operação de uma CPU
  - Condições
    - a – a CPU está esperando um processo ficar apto.
    - b – um processo ficou apto e está esperando ser executado.
    - c – a CPU está executando um processo.
    - d – o tempo do processo na CPU terminou.
  - Eventos
    - 1 – um processo fica apto.
    - 2 – a CPU começa a executar um processo.
    - 3 – a CPU termina a execução do processo.
    - 4 – o processo é finalizado.

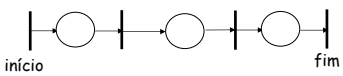
Eventos	Pré-condições	Pós-condições

Redes de Petri – Padrões

- Evolução dos Processos
  - Cooperação
  - Competição
  - Paralelismo
  - Seqüência
  - Decisões
  - Loops
- Modelagem
  - Causa-efeito
  - Dependências
  - Repetições

Redes de Petri – Padrões

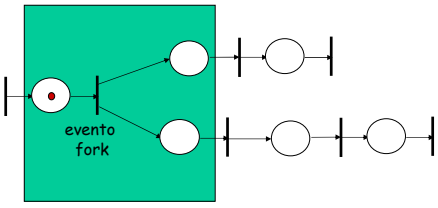
- Seqüência
  - Dependência de precedência linear de causa-efeito.
  - Seqüência de um processo de fabricação.
  - Atividades seqüenciais.



- Evolução assíncrona
  - Independência entre os disparos de transições.
  - Há várias possíveis seqüências de disparo.

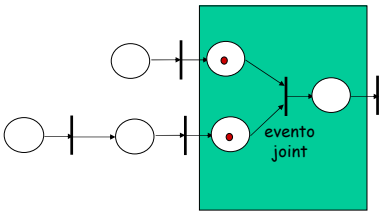
Redes de Petri – Padrões

- Separação (fork)
  - Evolução em paralelo.
  - Fork de processos.
  - Separação de produtos.



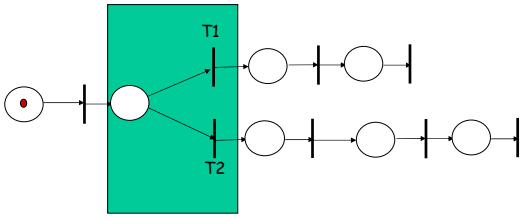
Redes de Petri – Padrões

- Junção (joint)
  - Ponto de sincronização
  - União de processos.
  - Composição de produtos.



Redes de Petri – Padrões

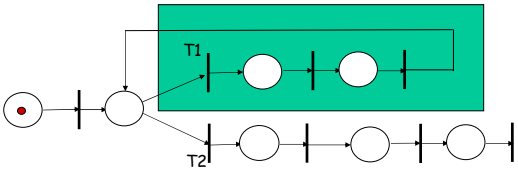
- Decisão (If)
  - Um caminho ou outro mutuamente excludentes.
  - Controle de decisão.



Decisão: T1 ou T2

Redes de Petri – Padrões

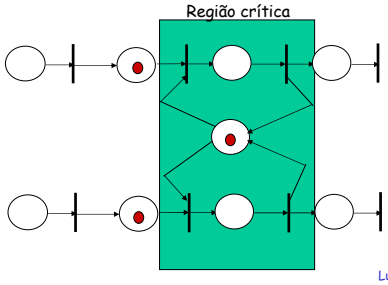
- Repetição (Loop)
  - Repetir até que ...
  - Há realimentação.



Repetir até que T2 ocorra antes de T1

Redes de Petri – Padrões

- Árbitro
  - Guarda região crítica.
  - Implementa exclusão mútua.
  - Mutex, semáforos, monitores.



Redes de Petri e Programação Concorrente

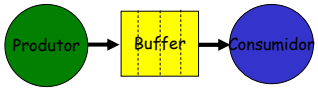
- Problemas Clássicos de Programação Concorrente
  - Produtor X Consumidor, com Buffer unitário
  - Produtor X Consumidor, com n Buffers
  - Produtor X Consumidor, com Buffer circular
  - Leitores X Escritores
  - Jantar dos Filósofos (deadlock)
  - Barbeiro Dorminhoco

Redes de Petri e Programação Concorrente

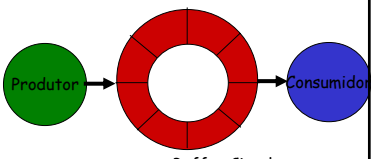
- Produtor X Consumidor, com Buffer unitário
  - Processos consumidor e produtor acessam um buffer unitário de forma intercalada.
    - Ciclo de produção, consumo, produção, consumo, ...
  - Faça o modelo em Rede de Petri e depois o Pseudo-código.
    - Há a necessidade de quantos semáforos na solução?
  - Esse é um problema recorrente em Sistemas Operacionais.

Redes de Petri e Programação Concorrente

- Produtor X Consumidor, com n Buffers
  - O buffer pode suportar até N dados.



- Produtor X Consumidor, com Buffer circular.



Buffer Circular

Redes de Petri e Programação Concorrente

- Leitores X Escritores
  - Há N processos **leitores** e K **escritores**.
  - Os leitores não são bloqueantes, mas os escritores são.
  - Problema típico de leitura e escrita.
  - Faça o modelo em Rede de Petri do problema e depois o seu pseudo-código.

Redes de Petri e Programação Concorrente

- Problema do Jantar dos Filósofos
  - Há 5 filósofos e 5 garfos
  - Cada filósofos possui 2 estados possíveis
    - Comendo ou pensando.
  - Para comer, há a necessidade de se pagar 2 garfos.
    - Os garfos só podem ser pegos 1 de cada vez
  - Faça o modelo em Rede de Petri e depois o pseudo-código livre de deadlock.
    - A solução deve ser o menos restritiva possível.

Redes de Petri e Programação Concorrente

- Problema do Barbeiro Dorminhoco
  - Há 5 cadeira numa dada barbearia.
  - O barbeiro só pode cortar o cabelo de 1 cliente por vez.
  - Se não houver cliente, o barbeiro fica dormindo.
  - Ao chegar, se não houver lugar, o cliente vai embora. Se houver lugar ele fica esperando. Se não houver ninguém esperando, ele bate na porta do barbeiro.
  - Faça o modelo em Rede de Petri do problema, com o seu respectivo pseudo-código.