Programação Concorrente e Paralela

Capítulo 6 – Vivacidade e Método Protegido

Marcial Porto Fernández
marcial@larces.uece.br

Semestre 2018.2

Sumário

- Vivacidade (Liveness)
  - Propriedade Progresso
- Deadlock
  - O problema do Jantar dos Filósofos
  - Detectar e evitar deadlock
- Métodos Protegidos
  - Verificar as condições de guarda
  - Lidar com as interrupções
  - Estruturação de notificação

Sumário

- Vivacidade (Liveness)
  - Propriedade Progresso
- Deadlock
  - O problema do Jantar dos Filósofos
  - Detectar e evitar deadlock
- Métodos Protegidos
  - Verificar as condições de guarda
  - Lidar com as interrupções
  - Estruturação de notificação

Vivacidade ou Progresso (Liveness)

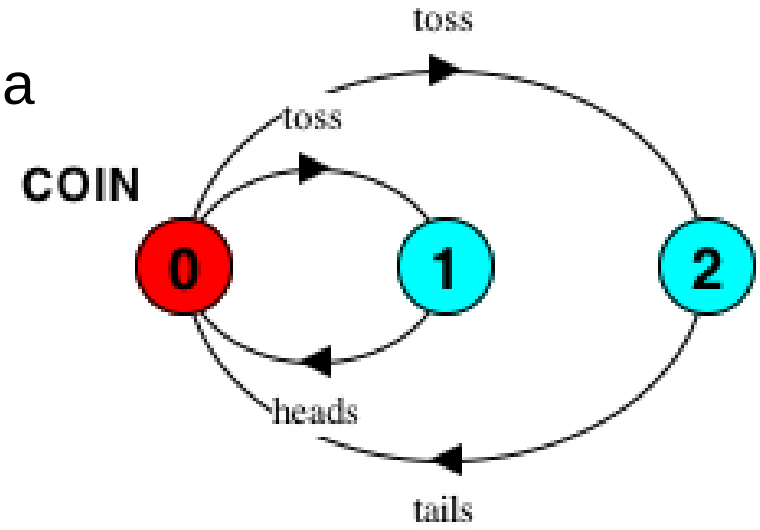
- A propriedade vivacidade ou progresso (liveness) afirma que algo de bom acontece eventualmente.
- A propriedade vivacidade ou progresso (liveness) afirma que sempre ocorrerá uma ação que será eventualmente executada.
- Os termos vivacidade e progresso remetem ao mesmo termo em inglês: liveness. Assim, serão considerados sinônimos.
- Progresso é o oposto de fome (starvation), o nome dado a uma situação em que em um programa concorrente não deixa uma outra ação ser executada.

Problemas de Vivacidade (liveness)

- Um programa pode ser "**seguro**", mesmo assim pode sofrer de vários tipos de problemas de vivacidade (liveness):
  - Fome (starvation): ("adiamento indefinido")
    - O sistema como um todo progride, mas alguns processos individuais não liberam os recursos.
  - Dormência:
    - Um processo de espera não é acordado
  - Terminação prematura:
    - Um processo é terminado antes de concluir
  - Impasse:
    - Dois ou mais processos estão bloqueados, cada um esperando que o outro libere um recursos.

Propriedade Progresso - Escolha Justa

Escolha Justa: Se a escolha sobre um conjunto de transições é executado infinitamente, então a transição em cada conjunto será executado infinitamente.



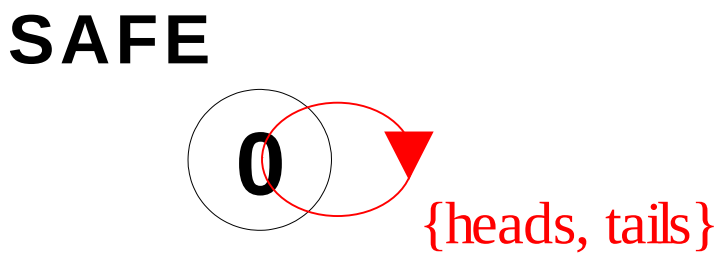
Se uma moeda for lançada um número infinito de vezes, seria de se esperar que ambas, cara (heads) e coroa (tail), cada uma seria escolhido infinitamente.

Isso pressupõe a escolha justa!

```
COIN =   ( toss -> heads -> COIN
   | toss -> tails -> COIN ).
```

Segurança vs Vivacidade

```
property SAFE =  ( heads -> SAFE
                 | tails -> SAFE ).
```



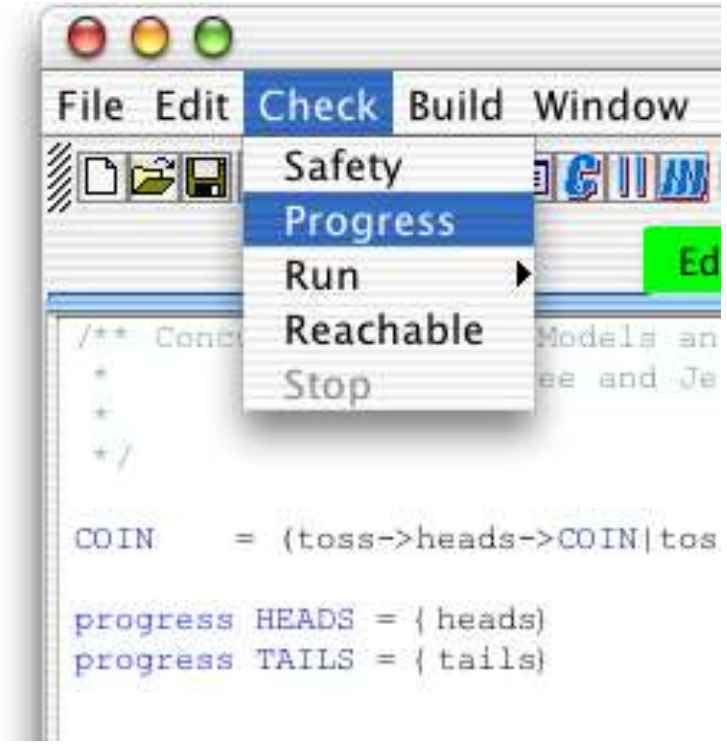
As características de segurança da moeda não são muito interessantes.

Como podemos expressar o que deve acontecer?

Propriedade Progresso

```
progress P = {a1,a2..an}
```

afirma que, em uma *execução infinita* de um sistema destino, *pelo menos uma* das ações a1, a2 ... será executada *infinitamente*.



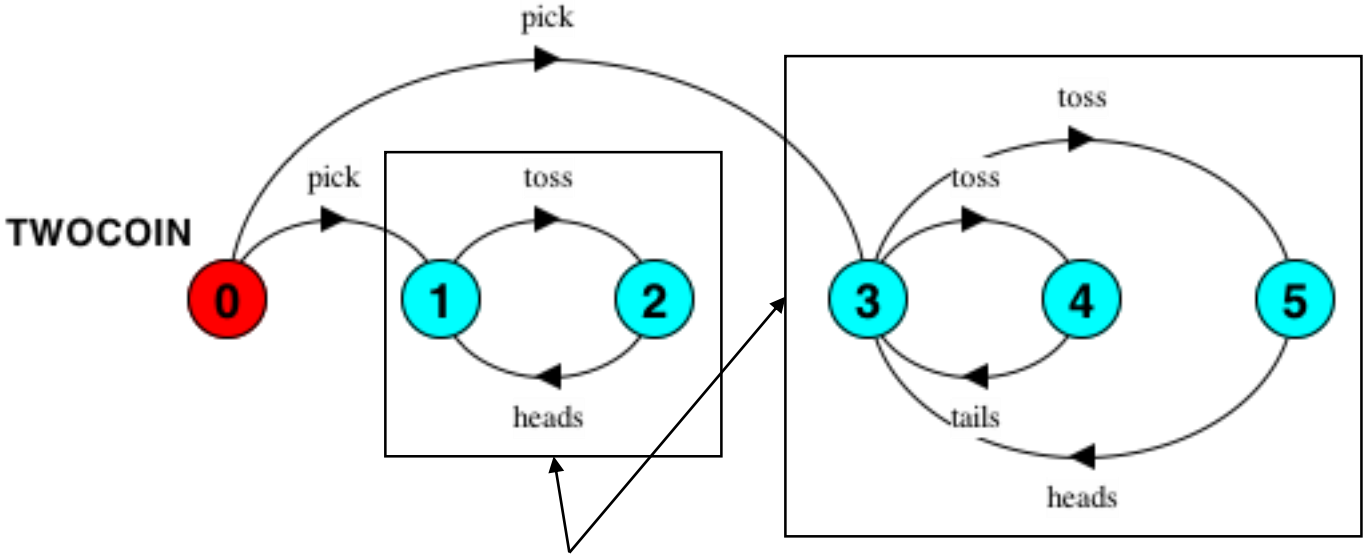

```
progress HEADS = {heads}
progress TAILS = {tails}
```

```
Nenhuma violação de progresso detectada.
```

Propriedade Progresso

- Suponha que você tenha duas moedas: uma normal e uma falsa (TRICK)

```
TWOCOIN   = ( pick->COIN | pick->TRICK ),
TRICK     = ( toss->heads->TRICK ),
COIN      = ( toss->heads->COIN | toss->tails->COIN ).
```



Terminal sets

Análise do progresso

- Um terminal set é um conjunto de estados em que cada estado é mutuamente alcançável mas nenhuma transição leva para fora do conjunto.
- O terminal set {1, 2} viola propriedade progresso.

```
progress HEADS = {heads}
progress TAILS = {tails}
progress HEADSorTAILS = {heads,tails}
```

```
Progress violation: TAILS
Trace to terminal set of states: pick
Actions in terminal set: {toss, heads}
```

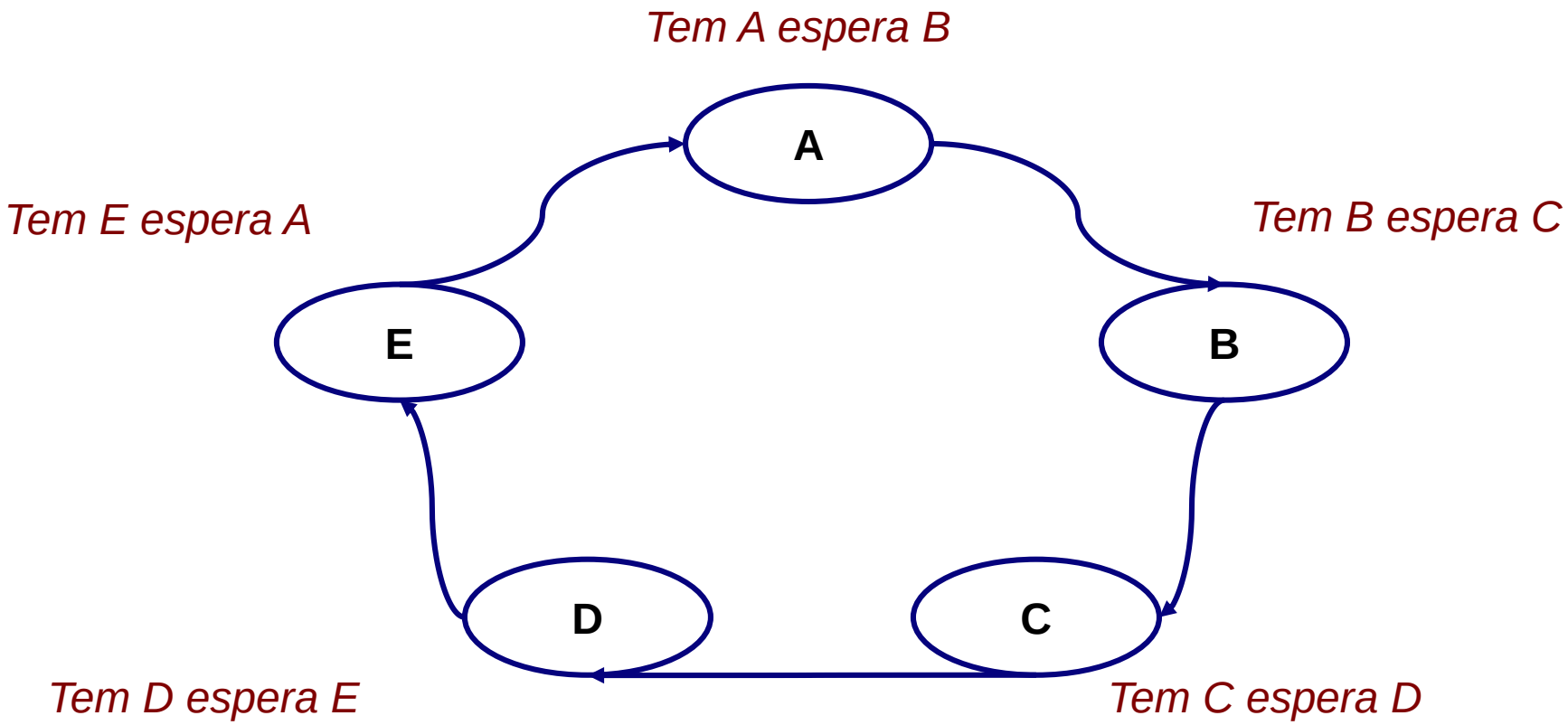
Sumário

- Vivacidade (Liveness)
  - Propriedade Progresso
- Deadlock
  - O problema do Jantar dos Filósofos
  - Detectar e evitar deadlock
- Métodos Protegidos
  - Verificar as condições de guarda
  - Lidar com as interrupções
  - Estruturação de notificação

Impasse (Deadlock)

- Existem quatro condições necessária e suficiente para impasse (deadlock):
  - **Uso de recursos reutilizáveis**: os processos em deadlock compartilham recursos com exclusão mútua.
  - **Aquisição incremental**: processos seguram os recursos adquiridos durante a espera esperando obter os recursos adicionais.
  - **Não preempção**: Uma vez adquirido por um processo, recursos não podem ser cedido a outro a não ser de forma voluntária.
  - **Ciclo de espera**: Existe um ciclo de processos, em que cada processo tem um recurso que o seu sucessor no ciclo está esperando para usá-lo.

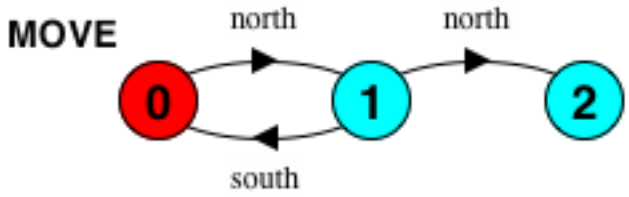
Ciclo de espera (Jantar dos Filósofos)



Análise de Impasse

- Um estado de deadlock é um estado sem transições de saída.

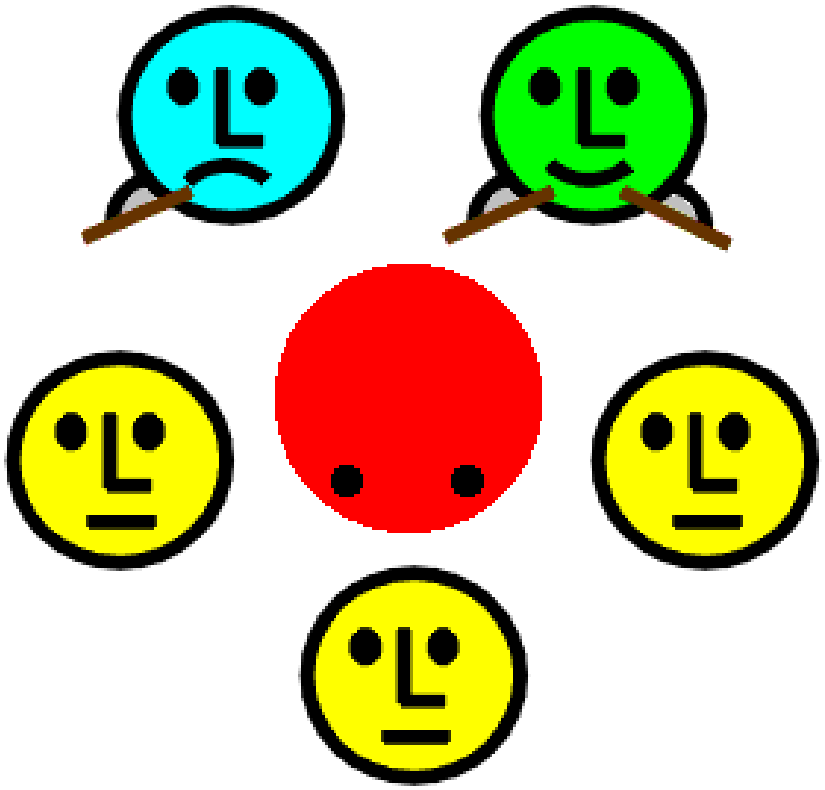
```
MOVE = ( north ->
         ( south -> MOVE
         | north -> STOP
         )
       ).
```




```
Progress violation for actions: {north, north}
```

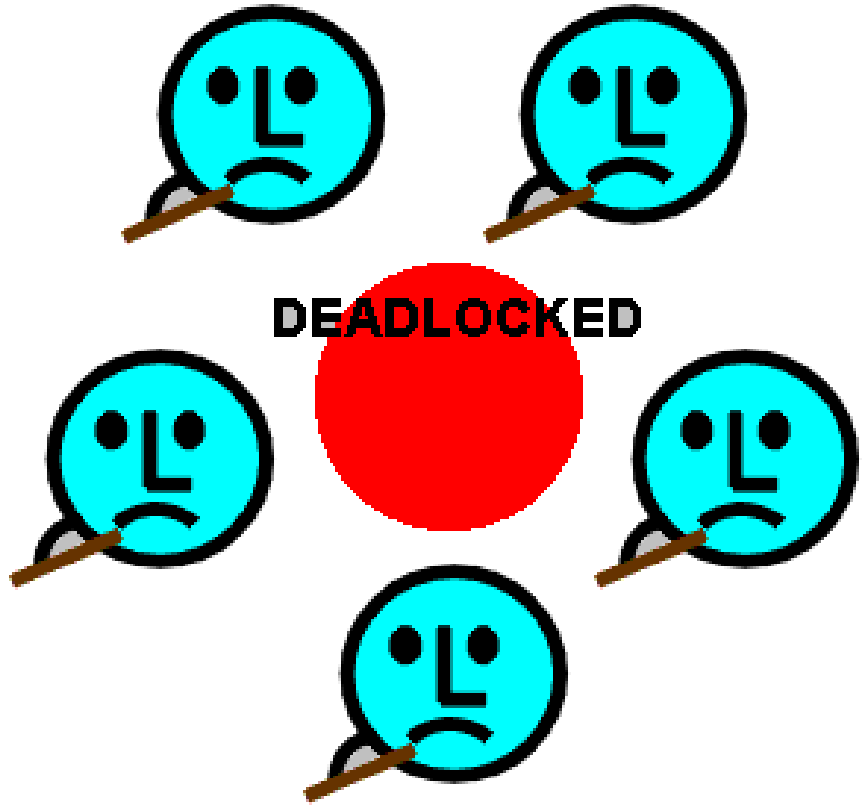
O Problema Jantar dos Filósofos

- Filósofos alternam entre pensar e comer.
- Um filósofo necessita de dois garfos para comer.
- Dois filósofos não podem segurar um garfo simultaneamente.
- Não deve haver bloqueio nem fome (starvation).
- O objetivo é obter um comportamento eficiente com ausência de contenção.



Filósofos Deadlocked

- Um bloqueio ocorre quando ocorre um ciclo de espera quando cada filósofo pega um garfo e fica esperando por outro garfo.



Filósofos: Segurança e vivacidade

O Jantar dos Filósofos ilustram problemas de segurança e muitas questões de progresso:

Exclusão Mútua	Cada garfo pode ser usado por um filósofo de cada vez
Condição de sincronização	Um filósofo precisa de dois garfos para comer
Comunicação com variável compartilhada	Filósofos compartilham garfos...
Comunicação baseada em mensagens	... ou eles podem passar garfos uns aos outros.
Espera ocupada	Um filósofo pode esperar por garfos ...
Espera bloqueada	... ou pode dormir até ser acordado por um vizinho
Livelock	Todos os filósofos podem pegar o garfo da esquerda e ficam esperando (espera ocupada) pelo garfo da direita ...
Deadlock	... ou pegar o da esquerda e aguardar (dormindo) pelo da direita
Fome (starvation)	Um filósofo pode morrer de fome se os vizinhos da esquerda e direita são sempre mais rápidos em pegar os garfos
Condições de corrida	Comportamento anômalo dependente do tempo

Modelagem do jantar dos filósofos

```
PHIL   =   (  sitdown
       -> right.get -> left.get
       -> eat -> left.put -> right.put
       -> arise -> PHIL ).

FORK   =   (get -> put -> FORK).

||DINERS(N=5)=  forall [i:0..N-1]
( phil[i]:PHIL || {phil[i].left,phil[((i-1)+N)%N].right}::FORK ).
```

Este sistema é seguro? É vivo?

Análise Jantar dos Filósofos

```
Trace to terminal set of states:
      phil.0.sitdown
      phil.0.right.get
      phil.1.sitdown
      phil.1.right.get
      phil.2.sitdown
      phil.2.right.get
      phil.3.sitdown
      phil.3.right.get
      phil.4.sitdown
      phil.4.right.get
Actions in terminal set: {}
```

Progresso não é possível devido ao ciclo de espera

Eliminando Deadlock

- Existem duas abordagens diferentes para eliminar impasse.
  - Detecção de Deadlock:
    - Repetidamente verifica a ocorrência de ciclo de espera. Quando detectada, escolher uma vítima e obrigá-lo a liberar seus garfos.
    - Comuns em sistemas transacionais, a vítima deve executar um "roll-back" e tentar novamente
  - Evitar Deadlock:
    - Projeto do sistema de modo que nunca deixe criar um ciclo de espera.

Soluções para Jantar dos Filósofos

- Há várias soluções que oferecem diferentes graus de garantias de progresso (liveness):
  - Quebrar o ciclo
    - Numerar os garfos. Filósofo pega sempre o garfo com menor número em primeiro lugar.
    - Um dos filósofo (e apenas um) pega os garfos na ordem inversa.
  - Filósofos esperam na fila para sentar
    - Permitir não mais que quatro filósofos sentar-se em um instante

Será que essas soluções evitar impasse?
E sobre a fome? São eles "justo"?

Alcançar Progresso

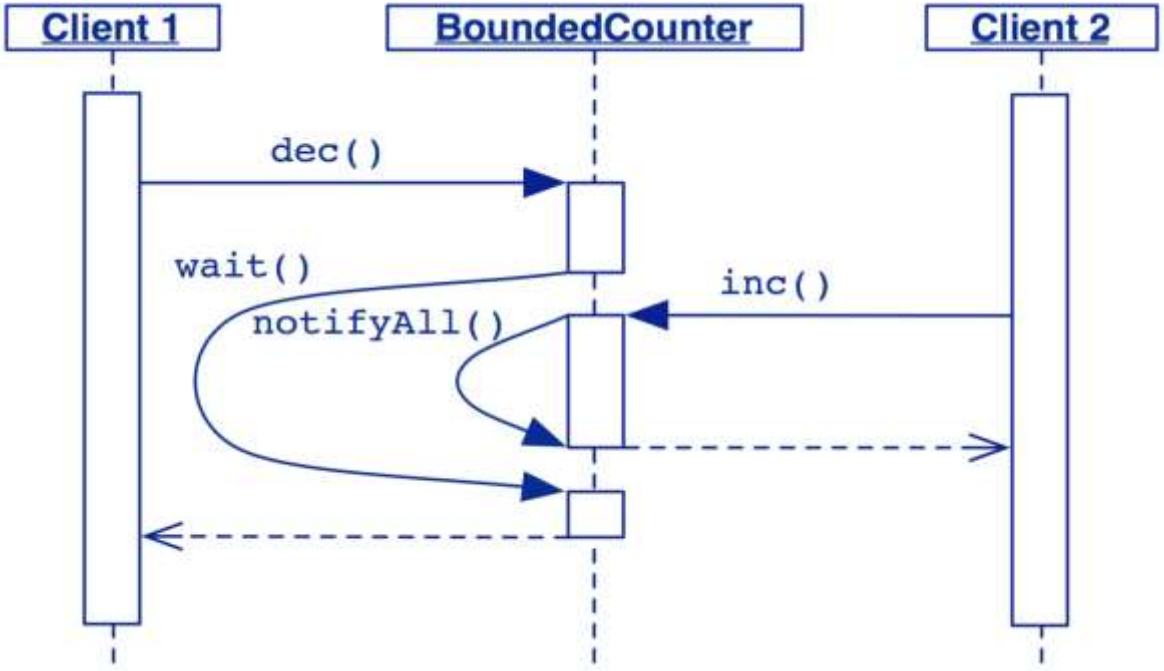
- Existem várias estratégias e técnicas para garantir o progresso:
  - Começar com um projeto seguro e remover seletivamente a sincronização
  - Começar com um projeto vivo e seletivamente adicionar segurança
  - Adotar padrões de projeto que limitam a necessidade de sincronização
  - Adotar arquiteturas padrão que evitem dependências cíclicas

Sumário

- Vivacidade (Liveness)
  - Propriedade Progresso
- Deadlock
  - O problema do Jantar dos Filósofos
  - Detectar e evitar deadlock
- Métodos Protegidos
  - Verificar as condições de guarda
  - Lidar com as interrupções
  - Estruturação de notificação

Padrão: Métodos Protegidos

- Ideia: Suspender temporariamente um thread de entrada quando um objeto não está no estado pronto (ready) para atender uma solicitação, e esperar que o estado mude ao invés de se esquivar (gerando uma exceção).



Métodos Protegidos – Aplicabilidade (1)

- Os clientes devem tolerar o adiamento por tempo indeterminado (podendo suspender).
- Você deve garantir que os estados necessários são atingíveis (através de outros processos), ou se não, que seja aceitável bloquear para sempre.
- Você deve providenciar para que notificações ocorram após ocorrerem todas as mudanças de estado relevantes. (Caso contrário, considerar um projeto baseado em espera ocupada).

Métodos Protegidos – Aplicabilidade (2)

- Você deve evitar ou lidar com os problemas de progresso (liveness) devido à thread em espera que retenham todos os bloqueios de sincronização.
- Você deve construir predicados computáveis descrevendo o estado em que as ações terão sucesso.
- Condições e ações devem ser gerenciados como um único objeto.

Métodos Protegidos - Etapas do Projeto

- A receita básica é usar wait em um loop condicional para bloquear até que seja seguro prosseguir, e usar notifyAll para acordar threads bloqueados.

```
public synchronized Object service() {
            while (wrong State) {
                  try { wait(); }
                  catch (InterruptedException e) { }
            }
            // fill request and change state ...
            notifyAll();
            return result;
      }
```

Verificar as condições de guarda...

- Definir um predicado que descreva com precisão as condições em que as ações podem prosseguir (Isto pode ser encapsulado como um método auxiliar).
- Preceder as ações condicional com uma loop de espera guardada da forma:

```
while (!condition) {
      try { wait(); }
      catch (InterruptedException ex) { ... }
}
```

Verificar as condições de guarda ...

- Se houver apenas uma condição para se verificar nesta classe (e todas as possíveis subclasses) e as notificações são emitidas somente quando a condição for verdadeira, então não há necessidade de re-verificar o estado após o retorno do wait()
- Garantir que o objeto está em um estado consistente (ou seja, mantém as invariantes de classe), antes de entrar em qualquer wait.
  - A maneira mais fácil de fazer isso é realizar as guardas antes de tomar qualquer ação.

Lidar com interrupções (1)

- Estabelecer uma *política* para lidar com InterruptedExceptions. As possibilidades incluem:
  - **Ignore as interrupções** (Isto é, uma cláusula catch vazio), que preserva a vivacidade em detrimento de acontecer algum problema de segurança.
  - **Terminar o thread atual** (parar). Isso preserva a segurança, embora brutalmente!

Lidar com interrupções (2)

- Possibilidades (continuação):
  - **Sair do método**, possivelmente gerando uma exceção. Isso preserva a vivacidade mas poderá exigir do chamador tomar alguma medida especial para preservar a segurança.
  - **Limpar** e reiniciar.
  - **Peça a intervenção do usuário** para decidir o que fazer antes de prosseguir.

Sinalizar mudanças de estado (1)

- Adicionar código de notificação para cada método da classe que mude de estado que possa afetar o valor de uma condição de guarda. Algumas opções são:
  - use **notifyAll** para acordar todos os threads que estão bloqueados na espera de algum evento.
  - use **notify** para acordar apenas um thread. Este é a forma ideal para uma otimização onde:
    - todos os threads bloqueados estão, necessariamente, à espera de condições sinalizado pelas mesmas notificações,
    - apenas um deles pode ser ativado por qualquer notificação dada.

Sinalizar mudanças de estado (2)

- Você pode construir seus próprios métodos de notificação para fins especiais usando notify e notifyAll.
  - Por exemplo, seletivamente notificar threads específicos para oferecer garantias de imparcialidade ou prioridade.

Fim do Capítulo 6

- Esse capítulo apresentou a modelagem de simultaneidade e sincronização condiciona com Java Active Object.
- Revise os conceitos apresentados.
- Resolver os exercícios da Lista.