Sistemas Distribuídos

Simultaneidade 2

António Rui Borges

Resumo

- *Princípios gerais de simultaneidade*
  - *Regiões críticas*
  - *Condições de corrida*
  - *Impasse e adiamento indefinido*
- *Dispositivos de sincronização*
  - *Monitores*
  - *Semáforos*
- *Biblioteca de simultaneidade Java*
- *Leituras sugeridas*

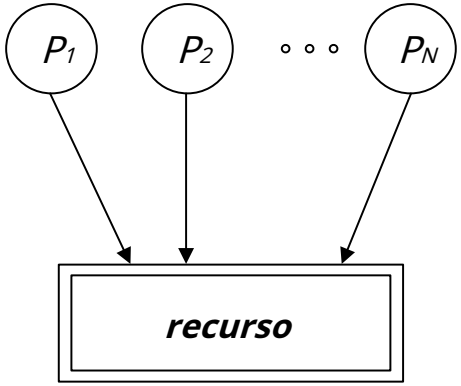
Princípios gerais de simultaneidade - 1

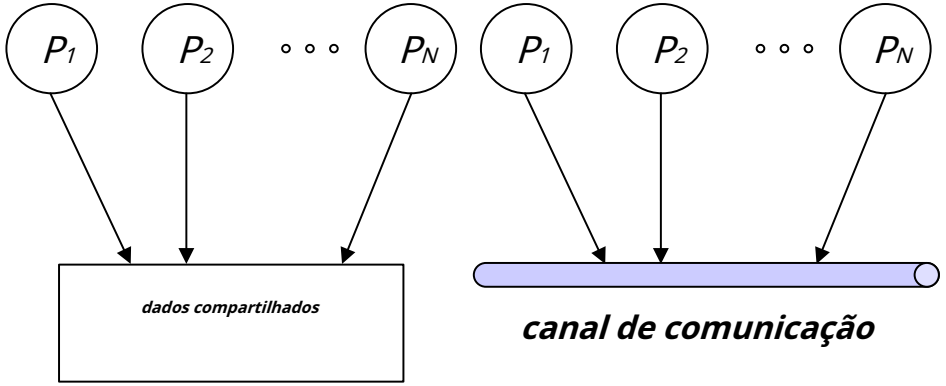
Em um ambiente multiprogramado, processos coexistentes podem apresentar diferentes comportamentos em termos de interação.

Eles podem atuar como

- *processos independentes*–quando são criados, vivem e morrem sem interagir explicitamente entre si; a interação subjacente é implícita e tem suas raízes no *concorrência*pelos recursos do sistema computacional; normalmente são processos criados por usuários diferentes, ou pelo mesmo usuário para finalidades diferentes, em um ambiente interativo, ou processos que resultam de*trabalho*processamento em um*lote*ambiente
- *processos cooperativos*–quando compartilham informações ou se comunicam de forma explícita;*compartilhamento*pressupõe um espaço de endereçamento comum, enquanto *comunicação*pode ser realizada compartilhando o espaço de endereçamento, ou através de um canal de comunicação que conecte os processos intervenientes.

Princípios gerais de simultaneidade - 2



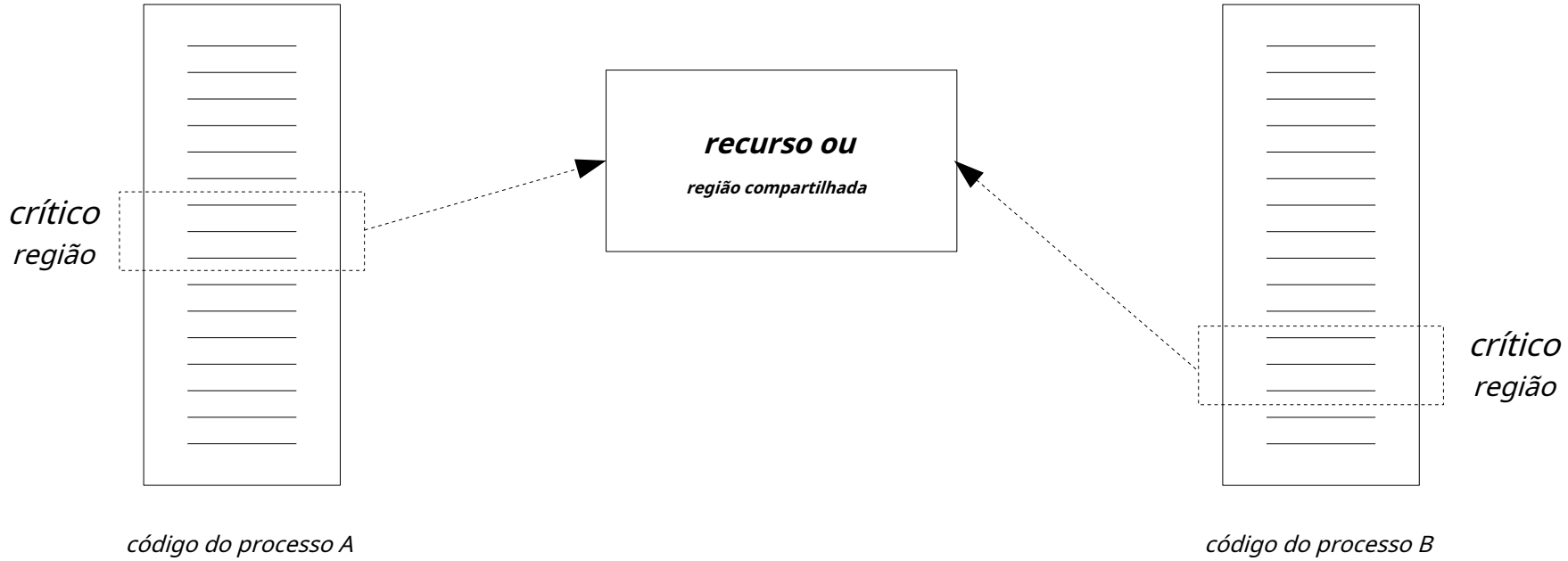


- *processos independentes* que competem pelo acesso a um recurso comum do sistema computacional
- é o *SO* responsabilidade de garantir que a atribuição de recursos seja realizada de forma controlada para que não haja perda de informações
- isso requer, em geral, que apenas um único processo possa ter acesso ao recurso por vez (*exclusão mútua*)

- *processos cooperativos* que compartilham informações ou se comunicam entre si
- é responsabilidade dos processos envolvidos garantir que o acesso à região compartilhada seja realizado de forma controlada para que não haja perda de informações
- isso requer, em geral, que apenas um único processo possa ter acesso por vez ao recurso (*exclusão mútua*)
- o canal de comunicação é tipicamente um recurso do sistema computacional; portanto, o acesso a ele deve ser visto como *concorrência* para acesso a um recurso comum

Princípios gerais de simultaneidade - 3

Tornando a linguagem precisa, sempre que se fala de *acesso por um processo a um recurso ou a uma região compartilhada*, estamos na realidade falando sobre o processador que executa o código de acesso correspondente. Este código, porque deve ser executado de forma a evitar a ocorrência de *condições de corrida*, que inevitavelmente leva à perda de informações, geralmente é chamado *região crítica*.



Princípios gerais de simultaneidade - 4

A imposição da exclusão mútua no acesso a um recurso, ou a uma região partilhada, pode ter, pelo seu carácter restritivo, duas consequências indesejáveis

- *impasse / livelock*–isso acontece quando dois ou mais processos estão esperando para sempre (bloqueados/em*ocupado esperando*) pelo acesso às respetivas regiões críticas, sendo dificultado por acontecimentos que, poder-se-á comprovar, nunca ocorrerão; como resultado, as operações não podem prosseguir
- *adiamento indefinido*–acontece quando um ou mais processos competem pelo acesso a uma região crítica e, devido a uma conjunção de circunstâncias onde novos processos surgem continuamente e competem com os primeiros por esse objetivo, o acesso é sucessivamente negado; estamos aqui, portanto, enfrentando um obstáculo real à sua continuação.

Pretende-se, ao projetar uma aplicação multithread, evitar que essas consequências patológicas ocorram e produzir código que tenha um*vivacidade*propriedade.

Problema de acesso a uma região crítica com exclusão mútua

Propriedades desejáveis que uma solução geral para o problema deve assumir

- *garantia eficaz da imposição de exclusão mútua*–o acesso à região crítica associada a um determinado recurso, ou região compartilhada, só pode ser permitido a um único processo por vez, entre todos que estão competindo pelo acesso simultaneamente
- *independência da velocidade relativa de execução dos processos intervenientes, ou do seu número*–nada deve ser presumido sobre esses fatores
- *um processo fora da região crítica não pode impedir a entrada de outro*
- *a possibilidade de acesso à região crítica de qualquer processo que desejar não pode ser adiada indefinidamente*
- *o tempo que um processo permanece dentro de uma região crítica é necessariamente finito*.

Recursos

De um modo geral, um *recurso* é algo que um processo precisa acessar. Os recursos podem ser *componentes físicos do sistema computacional* (processadores, regiões da memória principal ou de massa, dispositivos específicos de entrada/saída, etc.), ou *estruturas de dados comuns* definido no nível do sistema operacional (tabela de controle de processos, canais de comunicação, etc) ou entre processos de uma aplicação.

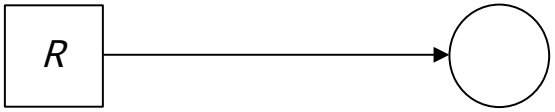
Uma propriedade essencial dos recursos é o tipo de processos de apropriação que eles fazem. Nesse sentido, os recursos são divididos em

- *recursos preemptivos* – quando puderem ser retirados dos processos que os sustentam, sem qualquer mau funcionamento decorrente do fato; o processador e regiões da memória principal onde é armazenado um espaço de endereçamento de processo, são exemplos desta classe em ambientes multiprogramados
- *recursos não preemptivos* – quando não for possível; a impressora ou uma estrutura de dados compartilhada, exigindo exclusão mútua para sua manipulação, são exemplos desta classe.

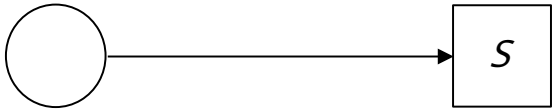
Caracterização esquemática de impasse

Em um *impasse* situação, apenas *não preemptivo* recursos são relevantes. Os restantes podem sempre ser retirados, se necessário, dos processos que os sustentam e atribuídos a outros para garantir que estes progridam.

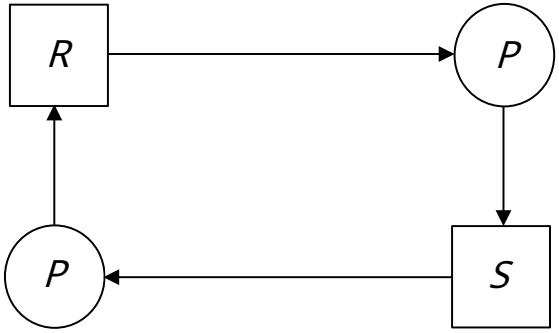
Assim, usando este tipo de estrutura, pode-se desenvolver uma notação esquemática que representa um *impasse* situação graficamente.



o processo P contém o recurso R



o processo P requer o recurso S



situação típica de impasse
(*o mais simples*)

Condições necessárias para a ocorrência de impasse

Pode-se mostrar que, sempre que *impasse* ocorre, há quatro condições que estão necessariamente presentes. Eles são o

- *condição de exclusão mútua* – cada recurso existente é gratuito ou foi atribuído a um único processo (sua posse não pode ser compartilhada)
- *condição de espera com retenção* – cada processo, ao solicitar um novo recurso, contém todos os outros recursos previamente solicitados e atribuídos
- *condição de não libertação* – nada, exceto o próprio processo, pode decidir quando um recurso previamente atribuído é liberado
- *condição de espera circular* (ou *círculo vicioso*) – forma-se uma cadeia circular de processos e recursos, onde cada processo solicita um recurso que está sendo mantido pelo próximo processo da cadeia.

Prevenção de deadlock - 1

As condições necessárias para a ocorrência de *impasse* levar à declaração

ocorrência de impasse ⇒ *exclusão mútua no acesso a um recurso* **e**
esperando com retenção **e** *sem*
liberação de recursos **e** *espera*
circular

o que equivale a

nenhuma exclusão mútua no acesso a um recurso **ou**
sem espera com retenção **ou**
liberação de recursos **ou**
sem espera circular ⇒ *nenhuma ocorrência de deadlock.*

Assim, para criar um impasse *impossível de acontecer*, basta negar uma das condições necessárias para a ocorrência de impasse. As políticas que seguem esta estratégia são chamadas *políticas de prevenção de impasses*.

Prevenção de deadlock - 2

O primeiro, *exclusão mútua no acesso a um recurso*, é muito restritivo porque só pode ser negado para recursos não preemptivos. De outra forma, *condições de corrida* são introduzidas informações que levam, ou podem levar, à inconsistência de informações.

A leitura do acesso de vários processos a um arquivo é um exemplo típico de negação dessa condição. Ressalta-se que, neste caso, também é comum permitir ao mesmo tempo um único acesso de escrita. Quando isso acontece, porém, *condições de corrida*, com a conseqüente perda de informações, não podem ser totalmente descartados. *Por que*?

Portanto, apenas as três últimas condições costumam ser objeto de negação.

Negação da condição de espera com retenção

Significa que um *processo deve solicitar de uma só vez todos os recursos necessários para a continuação*. Se conseguir contatá-los, a conclusão da atividade associada está garantida. Caso contrário, deverá esperar.

Deve-se notar que *adiamento indefinido* não está impedido. O procedimento deve também garantir que, mais cedo ou mais tarde, os recursos necessários serão sempre atribuídos a qualquer processo que os solicite. A introdução de *envelhecimento* políticas para aumentar a prioridade de um processo é um método muito popular usado nesta situação.

Impondo a condição de liberação de recursos

Significa que um *processo, quando não consegue obter todos os recursos necessários para a continuação, deve liberar todos os recursos em sua posse e iniciar posteriormente todo o procedimento de solicitação desde o início*. Alternativamente, também significa que um *o processo só pode conter um recurso por vez* (esta, no entanto, é uma solução específica e não é aplicável na maioria dos casos).

Deve-se tomar cuidado para que o processo não entre em *ocupado esperando* procedimento de solicitação/adquirência de recursos. Em princípio, o processo deve bloquear após liberar os recursos que contém e ser ativado somente quando os recursos que estava solicitando forem liberados.

No entanto, *adiamento indefinido* não está impedido. O procedimento deve também garantir que, mais cedo ou mais tarde, os recursos necessários serão sempre atribuídos a qualquer processo que os solicite. A introdução de *envelhecimento* políticas para aumentar a prioridade de um processo é um método muito popular usado nesta situação.

Negando a condição de espera circular

Isso significa *estabelecer uma ordenação linear dos recursos* e *fazer com que o processo, ao tentar obter os recursos de que necessita para a continuação, os solicite por ordem crescente do número associado a cada um deles*.

Desta forma, evita-se a possibilidade de formação de uma cadeia circular de processos detentores de recursos e solicitantes de outros.

Deve-se notar que *adiamento indefinido* não está impedido. O procedimento deve também garantir que, mais cedo ou mais tarde, os recursos necessários serão sempre atribuídos a qualquer processo que os solicite. A introdução de *envelhecimento* políticas para aumentar a prioridade de um processo é um método muito popular usado nesta situação.

Monitores - 1

A *monitor* é um dispositivo de sincronização, proposto independentemente por Hoare e Brinch Hansen, que pode ser pensado como um módulo especial definido dentro da linguagem de programação [concorrente] e consistindo em uma estrutura de dados interna, código de inicialização e um conjunto de primitivas de acesso.

```
monitor exemplo
   (* estrutura de dados interna
        acessível apenas de fora através das primitivas de acesso *)
   var
      val: DADOS;                                   (* região compartilhada *)
      c: doença;                    (*variável de condição para sincronização*)
   (* acessar primitivas *)
   procedimento pa1 (...);
   fim (*pa1 *)
   função pa2 (...):real; fim (*pa2 *)

   (*inicialização*)
   começar
      . . .
   fim
monitor final;
```

Monitores - 2

Uma aplicação escrita em uma linguagem concorrente, implementando o *paradigma de variáveis compartilhadas*, é visto como um conjunto de *tópicos* que competem pelo acesso a estruturas de dados compartilhadas. Quando as estruturas de dados são implementadas como *monitores*, a linguagem de programação garante que a execução de um *monitor* primitivo é realizado seguindo uma disciplina de exclusão mútua. Assim, o compilador, ao processar um *monitor*, gera o código necessário para impor esta condição de forma totalmente transparente ao programador da aplicação.

A *fio* entra em um *monitor* chamando uma de suas primitivas, que constitui a única forma de acessar a estrutura de dados interna. Como a execução primitiva implica exclusão mútua, quando outro *fio* está atualmente dentro do monitor, o *fio* está bloqueado na entrada, aguardando sua vez.

Monitores - 3

Sincronização entre *tópicos* o uso de monitores é gerenciado por *variáveis de condição*. *Variáveis de condição* são dispositivos especiais, definidos dentro de um monitor, onde uma thread pode ser bloqueada, enquanto espera por um evento que permita a sua continuação. Existem duas operações atômicas que podem ser executadas em um *variável de condição*

espere – o Chamado *fio* está bloqueado no *variável de condição* passado como argumento e é colocado *fora do monitor* para permitir outro *fio*, querendo entrar, prosseguir

sinal – se houver bloqueado *tópicos* no *variável de condição* passado como argumento, um deles é acordado; caso contrário, nada acontece.

Monitores - 4

Para evitar a coexistência de múltiplos *tópicos* dentro de um *monitor*, é necessária uma regra que estabeleça como a disputa suscitada por um *sinal* a execução está resolvida

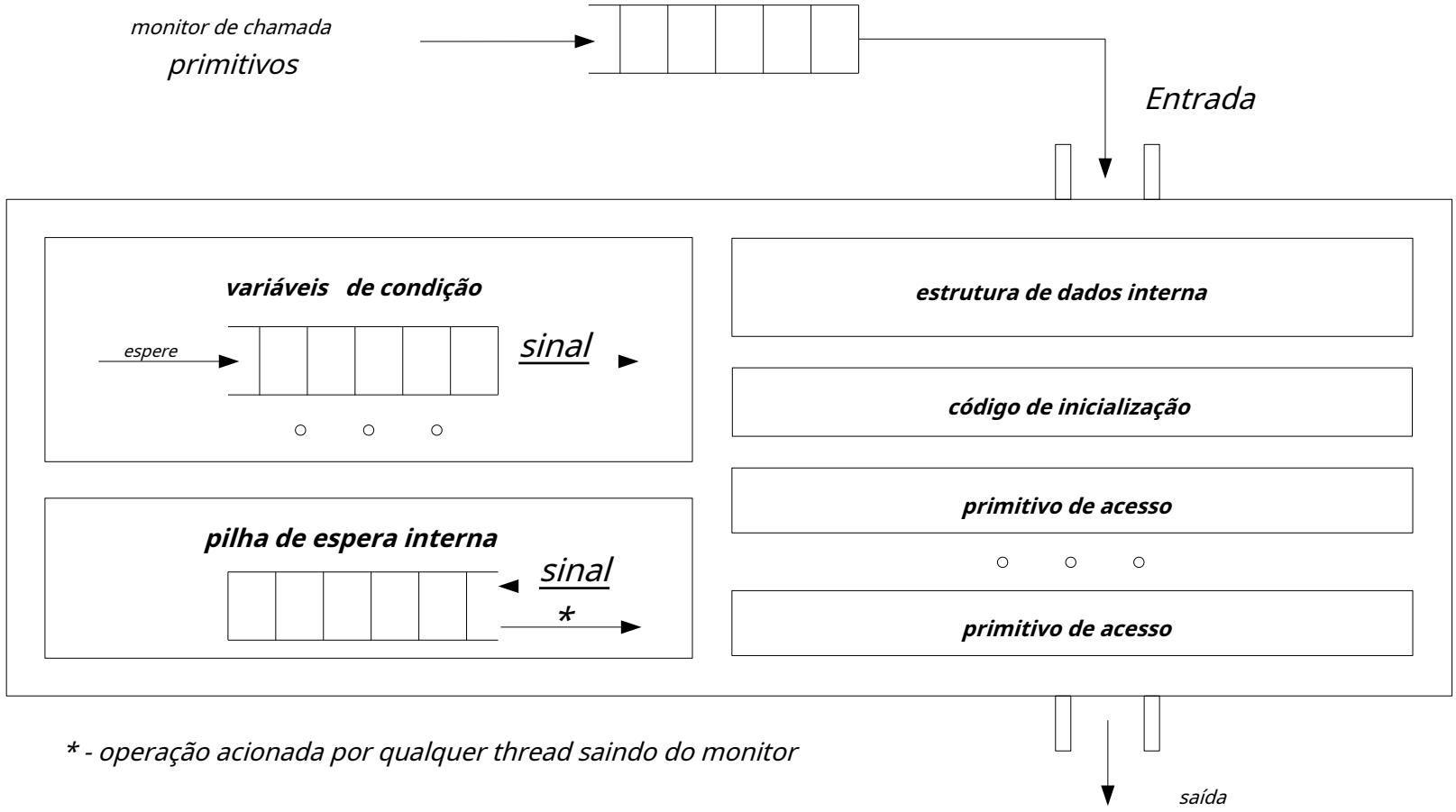
Monitor Hoare – o *fio* que chama *sinal* está localizado *fora do monitor* então que o acorde *fio* pode prosseguir; é uma solução muito geral, mas a sua implementação requer uma *pilha*, onde o *sinal* ligando *tópicos* são armazenados

Monitor Brinch Hansen – o *fio* que chama *sinal* deve sair imediatamente do *monitor* (*sinal* deve ser a última instrução executada em qualquer primitiva de acesso, exceto por uma possível *retornar*); é bastante simples de implementar, mas pode tornar-se bastante restritivo porque reduz o número de *sinal* chama para um

Monitor Lampson/Redell – o *fio* que chama *sinal* prossegue, o despertar *fio* é mantido *fora do monitor* e deve competir pelo acesso a ele novamente; ainda é simples de implementar, mas pode dar origem a *adiamento indefinido* de alguns dos envolvidos *tópicos*.

Monitores - 5

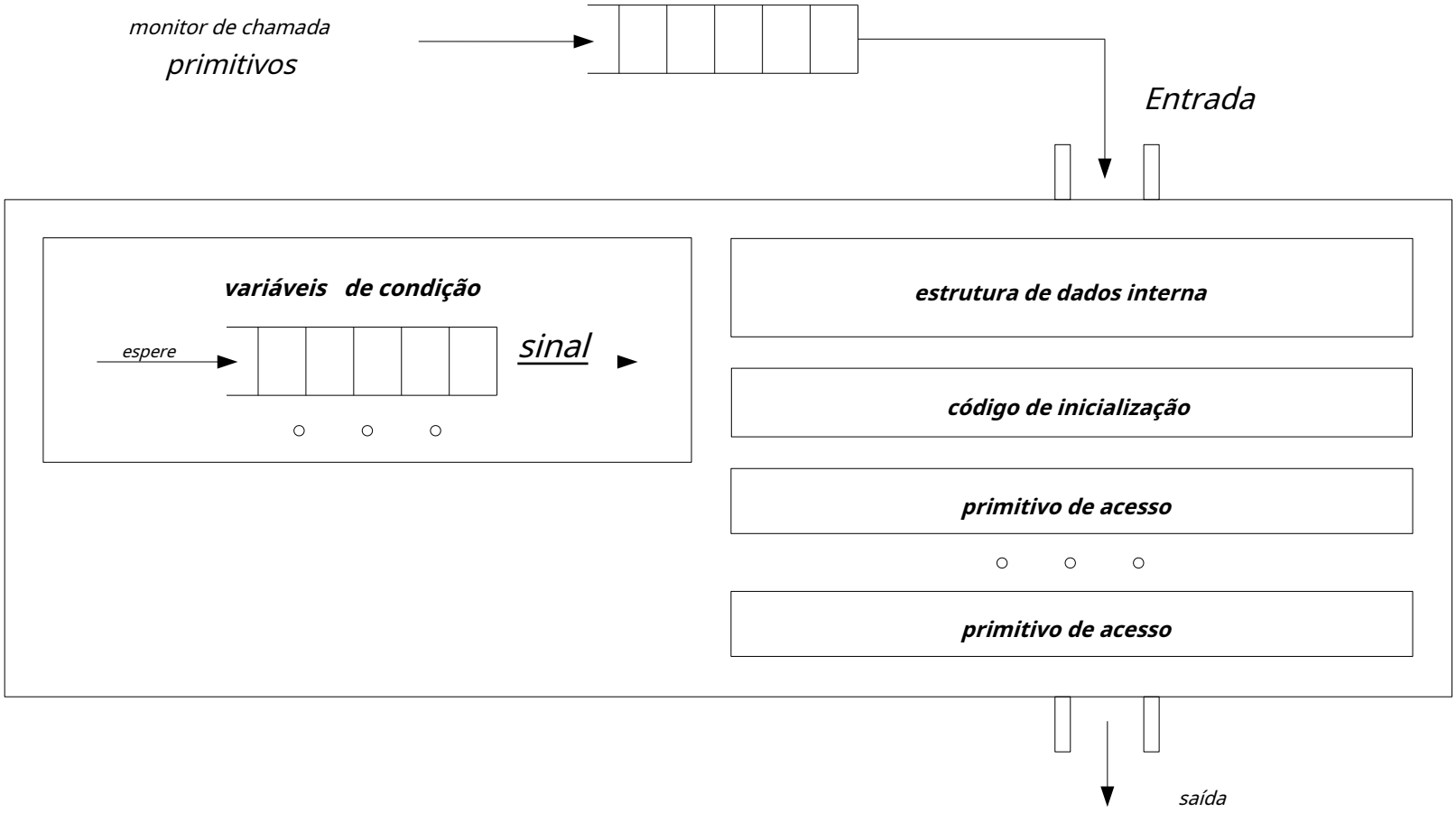
Monitor Hoare



** - operação acionada por qualquer thread saindo do monitor*

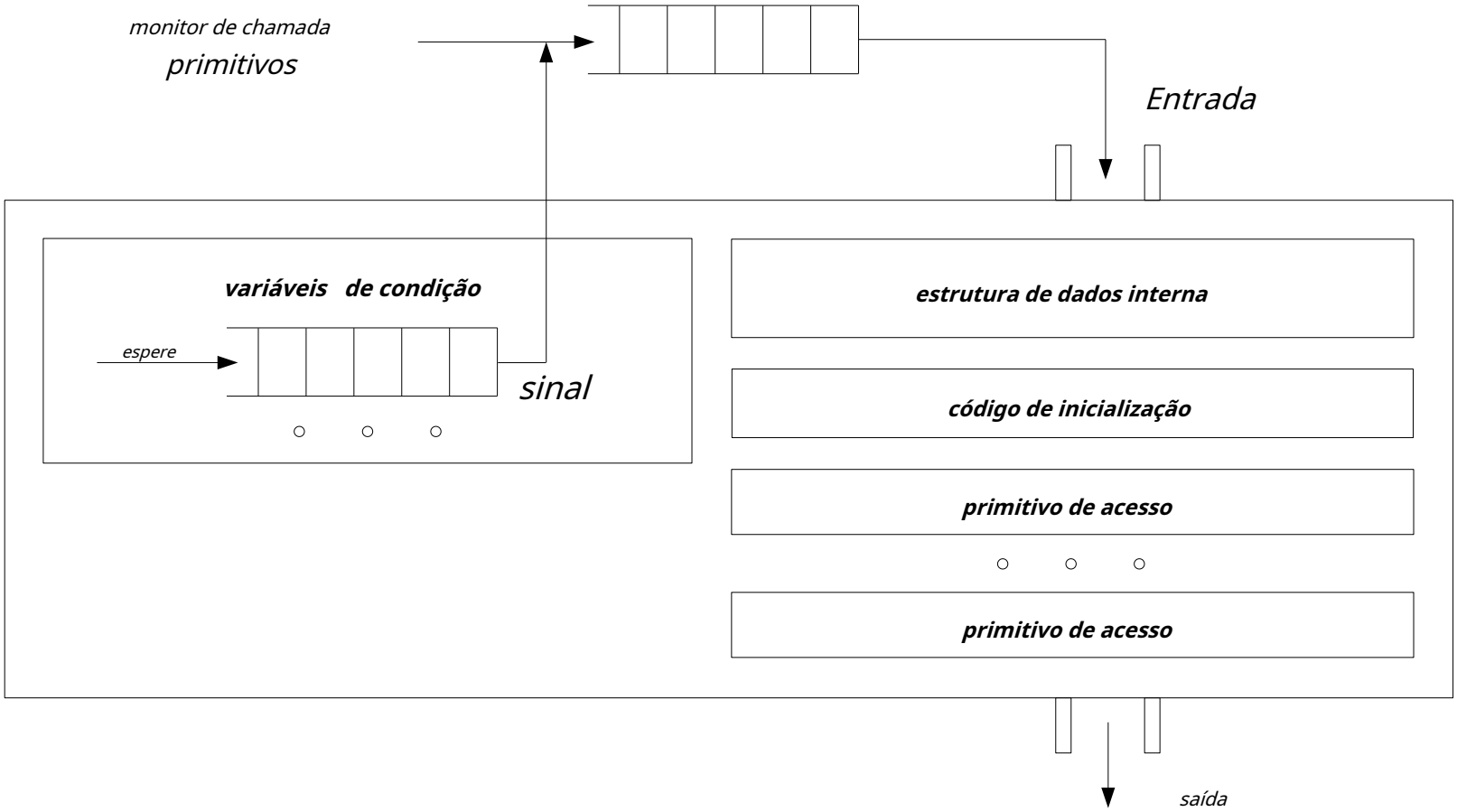
Monitores - 6

Monitor Brinch Hansen



Monitores - 7

Monitor Lampson/Redell



Monitores em Java - 1

Java suporta monitores Lampson/Redell como seu dispositivo de sincronização nativo

- cada tipo de dados de referência pode ser transformado em um monitor, permitindo assim garantir a exclusão mútua e *fio* sincronização quando estático métodos são chamados para isso
- cada objeto instanciado pode ser transformado em um monitor, permitindo assim garantir a exclusão mútua e *fio* sincronização quando métodos de instanciação são chamados.

Na verdade, e tendo em conta que Java é uma linguagem orientada a objectos, cada *fio*, sendo um objeto Java, também pode ser transformado em um monitor. Esta propriedade, se levada às últimas consequências, permite uma *fio* bloquear em seu próprio monitor!

Isto, no entanto, deveria nunca ser feito, pois introduz mecanismos de autorreferência que costumam ser muito difíceis de compreender pelos efeitos colaterais que geram.

Monitores em Java - 2

A implementação em Java de um monitor Lampson/Redell possui, no entanto, algumas peculiaridades

- o número de variáveis de condição é limitado a uma, referenciada de forma implícita através do objeto que representa em *tempo de execução* o tipo de dados de referência ou qualquer uma de suas instanciações
- o tradicional *sinal* operação é nomeada *notificar* e há uma variante desta operação, *notificar todos*, o mais utilizado, que permite o despertar de *todos* o *tópicos* atualmente bloqueado na variável de condição
- além disso, existe um método no tipo de dados d, nomeado *interromper*, qual quando chamado para um determinado *fio*, pretende acordá-lo jogando um *exceção* se estiver bloqueado em uma variável de condição.

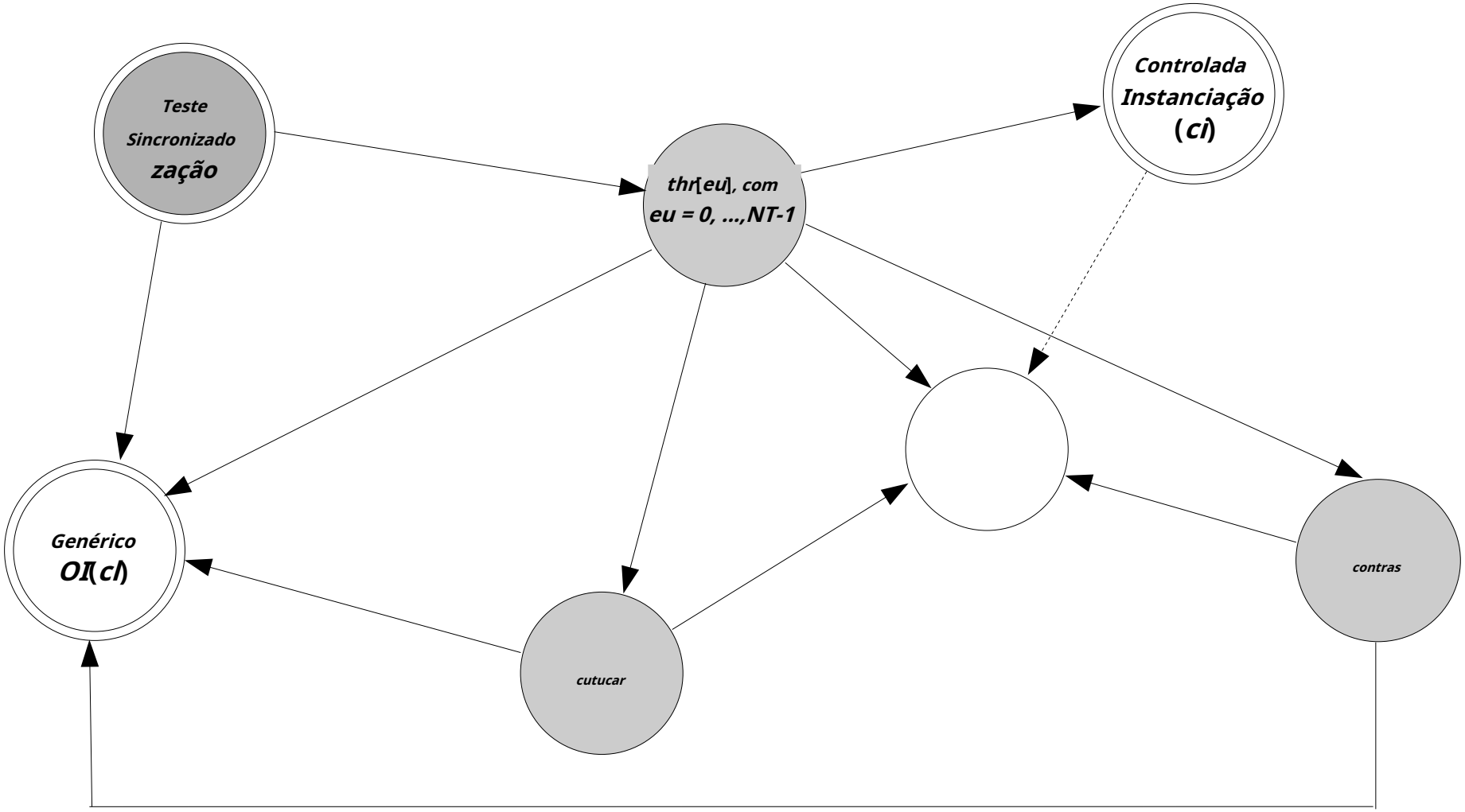
A necessidade da operação *notificar todos* fica evidente se pensarmos que, por ter uma única variável de condição por monitor, a única maneira de despertar um *fio* que está atualmente bloqueado aguardando que uma condição específica seja cumprida, é acordar *todos* o bloqueado *tópicos* e então determinar de forma diferencial qual é aquele que pode progredir.

Instanciação controlada - 1

O exemplo apresentado a seguir ilustra o uso de monitores em diferentes situações

- *fio* principal cria quatro *tópicos* do tipo de dados de referência TestThread que tentam instanciar objetos do tipo de dados ControlledInstanciação, cada um um local de armazenamento para transferência de valor entre *tópicos* de tipos de dados Processo e Contras
- o tipo de dados de referência não, no entanto, apenas permite a instanciação simultânea de no máximo dois objetos
- existem, portanto, três tipos de monitores em jogo
  - o monitor associado ao objeto que representa em *tempo de execução* o tipo de dados de referência ControlledInstanciação
  - o monitor associado a cada instanciação do tipo de dados ControlledInstanciação
  - o monitor associado ao objeto que representa em *tempo de execução* o tipo de dados de referência genclass.GenericIO
- o primeiro controla a instanciação de ControlledInstanciação objetos, o segundo é a transferência de valor entre threads do tipo de dados Prode Contras e a terceira a impressão de dados no *padrão* dispositivo de saída.

Instanciação controlada - 2



Instanciação controlada - 3

Operaçãoespereem um monitor definido no nível do tipo de dados de referência

```
público estático sincronizadoControlledInstantiation generateInst() {

    enquanto(n >= NMAX) {
    tentar
        { (Class.forName ("ControlledInstantiation")).wait (); }

        pegar(ClassNotFoundException e)
        { GenericIO.writelnString ("O tipo de dados ControlledInstantiation não era" +
                                                    "encontrado(geração)!");
            e.printStackTrace();
            Sistema.exit (1);
        }
        pegar(Exceção Interrompida e)
        { GenericIO.writelnString ("O método estático generateInst foi interrompido!"); }

    }
    n+= 1;
    nInst += 1;
    devolver novoControlledInstanciação();
}
```

Instanciação controlada - 4

Operaçãonotificarem um monitor definido no nível do tipo de dados de referência

```
público estático sincronizado vaziosreleaseInst() {

    n-= 1;
    tentar
    { (Class.forName ("ControlledInstantiation")).notify(); }

    pegar(ClassNotFoundException e)
    { GenericIO.writelnString ("O tipo de dados ControlledInstantiation não foi encontrado" +
                                                    "(liberar)!");
        e.printStackTrace();
        Sistema.exit (1);
    }
}
```

Instanciação controlada - 5

Operaçãoespereenotificarem um monitor definido no nível do objeto

```
vazio sincronizado públicocolocarVal (internovalor) {

    armazenar = val;
    enquanto(armazenar! =
    -1) { notificar ();
        tentar
        { espere ();
        }
        pegar(Exceção Interrompida e)
        { GenericIO.writelnString ("O método putVal foi interrompido!"); }

    }
}
```

Instanciação controlada - 6

Acesso com exclusão mútua a uma biblioteca não segura para threads

```
Classe<?> cl = nulo;                    // representação do tipo de dados da biblioteca na JVM

tentar
{ cl = Class.forName ("genclass.GenericIO"); }

pegar(ClassNotFoundException e)
{ System.out.println ("O tipo de dados genclass.GenericIO não foi encontrado!");
    e.printStackTrace();
    Sistema.exit (1);
}

sincronizado(cl)
{ GenericIO.writelnString ("Já criei os threads!"); }
```

Instanciação controlada - 7

Já criei os tópicos!
Eu, Thread_base_2, obtive o número de instanciação 2 do tipo de dados ControlledInstantiation!
Eu, Thread_base_1, obtive o número de instanciação 1 do tipo de dados ControlledInstantiation!
Eu, Thread_base_1, vou criar as threads que vão trocar o valor!
Eu, Thread_base_2, vou criar as threads que vão trocar o valor!
Eu, Thread_base_1_writer, vou escrever o valor 1 na instanciação número 1 do tipo de dados ControlledInstantiation!
Eu, Thread_base_2_writer, estou indo escrever o valor 2 na instanciação número 2 do tipo de dados ControlledInstantiation!
Eu, Thread_base_1_reader, li o valor 1 na instanciação número 1 do tipo de dados ControlledInstantiation!

Meu thread que grava o valor, Thread_base_1_writer, foi encerrado. Meu thread que lê o valor, Thread_base_1_reader, foi encerrado. Eu, Thread_base_1, vou liberar a instanciação número 1 do tipo de dados ControlledInstantiation!

Eu, Thread_base_0, obtive o número de instanciação 3 do tipo de dados ControlledInstantiation!
Eu, Thread_base_0, vou criar as threads que vão trocar o valor!
Eu, Thread_base_2_reader, li o valor 2 na instanciação número 2 do tipo de dados ControlledInstantiation!

Meu thread que grava o valor, Thread_base_2_writer, foi encerrado. Meu thread que lê o valor, Thread_base_2_reader, foi encerrado. Eu, Thread_base_2, vou liberar a instanciação número 2 do tipo de dados ControlledInstantiation!

Instanciação controlada - 8

Eu, Thread_base_3, obtive o número de instanciação 4 do tipo de dados ControlledInstantiation!
Eu, Thread_base_3, vou criar as threads que vão trocar o valor!
Eu, Thread_base_3_writer, vou escrever o valor 3 na instanciação número 4 do tipo de dados ControlledInstantiation!
Eu, Thread_base_0_writer, vou escrever o valor 0 na instanciação número 3 do tipo de dados ControlledInstantiation!
Eu, Thread_base_0_reader, li o valor 0 na instanciação número 3 do tipo de dados ControlledInstantiation!

Meu tópico que escreve O valor que, ase_0_wri ter, terminou.
Meu thread que lê o valor, Thread_base_0_reader, foi encerrado. Eu, Thread_base_0, vou liberar a instanciação número 3 do tipo de dados ControlledInstantiation!

Tópico_b

O thread Thread_base_0 foi encerrado. O thread Thread_base_1 foi encerrado. O thread Thread_base_2 foi encerrado.
Eu, Thread_base_3_reader, li o valor 3 na instanciação número 4 do tipo de dados ControlledInstantiation!

Meu thread que grava o valor, Thread_base_3_writer, foi encerrado. Meu thread que lê o valor, Thread_base_3_reader, foi encerrado. Eu, Thread_base_3, vou liberar a instanciação número 4 do tipo de dados ControlledInstantiation!

O thread Thread_base_3 foi encerrado.

Semáforos - 1

Uma abordagem diferente para resolver o problema do acesso com exclusão mútua a uma região crítica baseia-se na observação.

```
/* estrutura de dados de controle */
# definir R. . .                    /* número de processos que desejam acessar uma região crítica,
                                        pid = 0, 1, ..., R-1 */

compartilhado não assinado int acesso = 1;

/* primitiva para entrar na região crítica */ vazio enter_RC (int
não assinado próprio_pid)
{
    se(acesso == 0) dormir (own_pid);
        outro acesso -= 1;
}
```

operação atômica
(sua execução não pode ser interrompida)

```
/* primitiva para sair da região crítica */ vazio saída_RC()

{
    se(existem processos bloqueados) wake_up_one ();
        outro acesso += 1;
}
```

operação atômica
(sua execução não pode ser interrompida)

Semáforos - 2

A *semáforo* é um dispositivo de sincronização, originalmente inventado por Dijkstra, que pode ser pensado como uma variável do tipo

```
estrutura typedef
{int não assinado val;        /* valor de contagem não negativo */ /* fila de
   NÓ *fila;                  espera de processos bloqueados */
} SEMÁFORO;
```

onde duas operações atômicas podem ser realizadas

abaixo – se valor é diferente de zero (o semáforo tem tonalidade verde), seu o valor é diminuído em uma unidade; caso contrário, o processo que chamou a operação é bloqueado e seu id é colocado em fila

acima – se houver processos bloqueados em fila, um deles está acordado (de acordo a qualquer disciplina previamente prescrita); caso contrário, o valor de valor é aumentado em uma unidade.

Semáforos - 3

```
/* array de semáforo definido pelo kernel */
# definir R. . .                    /* id do semáforo - id = 0, 1, ..., R-1 */

estático SEMÁFORO sem[R];

/* operação para baixo */ vazio abaixo
(int não assinado eu ia) {

    desativar interrupções/bloquear sinalizador de acesso;
    se(sem[id].val == 0) sleep_on_sem (getpid(), id);
        outro sem[id].val -= 1;
    ativar interrupções/desbloquear sinalizador de acesso;
}

/* operação ascendente */ vazio acima
(int não assinado eu ia) {

    desativar interrupções/bloquear sinalizador de acesso;
    se(existem processos bloqueados em sem[id]) wake_up_one_on_sem (id);
        outro sem[id].val += 1;
    ativar interrupções/desbloquear sinalizador de acesso;
}
```

Semáforo Java Dijkstra

```
aula pública Semáforo
{
    privado interno val = 0,                    //indicador verde/vermelho
                    numbBlockThreads = 0;       // número de threads bloqueadas // no
                                                monitor

    vazio sincronizado público abaixo () {

        se(val == 0)
            {numbBlockThreads += 1;
                tentar
                { espere ();
                }
                pegar(InterruptedException e) {}
            }
            outro val -= 1;
    }

    vazio sincronizado público acima () {

        se(numbBlockThreads! = 0)
            { numbBlockThreads -= 1;
                notificar();
            }
            outro val += 1;
    }
}
```

Biblioteca de simultaneidade Java - 1

Javasimultaneidadebiblioteca fornece os seguintes dispositivos de sincronização relevantes

- *barreiras*–dispositivos que levam ao bloqueio de conjunto de*tópicos*até que uma condição para sua continuação seja cumprida; quando a barreira é levantada, todos os processos bloqueados são despertados
- *semáforos*–dispositivos que proporcionam uma implementação mais geral do conceito de semáforo do que a prescrita por Dijkstra, nomeadamente permitindo a chamada de*abaixo*e*acima*operações onde o campo internovaloré diminuído ou aumentado em mais de uma unidade por vez e tendo uma variante não bloqueadora de*abaixo*Operação
- *permutador*–dispositivos que permitem uma troca de valores entre*tópicos*pares através de um*encontro*tipo de sincronização.

Biblioteca de simultaneidade Java - 2

- *fechaduras*–Monitores Lampson-Redell de caráter geral (em contraste com Java *construídas em*monitor, é possível definir aqui múltiplas variáveis de condição e obter uma funcionalidade semelhante àquela fornecida pelo*thread*biblioteca
- *manipulação de variáveis atômicas*–*ler-modificar-escreve*mecanismos que permitem escrever e ler o conteúdo de diversos tipos de variáveis sem o risco de*condições de corrida*.

A biblioteca também fornece primitivas básicas de bloqueio de threads para criar bloqueios e outros dispositivos de sincronização.

Leitura sugerida

- *Sistemas Distribuídos: Conceitos e Design, 4ª Edição, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Capítulo 6:*Suporte a sistemas operacionais*
- *Sistemas Distribuídos: Princípios e Paradigmas, 2ª Edição,*Tanenbaum, van Steen, Pearson Education Inc.
  - Capítulo 3:*Processos*
- *On-line*documentação de suporte para ambiente de desenvolvimento de programas Java da Oracle (Java Platform Standard Edition 8)