Computação Distribuída

Concorrência 2

António Rui Borges

Sumário

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

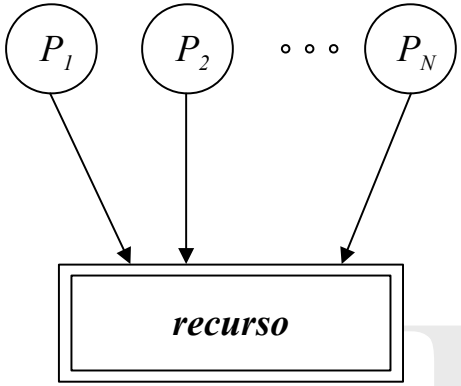
Princípios gerais de concorrência - 1

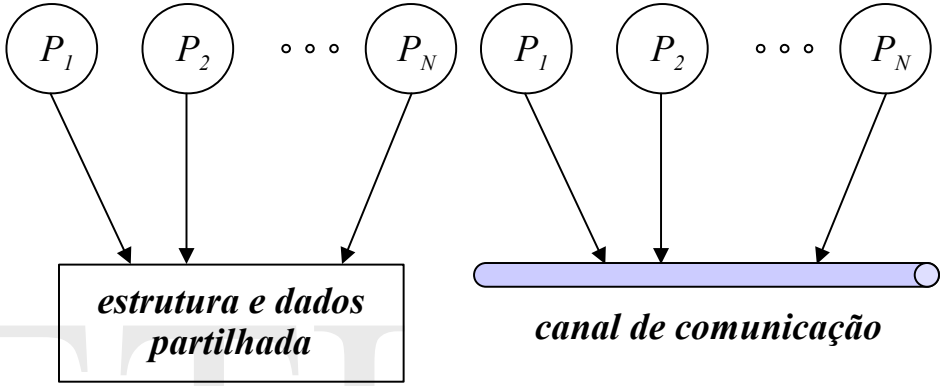
Num ambiente multiprogramado, os processos que coexistem podem apresentar comportamentos diversos em termos de interacção.

Constituem-se como

- *processos independentes* – quando são criados, têm o seu 'tempo de vida' e terminam sem interactuarem de um modo explícito; a interacção que ocorre é implícita e tem origem na sua *competição* pelos recursos do sistema computacional; trata-se tipicamente dos processos lançados pelos diferentes utilizadores num ambiente interactivo e/ou dos processos que resultam do processamento de *jobs* num ambiente de tipo *batch*;
- *processos cooperantes* – quando partilham informação ou comunicam entre si de um modo explícito; a *partilha* pressupõe um espaço de endereçamento comum, enquanto que a *comunicação* pode ser feita tanto através da partilha de um espaço de endereçamento, como através da existência de um canal de comunicação que interliga os processos intervenientes.

Princípios gerais de concorrência - 2





- *processos independentes* que competem por acesso a um recurso comum do sistema computacional;
- é da responsabilidade do *SO* garantir que a atribuição do recurso seja feita de uma forma controlada para que não haja perda de informação;
- isto impõe, em geral, que só um processo de cada vez pode ter acesso ao recurso (*exclusão mútua*).

- *processos cooperantes* que partilham informação ou comunicam entre si;
- é da responsabilidade dos processos envolvidos garantir que o acesso à região partilhada seja feito de uma forma controlada para que não haja perda de informação;
- isto impõe, em geral, que só um processo de cada vez pode ter acesso à região partilhada (*exclusão mútua*);
- o canal de comunicação é tipicamente um recurso do sistema computacional, logo o acesso a ele está enquadrado na *competição* por acesso a um recurso comum.

Princípios gerais de concorrência - 3

Tornando a linguagem precisa, *quando se fala em acesso por parte de um processo a um recurso, ou a uma região partilhada*, está na realidade a referir-se a execução por parte do processador do código de acesso correspondente. Este código, porque tem que ser executado de modo a evitar *condições de corrida* que conduzem à perda de informação, designa-se habitualmente de *região crítica*.

Princípios gerais de concorrência - 4

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

- *deadlock / livelock* – quando dois ou mais processos ficam a aguardar eternamente (bloqueados / em *busy waiting*) o acesso às regiões críticas respectivas, esperando acontecimentos que, se pode demonstrar, nunca irão acontecer; o resultado é, por isso, o bloqueio das operações;
- *adiamento indefinido* – quando um ou mais processos competem pelo acesso a uma região crítica e, devido a uma conjunção de circunstâncias em que surgem continuamente processos novos que competem com eles nesse desígnio, o acesso é-lhes sucessivamente adiado; está-se, por isso, perante um impedimento real à continuação dele(s).

Problema de acesso a uma região crítica com exclusão mútua

Propriedades desejáveis que a solução geral do problema deve assumir

- *garantia efectiva de imposição de exclusão mútua* – o acesso à região crítica associada a um mesmo recurso, ou região partilhada, só pode ser permitido a um processo de cada vez, de entre todos os processos que competem pelo acesso;
- *independência da velocidade de execução relativa dos processos intervenientes, ou do seu número* – nada deve ser presumido acerca destes factores;
- *um processo fora da região crítica não pode impedir outro de lá entrar*;
- *não pode ser adiada indefinidamente a possibilidade de acesso à região crítica a qualquer processo que o requeira*;
- *o tempo de permanência de um processo na região crítica é necessariamente finito*.

Recursos

Genericamente, um *recurso* é algo que um processo precisa para a sua execução. Os recursos tanto podem ser *componentes físicos do sistema computacional* (processadores, regiões de memória principal ou de memória de massa, dispositivos concretos de entrada / saída, etc), como *estruturas de dados comuns* definidas ao nível do sistema de operação (tabela de controlo de processos, canais de comunicação, etc), ou entre processos de uma mesma aplicação.

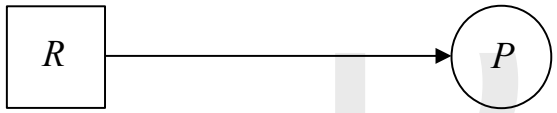
Uma propriedade essencial dos recursos é o tipo de apropriação que os processos fazem deles. Nestes termos, os recursos dividem-se em

- *recursos preemptable* – quando podem ser retirados aos processos que os detêm, sem que daí resulte qualquer consequência irreparável à boa execução dos processos; são, por exemplo, em ambientes multiprogramados, o processador, ou as regiões de memória principal onde o espaço de endereçamento de um processo está alojado;
- *recursos non-preemptable* – em caso contrário; são, por exemplo, a impressora, ou uma estrutura de dados partilhada que exige exclusão mútua para a sua manipulação.

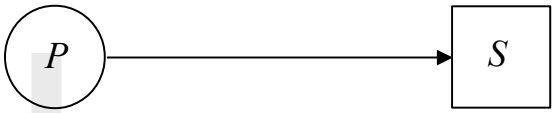
Caracterização esquemática de deadlock

Numa situação de *deadlock*, só os recursos *non-preemptable* são relevantes. Os restantes podem ser sempre retirados, se tal for necessário, ao(s) processo(s) que o(s) detêm e atribuídos a outros para garantir o prosseguimento da execução destes últimos.

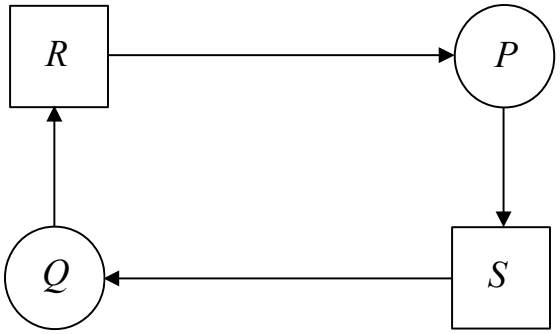
Assim, usando este tipo de classificação, torna-se possível desenvolver uma notação esquemática que representa graficamente situações de *deadlock*.



processo P mantém o recurso R na sua posse



processo P requer o recurso S



situação típica de deadlock
(*a mais simples possível*)

Condições necessárias à ocorrência de deadlock

Pode demonstrar-se que, sempre que ocorre *deadlock*, há quatro condições que ocorrem necessariamente.

São elas

- *condição de exclusão mútua* – cada recurso existente, ou está livre, ou foi atribuído a um e um só processo (a sua posse não pode ser partilhada);
- *condição de espera com retenção* – cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos anteriormente solicitados;
- *condição de não libertação* - ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido previamente atribuído;
- *condição de espera circular* (ou *ciclo vicioso*) - formou-se uma cadeia circular de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia.

Prevenção de deadlock no sentido estrito - 1

As condições necessárias à ocorrência de *deadlock* conduzem à proposição

há deadlock ⇒ *há exclusão mútua no acesso a um recurso* **and**
há espera com retenção **and**
há não libertação de recursos **and**
há espera circular

que é equivalente a

não há exclusão mútua no acesso a um recurso **or**
não há espera com retenção **or**
há libertação de recursos **or**
não há espera circular ⇒ *não há deadlock* .

Assim, desde que uma das condições necessárias à ocorrência de *deadlock* seja negada pelo algoritmo de acesso aos recursos, o *deadlock* torna-se impossível. Políticas com esta característica designam-se de *políticas de prevenção de dead-lock no sentido estrito* (*deadlock prevention*, em inglês).

Prevenção de deadlock no sentido estrito - 2

A primeira delas, *há exclusão mútua no acesso a um recurso*, é bastante restritiva porque só pode ser negada tratando-se de um recurso passível de partilha em simultâneo. Caso contrário, são introduzidas *condições de corrida* que conduzem, ou podem conduzir, a inconsistência de informação.

O acesso para leitura por parte de múltiplos processos a um dado ficheiro é um exemplo típico da negação desta condição. Note-se que, neste caso, é comum permitir também um acesso para escrita por parte de um processo de cada vez. Quando tal acontece, porém, não se pode impedir completamente a existência de *condições de corrida*, com a consequente inconsistência de informação. *Porquê*?

É, por isso, que só as três últimas condições são em geral objecto de negação.

Negando a condição de espera com retenção

Significa que um processo *tem que solicitar de um só vez todos os recursos que vai precisar para a sua continuação*.

Se os obtém, o completamento da acção associada está garantido.

Se não os obtém, terá que aguardar.

Note-se que a ocorrência de *adiamento indefinido* não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.

Impondo a condição de libertação de recursos

Significa que um processo *quando não obtém os recursos que necessita para a sua continuação, tem que libertar todos os recursos na sua posse, tentando mais tarde a sua obtenção a partir do princípio*. Alternativamente, isto significa também impor que um processo *só pode deter um recurso de cada vez* (o que é completamente inviável em muitas situações).

Um cuidado a ter numa solução deste tipo é que o processo não entre em *busy waiting*. O processo deve bloquear e ser mais tarde acordado quando houver libertação de recursos.

Note-se que a ocorrência de *adiamento indefinido* não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.

Negando a condição de espera circular

Significa *estabelecer uma ordenação linear dos recursos* e impor que um processo *quando procura obter os recursos que necessita para a sua continuação, o faça sempre por ordem crescente do número associado a cada um*.

Desta maneira, a possibilidade de formação de uma cadeia circular de processos e recursos está posta de parte.

Note-se que a ocorrência de *adiamento indefinido* não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.

Monitores - 1

Um *monitor* é um dispositivo de sincronização, proposto de uma forma independente por Hoare e Brinch Hansen, que pode ser concebido como um módulo especial, suportado pela linguagem de programação [concorrente] e constituído por uma estrutura de dados interna, por código de inicialização e por um conjunto de primitivas de acesso.

```
monitor exemplo
  (* estrutura de dados interna
     só acessível do exterior através das primitivas de acesso *)
  var
    val: DATA;                          (* região partilhada *)
    c: condition;                (* variável de condição para sincronização *)
  (* primitivas de acesso *)
  procedure pa1 (...);
  end (* pa1 *)
  function pa2 (...): real;
  end (* pa2 *)
  (* inicialização *)
  begin
    ...
  end
end monitor;
```

Monitores - 2

Uma aplicação escrita numa linguagem concorrente que implementa o *paradigma de variáveis partilhadas*, é vista como um conjunto de *threads* que competem pelo acesso a estruturas de dados partilhadas.

Quando as estruturas de dados são implementadas com *monitores*, a linguagem de programação garante que a execução de uma primitiva do *monitor* é feita em regime de exclusão mútua. Assim, o compilador, ao compilar um *monitor*, gera o código necessário para impor esta situação.

Um *thread* entra no *monitor* por invocação de uma das suas primitivas, o que constitui a única forma de acesso à estrutura de dados interna. Como a execução das primitivas decorre em regime de exclusão mútua, quando um outro *thread* está no seu interior, o *thread* é bloqueado à entrada, aguardando a sua vez de acesso.

Monitores - 3

A sincronização entre *threads* é gerida pelas *variáveis de condição*.

Existem duas operações que podem ser executadas sobre uma *variável de condição*

wait – o *thread* que invoca a operação é bloqueado na *variável de condição* argumento e é colocado *fora do monitor* para possibilitar que um outro *thread* que aguarda acesso, possa prosseguir;

signal – se houver *threads* bloqueados na *variável de condição* argumento, um deles é acordado; caso contrário, nada acontece.

Monitores - 4

Para impedir a coexistência de dois *threads* dentro do *monitor*, é necessária uma regra que estipule como a contenção decorrente de um *signal* é resolvida

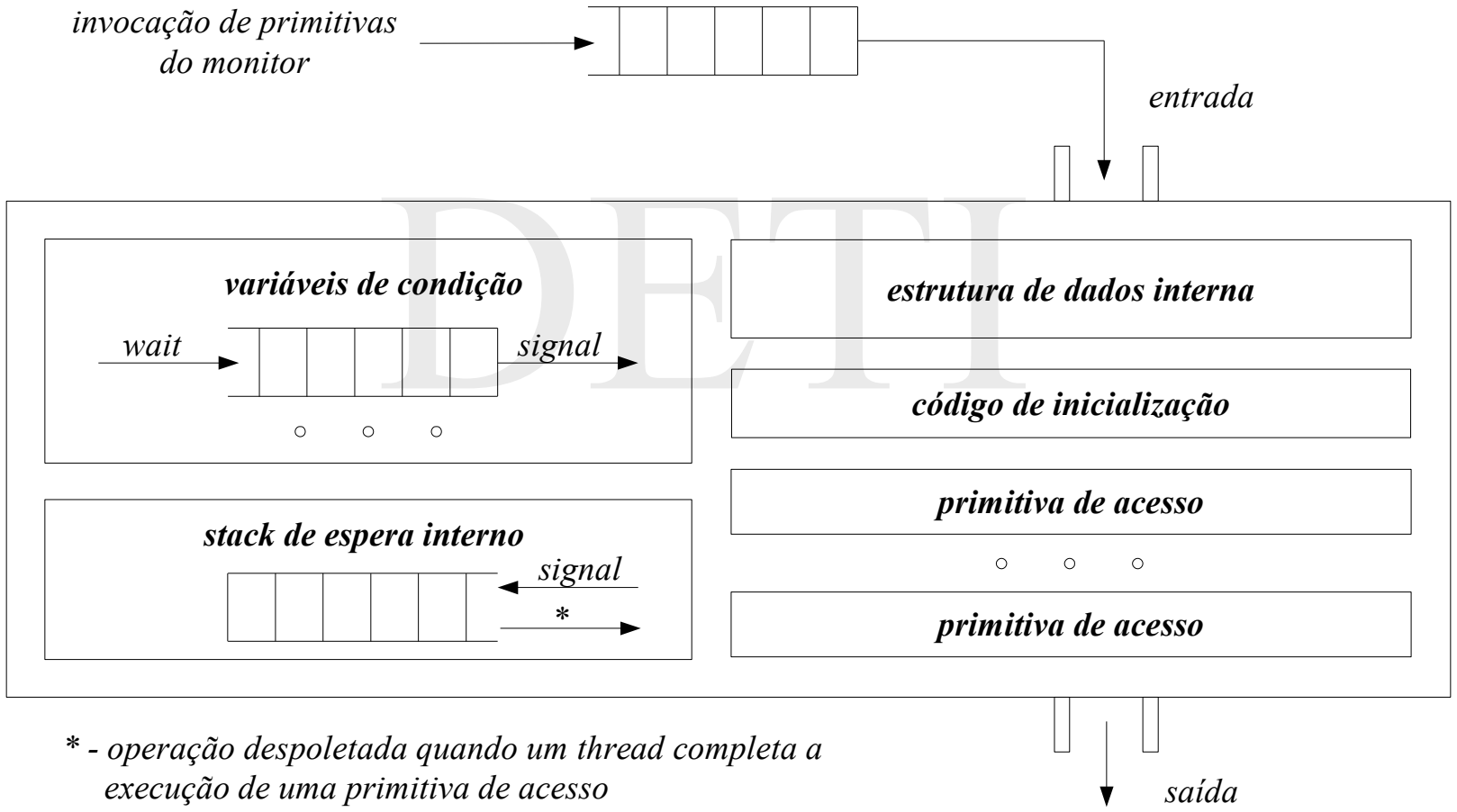
monitor de Hoare – o *thread* que invoca a operação de *signal* é colocado *fora do monitor* para que o *thread* acordado possa prosseguir; é muito geral, mas a sua implementação exige a existência de um *stack*, onde são colocados os *threads* postos *fora do monitor* por invocação de *signal*;

monitor de Brinch Hansen – o *thread* que invoca a operação de *signal* liberta imediatamente o *monitor* (*signal* é a última instrução executada); é simples de implementar, mas pode tornar-se bastante restritivo porque só há possibilidade de execução de um *signal* em cada invocação de uma primitiva de acesso;

monitor de Lampson / Redell – o *thread* que invoca a operação de *signal* prossegue a sua execução, o *thread* acordado mantém-se *fora do monitor* e compete pelo acesso a ele; é simples de implementar, mas pode originar situações em que alguns *threads* são colocados em *adiamento indefinido*.

Monitores - 5

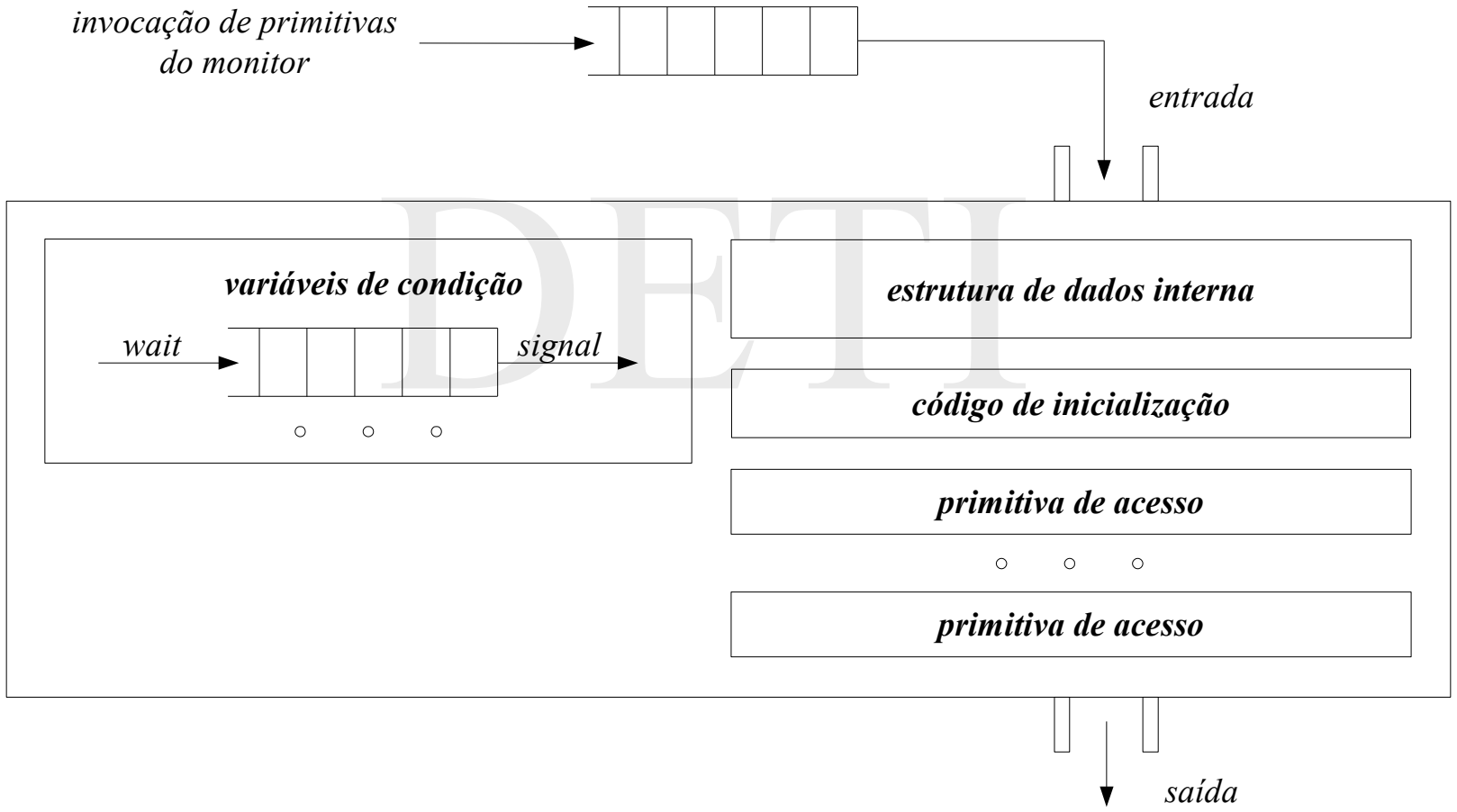
Monitor de Hoare



* - *operação despoletada quando um thread completa a execução de uma primitiva de acesso*

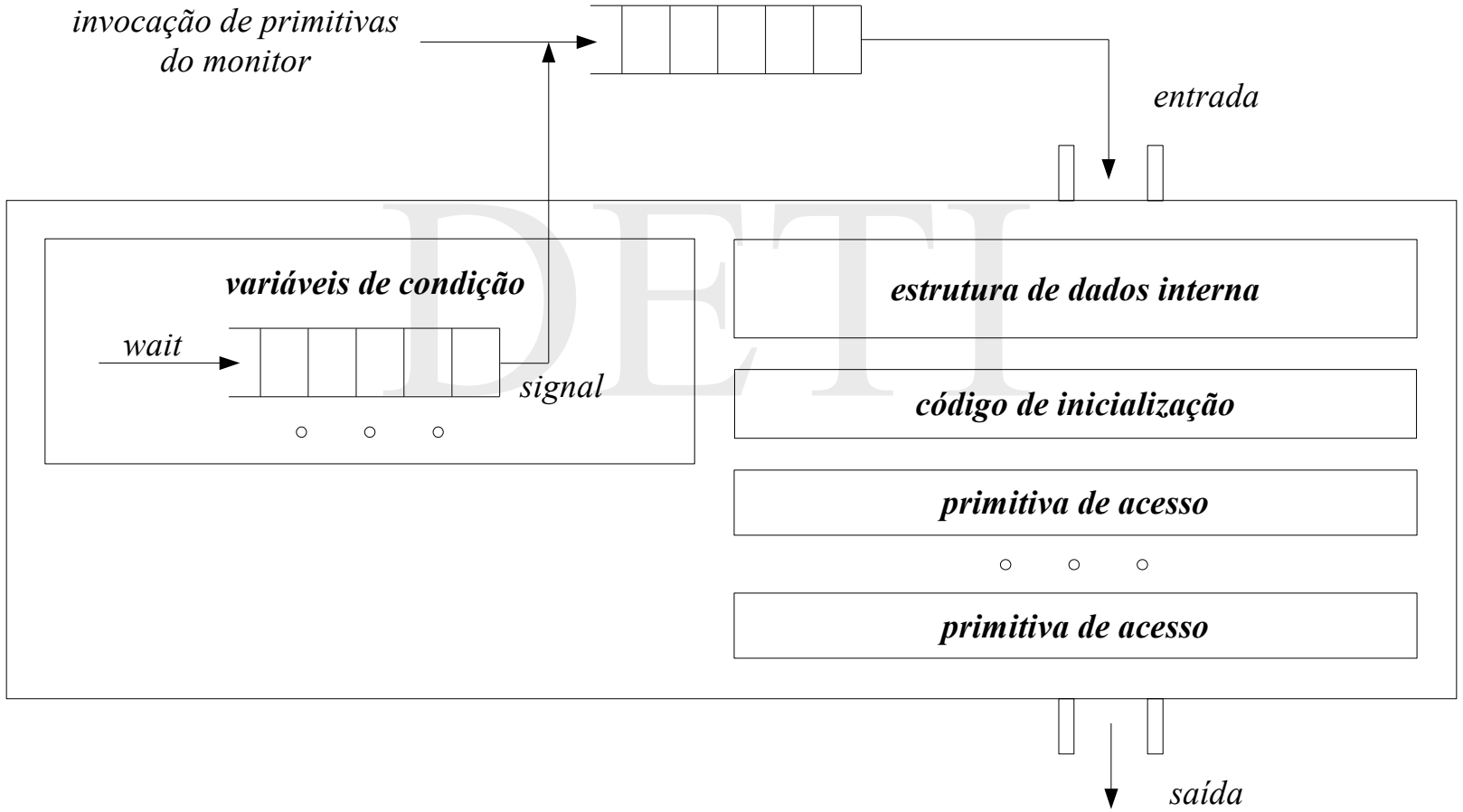
Monitores - 6

Monitor de Brinch Hansen



Monitores - 7

Monitor de Lampson / Redell



Monitores em Java - 1

A linguagem Java suporta monitores de tipo Lampson / Redell como dispositivo nativo de sincronização

- cada tipo de dados de referência tem associado um monitor que permite garantir a exclusão mútua e a sincronização de *threads* que invoquem sobre ele métodos de tipo `static`;
- cada objecto tem associado um monitor que permite garantir a exclusão mútua e a sincronização de *threads* que invoquem sobre ele métodos de instanciação.

Na realidade, e dada a característica orientada por objectos da linguagem, cada *thread*, sendo em última análise um objecto Java, tem também associado um monitor. Esta propriedade, se levada às últimas consequências, vai permitir que um *thread* bloqueie no seu próprio monitor!

Contudo, fazê-lo deve ser evitado já que introduz mecanismos de auto-referência que são normalmente de difícil compreensão nos efeitos colaterais que acarretam.

Monitores em Java - 2

A implementação em Java de um monitor de tipo Lampson / Redell apresenta, contudo, algumas peculiaridades

- o número de variáveis de condição está limitado a uma só, referenciada de uma maneira implícita através do objecto que representa em *runtime* o tipo de dados, ou uma das suas instanciações;
- a operação tradicional *signal* é aqui designada de *notify* e existe uma variante desta operação, *notifyAll*, a mais correntemente utilizada, que permite acordar *todos* os *threads* bloqueados na variável de condição;
- existe ainda uma operação pertencente ao tipo `Thread`, a operação *interrupt*, que, se executada sobre um *thread* particular, visa acordá-lo através do lançamento de uma *excepção*, caso ele esteja bloqueado numa variável de condição.

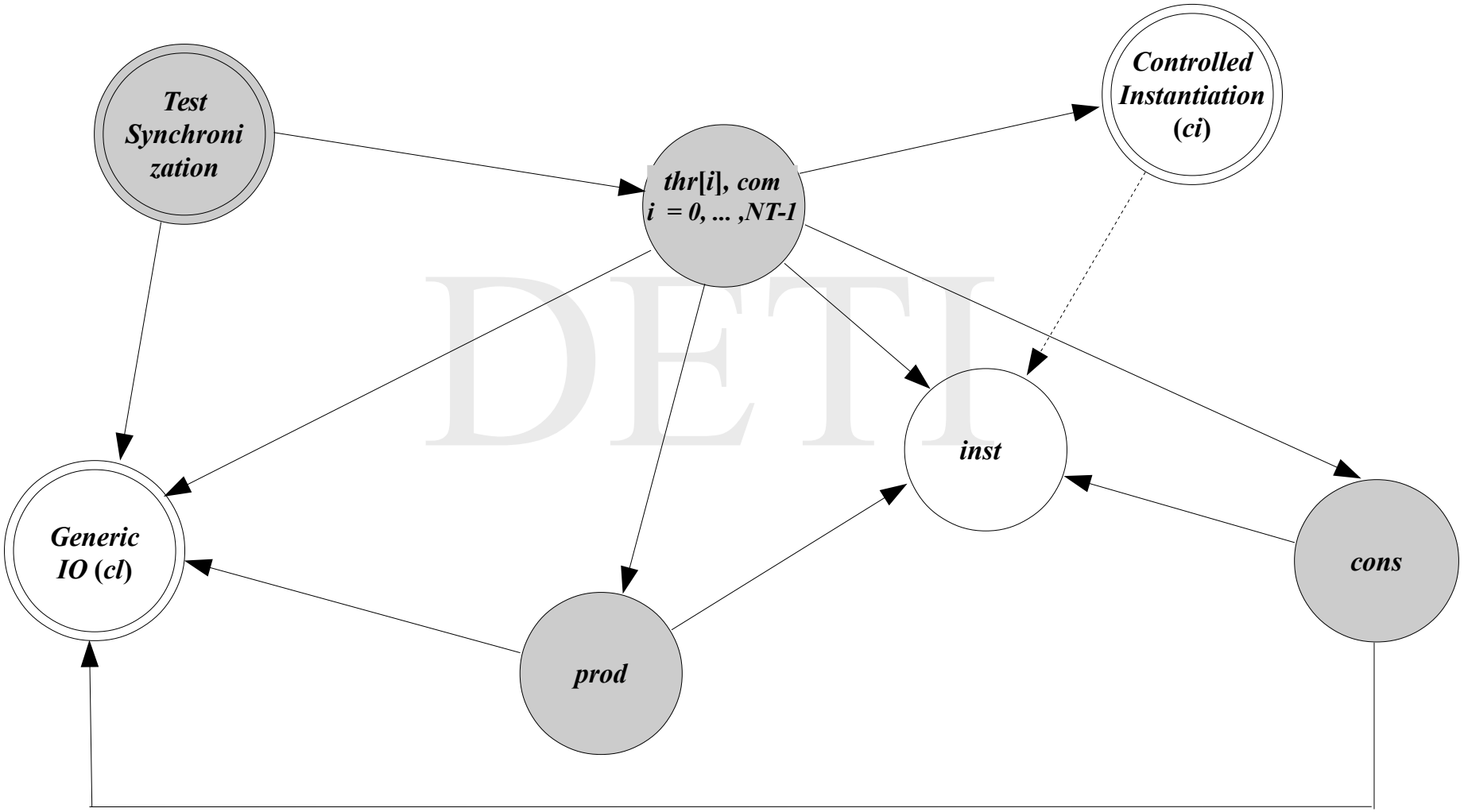
A necessidade da operação *notifyAll* surge clara se se pensar que, havendo apenas uma variável de condição por monitor, a única maneira de acordar um *thread* que esteja bloqueado numa condição específica é acordando *todos* os *threads* bloqueados e proceder de uma forma diferencial à determinação daquele que pode prosseguir.

Controlled instantiation - 1

O exemplo apresentado a seguir ilustra o recurso a monitores em diferentes situações

- o *thread* `main` lança quatro *threads* de tipo `TestThread` que procuram instanciar objectos de tipo `ControlledInstantiation`, cada um deles um objecto, e que irão constituir o local de transferência de um valor entre pares de *threads* posteriormente lançados, de tipo `Proc` e `Cons`, respectivamente;
- o tipo de dados `ControlledInstantiation`, porém, só permite a instanciação em simultâneo de um máximo de dois objectos;
- há, assim, três tipos de monitores presentes
  - o monitor associado ao objecto que representa em *runtime* o tipo de dados `ControlledInstantiation`;
  - o monitor associado a cada instanciação do tipo de dados `ControlledInstantiation`;
  - o monitor associado ao objecto que representa em *runtime* o tipo de dados `genclass.GenericIO`;
- o primeiro controla a instanciação de objectos de tipo `ControlledInstantiation`, o segundo a transferência do valor entre os threads de tipo `Prod` e `Cons` e o terceiro a impressão de informação no dispositivo de saída *standard*.

Controlled instantiation - 2



Controlled instantiation - 3

Valores impressos

```
I have already created the threads!
I, Thread_base_2, got the instantiation number 2 of data type
ControlledInstantiation!
I, Thread_base_1, got the instantiation number 1 of data type
ControlledInstantiation!
I, Thread_base_1, am going to create the threads that will exchange the value!
I, Thread_base_2, am going to create the threads that will exchange the value!
I, Thread_base_1_writer, am going to write the value 1 in instantiation number 1
of data type ControlledInstantiation!
I, Thread_base_2_writer, am going to write the value 2 in instantiation number 2
of data type ControlledInstantiation!
I, Thread_base_1_reader, read the value 1 in instantiation number 1 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_1_writer, has terminated.
My thread which reads the value, Thread_base_1_reader, has terminated.
I, Thread_base_1, am going to release the instantiation number 1 of data type
ControlledInstantiation!
I, Thread_base_0, got the instantiation number 3 of data type
ControlledInstantiation!
I, Thread_base_0, am going to create the threads that will exchange the value!
I, Thread_base_2_reader, read the value 2 in instantiation number 2 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_2_writer, has terminated.
My thread which reads the value, Thread_base_2_reader, has terminated.
I, Thread_base_2, am going to release the instantiation number 2 of data type
ControlledInstantiation!
```

Controlled instantiation - 4

```
I, Thread_base_3, got the instantiation number 4 of data type
ControlledInstantiation!
I, Thread_base_3, am going to create the threads that will exchange the value!
I, Thread_base_3_writer, am going to write the value 3 in instantiation number 4
of data type ControlledInstantiation!
I, Thread_base_0_writer, am going to write the value 0 in instantiation number 3
of data type ControlledInstantiation!
I, Thread_base_0_reader, read the value 0 in instantiation number 3 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_0_writer, has terminated.
My thread which reads the value, Thread_base_0_reader, has terminated.
I, Thread_base_0, am going to release the instantiation number 3 of data type
ControlledInstantiation!
The thread Thread_base_0 has terminated.
The thread Thread_base_1 has terminated.
The thread Thread_base_2 has terminated.
I, Thread_base_3_reader, read the value 3 in instantiation number 4 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_3_writer, has terminated.
My thread which reads the value, Thread_base_3_reader, has terminated.
I, Thread_base_3, am going to release the instantiation number 4 of data type
ControlledInstantiation!
The thread Thread_base_3 has terminated.
```

Semáforos - 1

O recurso directo a primitivas de tipo *sleep* e *wake up* não resolve por si só o problema. Continua a ser necessário garantir a *atomicidade* das operações!

```
/* estrutura de dados de controlo */
#define  R    ...        /* número de processos que pretendem acesso à região crítica,
                           pid = 0, 1, ..., R-1 */

shared unsigned int acesso = 1;

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
   if (acesso == 0) sleep (pid_proprio);
      else acesso -= 1;
}

/* primitiva de saída da região crítica */
void saida_de_RC (unsigned int pid_proprio)
{
   if (há processos bloqueados) wake_up_one ();
      else acesso += 1;
}
```

operação atómica
(a sua execução não pode ser interrompida)

operação atómica
(a sua execução não pode ser interrompida)

Semáforos - 2

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

```
typedef struct
{ unsigned int val;  /* valor de contagem */
  NODE *queue;       /* fila de espera dos processos bloqueados */
} SEMAPHORE;
```

sobre a qual é possível executar as duas operações atómicas seguintes

sem_down – se o campo `val` for não nulo, o seu valor é decrementado; caso con-trário, o processo que executou a operação é bloqueado e a sua identificação é colocada na fila de espera `queue`;

sem_up – se houver processos bloqueados na fila de espera `queue`, um deles é acordado (de acordo com uma qualquer disciplina previamente definida); caso contrário, o valor do campo `val` é incrementado.

Um semáforo só pode ser manipulado desta maneira e *é precisamente para garantir isso* que toda e qualquer referência a um semáforo particular é sempre feita de uma forma indirecta.

Semáforos - 3

```
/* array de semáforos definidos no kernel */
#define  R   ...            /* número de semáforos - semid = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

/* operação down */
void sem_down (unsigned int semid)
{
  inibição das interrupções;
  if (sem[semid].val == 0) sleep_on_sem (getpid(), semid);
     else sem[semid].val -= 1;
  activação das interrupções;
}

/* operação up */
void sem_up (unsigned int semid)
{
  inibição das interrupções;
  if (há processos bloqueados no semáforo sem[semid]) wake_up_one_on_sem (semid);
     else sem[semid].val += 1;
  activação das interrupções;
}
```

Semáforos - 4

A implementação das operações *sem_down* e *sem_up* apresentada é característica de um monoprocessador!

Porquê?

Como é que as mesmas operações poderão ser implementadas no caso de um multiprocessador?

Semáforos em Java

```
public class Semaphore
{
   private int val = 0,                        // green / red indicator
               numbBlockThreads = 0;           // number of the blocked threads
                                               //  in the monitor

   public synchronized void down ()
   {
     if (val == 0)
        { numbBlockThreads += 1;
          try
          { wait ();
          }
          catch (InterruptedException e) {}
        }
        else val -= 1;
   }

   public synchronized void up ()
   {
     if (numbBlockThreads != 0)
        { numbBlockThreads -= 1;
          notify ();
        }
        else val += 1;
   }
}
```

Biblioteca concurrency de Java - 1

A biblioteca `concurrency` de Java fornece os dispositivos de sincronização seguintes

- *barreiras* – dispositivos que conduzem ao bloqueio de grupos de *threads* até que estejam reunidas condições para a sua continuação; quando a barreira é levantada, todos os processos bloqueados prosseguem
- *semáforos* – trata-se de uma implementação de semáforo mais geral do que o modelo prescrito por Dijstra, permitindo nomeadamente a realização de operações de *down* e *up* em que o campo interno `val` é, respectivamente, decrementado e incrementado de mais do que uma unidade de cada vez e de uma operação de *down* não bloqueante
- *exchanger* – dispositivos que permitem a troca de valores entre pares de *threads* usando uma sincronização de tipo *rendez-vous*

Biblioteca concurrency de Java - 2

- *locks* – monitores de tipo Lampson-Redell de caracter geral (ao contrário do monitor *built-in* na linguagem, é possível definir-se aqui múltiplas variáveis de condição e obter-se uma funcionalidade semelhante à fornecida pela biblioteca *pthread*
- *manipulação atómica de variáveis* – mecanismo de tipo *read-modify-write* que permite operar sobre o conteúdo de diferentes tipos de variáveis sem a ocorrência de *condições de corrida*.

Leituras sugeridas

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Capítulo 6: *Operating systems support*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Capítulo 3: *Processes*