Sistemas Distribuídos

Simultaneidade 1

António Rui Borges

Resumo

- *Programa vs. Processo*
  - *Caracterização de um ambiente multiprogramado*
- *Processos vs. Threads*
  - *Caracterização de um ambiente multithread*
- *Ambiente de execução*
- *Tópicos em Java*
- *Leituras sugeridas*

Programa vs. Processo

De um modo geral, um *programa* pode ser definido como uma sequência de instruções que descreve a execução de uma determinada tarefa em um computador. Contudo, para que esta tarefa seja *na verdade* realizado, o programa correspondente deve ser executado.

A execução de um programa é chamada de *processo*.

Representando uma atividade que está ocorrendo, um *processo* é caracterizado por

- o *espaço de endereçamento* – o código e o valor atual de todas as suas variáveis associadas
- o *contexto do processador* – o valor atual de todos os registros internos do processador
- o *Contexto de E/S* – todos os dados que estão sendo transferidos para os dispositivos de entrada e de saída
- o *estado* da execução.

Modelagem dos processos - 1

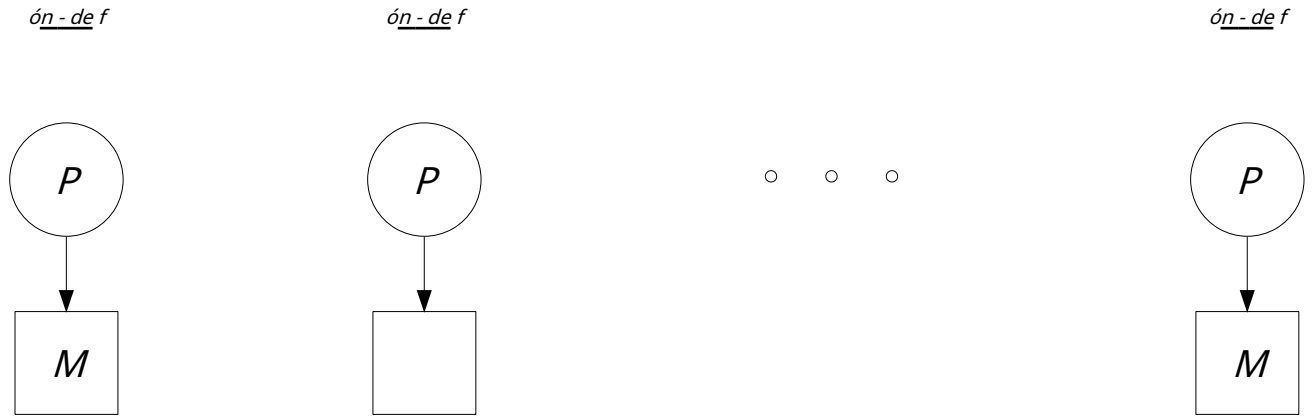
Multiprogramação, ao criar uma imagem de aparente simultaneidade na execução de diferentes programas pelo mesmo processador, dificulta muito a percepção das atividades que estão ocorrendo ao mesmo tempo.

Esta imagem pode ser simplificada se, em vez de tentar seguir o caminho de execução percorrido pelo processador em seu contínuo meandro entre processos, se supor que há um conjunto de processadores virtuais, um por processo que coexiste concorrentemente, e que os processos são executados em paralelo através da ativação (*sobre*) e a desativação (*desligado*) dos processadores associados.

Supõe-se ainda neste modelo que

- a execução do processo não é afetada pelo instante e pela localização do código onde ocorre a comutação
- nenhuma restrição é imposta ao tempo de execução total ou parcial.

Modelagem dos processos - 2



- o *comutação* do *contexto do processo* é simulado pela ativação e desativação dos processadores virtuais e é controlado por seu *estado*
- em um *monoprocessador*, o número de processadores virtuais ativos em qualquer instante é um, no máximo
- em um *processador multicore*, o número de processadores virtuais ativos em qualquer instante é igual ao número de processadores no núcleo, no máximo.

Diagrama de estado do processo - 1

Um processo pode estar em diferentes situações, chamadas *estados*, ao longo de sua existência. Os estados mais importantes são os seguintes

- *correr*–quando ele contém o processador e está, portanto, em execução
- *Pronto para correr*–quando espera a atribuição do processador para iniciar ou retomar a execução
- *bloqueado*–quando é impedido de prosseguir até que ocorra um evento externo (acesso a um recurso, conclusão de uma operação de entrada/saída, etc.).

As transições de estado são geralmente acionadas por uma fonte externa, o sistema operacional, mas podem ser acionadas pelo próprio processo em alguns casos.

A parte do sistema operacional que lida com as transições de estado [do processo] é chamada de *Agendador* (*agendador de processador*, neste caso), e forma parte integrante do seu núcleo, o *núcleo*, que é responsável pelo tratamento de exceções e pelo agendamento da atribuição do processador e de todos os outros recursos do sistema aos processos.

Diagrama de estado do processo - 2

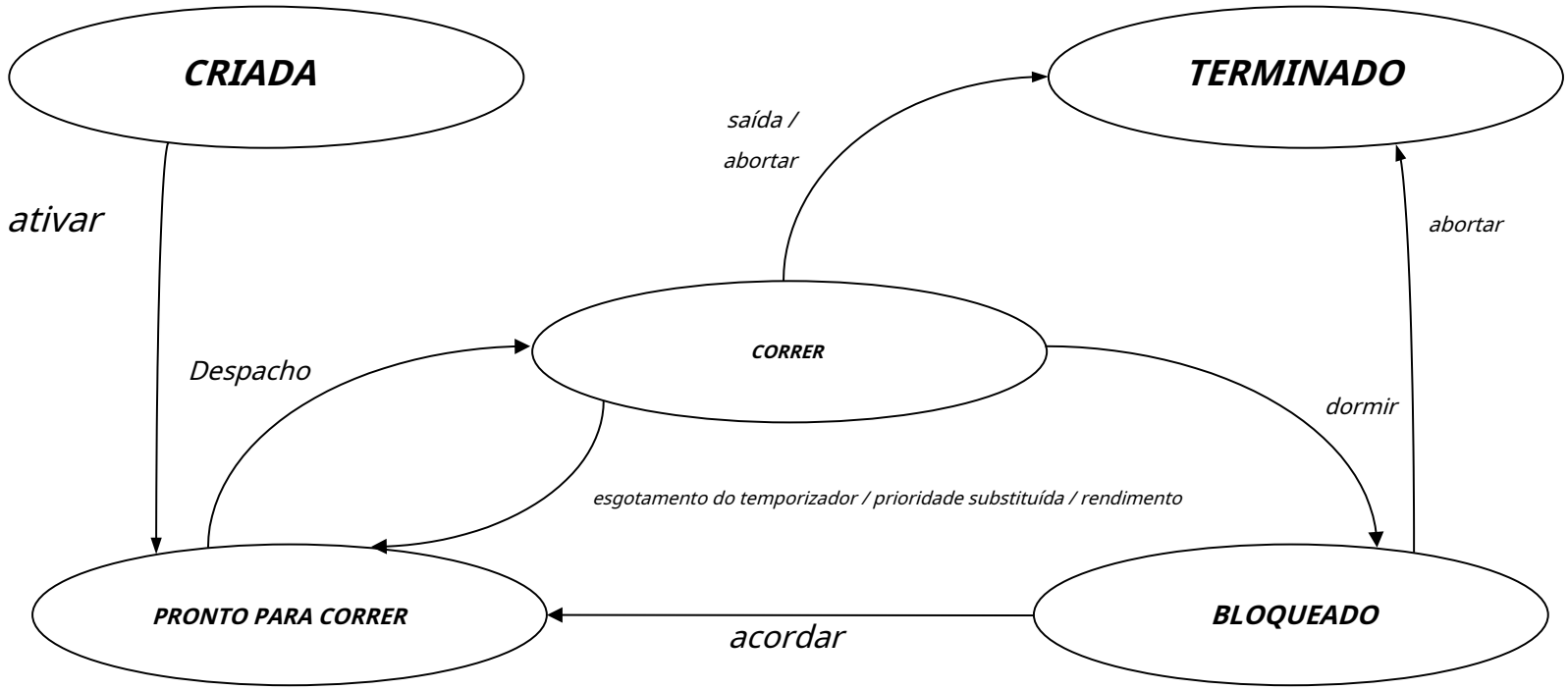


Diagrama de estado do processo - 3

ativar–um processo é criado e colocado no*fila pronta para execução*esperando para ser agendado para execução

Despacho -um dos processos*fila pronta para execução*é selecionado pelo agendador para execução

esgotamento do temporizador -o processo em execução esgotou o intervalo de tempo do processador que estava atribuído a ele (*agendamento preventivo*)

prioridade substituída–o processo em execução perde o processador porque o*Pronto para correr fila*agora contém um processo de maior prioridade que requer o processador (*agendamento preventivo*)

colheita–o processo libera voluntariamente o processador para permitir que outros processos sejam executados (*não-agendamento preventivo*)

dormir -o processo é impedido de prosseguir e deve aguardar a ocorrência de um evento externo

acordar -o evento externo que o processo estava esperando ocorreu

sair / abortar–o processo foi encerrado/é forçado a encerrar sua execução e aguarda os recursos que lhe foram atribuídos sejam liberados

Processos vs. Threads - 1

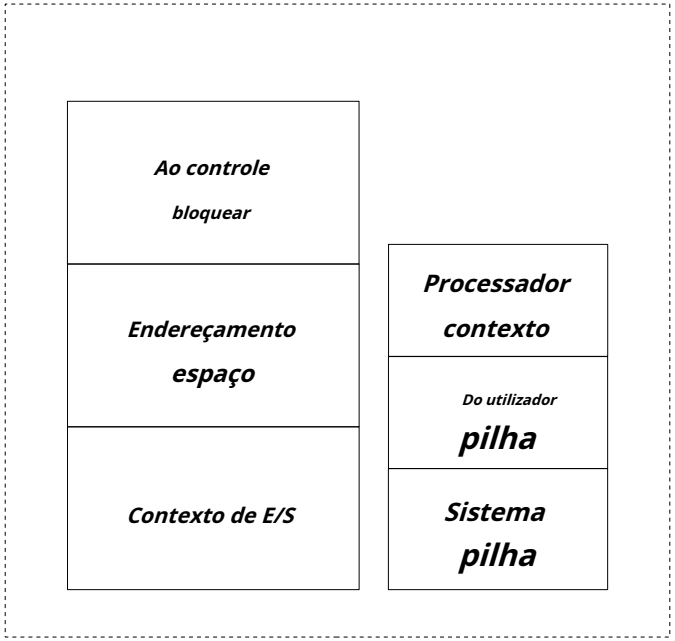
O conceito de *processo* incorpora as seguintes propriedades

- *propriedade de recursos* – um espaço de endereçamento privado e um conjunto privado de canais de comunicação com os dispositivos de entrada e saída
- *thread de execução* – a *contador de programa* que aponta para a instrução que deve ser executada a seguir, um conjunto de *registros internos* que contém os valores atuais das variáveis que estão sendo processadas e um *pilha* que mantém o histórico de execução (um *quadro* para cada rotina que foi chamada e ainda não retornou).

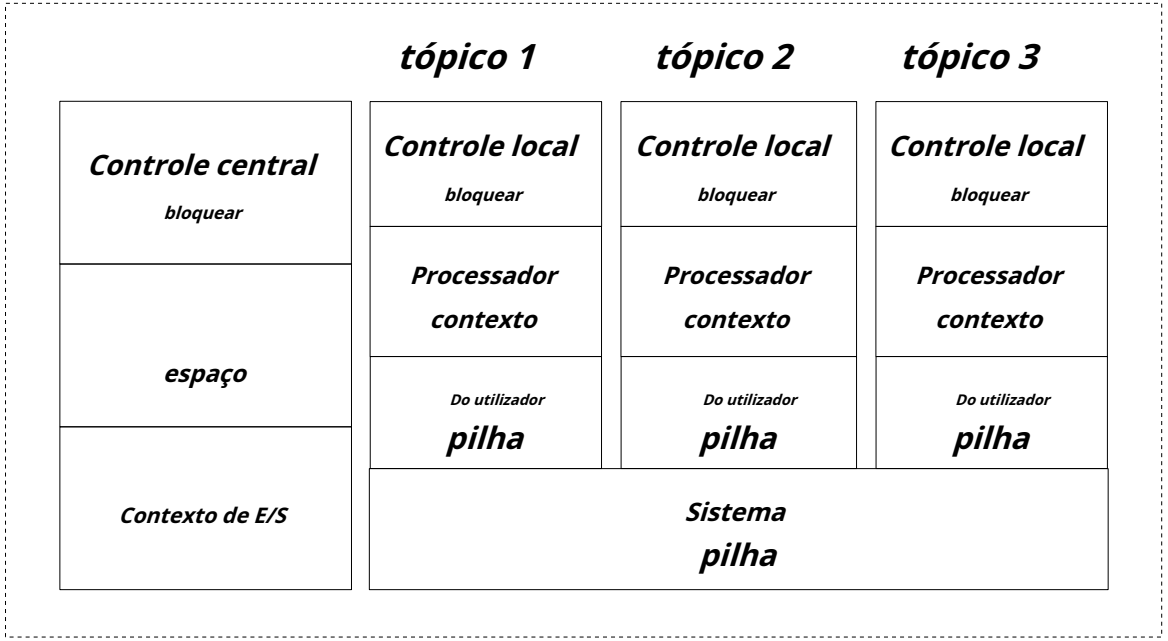
Estas propriedades, embora tomadas em conjunto num *processo*, pode ser tratado separadamente pelo ambiente de execução. Quando isso acontece, *processos* são concebidos como agrupando um conjunto de recursos e *tópicos*, também conhecido como *processos leves*, representam entidades independentes executáveis no contexto de um único processo.

Multithreading, então, significa um ambiente onde é possível criar múltiplos *threads de execução* dentro do mesmo processo.

Processos vs. Threads - 2



Rosqueamento único

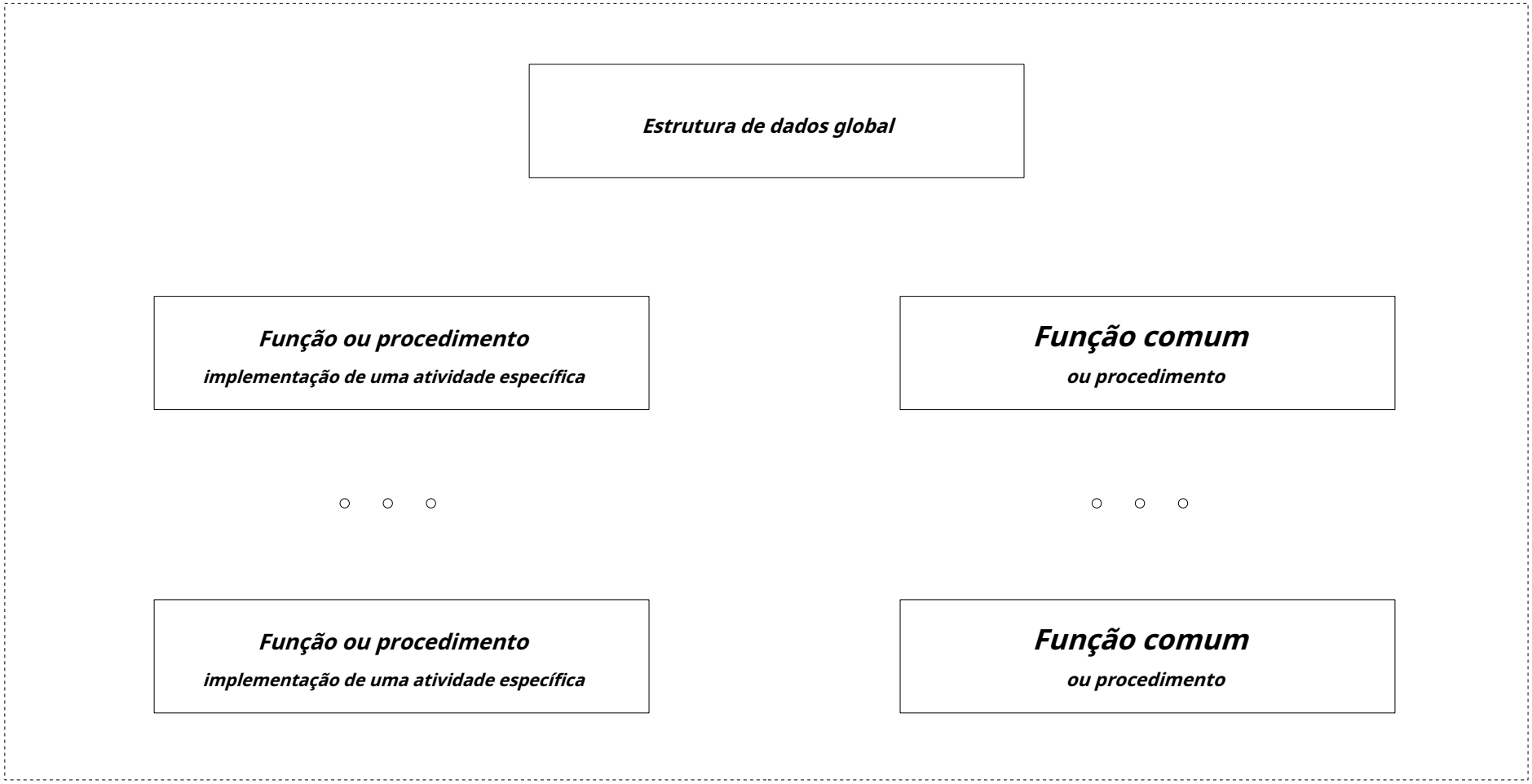


Multithreading

Vantagens de um ambiente multithread

- *maior simplicidade na decomposição da solução e maior modularidade na sua implementação*– programas que envolvem múltiplas atividades e atendem a múltiplas solicitações são mais fáceis de projetar e implementar em uma perspectiva concorrente do que em uma perspectiva puramente sequencial
- *melhor gerenciamento dos recursos do sistema de computador*–compartilhando o espaço de endereçamento e o contexto de E/S entre os *tópicos* de uma aplicação resulta na diminuição da complexidade do gerenciamento da ocupação da memória principal e do acesso aos dispositivos de entrada/saída
- *maior eficiência e velocidade de execução* – um sol decomposição da solução com base em *tópicos*, ao contrário daquela baseada em processos, requer menos recursos do sistema operacional, permitindo que operações como criação e encerramento de processos e comutação de contexto se tornem menos pesadas e, portanto, mais eficientes; além disso, torna-se possível, no multiprocessamento simétrico, agendar múltiplas operações para execução paralela. *tópicos* pertencentes à mesma aplicação, aumentando assim a velocidade de execução

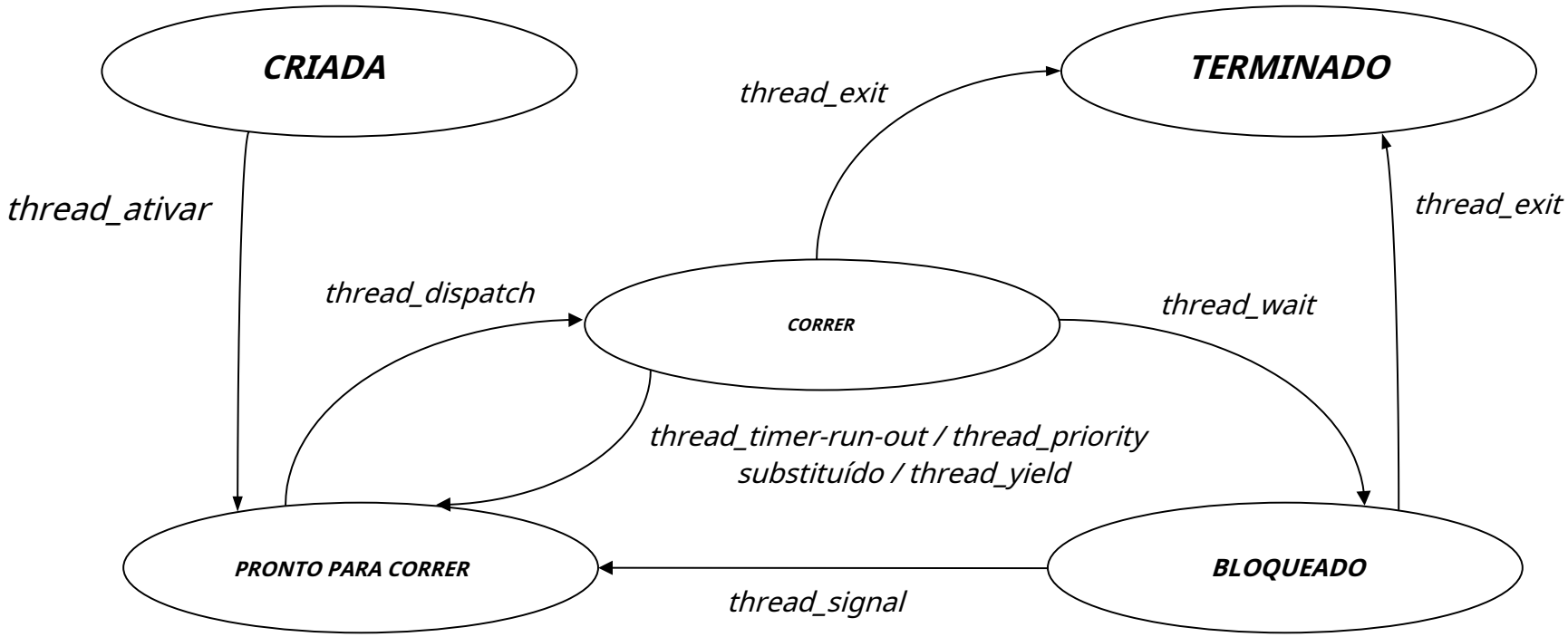
Organização de um programa multithread - 1



Organização de um programa multithread - 2

- cada *fio* normalmente está associado à execução de um *função ou procedimento que implementa uma atividade específica*
- o *dados globais* estrutura forma um espaço de compartilhamento de informações, definido em termos de variáveis e canais de comunicação com os dispositivos de entrada/saída, para ser acessado pelos múltiplos *tópicos* que coexistem em um determinado momento para escrever e para ler
- o *programa principal*, representado no diagrama por um *função ou procedimento que implementa uma atividade específica*, constitui o primeiro *fio* a ser criado e, normalmente, o último *fio* Para ser concluído

Diagrama de estado do thread



Suporte à implementação de um ambiente multithread

- *tópicos de nível de usuário*–*tópicos* são implementados por uma biblioteca específica em nível de usuário que dá suporte à criação, gerenciamento e agendamento de *fios sem* núcleo inteiro referência; seu gene come um cinco muito versátil e portátil, mas implementação ineficiente, pois, como o kernel percebe apenas processos, quando um determinado *fio* invoca um bloqueio *chamada de sistema*, todo o processo é bloqueado, mesmo que houvesse *tópicos* pronto para ser executado
- *threads em nível de kernel*–*tópicos* são implementados no nível do kernel, fornecendo diretamente as operações para a criação, gerenciamento e escalonamento de *tópicos* ; a implementação é específica do sistema operacional, mas o bloqueio de um determinado *fio* não afeta o despacho do restante para execução e a execução paralela em um processador multicore torna-se possível

Ambiente de execução - 1

Máquina Virtual JAVA(JVM) constitui o ambiente de execução para um aplicativo codificado em Java. Em princípio, a JVM roda no topo do sistema operacional da plataforma de hardware que executa um programa Java e estabelece com ele uma conexão muito íntima. O acesso às informações relativas ao ambiente de execução pode ser obtido através da invocação de métodos em dois tipos de dados de referência da biblioteca base Java,java.lan: e .

Dentre as informações fornecidas, destacam-se as seguintes

- número de processadores e tamanho de memória disponível para executar o código
- referências aos fluxos associados a dispositivos de entrada padrão, saída padrão e erro padrão
- acesso a um conjunto modificável de definições, chamado*propriedades*, que caracterizam o ambiente de execução
- acesso somente leitura às definições das variáveis da interface de usuário do sistema operacional, o*concha*, onde o comando java foi executado – chamado neste contexto*ambiente*.

meio ambiente - 1

[ambiente ruib@ruib-laptop]$java CollectEnvironmentData

Caracterização de Máquina Virtual Java (JVM)
N. de processadores disponíveis = 8
Tamanho da memória dinâmica atualmente livre (em bytes) = 248250352
Tamanho da memória dinâmica total (em bytes) = 249561088
Tamanho máximo da memória principal disponível da plataforma de hardware onde a máquina virtual Java está instalada (em bytes) = 367840460

Propriedades do ambiente de execução java.runtime.name = Java(TM)
SE Runtime Environment sun.boot.library.path = /opt/jdk1.8.0_241/jre/
lib/amd64 java.vm.vendor = Oracle Corporation

java.vendor.url = http://java.oracle.com/
path.separator = :
java.vm.name = VM do servidor Java HotSpot(TM) de 64 bits
user.dir =
/home/ruib/Teaching/SD/2020_2021/aulas teóricas/exemplos demonstrativos/
threadBasics/ambiente
java.runtime.version = 1.8.0_241-b07
java.io.tmpdir = /tmp
os.nome = Linux
sun.jnu.encoding = UTF-8 os.version =
5.10.22-100.fc32.x86_64 user.home = /home/
ruib

. . .

meio ambiente - 2

Variáveis do ambiente de execução

CAMINHO = /opt/jdk1.8.0_241/bin:/opt/jdk1.8.0_241/jre/bin:/opt/mpich/bin:/home/ ruib/.local/bin:/home/ ruib/bin:/usr /lib64/ccache:/usr/local/bin:/usr/bin:/ bin:/usr/local/sbin:/usr/sbin

LC_MEASUREMENT = pt_PT.UTF-8

LC_COLLATE = pt_PT.UTF-8 LOGNAME

= ruib

PCD = /home/ruib/Ensino/SD/2020_2021/aulas teóricas/exemplos demonstrativos/threadBasics/environment

c=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx= 00;36 :*.xspf=00;36:

XDG_SESSION_DESKTOP = KDE

SHLVL = 1

LC_MONETARY = pt_PT.UTF-8

DISPLAY = :0

LC_NUMERIC = pt_PT.UTF-8

HOME = /home/ruib

. . .

Ambiente de execução - 2

O ambiente de execução também permite a execução de comandos diretamente na interface de usuário do sistema operacional subjacente. Dois tipos de dados de referência da biblioteca base Java,java.lang,são fundamentais para cumprir esse propósito:Construtor de Processose Processo.

CadaProcessoBuild objeto gerencia atributos do processo, como o*concha*com-comando a ser executado e a configuração do diretório de trabalho e dos fluxos associados aos dispositivos de entrada padrão, saída padrão e erro padrão. Vários processos podem ser criados em sucessão a partir do mesmo objeto, compartilhando assim a mesma configuração de atributos.

Cada processo criado é uma instância doProcessotipo de dados. Aqui são fornecidos meios para verificar se ele ainda está em execução, aguardar seu encerramento, obter seu status de encerramento, eliminá-lo e obter referências aos fluxos associados à sua entrada padrão, saída padrão e dispositivos de erro padrão.

comando de execução

[ruib@ruib-laptop runCommand]$java ListWorkDir Listando
o diretório de trabalho atual

- total 16

drwxrwxr-x. 2 ruib ruib 4096 17 de março 12h09 .
drwxrwxr-x. 8 ruib ruib 4096 26 de fevereiro de 2018 ..
- rw-rw-r--. 1 ruib ruib 1298 17 de março 12:09 ListWorkDir.class
- rw-rw-r--. 1 ruib ruib 1626 28 de fevereiro de 2017 ListWorkDir.java
- -
status de saída = 0

Tópicos em Java - 1

Sendo Java uma linguagem de programação concorrente, *tópicos* são suportados pela própria linguagem. Conceitualmente, a criação de um *fio* pressupõe a existência de duas entidades na máquina virtual Java: um objeto que representa um thread autônomo de execução, o *fio* em si, e um tipo de dados de referência não instanciado, ou um objeto instanciado a partir dele, que define o método executado pelo *fio*, seu ciclo de vida.

Máquina Virtual JAVA gerencia o *multithread* ambiente de acordo com as seguintes regras

- todo programa em execução consiste em pelo menos um *fio* que é criado implicitamente quando a máquina virtual, após inicializar o ambiente de execução, chama o método principal no tipo de dados inicial
- o restante *tópicos* são explicitamente criados pelo *fio* principal, ou por qualquer *fio* criado sucessivamente a partir do *fio* principal
- o programa termina quando todos os criados *tópicos* terminaram de executar seu método associado.

Biblioteca base Java,java.lang,fornece dois tipos de dados de referência, um usando o construtor*interface*e o outro usando o construtor*aula*, que são fundamentais para a construção de um*multithread*ambiente.

```
interface pública Executável {

    vazio público correr ();
}


aula pública Fio
{
        . . .
    vazio público correr ()
    vazio público começar ()
        . . .
}
```

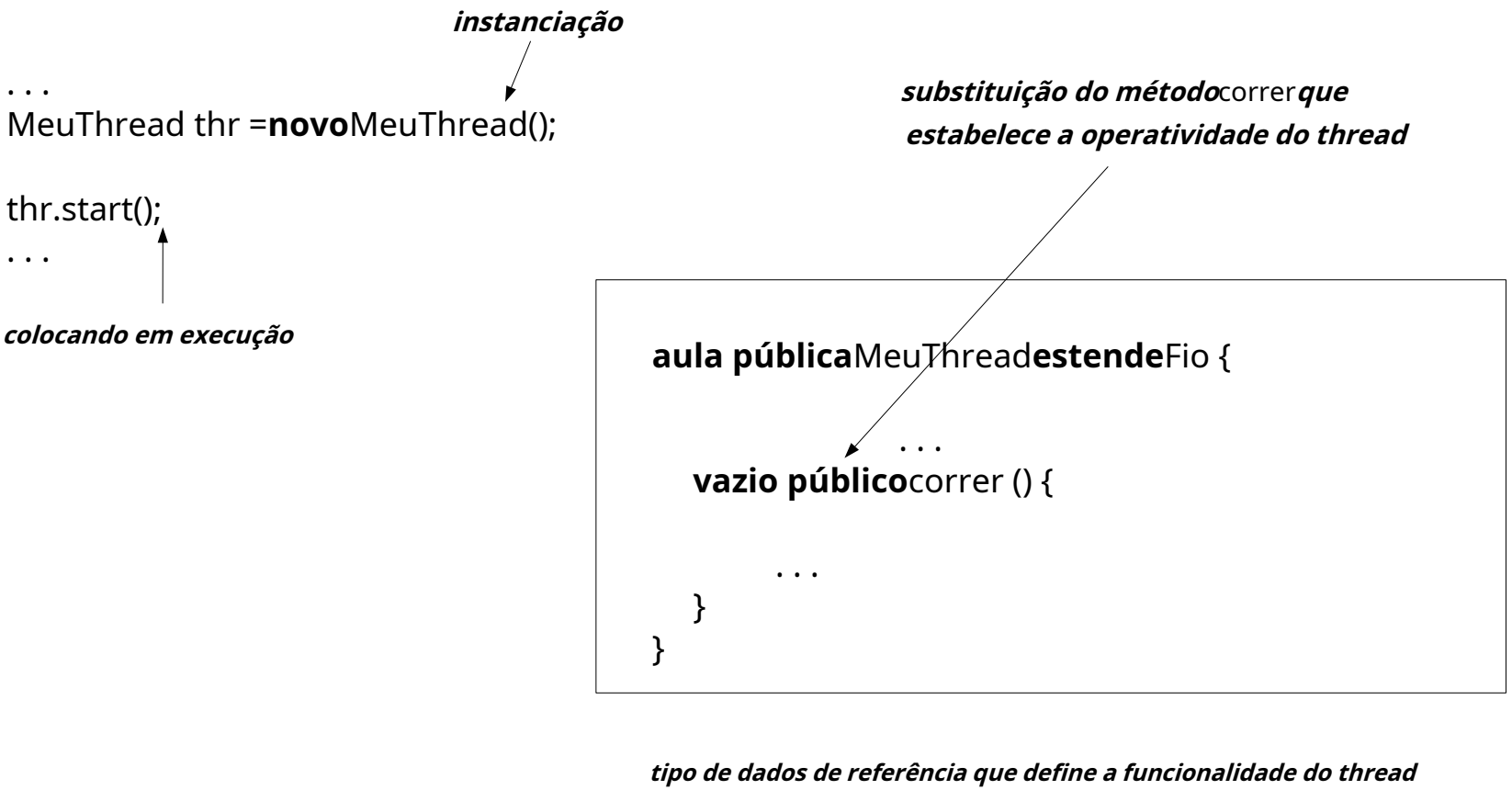
Cada thread autônomo de execução é uma instanciação do tipo de dados de referênciaFio.Define dois métodos que são operacionalmente relevantes neste contexto

- correr -que é chamado quando o*fio*é colocado em execução (iniciado) e que representa seu ciclo de vida
- começar -que é chamado para*começar*o*fio*.

Contudo, não é estritamente necessário criar novos tipos de dados de referência, derivados deFioe qual*sobrepor*o métodocorrer,para garantir a execução de tarefas específicas. O mesmo objetivo pode alternativamente ser alcançado criando um tipo de dados de referência independente que implemente a interface Executávele que, consequentemente, definecorrer.

A última é geralmente relatada como a abordagem preferida na literatura relacionada a Java, mas a primeira permite uma solução Java para herança múltipla, o que é bastante útil na construção de servidores que possuem uma diferenciação de serviço de cliente.

Criação de tópico(*abordagem 1*)



Criação de tópico (*abordagem 2*)

instanciação

```
. . .
Tópico thr = novo Fio (novo MeuThread());

thr.start();
. . .
```

colocando em execução

implementação do método correr *qual estabelece a operatividade do thread*

```
aula pública MeuThread implementos Executável {

        . . .
    vazio público correr () {

        . . .
    }
}
```

tipo de dados de referência que define a funcionalidade do thread

A *fio* tem os seguintes atributos

- *nome* – nome atribuído (por padrão, o ambiente de execução gera um *corda* de formato Fio-#, onde # é o número de criação incrementado sucessivamente de zero)
- *identificador interno* – número do tipo longo que é único e é mantido inalterado durante o *fio* vida
- *grupo* – agrupar o *fio* pertence a (todos os *tópicos* do mesmo aplicativo pertencem por padrão ao mesmo grupo, o grupo)
- *prioridade* – pode variar de 1 (*MIN_PRIORITY*) a 10 (*MAX_PRIORITY*), por padrão, o ambiente de execução atribui o valor 5 (*NORM_PRIORITY*)
- *estado* – *fio* Estado atual
  - NOVO (*CRIADA*), após a instanciação de uma variável de referência do tipo de dados Fio, ou de um tipo de dados derivado
  - EXECUTÁVEL (*PRONTO PARA CORRER* ou *CORRER*), quando espera pela execução ou está em execução
  - BLOQUEADO, ESPERANDO ou TIME_WAITING (*BLOQUEADO*), quando está bloqueado
  - TERMINADO (*TERMINADO*), após o término.

threadInfo

[ruib@ruib-laptop threadInfo]$java CollectThreadData

Caracterização de thread

Nome = principal
Identificador interno = 1
Grupo = principal
Prioridade = 5
Estado atual = EXECUTÁVEL

Estados possíveis:

NOVO
EXECUTÁVEL
BLOQUEADO
ESPERANDO
TIMED_WAITING
TERMINADO

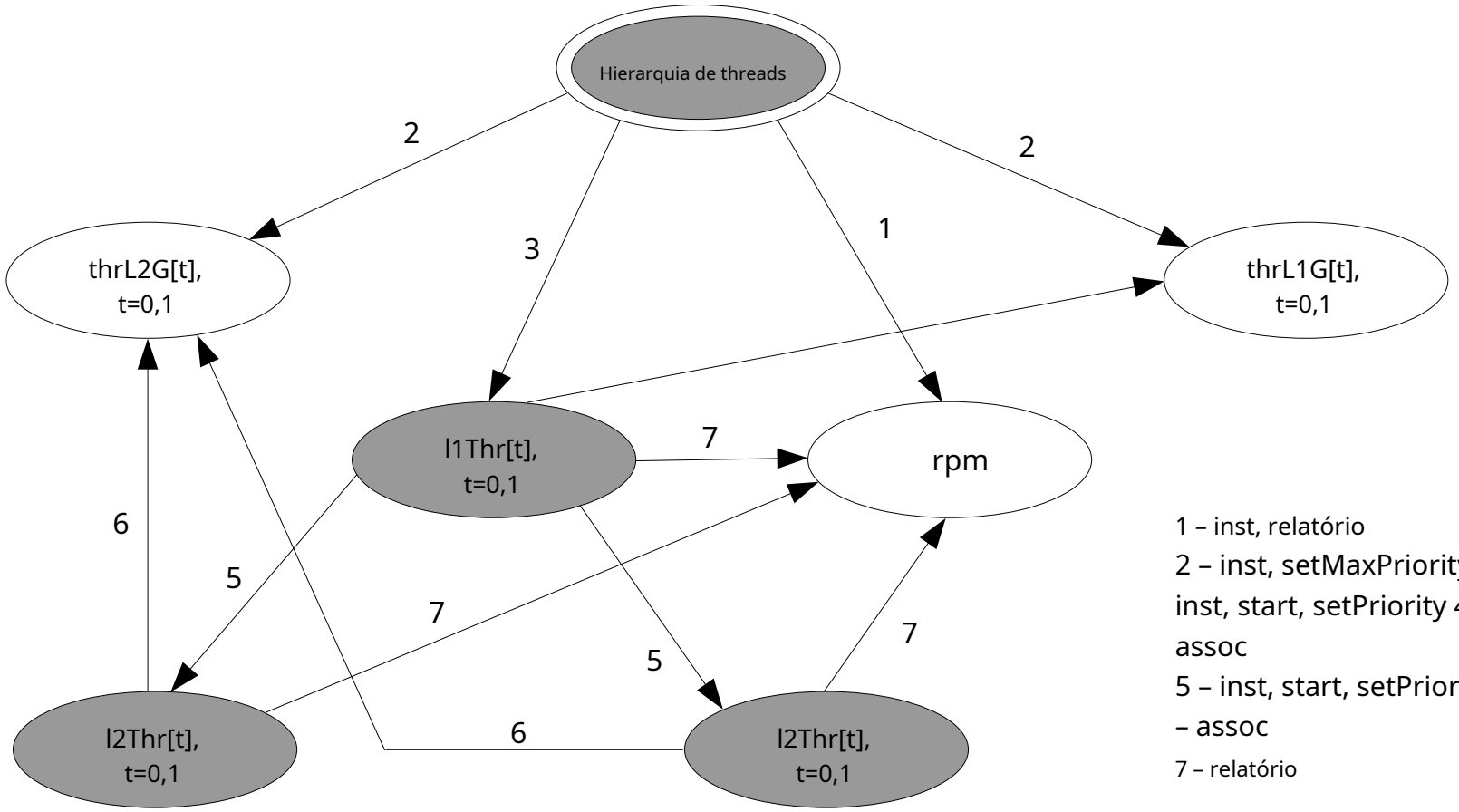
Tópicos em Java - 7

A partição em diferentes grupos do *tópicos* que são instanciados sucessivamente, permite organizar de forma hierárquica e funcional uma aplicação, bem como aproveitar as especificidades do Java na definição de propriedades comuns e na interação com elas. Biblioteca base Java, java.lang, fornece o tipo de dados de referência Três para cumprir esse propósito.

Assim, verifica-se que é possível

- definir logo no início a prioridade máxima e a propriedade de ser um *daemon* associados a um determinado grupo – todos os *tópicos* posteriormente instanciado, e pertencente a este grupo, manterá essas propriedades
- enviar uma interrupção para *todos* o *tópicos* pertencer ao mesmo grupo em uma operação única, ao invés de fazê-lo separadamente para cada membro do grupo.

hierarquia - 1



1 – inst, relatório
2 – inst, setMaxPriority 3 – inst, start, setPriority 4 – assoc
5 – inst, start, setPriority 6 – assoc
7 – relatório

hierarquia - 2

nível 0

nome do tópico: principal
threadId: 1
Linha prioritária: 5
threadGroupName: principal
threadParentGourpName: sistema

nível 1

nome do tópico: Tópico_L1.1
threadId: 8
Linha prioritária: 9
threadGroupName: Tópico_G1.1
threadParentGroupName: principal

nome do tópico: Tópico_L1.2
threadId: 9
Linha prioritária: 8
threadGroupName: Tópico_G1.2
threadParentGroupName: principal

nome do tópico: Thread_L1.1_L2.1
threadId: 11
Linha prioritária: 8
threadGrpName: Tópico_G1.1_G2
threadPrtGrpName: Tópico_G1.1

nome do tópico: Thread_L1.1_L2.2
threadId: 12
Linha prioritária: 8
threadGrpName: Tópico_G1.1_G2
threadPrtGrpName: Tópico_G1.1

nome do tópico: Thread_L1.2_L2.1
threadId: 13
Linha prioritária: 7
threadGrpName: Tópico_G1.2_G2
threadPGroupName: Tópico_G1.2

nome do tópico: Thread_L1.2_L2.2
threadId: 14
Linha prioritária: 7
threadGrpName: Tópico_G1.2_G2
threadPGroupName: Tópico_G1.2

nível 2

hierarquia - 3

Valores impressos

[ruib@ruib-hierarquia de laptop_1]$java ThreadHierarquia1

Número de threads de nível 1? 2
Número de threads de nível 2 por threads de nível 1? 2

Nome do tópico: principal
ID do tópico: 1
Linha prioritária: 5
Nome do grupo de threads: principal
Nome do grupo pai do grupo de threads: sistema
N. de threads ativos no grupo de threads: 1 *Nome dos threads ativos no grupo de threads*: principal
N. de subgrupos ativos no grupo de threads: 0

Nome do tópico: Thread_L1.1_L2.1 *ID do tópico*: 11
Linha prioritária: 8
Nome do grupo de threads: Tópico_G1.1_G2
Nome do grupo pai do grupo de threads: Tópico_G1.1
N. de threads ativos no grupo de threads: 2
Nome dos threads ativos no grupo de threads: Tópico_L1.1_L2.1 -
N. de subgrupos ativos no grupo de threads: 0

. . .

Tópicos em Java - 8

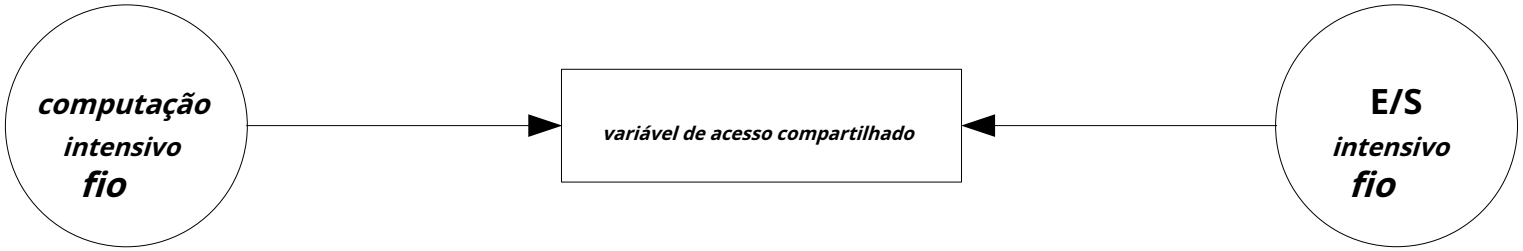
A máquina virtual Java supõe uma política de *agendamento não preemptivo* baseado em um sistema de prioridades estáticas com 10 níveis

- as transições entre o estado *CORRER* e o estado *PRONTO PARA CORRER* são do tipo *thread_priority_superseded* e *thread_rendimento*
- a prioridade atribuída a um *fio*, definido na instanciação ou modificado antes de sua criação, permanece inalterado durante seu tempo de vida ativo.

A máquina virtual Java, no entanto, não impõe estritamente a *agendamento* política. A implementação tem muita liberdade sobre como funciona. Isto é particularmente verdadeiro quando a máquina virtual Java é executada no topo de um sistema operacional multitarefa de uso geral!

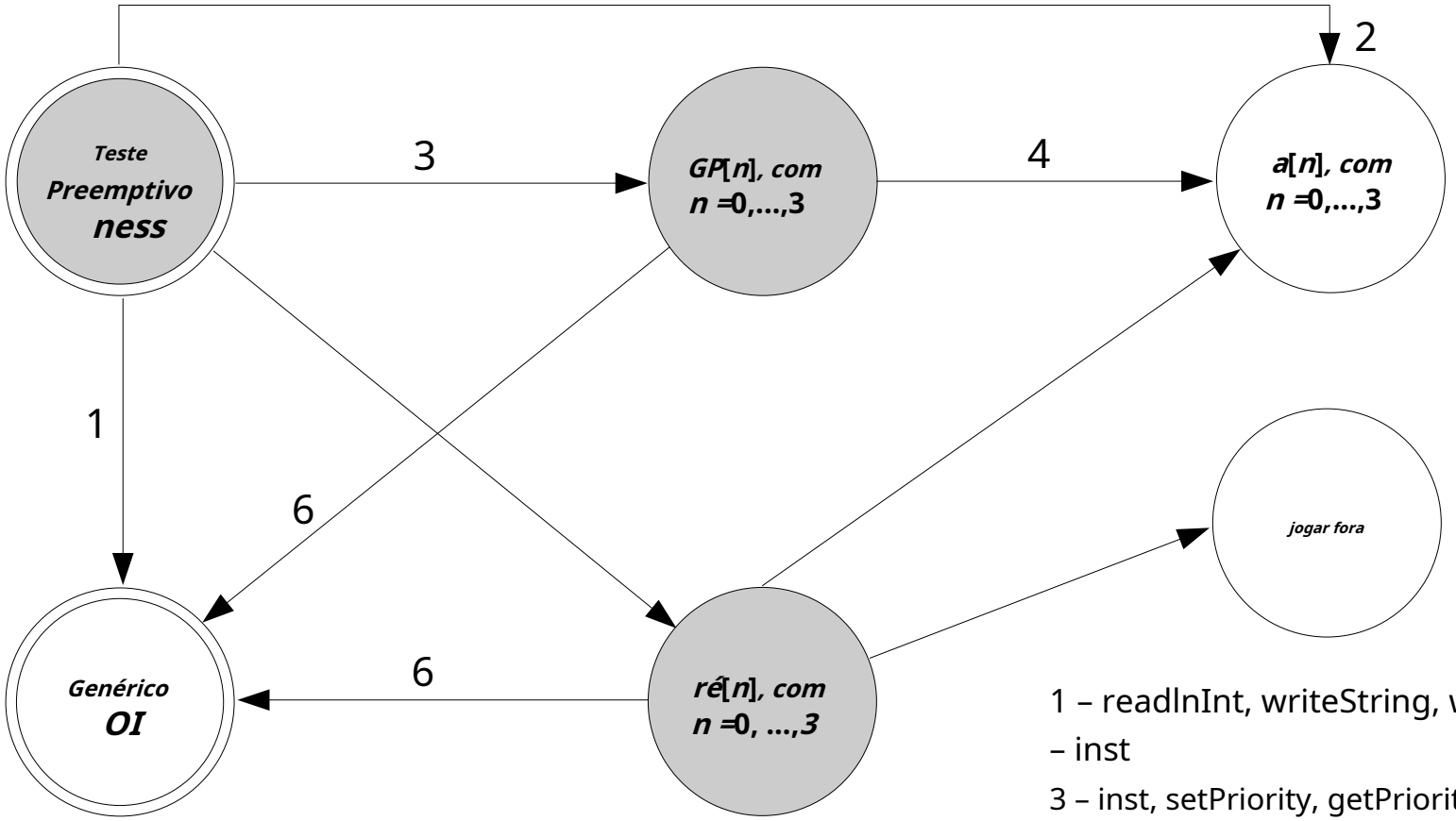
No Linux, por exemplo, threads Java são *núcleo* threads de nível, aproveitando *multicore* processadores para melhorar a execução paralela, e o predominante *agendamento* a política é a local.

testePreemptividade - 1



- existem quatro conjuntos, cada um consistindo de um thread intensivo de computação, um thread intensivo de E/S e uma variável de acesso compartilhado
- o thread de computação intensiva aumenta sucessivamente em uma variável de acesso compartilhado 10 milhões de vezes
- o thread intensivo de E/S lê e imprime sucessivamente a variável de acesso compartilhado até que seu valor atinja 10 milhões
- prioridades de thread podem ser alteradas e *colheita* pode ser introduzido após cada operação de incremento do thread de computação intensiva ser executada
- o número de leituras e impressões realizadas pelo thread intensivo de E/S é usado como uma figura de mérito para estimar a velocidade relativa de execução de ambos os threads

testePreemptividade - 2



1 – readlnInt, writeString, writelnString 2 – inst

3 – inst, setPriority, getPriority, start 4 – variável de escrita/leitura

5 – ler variável 6 – writelnString

7 – inst, escrever, liberar

testePreemptividade - 3

Sem rendimento de rosca

[ruib@ruib-laptop testPreemptividade]$java TestPreemptiveness

*Nível de prioridade de threads de computação intensiva?*1 *Nível de prioridade de threads intensivos de E/S? 10* Prioridade de threads com uso intensivo de computação = 1 Prioridade de threads com uso intensivo de E/S = 10
Já iniciei os threads intensivos de E/S!
Já comecei os threads de computação intensiva! Eu terminei meu trabalho!

N. de iterações na impressão de valores de A = 59702 A = 10000000
N. de iterações na impressão de valores de D = 123791 D = 10000000
N. de iterações na impressão de valores de C = 111804 C = 10000000
N. de iterações na impressão de valores de B = 102411 B = 10000000

testePreemptividade - 4

Sem rendimento de rosca

[ruib@ruib-laptop testPreemptividade]$java TestPreemptiveness

*Nível de prioridade de threads de computação intensiva?*10 *Nível de prioridade de threads intensivos de E/S?*1 Prioridade de threads com uso intensivo de computação = 10 Prioridade de threads com uso intensivo de E/S = 1
Já iniciei os threads intensivos de E/S!
Já comecei os threads de computação intensiva! Eu terminei meu trabalho!

N. de iterações na impressão de valores de C = 62180 C = 10000000
B = 1.000.000
N. de iterações na impressão de valores de B = 103934
N. de iterações na impressão de valores de D = 91698 D = 10000000
UMA = 1.000.0000
N. de iterações na impressão de valores de A = 119485

testePreemptividade - 5

Com rendimento de rosca

[ruib@ruib-laptop testPreemptividade]$java TestPreemptiveness

*Nível de prioridade de threads de computação intensiva?*1 *Nível de prioridade de threads intensivos de E/S?*10 Prioridade de threads com uso intensivo de computação = 1 Prioridade de threads com uso intensivo de E/S = 10
Já iniciei os threads intensivos de E/S!
Já comecei os threads de computação intensiva! Eu terminei meu trabalho!

N. de iterações na impressão de valores de C = 4834998 C = 10000000
N. de iterações na impressão de valores de D = 5085208 D = 10000000
N. de iterações na impressão de valores de A = 4931095 A = 10000000
N. de iterações na impressão de valores de B = 5193997 B = 10000000

testePreemptividade - 6

Com rendimento de rosca

[ruib@ruib-laptop testPreemptividade]$java TestPreemptiveness

*Nível de prioridade de threads de computação intensiva?*10 *Nível de prioridade de threads intensivos de E/S?*1 Prioridade de threads com uso intensivo de computação = 10 Prioridade de threads com uso intensivo de E/S = 1
Já iniciei os threads intensivos de E/S!
Já comecei os threads de computação intensiva! Eu terminei meu trabalho!

N. de iterações na impressão de valores de D = 5053297 D = 10000000
UMA = 1.000.0000
N. de iterações na impressão de valores de A = 5151587
N. de iterações na impressão de valores de B = 5461388 B = 10000000
N. de iterações na impressão de valores de C = 5392911 C = 10000000

Deve-se notar que o campoado tipo de dados de referênciaVariávelinclui o modificadorvolátil.Seu significado preciso é informar ao compilador Java que o *tópicos*, durante a sua execução, deverão observar permanentemente uma*consistente*valor na variávela.

*Consistência*significa, neste sentido, que o acesso à variáveladeve ocorrer sempre da maneira exata prescrita pelo código de cada*fio*.

Esta informação é crucial aqui porque o modelo de memória Java permite ao compilador, ao gerar o*bytecódigo*de um determinado tipo de dados de referência, bem como da máquina virtual Java, ao interpretar este*bytecódigo*, para realizar otimização de código que, sendo totalmente consistente em um*singlethreaded*ambiente, pode produzir uma execução paradoxal em um *multithread*ambiente.

Tópicos em Java - 10

Um Java *multithread* aplicação termina em princípio quando todos os seus constituintes *tópicos* terminar. Em aplicações complexas, onde o número de suportes *tópicos* é muito grande, lidando com situações excepcionais que podem exigir o aborto das operações, pode tornar-se bastante extenuante e exigir a introdução de código específico cuja utilidade prática é questionável.

Para simplificar o problema, Java apresenta duas alternativas

- *chamando o método* para nós) -que forçosamente encerra a máquina virtual Java, retornando o comunicado *status* de operação
- transformando o instanciado *tópicos*, direta ou indiretamente, criado a partir do *fio* principal em *demônios* – a máquina virtual Java termina assim que todos os restantes *tópicos* tem essa propriedade.

Tópicos em Java - 11

No entanto, a habitual rescisão de um *multithread* a aplicação é feita fazendo o primeiro ou principal *fio* aguardando o término de todos os *tópicos* que pode ter sido criado a partir dele.

Em Java, temos uma situação semelhante. O tipo de dados de referência `Fio` tem um método chamado `juntar` que, como é tradicional na programação concorrente, e numa perspectiva orientada a objetos, bloqueia a chamada *fio* até o referenciado *fio* termina.

Leitura sugerida

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