Múltiplas Linhas de Execução

Propriedades de Linha de Execução

Você provavelmente está familiarizado com a multitarefa: a capacidade de ter mais de um programa funcionando no que parece ser simultâneo. Por exemplo, você pode imprimir enquanto edita ou envia um fax. É claro que, a menos que você tenha uma máquina com vários processadores, o que ocorre realmente é que o sistema operacional está repartindo recursos para cada programa, dando a impressão de atividade paralela. Essa distribuição de recursos é possível pois, enquanto o usuário pode pensar que está mantendo o computador ocupado para, por exemplo, introduzir dados, a maior parte do tempo da CPU estará ociosa.

A multitarefa pode ser realizada de duas maneiras, se o sistema operacional interromper o programa sem consultar primeiro ou se os programas são interrompidos apenas quando estão querendo produzir controle. O primeiro caso é chamado de multitarefa preemptiva; o último é chamado de multitarefa cooperativa. O Windows 3.1, 95, 98 e NT são sistemas multitarefa cooperativos e os Unix são preemptivos.

Os programas de múltiplas linhas de execução ampliam a idéia da multitarefa levando-a um nível mais abaixo: os programas individuais parecerão realizar várias tarefas ao mesmo tempo. Normalmente, cada tarefa é chamada de linha de execução (Thread) que é a abreviação de linha de execução de controle. Diz-se que os programas que podem executar mais de uma linha de execução simultaneamente são multilinhas, ou têm múltiplas linhas de execução. Considere cada linha de execução como sendo executada em um contexto separado: os contextos fazem paracer que cada linha de execução possui sua própria CPU com registradores, memória e seu próprio código.

Então, qual é a diferença entre múltiplos processos e múltiplas linhas de execução? A diferença básica é que, enquanto cada processo tem um conjunto completo de variáveis próprias, as linhas de execução compartilham os mesmos dados. Isso parece um tanto arriscado e, na verdade, pode ser, conforme você verá posteriormente. Mas é necessário muito menos sobrecarga para criar e destruir linhas de execução individuais do que para ativar novos processos e esse é o motivo pelo qual todos os sistemas operacionais modernos oferecem suporte a múltiplas linhas de execução. Além disso, a comunicação entre processos é muito mais lenta e restritiva do que a comunicação entre linhas de execução.

As múltiplas linhas de execução são extremamente úteis na prática: por exemplo, um navegador deve tratar com vários hosts, abrir uma janela de correio eletrônico ou ver outra página, enquanto descarrega dados. A própria linguagem de programação Java usa uma linha de execução para fazer coleta de lixo em segundo plano evitando assim o problema de gerenciar memória! Os programas GUI têm uma linha de execução separada

para reunir eventos da interface com o usuário do ambiente operacional hospedeiro.

O que são linhas de Execução?

Vamos começar vendo um programa que não utiliza múltiplas linhas de execução e que como conseqüência, torna difícil para o usuário realizar várias tarefas com o programa. Após o dissecarmos, mostraremos então como é fácil fazer esse programa executar linhas de execução separadas. Esse programa anima uma bola que quica continuamente, movendo a bola, descobrindo se ela quica contra uma parede e depois redesenhando-a.

Assim que você clica no botão “Start”, o programa lança uma bola a partir do canto superior esquerdo da tela e a bola começa a quicar. A rotina de tratamento do botão “Start” chama o método bounce() da Classe Ball, que contém um laço executado 1000 vezes. Após cada movimento, chamamos o método estático sleep da classe Thread para fazer uma pausa na bola, por 15 segundos.

```
class Ball {
  public void bounce() {
    draw();
    for(int i = 1; i <= 1000; i++) {
      move();
      try {
        Thread.sleep(5);
      } catch(InterruptedException e) { }
    }
  }
}
```

A chamada Thread.sleep() não cria uma nova linha de execução – sleep é um método estático da classe Thread que coloca a linha de execução corrente em estado de suspensão.

O método sleep pode lançar uma exceção InterruptedException. Vamos discutir essa exceção e seu tratamento correto posteriormente; vamos apenas ignorá-la, por enquanto.

Se você executar o programa, poderá ver que a bola quica normalmente, mas assume o controle completo do aplicativo. Se você cansar da bola quicando antes que ela faça isso 1000 vezes e se der um clique no botão “Close”, a bola continuará a quicar. Você não pode interagir com o programa até que a bola tenha terminado de quicar.

Essa não é uma boa situação na teoria ou na prática e está se tornando cada vez mais problemático à medida que as redes se tornam mais importantes. Afinal, quando

você está lendo dados através de uma conexão de rede, é muito comum ficar preso em uma tarefa demorada que você gostaria realmente de interromper. Por exemplo, suponha que você descarregue uma imagem grande e decida, após ver uma parte dela, que não precisa ou não quer ver o restante; você certamente gostaria de poder dar um clique em um botão “Parar” ou “Voltar” para interromper o processo de download.

Vide o código Bounce.Java.

Usando Linhas de Execução para das uma chance às outras tarefas

Tornaremos nosso programa da bola que quica mais responsivo, executando o código que move a bola uma linha de execução separada.

Em nosso próximo programa, usamos duas linhas de execução : uma para a bola que quica e outra para a linha de execução principal, que cuida da interface com o usuário. Como cada linha de execução tem uma chance de ser executada, a linha de execução principal tem a oportunidade de notar quando você dá um clique no botão “Close”, enquanto a bola está quicando. Ele pode então processar a ação de “fechamento”.

Existe um procedimento simples para executar o código em uma linha de execução separada: insira o código no método run de uma classe derivada de Thread.

Para fazer nosso programa da bola que quica em uma linha de execução separada, precisamos apenas de derivar Ball e Thread e renomear o método bounce como run, como no código a seguir:

```
class Ball extends Thread {
  public run() {
    try {
      draw();
      for(int i = 1; i <= 1000; i++) {
        move();
        sleep(5);
      }
    } catch(InterruptedException e() {}
  }
}
```

Você pode ter notado que estamos capturando uma exceção chamada InterruptedException. Métodos como sleep e wait lançam essa exceção quando sua thread é interrompida, em função de outra linha de execução ter chamado o método interrupt. A interrupção de uma linha de execução é um modo muito drástico para obter a atenção da linha de execução, mesmo quando ela não está ativa. Normalmente, uma

linha de execução é interrompida para ser finalizada. Dessa maneira, nosso método run termina quando ocorre uma InterruptException.

Executando e iniciando linhas de execução

Quando você constrói um objeto derivado de Thread, o método run não é chamado automaticamente.

```
Ball b = new Ball(...); // não será executado ainda
```

Você deve chamar o método start em seu objeto, para realmente iniciar uma linha de execução.

```
b.start();
```

Na linguagem de programação Java, uma linha de execução precisa dizer às outras linhas de execução quando está ociosa, para que as outras linhas de execução tenham a chance de executar o código de seus procedimentos run. O modo normal de fazer isso é atracés do método estático sleep. O método run da classe Ball usa a chamada a sleep(5) para indicar que a linha de execução ficará ociosa pelos próximos 5 milissegundos. Após 5 milissegundos ela será iniciada novamente, mas nesse meio tempo, outras linhas de execução têm uma chance de fazer seu trabalho.

Do ponto de vista do projeto, parece estranho fazer a classe Ball estender a classe Thread. Uma bola (Ball) é um objeto que se move na tela e quica nos campos. Há uma plicação de regra de herança aqui? Uma bola é linha de execução? Na verdade, não. Aqui, estamos usando herança por razões estritamente técnicas. Para obter uma linha de execução que possa controler, você precisa de um objeto thread com um método run. Também poderíamos incluir esse método run na classe cujos métodos e campos de instância o método run utiliza. Portando, tornamos Ball uma classe filha de Thread. Além disso, isso torna o exemplo mais fácil de entender: todo objeto Bal que você constrói e inicia, executa seu método run em sua própria linha de execução. Posteriormente você verá como pode usar a interface Tunnable para não precisar extender a classe Thread.

Vide o código BounceThread.java.

Executando diversas linhas de execução

Execute o programa BounceThread. Agora, clique no botão “Start” novamente, enquanto a bola está se movendo. Clique nele mais algumas vezes. Você verá diversas bolas quicando. Cada bola se moverá 1000 vezes, até chagar à sua posição filan de repouso.

Esse exemplo demostra uma grande vantagem da arquitetura de linhas de execução na linguagem de programação Java. É muito fácil criar qualquer número de objetos autônomos, que parecem ser executados em paralelo.

Propriedades de linhas de execução

Estados de linha de execução

As linhas de execução podem estar em um dos quatro estados:

- nova
- passível de execução
- bloqueada
- morta

Cada um desses estados está explicado a seguir.

Linhas de execução novas

Quando você cria uma linha de execução com o operador new – por exemplo:

```
new Ball()
```

A linha de execução ainda não está em execução. Isso significa que ela está no estado novo. Quando uma linha de execução está no estado novo, o programa não começou a executar o código dentro dele. Uma certa preparação precisa ser feita, antes que uma linha de execução possa ser executada. Fazer a preparação e alocar os recursos necessários são tarefas do método start.

Linhas de execução passível de execução

Quando você chama o método start, a linha de execução é passível de execução (runnable). Uma linha de execução passível de execução pode não estar sendo executada ainda. Isso fica por conta do sistema operacional dar tempo para a linha de execução ser executada. Quando o código dentro da linha de execução começa a ser executada, ela está em execução. (Contudo, a documentação da plataforma Java não chama isso de um estado separado. Uma linha de execução em execução ainda está no estado de passível de execução.)

Como isso acontece fica por conta do sistema operacional. O pacote de linhas de execução da plataforma Java precisa trabalhar com o sistema operacional. Apenas o sistema operacional pode fornecer os ciclos de CPU. O chamado pacote conhecido como green threads, que é usado pela plataforma Java 1.x no Solaris, por exemplo, mantém uma linha em execução ativa até que a linha de execução de prioridade mais alta seja despertada e assuma o controle. Outros sistemas operacionais (como Windows 95 e NT) fornecem a cada linha passível de execução uma divisão do tempo para realizar sua tarefa. Quando essa divisão do tempo termina, o sistema operacional dá à outra linha uma oportunidade de trabalhar. Essa estratégia é mais sofisticada e faz melhor uso dos recursos de múltiplas linhas de execução da linguagem de programação Java. As versões

atuais da plataforma Java em Solaris podem ser configuradas de forma a permitir o uso das linhas de execução Solaris nativas, que também realizam divisão do tempo.

Lembre-se sempre de que uma linha de execução passível de execução pode ou não estar sendo executada em determinado momento. (É por isso que o estado é chamado de “passível de execução” e não “em execução”.)

Linhas de execução bloqueadas

Uma linha de execução entra no estado bloqueada quando uma das seguintes ações ocorre:

1. Alguém chama o método sleep() da linha de execução.
2. A linha de execução chama uma operação que está bloqueando entrada/Saída (I/O); isto é, uma operação que não retornará ao seu chamador até que uma ou mais operações de I/O estejam concluídas.
3. A linha de execução chama o método wait().
4. A linha de execução tenta bloquear um objeto que está bloqueado por outra execução.
5. Alguém chama o método suspend() da linha de execução. Entretanto, esse método foi desaprovado e você não deve chamá-lo em seu código.

Quando uma linha de execução está bloqueada (ou, é claro, quando ela morre), outra linha de execução é escalada para execução. Quando uma linha de execução bloqueada é reativada (por exemplo, porque ela ficou desativada pelo número de milissegundos exigido ou porque se está esperando a conclusão da operação de I/O), o agendador (scheduler) verifica se ela possui uma prioridade mais alta do que a linha que está em execução. Se assim for, ele apropria-se (fica a preempção) da linha de execução atual e escolhe uma nova linha de execução para executar. (Em uma máquina com múltiplos processadores, cada processador pode executar uma linha de execução e você pode ter várias linhas de execução executando em paralelo. Em tal máquina, uma linha de execução só é apropriada se uma linha de execução de prioridade mais alta se tornar passível de execução e não houver processador disponível para executá-la.)

Saindo de um Estado de Bloqueio

A linha de execução precisa sair do estado de bloqueio e voltar para o estado passível de execução, usando a rota oposta da que a colocou no estado bloqueado.

1. Se uma linha de execução foi colocada em estado “dormente”, o número especificado de milissegundos deve expirar.
2. Se uma linha de execução está esperando pela conclusão de uma operação de I/O, então a operação deve ter terminado.

3. Se uma linha de execução chamou wait, então outra linha de execução deve chamar notify ou notifyAll.

4. Se uma linha de execução está esperando por um bloqueio de objeto pertencente a outra linha de execução, então a outra linha precisa ter cedido o bloqueio.

5. Se uma linha de execução foi suspensa, então alguém deve chamar seu método resume. Entretanto, como o método suspend foi desaprovado, o método resume também foi, e você não deve chamá-lo em seu código.

Se você chamar em uma linha de execução, um método que seja incompatível com seu estado, então a máquina virtual lançará uma IllegalThreadStateException. Por exemplo, isso acontece quando você chama sleep em uma linha de execução que está atualmente bloqueada.

Linhas de Execução Mortas

Uma linha de execução torna-se morta por uma de duas razões:

- Ela morre de morte natural, pois o método run encerra normalmente.
- Ela morre abruptamente, pois uma exceção não capturada encerra o método run.

Em particular, é possível matar uma linha de execução chamando seu método stop. Esse método lança um objeto de erro ThreadDeath que mata a linha de execução. Entretanto, o método stop foi depreciado e você não deve chamá-lo em seu código.

Para descobrir se uma linha de execução está viva (isto é, se é passível de execução ou se está bloqueada), use o método isAlive. Esse método retorna true se a linha de execução passível de execução ou se estiver bloqueada, false se a linha de execução ainda for nova e não for passível de execução ou se estiver morta.

Interrompendo uma linha de execução

Uma linha de execução termina quando seu método run retorna. Como o método stop foi agora depreciado, não existe outro modo interno de terminar uma linha de execução. Isso significa que o método run de uma linha de execução precisa verificar de vez em quando se deve terminar. Para terminar, o método run simplesmente encerra.

```
public void run() {
  while(nenhum pedido para terminar && mais trabalho parafazer){
    realiza mais trabalho
  }
  sai do método run e termina a linha de execução
```

```
}
```

Entretanto, conforme você aprendeu, uma linha de execução não deve funcionar continuamente, mas “dormir” ou esperar de vez em quando, para dar às outras linhas uma chance de fazerem seu trabalho. Mas quando um linha está dormente, ela não pode verificar ativamente se deve terminar. É aí que o método interrupt entra em cena. Quando o médoto interrupt é chamado em um objeto thread que está correntemente bloqueado, a chamada de bloqueio(como sleep ou wait) é terminada por uma InterruptedExeption.

Não há requisito de linguagem dizendo que uma linha de execução que é interrompida deve terminar. A interrupção de uma linha de execução simplesmente chama sua atenção. A linha de execução interrompida pode decidir como vai reagir à interrupção, inserindo ações apropriadas na cláusula catch que trata da InterruptedException. Algumas linhas são tão importantes, que devem simplesmente ignorar sua interrupção, capturando a exceção e continuando. Mas, com muita freqüência, uma linha de execução simplesmente desejará interpretar uma interrupção como um pedido de encerramento. O método run de uma linha de execução assim tem a seguinte forma.

```
public void run() {
  try {
    ...
    while(mais trabalho parafazer){
      realiza mais trabalho
    }
  } catch(InterruptedException e) {
    a linha de execução foi interrompida durante sleep ou wait
  }
  sai do método run e termina a linha de execução
}
```

Entretanto, há um problema nesse esqueleto de código. Se o método interrupt fosse chamado enquanto a linha de execução estivesse dormente ou em estado de espera, então nenhuma InterruptedException teria sido gerada. A linha de execução precisa chamar o método interrupted para descobrir se foi interrompida recentemente.

```
    while(!interrupted() && mais trabalho para fazer){
      realiza mais trabalho
    }
```

Em particular, se uma linha de execução foi bloqueada enquanto esperava por I/O, estas operações não são terminadas com a chamada a interrupt. Quando a operação de bloqueio tiver retornado, você precisará chamar o método interrupted para descobrir se a linha de execução corrente foi interrompida.

Prioridades da linha de execução

Na linguagem de programação Java, toda linha de execução tem uma prioridade. Por definição, uma linha de execução herda a prioridade de sua linha de execução progenitora. Você pode aumentar ou diminuir a prioridade de qualquer linha de execução com o método setPriority. Você pode configurar a prioridade para qualquer valor entre MIN_PRIORITY (configurada como 1 na classe Thread) e MAX_PRIORITY (configurada como 10). NORM_PRIORITY (configurada como 5).

Quando o agendador (scheduler) de linhas de execução tem uma chance de escolher uma nova linha de execução, geralmente ele escolhe a linha de execução de prioridade mais alta que é passível de execução.

A linha de execução passível de execução de prioridade mais alta continua sendo executada até que:

- Ela ceda a vez, chamando o médoto yield, ou
- Ela cesse de ser passível de execução (ou morrendo ou entrando no estado bloqueado), ou
- Uma linha de execução de prioridade mais alta se torne passível de execução (devido à linha de execução de prioridade mais alta ter ficado dormente por tempo suficiente, sua operação de I/O terminou ou alguém chamou se método notify).

Então, o agendador (scheduler) seleciona uma nova linha para execução. A linha de execução de prioridade mais alta restante é escolhida dentre aquelas que são passíveis de execução.

O que acontece se há mais de uma linha passível de execução com a mesma prioridade (mais alta)? Uma das linhas de prioridade mais alta é escolhida. Fica totalmente por conta do agendador de linhas de execução como arbitrar entre as linhas de execução de mesma prioridade. A linguagem de programação Java não garante que todas as linhas de execução sejam tratadas de modo imparcial. É claro que seria desejável que todas as linhas de execução de mesma prioridade fossem atendidas uma por vez, para garantir que cada uma delas tenha uma chance de prosseguir. Mas, o mínimo, é teoricamente possível que, em algumas plataformas, um agendador de linhas de execução escolha uma linha de execução aleatória ou fique escolhendo a primeira disponível. Essa é uma deficiência da linguagem de programação Java e é difícil escrever programas com múltiplas linhas de execução que com certeza funcionem de modo idêntico em todas as máquinas virtuais.

Considere, por exemplo, uma chamada a yield. Ela talvez não tenha efeito em algumas implementações. Os níveis de todas as linhas passíveis de execução poderiam ser mapeados no mesmo nível de linhas de execução do hospedeiro. O agendador do computador hospedeiro poderia não fazer nenhum esforço de imparcialidade e

simplesmente continuar reativando alinha de execução que cede a vez, mesmo que outras linhas de execução queiram ser executadas. É uma boa idéia chamar sleep, em vez disso – pelo menos, você sabe que a linha de execução corrente não será escolhida outra vez, imediatamente.

Como uma questão prática, as máquinas virtuais têm várias (embora não necessariamente 10) prioridades e os agendadores de linha de execução fazem algum esforço para percorrer as linhas de execução de prioridade igual. Por exemplo, se você observar o número de linhas de execução de bola no exemplo de programa anterior, todas as bolas progrediram até o fim e pareceram ser executadas aproximadamente na mesma velocidade. É claro que isso não é garantido. Se você precisa garantir um plano de agendamento imparcial, então deve implementá-lo você mesmo.

Considere o seguinte exemplo de programa, que modifica o programa anterior para executar linhas de execução de um tipo de bolas.

Se você der um clique no botão “Start, cinco linhas de execução serão ativadas com prioridade normal, animando cinco bolas pretas. Se você der um clique no botão “Express”, então ativará cinco bolas vermelhas, cuja execução é executada em uma prioridade mais alta do que as bolas normais.

```
  public BounceExpress() {
    addButton(p, “Start”, new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
        for(int i = 0;i < 5;i++) {
          Ball b = new Ball(canvas, Color.black);
          b.serPriority(Thread.NORM_PRIORITY);
          b.start();
        }
      }
    });

    addButton(p, “Express”, new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
        for(int i = 0;i < 5;i++) {
          Ball b = new Ball(canvas, Color.red);
          b.serPriority(Thread.NORM_PRIORITY + 2);
          b.start();
        }
      }
    });
```

```
    ...
}
```

Experimente. Ative um conjunto de bolas normais e um conjunto de bolas expressas. Você notará que as bolas expressas parecem ser executadas mais rapidamente. Isso é apenas um resultado de sua prioridade mais alta e não porque as bolas vermelhas são executadas em velocidade mais alta. O código para mover as boas expressas é o mesmo do das bolas normais.

Aqui está o motivo pelo qual essa demonstração funciona: 5 milissegundos após a linha de execução expressa dormir, o agendador a ativa. Agora:

- O agendador avalia novamente as prioridades de todas as linhas de execução passíveis de execução.
- Ele verifica quais linhas de execução expressas têm a prioridade mais alta.

Uma das bolas expressas é colocada em execução novamente. Ela pode ser uma das que acabaram de ser ativadas ou talvez seja outra linha de execução expressa – você não tem meios de saber. As linhas de execução expressas são colocadas em execução novamente e somente quando todas estiverem dormindo é que o agendador dá às linhas de execução de prioridade mais baixa uma chance de serem executadas.

Note que as linhas de execução de prioridade mais baixa não teriam nenhuma chance de serem executadas, se as linhas de execução expressas tivessem chamado yield, em vez de sleep,

Mais uma vez, avisamos que esse programa funciona bem em NT e Solaris, mas que as especificação da linguagem de programação Java não garante que ele vá funcionar de forma idêntica em outras implementações.

Vide o código BounceExpress.java.

Linhas de Execução egoístas

Nossas linhas de execução de bola se comportaram bem e deram umas às outras uma chance de executar. Elas fizeram isso chamando o método sleep para esperar sua vez. O método sleep bloqueia as linhas de execução e dá às outras linhas de execução uma oportunidade de serem agendadas. Mesmo que uma linha de execução não queira dormir por qualquer período de tempo, ele pode chamar yield() quando não se importar de ser interrompida. Uma linha de execução sempre deve chamar yield ou sleep, quando estiver executando um laço longo, para garantir que ela não monopolize o sistema. Uma linha de execução que não siga essa regra é chamada de egoísta.

O programa a seguir mostra o que acontece quando uma linha de execução contém um laço pesado, um laço no qual ela realiza muito trabalho, sem dar uma chance a outras linhas de execução. Quando você dá um clique no botão “Selfish”, é lançada uma bola

azul cujo método run contém um laço pesado.

```
class SelfishBall extends Ball {
  public void run() {
    draw();
    for(int i = 1; i < 1000;i++) {
      move();
      long t = System.currentTimeMillis();
      while(System.currentTimeMillis() < t + 5);
    }
  }
}
```

O procedimento run durará cerca de 5 segundos antes de retornar, finalizando a linha de execução. Nesse tempo, ele nunca chama yield ou sleep.

O que acontece realmente quando você executa esse programa depende de seu sistema operacional e da escolha de implementação de linha de execução. Por exemplo, quando você executar esse programa sob o Solaris com a implementação “green thread”, em oposi;cão à implementação de “linhas de execução nativas”, descobrirá que a bola egoísta toma conta do aplicativo inteiro. Tente fechar o programa ou lançar outra bola; você terá dificuldade até de dar um clique de mouse no aplicativo. Entretanto, quando você executar o mesmo programa no Windows, nada adverso ocorrerá. As bolas azuis podem ser executadas em paralelo com as outras bolas.

O motivo dessa diferença comportamental é que o pacote de linha de execução do Windows realiza divisão de tempo. Ele interrompe periodicamente as linhas de execução no meio do caminho mesmo que elas não estejam cooperando. Quando uma linha de execução (mesmo uma linha de execução egoísta) é interrompida, o agendados ativa outra linha de execução – escolhida entre as linha de execução passíveis de execução de prioridade superior. A implementação de green threads do Solaris não realiza divisão de tempo, mas o pacote de linhas de execução nativas, sim. (Por que todo mundo não usa simplesmente as linhas de execução nativas? Até recentemente, o X11 e o Motif não tinham segurança para a linha de execução e usar linha de execução nativa podia travar o gerenciador de janelas.) Se você sabe que seu programa vai ser executado em uma máquina cujo sistema de linhas de execução realiza a divisão do tempo, então não precisa se preocupar em tornar suas linhas de execução corteses. Mas a questão da computação para Internet é que você geralmente não conhece os ambientes das pessoas que usarão seu programa. Portanto, você deve planejar para o pior caso e colocar chamadas a yield ou sleep em cada laço.

Vide o código BounceSelfish.java.

Grupo de linhas de Execução

Alguns programas contêm muitas linhas de execução. Então, se torna interessante classificá-las pela funcionalidade. Por exemplo, considere um navegador da Internet. Se muitas linhas de execução tiverem tentando adquirir imagens de um servidor e o usuário der um clique no botão “Parar” para interromper o carregamento da página corrente, então é útil ter um modo de interromper todas essas linhas de execução simultaneamente. A linguagem de programação Java permite que você construa o que ela chama de grupo de linha de execução, para que possa trabalhar simultaneamente com um grupo delas.

Você constrói um grupo de linha de execução com o construtor:

```
String nomeGrupo = ...;
ThreadGroup g = new ThreadGroup(nomeGrupo);
```

O argumento string do construtor de ThreadGroup identifica o grupo e deve ser exclusivo. Então, você insere a linha de execução no grupo, especificando os grupos de linhas de execução no construtor de linha de execução:

```
Thread t = new Thread(g, thread.Name);
```

Para saber se algumas linhas de execução de um grupo em particular ainda são passíveis de execução , use o método activeCount.

```
if(g.activeCount() = 0) {
  //Todos os threads no grupo g pararam
}
```

Para interromper todas as linhas de execução em um grupo, basta chamar interrupt no objeto grupo.

```
g.interrupt(); // interrompe todas as linhas de execução no
                  grupo g
```

Os grupos de linhas de execução podem ter subgrupos filhos. Por definição, um grupo de linhas recentemente criado se torna filho do grupo de linhas de execução corrente. Mas você também pode nomear explicitamente o grupo progenitor no construtor (veja as notas de API). Métodos como activeCount e interrupt se referem a todas as linhas de execução de seu grupo e a todos os grupos filhos.

Sincronismo

Na maioria dos aplicativos com múltiplas linhas de execução duas ou mais linhas precisam compartilhar o acesso aos mesmos objetos. O que acontece se duas linhas de execução têm acesso ao mesmo objeto e cada uma chama um método que modifica o estado do objeto? Conforma você poderia imaginar, as linhas de execução pisam nos pés umas das outras. Dependendo da ordem em que os dados foram acessados o resultado

pode ser, objetos danificados. Tal situação é freqüentemente chamada de condição de corrida (race condition).

Comunicação de linhas de execução sem sincronismo

Para evitar o acesso simultâneo de um objeto compartilhado por diversas linhas de execução, você deve aprender a sincronizar o acesso.

No próximo programa de teste, simulamos um banco com 10 contas. Geramos, aleatoriamente, transações que movimentam dinheiro entre essas contas. Existem 10 linhas de execução, uma para cada conta. Cada transação movimenta uma quantidade aleatória de dinheiro da conta atendida pela linha de execução, para outra conta aleatória.

O código da simulação é simples. Temos a Classe Bank, com o método transfer. Esse método transfere alguma quantidade de dinheiro de uma conta para outra. Se a conta de origem não tiver dinheiro suficiente, então a chamada simplesmente retorna. Aqui está o código do método transfer da classe Bank.

```
// ATENÇÃO: inseguro quando chamado a partir de diversas linhas
//          de execução
public void transfer(int from, int to, double amount) {
  if(accounts[from] < amount)
    return;

  accounts[from] -= amount;
  accounts[to] += amount;
  ntransacts++;

  if(ntransacts % NTEST == 0)
    test();
}
```

Aqui está o código da classe TransferThread. Seu método run se mantém movimentando dinheiro de uma conta de banco fixa. Em cada interação, o método run escolhe uma conta de destino aleatório e um valor aleatório, chama transfer no objeto bank e depois dorme.

```
class TransferThread extends Thread {
  private Bank bank;
  private int fromAccount;
  private int maxAmount;

  public TransferThread(Bank b, int from, int max) {
```

```
      bank = b;
      fromAccount = from;
      maxAmount = max;
    }

    public void run() {
      try {
        while(!interrupted()) {
          int toAccount = (int)(bank.size() * Math.random());
          int amount = (int)(maxAmount * Math.random());
          bank.transfer(fromAccount, toAccount, amount);
          sleep(1);
        }
      } catch(InterruptedException e) {}
    }
  }
```

Quando essa simulação é executada, não sabemos quanto dinheiro existe em cada conta bancária no momento. Mas sabemos que a quantidade total de dinheiro em todas as contas deve permanecer inalterada, pois apenas movemos dinheiro de uma conta para outra.

A cada 100.000 transações, o método transfer chama um método test que recalcula o total e o imprime.

Esse programa nunca termina. Basta pressionar Ctrl+C para interromper o programa.

Aqui está um resultado típico:

```
Transdactions:10000 Sum 100000
Transdactions:20000 Sum 100000
Transdactions:30000 Sum 100000
Transdactions:40000 Sum 100000
Transdactions:50000 Sum 100000
Transdactions:60000 Sum 100000
Transdactions:70000 Sum 100000
Transdactions:80000 Sum 100000
Transdactions:90000 Sum 100000
Transdactions:100000 Sum 100000
Transdactions:110000 Sum 100000
Transdactions:120000 Sum 100000
```

```
Transdactions:130000 Sum 94792
Transdactions:140000 Sum 94792
Transdactions:150000 Sum 94792
```

Conforme você pode ver, algo está errado. Para tantas transações, o saldo bancário permanece em 100000, que é o total correto das 10 contas de 10000 cada. Mas após algum tempo, o saldo muda ligeiramente. Quando você executar esse programa, poderá ver que erros acontecem rapidamente ou que pode demorar bastante para que o saldo seja corrompido. Essa situação não inspira confiança e provavelmente não desejaríamos depositar nesse banco nosso dinheiro duramente ganho.

Vide o código UnsynchBankTest.java.

Sincronizando o acesso a recursos compartilhados

Na seção anterior, executamos um programa em que várias linhas de execução atualizavam saldos de conta bancária. Após algum tempo, apareceram erros e certa quantidade de dinheiro ou foi perdida ou espontaneamente criada. Esse problema ocorre quando duas linhas de execução estão tentando atualizar uma conta simultaneamente. Suponha que duas linhas de execução executem simultaneamente a seguinte instrução:

```
accounts[to] += amount;
```

O problema é que essas não são operações atômicas. A instrução poderia ser processada como segue:

1. Carrega accounts[to] em um registradores.
2. Soma amount.
3. Move o resultado de volta para accounts[to].

Agora, suponha que a primeira linha de execução execute os passos 1 e 2, e depois seja interrompida. Suponha que a segunda linha de execução seja acordada e atualizada a mesma entrada no array account. Logo, a primeira linha de execução será acordada e finalizada no passo 3.

Essa ação elimina a modificação da outra linha de execução. Como resultado, o total não está mais correto.

Qual é a chance desse erro estar ocorrendo? Se todas as linhas de execução estiverem sendo executadas com a mesma prioridade ela será muito baixa, pois cada linha de execução realiza tão pouco trabalho antes de entrar em estado de suspensão que é improvável que o agendador faça preempção dela. Depois de algumas tentativas, verificamos que poderíamos aumentar a probabilidade de dados, atribuindo à metade das linhas de execução de transferência uma prioridade mais alta do que à outra metade. Quando uma linha de execução de transferência de prioridade mais alta acorda do seu sono, ele fará preempção de uma linha de execução de transferência de prioridade mais

baixa.

O problema é que o trabalho do método transfer pode ser interrompido no meio. Se pudéssemos garantir que o método fosse executado até o fim, antes que a linha de execução perdesse o controle, então o estado do objeto conta bancária não seria o de danificado.

Muitas bibliotecas de linhas de execução obrigam o programador a se preocupar com semáforos e seções críticas para obter acesso ininterrupto a um recurso. Isso é suficiente para a programação procedural, mas dificilmente é orientado a objetos. A linguagem de programação Java tem um mecanismo melhor, inspirado pelos monitores, inventados por Tony Hoare.

Você simplesmente identifica toda a operação que não deve ser interrompida com synchronized, como seque:

```
public synchronized void transfer(int from, int to, double
                  amount) {
  if(accounts[from] < amount)
    return;

  accounts[from] -= amount;
  accounts[to] += amount;
  ntransacts++;

  if(ntransacts % NTEST == 0)
    test();
}
```

Quando uma linha de execução chama um método synchronized, é garantido que o método todo concluirá sua execução, antes que outro possa executar qualquer método synchronized no mesmo objeto. Quando uma linha de execução chama transfer e depois outra linha também chama transfer, a segunda linha não pode continuar. Em vez disso, ela é desastrada e deve esperar que a primeira linha de execução termine de executar o método transfer.

Bloqueio de Objetos

Quando uma linha de execução chama um método synchronized, o objeto se torna “bloqueado”. Considere cada objeto como tendo uma chave da porta da frente. Inicialmente, a chave fica na soleira da porta. Quando uma linha de execução entra em um método synchronized, ela pega a chave, entra no objeto e a tranca na porta por dentro. Quando outra linha de execução tenta chamar um método synchronized no mesmo objeto, ela não pode abrir a porta. Então, ela procura a chave e não consegue

encontrá-la; portanto, pára de executar. Finalmente, a primeira linha de execução termina seu método synchronized, deixa o objeto e deposita a chave novamente na soleira da porta.

Periodicamente, o agendador de linhas de execução ativa as linhas de execução que estão esperando pela chave, usando suas regras de ativação normais, que já discutimos. Quando uma das linhas de execução que deseja usar o objeto é novamente executada, ela verifica se o objeto ainda está bloqueado. Se não estiver, ela fica sendo a próxima a obter acesso exclusivo ao objeto.

Entretanto, outras linhas de execução estão livres para chamar métodos unsynchronized em um objeto bloqueado. Por exemplo, o método size da classe BankAccount não precisa se synchronized.

Quando uma linha de execução deixa um método synchronized lançando uma exceção , ela ainda renuncia ao bloqueio do objeto. Isso é bom – você não desejaria que uma linha de execução mantivesse o bloqueio de um objeto depois que tivesse terminado o método synchronized.

Se uma linha de execução possui o bloqueio de um objeto e chama outro método synchronized do mesmo objeto, então ela tem acesso garantido automaticamente. A linha de execução só libera o bloqueio quando termina o último método synchronized.

Uma linha de execução pode possuir os bloqueios de vários objetos simultaneamente, simplesmente chamando um método synchronized em um objeto, enquanto executa um método synchronized de outro objeto. Mas, é claro, o bloqueio de um objeto só pode pertencer a um thread em um certo momento.

Os Métodos wait e notify

Vamos refinar nossa simulação do banco. Não queremos transferir dinheiro de uma conta que não tenha fundos. Note que não podemos usar código como o seguinte:

```
if(bank.getBalance(from) >= amount)
   bank.transfer(from. to, amount);
```

É totalmente possível que a linha de execução corrente seja desativada entre a saída com êxito do teste e a chamada a transfer.

```
if(bank.getBalance(from) >= amount)
   //o thread poderia ser desativado neste ponto
   bank.transfer(from, to, amount);
```

Quando a linha de execução estiver novamente em execução, o saldo da conta poderá ter caído abaixo do valor de retirada. Você deve certificar-se de que a linha de execução não possa ser interrompida entre o teste e a inserção. Você faz isso colocando o teste e a ação de transferência dentro do mesmo método synchronized:

```
public synchronized void transfer(int from, int to, int amount)
                    {
  while(accounts[from] < amount) {
    // espera
  }
  // transfere fundos
}
```

Agora, o que fazemos quando não há dinheiro suficiente na conta? Esperamos até que alguma outra linha de execução tenha incluído os fundos. Mas o método transfer é synchronized. Essa linha de execução acabou de obter acesso exclusivo ao objetivo bank; portanto, nenhuma outra linha de execução tem uma chance de fazer um depósito. Um segundo recurso dos métodos synchronized trata dessa situação. Você usa o método wait na classe de linha de execução, se precisar esperar dentro de um método synchronized.

Quando wait é chamado dentro de um método synchronized, a linha de execução corrente é bloqueada e libera o bloqueio de objeto. Isso permite que outra linha de execução, esperamos, aumente o saldo da conta. Note que o método wait é um método da classe Object e não da classe Thread. Ao se chamar wait, o objeto Bank bloqueia a linha de execução e se desbloqueia.

Há uma diferença fundamental entre uma linha de execução que está esperando para entrar em um método synchronized e uma linha de execução que chamou wait. Quando uma linha de execução chama o método wait, ela entra em uma lista de espera. Até que a linha de execução seja removida da lista de espera, o agendador a ignora e ela não tem uma chance de continuar a ser executada.

Para se remover a linha de execução da lista de espera, alguma outra linha de execução deve chamar o método notify ou notifyAll no mesmo objeto. O médoto notify remove uma linha de execução selecionada arbitrariamente da lista de espera. O método notifyAll remove todas elas. Quando as linhas de execução forem removidas da lista de espera, o agendador eventualmente as ativará novamente. Nesse momento, elas tentarão entrar de novo no objeto. Assim que o bloqueio de objetos estiver disponível, uma delas voltará a entrar no objeto e continuará a partir de onde estava, após a chamada de wait.

É de fundamental importância que alguma linha de execução chame o método notify ou notifyAll periodicamente. Quando uma linha de execução chama wait ela não tem meios de desbloquear-se. Ela conta com as outras linhas de execução. Se nenhuma delas desbloquear a linha de execução que está esperando, ela nunca mais será executada. Isso pode levar a situações de bloqueio mutuamente exclusivo (deadlock) desagradáveis. Se todas as outras linhas de execução estiverem bloqueadas e a última linha de execução ativa chamar wait sem desbloquear uma das outras, então ela também será bloqueada. Não resta nenhuma linha de execução para desbloquear as outras, e o

programa trava. As linhas de execução que estão esperando não são reativadas automaticamente, quando nenhuma outra linha de execução está trabalhando no objeto. Vamos discutir os bloqueios mutuamente exclusivos mais adiante.

Na prática, é perigoso chamar notify, pois você não tem controle sobre qual linha de execução é desbloqueada. Se a linha de execução errada for desbloqueara, talvez ela não possa prosseguir. Recomendamos simplesmente que você use o método notifyAll e que todas as linhas de execução sejam desbloqueadas.

Quando você deve chamar notifyAll? A regra geral é chamar notifyAll quando o estado de um objeto mudar de um modo que seria vantajoso para as linhas de execução que estão esperando. Por exemplo, quando um saldo de conta muda, as linhas de execução que estão esperando devem ter outra chance de inspecionar o saldo. Em nosso exemplo, chamaremos notifyAll quando tivermos terminado com a transferência de fundos.

```
public synchronized void transfer(int from, int to, int amount)
                  {
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   notifyAll();
   ...
}
```

Essa notificação dá às linhas de execução que estão esperando uma chance de serem executadas novamente. Uma linha de execução que esteja esperando por um saldo mais alto tem então uma chance de verificar o saldo novamente. Se o saldo for suficiente, a linha de execução realiza a transferência. Se não, ela chama wait novamente.

Note que a chamada a notifyAll não ativa imediatamente uma linha de execução que esteja esperando. Ela apenas desbloqueia as linhas de execução que estão esperando para que elas possam concorrer para a entrada no objeto, depois que a linha de execução corrente river terminado o método synchronized.

Se você executar o exemplo de programa com a versão sincronizada do método transfer, notará que nada jamais dá errado. O saldo total permanece em 100000 para sempre. (Novamente, você precisa pressionar Crtl+C para terminar o programa.)

Você também notará que o programa é executado um pouco mais lentamente – esse é o preço pago pela contabilidade envolvida no mecanismo de sincronismo.

Vide o código SynchBankTest.java.

Aqui está um resumo de como o mecanismo de sincronismo funciona.

1. Para chamar um método synchronized, o parâmetro implícito não pode ser bloqueado. Chamar o método bloqueia o objeto. Retornar da chamada desbloqueia o objeto parâmetro implícito. Assim, apenas uma linha de execução por vez pode executar métodos synchronized em um objeto específico.

2. Quando uma linha de execução executa uma chamada a wait, ela renuncia ao bloqueio do objeto e entra em uma lista de espera.

3. Para remover uma linha de execução da lista de espera, alguma outra linha de execução deve fazer uma chamada a notifyAll ou notify, no mesmo objeto.

As regras de agendamento são incontestavelmente complexas, mas é realmente muito simples colocá-las em prática. Basta seguir estas quatro regras:

1. Se duas ou mais linhas de execução modificam um objeto, declare os métodos que realizam as modificações como synchronized. Os métodos somente de leitura que são afetados por modificações de objeto também devem ser synchronized.

2. Se uma linha de execução precisa esperar que o estado de um objeto mude, ela deve esperar dentro do objeto, e não fora, entrando em um método synchronized e chamando wait.

3. Quando um método muda o estado de um objeto, ele deve chamar notifyAll. Isso dá às linhas de execução que estão esperando uma chance de ver se as circunstâncias mudaram.

4. Lembre-se de que wait e notify/notifyAll são métodos da classe Object e não da classe Thread. Confira se suas chamadas a wait correspondem a uma notificação no mesmo objeto.

Impasses

O recurso de sincronismo na linguagem de programação Java é conveniente e poderoso, mas não pode resolver todos os problemas que surgem no uso de múltiplas linhas de execução. Considere a seguinte situação:

Conta 1: 2000

Conta 2: 3000

A linha de execução 1 transfere 3000 da Conta 1 para a Conta 2

A linha de execução 2 transfere 4000 da Conta 2 para a Conta 1

As Linhas de execução 1 e 2 estão claramente bloqueadas. Nenhuma delas pode prosseguir, pois os saldos nas contas 1 e 2 são insuficientes.

É possível que todas as 10 linhas de execução sejam bloqueadas porque cada uma está esperando por mais dinheiro? Tal situação é chamada de bloqueio mutuamente

exclusivo (deadlock).

Em nosso programa, um bloqueio mutuamente exclusivo não poderia ocorrer por um motivo simples. Cada valor de transferência é de, no máximo, 10000. Como existem 10 contas e um total de 100000 nelas, pelo menos uma das contas deverá ter mais de 10000 em certo momento. A linha de execução que está movendo dinheiro dessa conte pode, portanto, prosseguir.

Mas se você mudas o método run das linhas de execução para remover o limite de transação de 10000, bloqueios mutuamente exclusivos poderão ocorrer rapidamente. Experimente. Construa cada TransferThread com um valor de maxAmount igual a 14000 e execute o programa. O programa será executado por algum tempo e depois parará.

Infelizmente, não existe nada na linguagem de programação Java para evitar ou solucionar esses bloqueios mutuamente exclusivos. Você precisa fazer o projeto de suas linhas de execução de forma a garantir que uma situação de bloqueio mutuamente exclusivo não possa ocorrer. Você precisa analisar seu programa e garantir que toda linha de execução bloqueada seja finalmente notificada e que pelo menos uma delas sempre possa prosseguir.

Por que os métodos stop e suspend foram depreciados

A plataforma Java 1.0 definiu um método stop que simplesmente encerra uma linha de execução e um método suspend, que bloqueava uma linha de execução atá que outra linha de execução chamasse resume. Esses dois métodos foram depreciados na plataforma Java 2. O método stop é inerentemente inseguro e a experiência tem mostrado que o método suspend freqüentemente leva a bloqueios mutuamente exclusivos.

Vamos ver primeiro o método stop. Quando uma linha de execução é interrompida, ela abandona imediatamente os bloqueios em todos os objetos que tiver bloqueado. Isso pode deixar objetos em um estado inconsistente. Por exemplo, suponha que uma TransferThread seja interrompida no meio da movimentação de dinheiro de uma conta para outra, após a retirada e antes do depósito. Agora o objeto bank está danificado.

Pelo mesmo motivo, qualquer exceção não capturada em um método synchronized pode fazer o método terminar prematuramente e levar a objetos danificados.

Quando uma linha de execução quer interromper outra, ela não tem meios de saber quando o método stop é seguro e quando ele leva a objetos danificados. Portanto, o método foi depreciado.

Se você precisar interromper uma linha de execução com segurança, pode fazer a linha de execução verificar uma variável periodicamente, a qual indica se uma parada foi solicitada.

```
public class MyThread extends Thread {
```

```
    private boolean stopRequested;

    public void run() {
      while(!stopRequested && mais trabalho a realizar ) {
        realiza mais trabalho
      }
    }

    public void requestStop() {
      stopRequested = true;
    }
  }
```

Esse código deixa o método run no controle ao terminar e fica por conta desse método garantir que nenhum objeto seja deixado em um estado danificado.

O teste da variável stopRequested no laço principal de uma linha de execução funciona vem, exceto se a linha de execução estiver bloqueada. Nesse caso, a linha de execução só terminará depois que for desbloqueada. Você pode forçar a saída de uma linha de execução de um estado bloqueado, interrompendo-a. Assim, você deve definir que o método requestStop chame interrupt:

```
  public void requestStop {
    stopRequested = true;
    interrupt();
  }
```

Você pode testar a variável stopRequested na cláusula catch de InterruptedException. Por exemplo,

```
  try {
    wait();
  } catch(InterruptedException e) {
    if(stopRequested)
      return; // sai do método run
  }
```

Na verdade, muitos programadores consideram que o único motivo de interromper uma linha de execução é para pará-la, Então, você não precisa testar a variável stopRequested – basta sair do método run que não perceber uma interrupção.

Ao contrário de stop, suspend não danificará objetos. Entretanto, se você suspender uma linha de execução que possui um bloqueio em um objeto, este ficará indisponível até que a linha de execução seja retomada. Se a linha de execução que chama o método

suspend tentar adquirir o bloqueio para o mesmo objeto, antes de chamar resume, o programa entrará em um bloqueio mutuamente exclusivo: a linha de execução suspensa esperará para ser retomada e a linha que causou a suspensão esperará que o objeto seja desbloqueado.

Essa situação ocorre freqüentemente em interfaces do usuário gráficas. Suponha que tenhamos uma simulação gráfica de nosso banco. Temos um botão chamado “Pausa” (pause), que suspende as linhas de execução de transferência, e um botão chamado “Continuar” (resume), que os retoma.

```
public void actionPerdormed(ActionEvent event) {
  Object source = event.getSource();

  if(source == pauseButton)
    for(int i = 0;i < threads.length; i++)
      threads[i].suspend(); // não faça isto
  else if(source == resumeButton)
    for(int i = 0;i < threads.length; i++)
      threads[i].resume(); // não faça isto
}
```

Um método paintComponent pinta um gráfico de cada conta, chamando o método bank.getAccount, e esse método é synchronized.

As duas ações de botão e a repintura ocorrem na mesma linha de execução, a linha de execução de envio de eventos.

Agora, considere o seguinte cenário:

1. Uma linha de execução de transferência adquire o bloqueio sobre o objeto bank.
2. O usuário dá um clique no botão “Pausa”.
3. Todas as linhas de execução de transferência são suspendas; uma delas ainda detém o bloqueio sobre o objeto bank.
4. Por algum motivo, o gráfico da conta precisa ser refeito.
5. O método paintComponent chama o método synchronized bank.getBalance.

Agora o programa está travado.

A linha de execução de envio de eventos não pode prosseguir, pois o objeto bank está vloqueado por uma das linhas de execução suspensas. Assim, o usuário não pode dar um clique no botão “Continuar” e as linhas de execução jamais serão retomadas.

Se você quiser suspender a linha de execução com seguranç, deve introduzir uma

variável suspendRequested e testá-la no laço principal de seu método run. Quando você verificar que a variável foi ativada, continue esperando até que ela se torne novamente disponível. Entretanto, você não quer implementar uma espera ocupada:

```
while(suspendRequested)
  sleep(1); //NÃO
```

Em vez disso, chame wait e defina um método requestResume para chamar notify. Então, a linha de execução permanece bloqueada até que seja retomada.

```
class Mythread extends Thread {
  private boolean suspendRequested;

  public void requestSuspend() {
    suspendRequested = true;
  }

  private synchronized void checkSuspended() throws
                  InterruptedException {
    while(suspendRequested)
      wait();
  }

  public synchronized void requestResume() {
    suspendRequested = false;
    notify();
  }

  public void run() {
    while(mais trabalho a realizar) {
      checkSuspended();
      realiza mais trabalho
    }
  }
}
```

Por que essa estratégia resolve o problema de interdaces do usuário travadas? O método run chama checkSuspended em um momento ocioso, quando ele não está bloqueando nenhum outro objeto. Portando, a linha de execução só bloqueia a si mesma, quando for seguro fazer isso.

É claro que evitar o método Thread.suspend não evita bloqueios mutuamente

exclusivos automáticamente. Se uma linha de execução chamar wait, ela também poderia não ser ativada. Mas há uma diferença básica. Uma linha de execução pode controlar quando faz a chamada a wait. Mas o método Thread.suspend pode ser chamado externamente em uma linha de execução, a qualquer momento, sem o consentimento da linha de execução. O mesmo vale para o método Thread.stop. Por isso, esses dois métodos foram depreciados.

A interface Runnable

Quando você precisar usar múltiplas linhas de execução em uma classe que já é derivada de uma classe diferente de Thread, pode fazer a classe implementar a interface Runnable. Quando você tiver derivado de Thread, coloque o código que precisar ser executado no método run. Por exemplo,

```
class Animation extends JApplet implements Runnable {
  public void run() {
  // a ação da linha de execução fica aqui
  }
}
```

Você ainda precisa fazer um objeto linha de execução ativar a linha de execução. Forneça a essa linha de execução uma referência ao objeto Runnable em seu construtor. A linha de execução então chama o método run desse objeto.

Essa chamada normalmente é feita no método start de uma applet, como no exemplo a seguir:

```
class Animation extends Japplet implements Runnable {
  private Thread runner;

  public void start() {
    if(runner == null) {
      runner = new Thread(this);
      runner.start();
    }
  }

  public void run() {
    //a ação da linha de execução fica aqui
  }
  ...
}
```

Nesse caso, o argumento this do construtor Thread especifica que o objeto cujo método run deve ser chamado quando a linha de execução for executada é uma instância do objeto Animation.

Não seria mais fácil se apenas definíssemos outra classe de Thread e a ativássemos no applet?

```
class AnimationThread extends Thread {
  public void run() {
    //a ação da linha de execução fica aqui
  }
}

class Animation extends Japplet {
  private Thread runner;

  public void start() {
    if(runner == null) {
      runner = new AnimationThread();
      runner.start();
    }
  }
  ...
}
```

Na verdade, isso seria claro e simples. Entretanto, se o método run precisar ter acesso aos dados privativos de um applet, então faz sentido manter o método run com o applet e usar a interface Runnable em seu lugar.