Introdução à Programação Concorrente para a Internet

Dilma Menezes da Silva

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo
dilma@ime.usp.br
http://www.ime.usp.br/~dilma/

Resumo

A popularização de novas categorias de aplicações, em particular a disponibilidade de programas que executam a partir de *browsers*, muitas vezes obriga o desenvolvedor a lidar com aspectos de concorrência antes restritos a comunidades bem específicas. Os conceitos básicos da área em geral são apresentados como parte de um curso de sistema operacional ou banco de dados, mas neste mini-curso enfocamos o assunto tendo como motivação seu uso para o desenvolvimento de programas concorrentes que executam na internet como Applets Java.

Abstract

The increasing popularity of new categories of applications, particularly applications in the internet that are accessed by browsers, many times requires that the developer deal with concurrency aspects so far usually restricted to specific computer science communities. The basic concepts in the area are usually presented as part of an operating systems or data base course, but in this short course we focus the subject through its use in the development of concurrent programs that execute as Java Applets.

1. Introdução e motivação

A área de programação concorrente vem sendo explorada há várias décadas, sob vários enfoques. Em muitos desses enfoques o trabalho foi bem sucedido:

problemas básicos foram definidos e solucionados, aspectos teóricos caracterizados e estudados, aspectos de implementação largamente experimentados e analisados. Em geral, o impacto dos resultados obtidos por esses estudos estavam confinados a comunidades específicas, estando o desenvolvedor de sistemas razoavelmente "protegido" dos desafios e problemas da área, a menos que desenvolvesse software básico (sistemas operacionais, gerenciadores de bancos de dados) ou atuasse diretamente em áreas específicas como computação paralela ou sistemas de tempo real. Mas novas categorias de aplicações, com crescente impacto na área de computação, muitas vezes obrigam o desenvolvedor a enfrentar diretamente as questões relativas à concorrência. Conceitos, mecanismos e implementações antes restritos a livros acadêmicos passaram a exercer um papel importante em livros técnicos direcionados aos desevolvedores de software. Vários textos recentes [Magee-Kramer 1999, Butenhof 1997, Lea 1997] apresentam formas de lidar com concorrência do ponto de vista de tecnologias específicas, enquanto outros textos abordam a questão de forma totalmente conceitual, sem abordar diretamente tecnologias recentes e bem difundidas atualmente [Milner 1995, Andrews 1991]. O objetivo deste texto é aliar os dois extremos, isto é, apresentar uma introdução à área de concorrência que contemple tanto seu uso imediato no desenvolvimento de aplicativos (em especial, aplicativos para a internet) quanto a formação conceitual básica comum às várias vertentes da Ciência da Computação relacionadas a programação concorrente.

Muitos dos conceitos e problemas apresentados aqui fazem parte dos currículos usuais de graduação na área de Ciência da Computação, aparecendo em geral como um item entre vários a serem abordados nas disciplinas de sistemas operacionais, banco de dados, computação paralela ou computação distribuída. Este mini-curso se diferencia por enfocar os problemas fora dos contextos e requisitos específicos destas disciplinas. O mini-curso foi elaborado de forma a contemplar um público que se encaixe de alguma forma nas seguintes premissas:

- O "aluno" está interessado no desenvolvimento de aplicativos para a internet (isto é, programas que possam ser utilizados a partir de *browsers)* e disposto a aprender conceitos e mecanismos úteis para melhorar a qualidade de tais aplicativos;
- O "aluno" já estudou conceitos de concorrência (por exemplo, no contexto de sistemas operacionais) e gostaria de aplicá-los no desenvolvimento de aplicativos gerais;
- O "aluno" está interessado em entender o que há por trás do pacote de Threads fornecido pela linguagem de programação Java, ou qual é seu propósito geral;
- "aluno" gosta de desafios, e está disposto a lidar com programas onde a dificuldade de depuração e análise é bem maior do que a usual.

Além do conteúdo idealizado para o público alvo acima descrito, foi incluído no texto material introdutório sobre modelagem de sistemas. Esta parte é apresentada de forma relativamente independente do restante do material, e foi incluída para exemplificar um caminho que sirva de fundação para uma abordagem disciplinada e metódica de programação concorrente.

Este mini-curso assume que o aluno tem uma maturidade em programação no nível de segundo anistas de um curso de graduação em computação, e que ele se sente confortável com a utilização de estruturas de dados básicas como pilha, fila e listas ligadas. Na quase totalidade dos exemplos é utilizada a linguagem Java; como não é assumido que o aluno domine esta linguagem, os programas são documentados pesadamente, de forma a facilitar a compreensão. Todos os exemplos estão disponíveis na *home page* deste texto (http://www.ime.usp.br/~dilma/jai99/). Neste sítio[1] você também encontrará atualizações do texto e um endereço para envio de correções, sugestões ou dúvidas. Mais importante: os exemplos lá apresentados são mais gerais e ligados à programação concorrente para internet de que os aqui desenvolvidos, em função do espaço que ocuparia a inclusão, com documentação detalhada, de programas completos cuja funcionalidade fosse um pouco mais interessante.

O restante desta seção define informalmente a noção de concorrência e descreve o tipo de aplicação que motiva o restante do nosso texto A seção 2 apresenta os conceitos básicos e definições da área. Na seção 3 abordamos os aspectos práticos da programação concorrente, focalizando no suporte oferecido pela linguagem Java. A seção 4 lida, superficialmente, com os problemas de corretude e desempenho. A seção 5 apresenta aspectos sobre applets, concorrência e distribuição, mas as aplicações são bem simples (exemplos mais interessantes estão disponíveis na página deste texto). A última seção do texto descreve brevemente aspectos de modelagem de sistemas concorrentes.

1.1. O que é concorrência ?

Sistemas concorrentes devem lidar com atividades separadas que estão progredindo ao mesmo tempo. Informalmente, dizemos que duas atividades são *concorrentes* se em algum momento ambas tenham sido iniciadas mas não terminadas, ou seja, estão em algum ponto intermediário da execução. *Programação concorrente* é a atividade de construir um programa contendo múltiplas atividades que progridem em paralelo. Atividades podem progredir em paralelo de duas maneiras:(1) sendo realmente executadas em paralelo, cada uma de posse de um processador diferente ou (2) tendo o processador se revezando entre as atividades, de forma que cada atividade possa fazer uso do processador

[1] Tradução para “site”.

durante um período de tempo e depois espere sua próxima chance de avançar na computação.

A programação concorrente lida com atividades que, embora executem separadamente, estejam relacionadas de alguma forma. Em alguns casos, as atividades estão relacionadas por terem um objetivo comum, por estarem *cooperando* para solucionar um problema. Em outros casos, a relação entre as atividades é mais frouxa, estas podendo ser totalmente independentes em seus propósitos, mas precisando compartilhar os recursos pelos quais competem por estarem executando em um ambiente comum.

1.2. Por que concorrência é relevante ?

Em muitos cenários, concorrência não pode ser evitada porque é uma parte inerente do sistema a ser desenvolvido, pois o sistema manuseia estímulos que podem ocorrer simultaneamente no mundo externo ao sistema computadorizado. Por exemplo:

- vários clientes de um banco podem solicitar transações bancárias ao mesmo tempo;
- vários visitantes de uma livraria virtual (isto é, visitantes de sua página na internet) podem ao mesmo tempo solicitar buscas de informação ou compras de livros;
- um sistema de monitoramento de segurança de um prédio deve lidar ao mesmo tempo com as várias imagens e demais informações sendo coletadas.

Mas os requisitos de concorrência não aparecem apenas nas aplicações que lidam com estímulos externos inerentemente concorrentes. Com o avanço da tecnologia das últimas duas décadas, é cada vez mais viável a implementação de sistemas de forma distribuída sobre uma rede de computadores. Implementações distribuídas, por oferecerem em geral várias portas de entrada para o uso do sistema, são obrigadas a lidar com a concorrência das atividades que executam simultaneamente nos diferentes nós da rede. Algumas vezes as razões que levam à opção por uma implementação distribuída ao invés de centralizada servem como argumentos para que se empregue concorrência mesmo em uma implementação centralizada do sistema.

Outra motivação para o uso de concorrência é a busca por soluções que permitam que várias partes de um mesmo problema sejam atacadas em paralelo. Com este enfoque, se a plataforma de execução dispor de muitos processadores (o que já é possível com estações de trabalho de custo relativamente baixo), o problema poderá ser resolvido em menos tempo, ou pelo menos soluções parciais úteis ao usuário poderão ser rapidamente fornecidas, e incrementalmente refinadas. A divisão de uma tarefa a ser executada em várias atividades concorrentes que

colaboram para atingir o resultado esperado introduz muitas dificuldades algoritmicas e de implementação. As dificuldades básicas são abordadas com detalhe neste texto (Seções 2.1 e 2.2).

Para algumas novas categorias de aplicações, que se tornam cada vez mais populares, os aspectos de concorrência se tornaram particularmente relevantes:

- **aplicações multímidia.** O desenvolvimento de aplicações que manuseiam som e imagem de várias fontes vem sendo muito explorada. Aplicações como vídeo-fone, vídeo-mail e vídeo-conferência operam com requisitos de *tempo real*, isto é, uma solução deve não só obter resultados computacionais corretos, como também respeitar restrições temporais (chegar aos resultados dentro dos limites impostos pela aplicação). Sistemas de tempo real podem ser de dois tipos: (1) *hard*, em que o não atendimento a uma restrição temporal significa falha total do sistema ou (2) *soft*, em que se procura respeitar as restrições temporais, mas a falha em uma ou outra restrição de tempo ocasional não tem conseqüências sérias, e portanto não constitui uma falha completa do sistema. A área de tempo real vem sendo estudada há muito tempo, e nela é essencial que o gerenciamento de tarefas simultâneas seja feito de forma eficiente, escolhendo as tarefas sempre de forma a garantir que limites de conclusão das várias tarefas pendentes sejam respeitados. A popularização do desenvolvimento de aplicações multimídia implica na necessidade de ambientes computacionais eficientes e flexíveis no gerenciamento de concorrência, de forma que o desenvolvedor da aplicação multimídia ganhe "de graça" o controle das restrições temporais de sua aplicação. Muitos trabalhos recentes investigam a construção de tais ambientes computacionais [Smart 1997, Eclipse 1998, Rosu 1997], mas o problema está longe de ser resolvido.
- **interfaces baseadas em janelas.** Aplicações com interface via janelas gráficas tornam a concorrência ainda mais evidente para o usuário, já que estas oferecem vários serviços ao usuário via componentes como botões e menus que podem ser acionados quase que "ao mesmo tempo", causando a execução simultânea dos serviços solicitados. Por exemplo, um usuário que utilize uma interface gráfica para receber e enviar e-mails pode solicitar leitura de novos e-mails a qualquer momento, inclusive enquanto utilize a interface para compor/enviar novos e-mails ou ler e-mails antigos. A implementação deste aplicativo de e-mail deve ter sido projetada de forma a lidar com as tarefas de envio e recebimento de e-mail executando simultaneamente e competindo por recursos (por exemplo, pelo repositório de mensagens recebidas).

- **servidores de informação na Teia**[2]. Cada usuário da Teia pode operar sequencialmente, isto é, visitando uma página de cada vez. Mas o servidor (programa que gerencia o envio de páginas solicitadas) poderá receber vários pedidos concorrentes de páginas. Conforme o uso da Teia vá aumentando, veremos cada vez mais grandes bancos de dados acessíveis via browsers, aumentando o potencial para acesso concorrente dos mesmos. Mais recentemente, vem aumentando a presença de animação nas páginas da Teia; processos concorrentes poderiam ser utilizados do lado do cliente de forma a agilizar a animação e permitir que o usuário interaja com a página (preenchendo formulários, por exemplo) enquanto a animação progride.
- **middleware.** Um enfoque amplamente utilizado para lidar com heterogeneidade em sistemas distribuídos é construir uma camada de software acima dos sistemas operacionais heterogêneos de forma a apresentar uma *plataforma uniforme* sobre as quais aplicações distribuídas possam ser executadas. Esta camada – o middleware – em geral é responsável por gerenciar a execução concorrente dos vários componentes do sistema distribuído.

1.3. Aplicação Motivadora

Como já mencionado, existem vários tipos de aplicações que nos levam a lidar com concorrência. Neste mini-curso o objetivo é lidar com um tipo tão corriqueiro e desvinculado de comunidades específicas quanto possível. Em outras palavras, nosso objetivo é aprender sobre concorrência a fim de desenvolver pequenos programas que possam ser executados a partir de browsers na internet, e que mantenham algum estado simples, do tipo que pode ser facilmente manipulado com alguns poucos arquivos, não necessitando do controle de um gerenciador de banco de dados. Mais especificamente, queremos desenvolver applets que lidem com a possibilidade de vários usuários utilizarem tais programas ao mesmo tempo. No contexto deste curso, foram desenvolvidos dois pequenos programas que envolvem os principais aspectos de programação concorrente (de pequena escala, ou seja, em que não estejamos instalando um banco de dados e integrando-o com nosso applet) para a internet. Os exemplos e suas documentações se encontram disponíveis em http://www.ime.usp.br/~dilma/jai99.

O primeiro programa tem como objetivo dar apoio a uma brincadeira usual da época de festividades de fim de ano: o “amigo secreto”. Este programa permite que sejam definidos os partipantes. A partir daí, o programa sorteia quem é amigo secreto de quem, e torna disponível a troca de bilhetes entre os participantes. Também é possível cadastrar pseudônimos para as trocas de bilhetes. Os bilhetes podem ser lidos através do página da internet de acesso ao programa, ou o usuário

[2] World Wide Web.

pode especificar que deseja ter seus bilhetes automaticamente enviados para seu e-mail. Bilhetes podem ser enviados para participantes ou para pseudônimos cadastrados, sendo analogamente assinados por participantes ou por pseudônimos. O programa lida com aspectos de segurança, isto é, não torna disponível o repositório de informações sobre o amigo secreto.

O segundo programa ajuda na organização de uma festa, permitindo que as pessoas confirmem sua presença ou ausência, verifiquem quem já confirmou a ida a festa, e especifique qual será sua contribuição para a festa (que bebida ou comida, de uma lista disponibilizada pelo promotor da festa).

2. Conceitos Básicos e Definições

Os elementos ativos que constituem uma aplicação em execução são referenciados na literatura em computação por vários termos: atividades, processos, tarefas ou *threads*. Estes termos podem assumir diferentes significados dependendo do contexto ou da tecnologia em questão. Adota-se o termo clássico **processo** para denotar uma entidade abstrata que executa um programa (seqüência de instruções[3]) em um processador [Bacon 1998]. Historicamente, a abstração de processo captura a execução dinâmica em um computador, com as operações executadas em um processo sendo feitas estritamente uma por vez. Hoje esta visão já não faz sentido, pois é comum que um mesmo programa ou aplicativo venha a requerer a existência de várias atividades que progridam ao mesmo tempo. Usaremos o termo processo para designar a ativação inicial de um programa. Se este programa é composto por várias atividades separadas, chamaremos estas atividades de **threads** (linhas de execução que compõe o processo). Uma outra forma de denotar esta composição é chamar as várias atividades a serem executadas de *processos*, e o conjunto das atividades de *programa*. A escolha da nomenclatura depende bastante do contexto. Por exemplo, no UNIX originariamente a unidade de execução era o processo, estando disponível na interface para o programador primitivas para a criação e remoção de processos; assim aplicativos que empregavam concorrência eram vistos como programas compostos por vários processos. Mais recentemente as diversas implementações do UNIX disponibilizam também pacotes de threads — também conhecidos como "processos leves" (lightweight processes), por implicarem em custos de criação e manutenção menores —, isto é, interfaces através das quais threads podem ser criados e adicionados/removidos de um processo. Neste texto nos referiremos às atividades concorrentes que compõem o aplicativo como threads, já que este é o termo dos ambientes de programação concorrentes utilizados nos exemplos.

---

[3] Mais formalmente, a descrição de uma computação em uma linguagem formal [Hansen 1973].

Threads (ou processos em programas concorrentes em geral) devem colaborar entre si por dois motivos: (1) para que a funcionalidade esperada do programa seja atingida e (2) para que eles possam se coordenar quanto ao uso de recursos compartilhados. Esta colaboração é feita através de *comunicação*. Por exemplo, se uma aplicação de busca de um usuário chamado "Jassei Java" em uma dada rede de computadores é composta por várias threads, em que cada thread busca o usuário em um dado subdomínio da rede, se um dos threads encontra o usuário ele deve comunicar este fato para os outros threads, de forma que estes cessem as suas buscas. Se a busca é pelo número de usuários com o sobrenome "Java", as quantidades encontradas pelos vários threads devem ser somadas para formar o resultado final da aplicação. Por outro lado, se a aplicação exige que toda vez que um usuário com sobrenome "Java" seja encontrado lhe seja enviado um FAX oferecendo um curso de Java de graça, vários threads podem requerer simultaneamente o uso da placa de FAX, e devem colaborar para que todos consigam acessoao recurso compartilhado.

Comunicação se dá através do uso de variáveis compartilhadas ou do envio de mensagens. Para que a comunicação ocorra, é necessário que os envolvidos estejam prontos em seus postos: um thread pronto para dar a comunicação, outro(s) para receber. Em outras palavras, comunicação exige sincronização. Duas formas de sincronização ocorrem em programas concorrentes [Andrews 1991]: **exclusão mútua** e **sincronização condicional**.

Exclusão mútua tem como objetivo sincronizar threads de forma a garantir que determinadas partes do código sejam executadas por no máximo um thread de cada vez, servindo assim para evitar que recursos compartilhados sejam acessados simultaneamente por vários threads. O desenvolvedor do aplicativo deve identificar quais são as partes do código que exigem este tipo de sincronização (isto é, quais são as *regiões críticas* do programa) e utilizar alguma primitiva de comunicação entre threads disponível em seu ambiente para delinear o início e o fim da região crítica onde se deseja exclusão mútua**.**

Sincronização condicional permite que o programador faça com que um thread espere uma dada condição ser verdadeira antes de continuar sua execução. Com esta forma de controle do andamento de um thread é possível orquestrar o andamento da computação, por exemplo garantindo que um thread espere pela informação computada por um outro thread antes de progredir com sua computação. O desafio para o desenvolvedor está em identificar onde colocar sincronizações condicionais de forma a dirigir corretamente o andamento coletivo dos threads, e também escolher as condições lógicas adequadas a cada sincronização condicional.

Na seção 4 abordaremas as primitivas disponíveis para criação e sincronização de threads em Java e em C/UNIX, e como programar com elas. A fim de esclarecer

como o suporte para threads nestes ambientes funciona, estudaremos como algumas primitivas de sincronização "baixo nível" funcionam e foram implementadas. O objetivo não é cobrir detalhes, e sim evidenciar as dificuldades encontradas em implementar e utilizar tais suportes para sincronização.

2.1 Suporte para Sincronização Condicional: Wait e Signal

Conforme já discutido, threads que cooperam precisam sincronizar uns com os outros. Para isso eles precisam ser capazes de esperar (WAIT) um pelos outros e avisar (SIGNAL) uns aos outros que já é possível prosseguir, ou seja, que o encontro de sincronização terminou. Os threads que competem por recursos devem conseguir esperar (WAIT) para adquirem o recurso compartilhado e avisar (SIGNAL) da liberação de recursos.

Muitos sistemas operacionais oferecem primitivas que permitem especificar espera por um evento e aviso da existência (ou criação) de um evento:

- WAIT (*e*): a execução aguarda até que seja avisada da ocorrência do evento *e*
- SIGNAL(*e*): sinaliza que o evento *e* foi gerado.

Por exemplo, ao especificar a espera por uma interrupção de hardware, indica-se uma sincronização com um hardware. Já com interrupções de software, pode-se especificar sincronização entre programas ou atividades de um programa.

Estes mecanismos podem ser generalizados de forma a cobrir os casos em que se espera qualquer evento oriundo de uma dada atividade, ou um evento específico oriundo de qualquer atividade. As duas primitivas são úteis para expressar sincronização condicional, mas dependendo da forma com que são implementadas, seu uso pode se tornar inviável, como exemplificamos a seguir.

Considere uma aplicação formada por duas atividades: (1) um thread que acha números primos, isto é, consecutivamente procura o próximo inteiro que seja primo e (2) um thread que notifica por e-mail o chefe do laboratório de pesquisa em primos quando um novo primo é encontrado. Os dois threads precisam sincronizar, isto é, o que busca primos deve notificar que um primo foi encontrado, e o thread que envia e-mail deve aguardar por tal notificação (esperar até que uma condição, o encontro de um número primo, seja verdadeira).

```
        Thread Procura
while ( ) { // executa para sempre
        // busca próximo primo
        acha_próximo_primo();
        // avisa ao outro Thread
        SIGNAL (Informa);
}
```

```
Thread Informa
while ( ) { // executa para     sempre
        WAIT(Procura);
        envia_email_para_chefe();
}
```

A aplicação principal simplesmente criaria os dois threads. Suponha que o thread *Procura* foi criado e iniciado antes do thread *Informa*, e que no momento em que *Procura* executa "SIGNAL (*Informa*)", o thread *Informa* ainda não executou WAIT(*Procura*). Dependendo da implementação (e este é o caso em muitas das implementações disponíveis em ambientes Unix), o sinal enviado por *Procura* para *Informa* é perdido (neste exemplo, consequentemente o primeiro primo encontrado não causaria o envio de um e-mail). Outra situação análoga surge quando o SIGNAL de *Procura* faz com que *Informa* continue sua execução e execute envia_email_para_chefe(), mas enquanto esta rotina é processada por *Informa* vários outros primos são encontrados em *Procura*, com os respectivos SIGNAL (*Informa*) sendo perdidos, já que Informa ainda não está atingiu novamente WAIT(*Procura*). Estes problemas podem ser evitados se a implementação de WAIT/SIGNAL garantir que os sinais que cheguem adiantados ao comando WAIT sejam acumulados e entregues para a aplicação que os espera, mas isto tornaria as primitivas mais custosas (trabalho extra de gerenciar que sinais já foram tratados e quais devem ser entregues no próximo WAIT).

2.2. Suporte para Exclusão Mútua

A sincronização via exclusão mútua tem como objetivo garantir que regiões críticas do código (por exemplo, acesso a recursos compartilhados) sejam executadas "sequencialmente", isto é, no máximo um thread deve estar ativo na região crítica por vez. Como a colaboração entre threads muitas vezes é feita através do acesso à memória compartilhada, é particularmente importante que as partes de código que manipulem variáveis compartilhadas sejam protegidas da interferência entre threads. Arquiteturas convencionais permitem que assumamos que:

- A operação de leitura de uma posição de memória é indivisível (ou atômica), isto é, ela não pode ser interrompida no meio com o processador passando a atender outro thread;
- A operação de escrita em uma posição de memória também é atômica.

Se dois threads estão executando concorrentemente, potencialmente qualquer entrelaçamento das instruções dos dois threads pode ocorrer. Por exemplo, considere os threads *SomaUm* e *SomaDois* com os seguintes códigos:

SomaUm	SomaDois
$x = x + 1;$	$x = x + 2;$

Cada um desses threads têm um único comando, mas três instruções : (1) LD *x* (lê *x* da memória), (2) adiciona constante, e (3) e ST *x* (escreve o novo valor de *x* na memória). As duas execuções a seguir de um programa composto por estes dois threads são válidas. A descrição das execuções é apresentada como uma progressão no tempo que indica a ordem em que as instruções foram executadas. A fim de tornar explícito a quem pertence a instrução executada, a mesma é escrita na coluna correspondente ao seu thread.

Histórico da Execução 1:

SomaUm	*SomaDois*
LD *x*	
Incrementa em 1	
ST *x*	
	LD *x*
	Incrementa em 2
	ST *x*

Histórico da Execução 2:

SomaUm	*SomaDois*
LD *x*	
Incrementa em 1	
	LD *x*
	Incrementa em 2
ST *x*	
	ST *x*

O primeiro histórico corresponde a execução sequencial (sem atividades concorrentes) de *SomaUm* seguida da execução sequencial de *SomaDois*. O segundo histórico corresponde a uma execução concorrente dos dois threads em que *SomaUm* começa a execução e é interrompido após duas instruções. Assumindo que a variável *x* como valor inicial zero, ao final da execução 1 ela armazena o valor 3, enquanto que ao final da execução 2 *x* tem o valor 2. Em outras palavras, o programa formado pelos dois threads não tem um resultado determinístico, já que diferentes escalonamentos dos threads (isto é, diferentes escolhas sobre qual thread deve progredir em sua execução) resultam em valores finais diferentes. O mesmo problema ocorre se o programa for executado em uma plataforma com vários multiprocessadores, pois como os processadores podem ter diferentes velocidades de execução ou de acesso à memória, não se sabe a priori a ordem relativa entre as instruções dos dois threads. Este indeterminismo pode ser evitado se o acesso à variável compartilhada *x* for feito com exclusão mútua, isto é, se forem impossibilitadas execuções em que mais de um thread esteja executando alguma parte do acesso ao recurso compartilhado (isto é, estiver na região crítica). A região crítica do exemplo acima é bem curta (um comando em linguagem de alto nível apenas), mas se o recurso compartilhado for uma estrutura de dados complexa (como uma B-árvore ou grafo dirigido), a seção do código que manipula a estrutura pode ser longa e de execução demorada.

O problema a ser resolvido é como fornecer a um thread acesso exclusivo a uma estrutura de dados para um número arbitrário de leituras e gravações. As subseções a seguir discutem soluções clássicas para o problema. Observe que além de garantir o acesso exclusivo, é desejável que soluções para o problema tenham as seguintes características:

- Se dois ou mais threads estão tentando obter acesso para a estrutura de dados, então pelo menos um deles vai ter sucesso. Em outras palavras, a execução do programa não fica "congelada": não ocorre *deadlock*[4].
- Se um thread T está tentando obter acesso exclusivo a um recurso (como uma estrutura de dados, por exemplo) e os outros threads estão ocupados com tarefas que não se relacionam com o recurso compartilhado, então nada impede que o thread T obtenha o acesso exclusivo. Isto fica mais claro se pensamos em uma solução em que esta característica não é atingida: suponha o esquema de acesso a uma placa de FAX compartilhada por dois threads em que o acesso exclusivo é obtido com a seguinte regra: durante os 30 primeiros minutos de

[4] Dizemos que um conjunto de threads está em deadlock se todos os threads do conjunto estão aguardando eventos que só podem ser gerados por threads pertencentes ao conjunto. Em outras palavras, eles estão bloqueados esperando por algo que não pode ser fornecido por elementos externos ao conjunto.

qualquer hora só o thread 1 pode usar a placa, e nos últimos 30 minutos só o thread 2 tem acesso a este recurso compartilhado. Com este esquema, temos que se o thread 2 quer acesso às 14:10, ele terá que esperar (pelo menos 20 minutos) mesmo se o thread 1 não estiver utilizando o recurso.

- Se um thread está tentando conseguir acesso ao recurso, alguma hora ele consegue.

2.2.1. Com suporte do hardware:Test-and-Set

Os vários threads competindo por uma região crítica (isto é, pelo acesso a um recurso compartilhado) podem adotar a convenção de que *sempre* antes de entrar na região crítica eles testam o valor de uma variável que indica se a região está livre ou ocupada. No exemplo da aplicação formada pelos threads SomaUm e SomaDois, o desenvolvedor faria:

SomaUm	*SomaDois*
Se (flag == 0) {	Se (flag == 0) {
flag = 1;	flag = 1;
x = *x* + 1;	*x* = *x* + 2;
flag = 0;	flag = 0;
}	}

O código acima obviamente continua com problema, uma vez que o teste do valor da variável *flag* e sua mudança para 1 não são atômicas. Em uma plataforma com vários processadores, existe uma chance de que os dois threads fossem executados exatamente ao mesmo tempo, e portanto ambos achassem que a região crítica estava livre. Mesmo em processadores com um único processador, o problema continua: é possível que o primeiro thread carregue da memória o valor de *flag*, compare com 0 e seja interrompido exatamente antes de ter a chance de escrever 1 na variável (com isso indicando que a região crítica está ocupada). Assim, o segundo thread ao carregar o valor de *flag* também obteria 0, e teríamos os dois threads ativos na região crítica.

Muitos computadores têm alguma instrução que executa atomicamente leitura, teste de valor e escrita. Isto é, a instrução executa em um único ciclo, a operação de (1) verificar se a região crítica está livre e (2) marcá-la como ocupada se este é o caso não pode ser interrompida. Um exemplo de tal instrução é a *Test-And-Set*, que tem tipicamente a seguinte forma:

```
se a booleana indica que a região está livre {
        mude a booleana para indicar região ocupada;
        pule a próxima instrução;
}
senão
        execute a próxima instrução;  // em geral uma instrução que desvia o
                                      // fluxo para fazer o teste de novo
```

Outra forma de enxergar as implementações de Test-And-Set disponíveis nos hardwares usuais é como uma instrução que dadas duas posições de memória, atomicamente copia o conteúdo da segunda na primeira e coloca 1 na segunda. A entrada na região crítica ficaria então:

```
valor_lido = 1;
flag = 0;
while (valor_lido == 1) {
        Test-And-Set (valor_lido, flag); // atômico
        if ( valor_lido == 0 ) {    // se estava zero antes de eu pedir para colocar
                < código da região crítica >
                flag = 0;
        }
}
```

Este tipo de solução, que envolve repetidamente testar se a região crítica ficou livre, é chamada de *espera ocupada* (*busy wait* ou *spin lock*), já que enquanto o thread espera para entrar na região crítica ele continua "gastando" a CPU com comparações e chamadas do Test-And-Set. Alternativamente, o thread poderia ser bloqueado, isto é, sair da CPU por algum tempo, voltando para tentar novamente mais tarde (preferencialmente, nos momentos em que a região crítica foi desocupada e portanto há chance de sucesso na tentativa de obter acesso).

Outras instruções de hardware (por exemplo, Compare-And-Swap atômico) tornam possível o uso de uma variável para indicar se a região crítica está livre ou ocupada. Observe, no entanto, que este tipo de solução exige que a variável faça parte do espaço de endereçamento de todos os threads envolvidos, isto é, que as atividades concorrentes possam compartilhar posições de memória. Em ambientes com vários processadores, o compartilhamento de uma variável como o que foi feito com o Test-And-Set no exemplo acima pode causar tremendos prejuízos de desempenho, pois a cada vez que se escreve 1 em *flag* (o que é feito toda fez que Test-And-Set é executada, estando a região crítica livre ou não) as entradas dos *caches* correspondentes à variável em cada processador podem ter que ser invalidadas.

Outra deficiência de uma solução baseada em Test-And-Set é a questão de justiça: nada impede que um thread consiga entrar sucessivamente na região crítica, enquanto outro persiste na tentativa sem ter sucesso.

Em ambientes com apenas um processador, uma técnica comum para proibir que a execução seja interrompida é simplesmente desabilitar as interrupções durante a execução da região crítica, tornando impossível que outro thread tenha a chance de executar[5]. Esta solução se torna inaceitável se a região crítica for longa, pois interrupções importantes (como, por exemplo, as relacionadas à entrada e saída) poderiam ser perdidas.

2.2.2. Sem suporte de hardware

O problema de obter exclusão mútua sem usar instruções específicas de hardware e sem apelar para desabilitação de interrupções (já que isto não resolve o problema em máquinas com vários processadores) recebeu a atenção de cientistas importantes da área de computação e matemática. O problema começou a ser discutido por Dijkstra em 1965 e o matemático Dekker publicou a primeira solução correta para lidar com a disputa por dois threads. Dijkstra então generalizou a solução para funcionar para um número arbitrários de threads que competem pela região crítica. Por algum tempo o problema foi considerado de pouco interesse prático, já que em máquinas com um único processador o enfoque descrito na seção 2.2.1 pode funcionar bem. Mas com o tempo o interesse na construção de máquinas com vários processadores a partir da conexão de computadores com memória e processador padrões trouxe novamente a atenção para o problema, tentando-se obter soluções alternativas práticas para obter exclusão mútua através de memória compartilhada. Este interesse resultou em muitas soluções alternativas e versões cada vez mais eficientes sendo propostas nas décadas de 70 e 80.

Os algoritmos que solucionam o problema da exclusão mútua não são simples de entender. Uma descrição detalhada e bem formalizada de soluções presentes na literatura pode ser encontrada em [Andrews 1991]. Nesta seção os algoritmos são discutidos superficialmente; suas implementações em Java estão disponíveis, de forma detalhada e discutida, em http://www.ime.usp.br/~dilma/jai99/.

O algoritmo de “senha” (Ticket algorithm) se baseia no procedimento adotado em grandes açougues, confeitarias ou centrais de atendimento a fim de garantir que os clientes sejam atendidos na ordem de chegada sem que se forme uma fila física. Há um dispositivo que libera cartões numerados sequenciamente (“senhas”). O cliente ao chegar se dirige ao dispositivo e retira um número, e fica esperando até que seu

[5] Pois a troca de threads é baseada no uso de interrupções.

número seja chamado. Se pensamos nesta solução em um cenário em que apenas um atendente está disponível, vemos os clientes competindo pelo atendente da mesma forma em que threads competem pelo acesso à região crítica. Uma implementação dessa idéia pode ser feita utilizando-se duas variáveis compartilhadas: *número* e *próximo*, ambas tendo 1 como valor inicial. Cada thread então teria a seguinte estrutura algoritmica:

```
int minha_vez;                  // variável interna ao thread,
                                // não compartilhada.
minha_vez = número; número ++;  // ATÔMICO !
while (próximo != minha_vez) ;  // espera ocupada
executa_região_crítica();       // aqui está o código a ser protegido
                                // de execução simultânea
próximo++;                      // ATÔMICO: passa posse da região
                                // crítica pro seguinte
```

Os números distribuídos aos threads que desejam entrar na região crítica não se repetem, e se a obtenção de um número por um thread e a geração do número seguinte forem feitas de forma indivisível, então há garantia de que no máximo um thread por vez tem acesso ao código executa_região_crítica(). Cada cliente (cada thread) é atendido assim que chega a sua vez e há garantia de que todo thread pleiteando acesso vai conseguir alguma hora. Mas a solução tem um problema: as variáveis *próximo* e *número* crescem ilimitadamente; na prática se o programa rodar por muito tempo, as variáveis atingiriam o limite máximo, em muitas arquiteturas voltando para zero. Isto seria um problema se o número de threads esperando pela sua vez excedesse ao maior inteiro armazenável, pois dois threads estariam esperando com o mesmo número de atendimento.

Outra questão importante é que esta solução exige que alguns trechos (delimitados por um retângulo no código acima) sejam executados atomicamente, podendo ser implementados em máquinas que tenham uma instrução atômica de Fetch-And-Add[6] ou através da desabilitação temporária de interrupções nestes pequenos trechos de código.

O algoritmo da Padaria (Bakery Algorithm), proposto por Lamport em 1974, aproveita a idéia da distribuição de senhas, com a vantagem de não exigir um gerador atômico do próximo número e nem a manutenção de uma variável

[6] Fetch-And-Add carrega o valor de uma variável e imediatamente incrementa-a.

próximo. O algoritmo é mais complicado, mas ilustra uma idéia bem útil: quebrar empates quando dois threads estão com senhas com o mesmo número. A idéia do algoritmo também vem do mundo real, como no caso do algoritmo com distribuição de senhas. Quando a gente vai a uma padaria ou açougue mais simples, em geral não encontramos um dispositivo para a gente pegar um número de atendimento. Simplesmente a gente olha pros lados, observa quem está esperando para ser atendido, e se situa sobre quando seremos atendidos (só quando os que já estavam lá tiverem sido atendidos). Em outras palavras, o thread escolhe para si um número que seja maior do que qualquer outro número que esteja em posse de algum thread esperando pelo acesso, e o thread sabe que está na hora de executar quando o seu número for o menor entre todos que estejam esperando. O seguinte pseudo-código descreve o que cada thread faz para entrar na região crítica, utilizando-se uma vetor de inteiros vez[], onde vez[t] armazena 0 se o thread t não está interessado em entrar na região crítica ou armazena um número > 0 que representa seu número de atendimento.

```
// início da tentativa de entrar na região crítica
// t é um identificador número do thread
vez[t] = máximo(vez) + 1;          //atribuição de número de atendimento
enquanto (vez[t] > mínimo_não_nulo(vez));      // espera pela sua vez
executa_região_crítica();
vez[t] = 0;      // não está mais interessado na região crítica
```

Obviamente, se as ações identificadas por retângulos acima não forem executadas com exclusividade pelo thread, mais de um thread pode acabar escolhendo o mesmo número de atendimento. O algoritmo permite que isto aconteça, ou seja, que mais de um thread receba um mesmo número de atendimento, mas impõe um desempate entre os que têm o mesmo número, de forma que apenas um deles acabe entrando na região crítica.O desempate se dá através do número identificador de cada thread. Ou seja, ao invés de comparar vez[a] > vez[b], comparamos (vez[a], a) > (vez[b], b) onde esta expressão é verdadeira se vez[a] > vez[b] ou (vez[a] = vez[b] e a> b). O algoritmo fica:

```
// início da tentativa de entrar na região crítica
// t é um identificador número do thread
vez[t] = máximo(vez) + 1;     // atribuição de número de atendimento
                              // OK se números repetidos forem fornecidos
para todo thread j que não seja este thread t {
      enquanto (vez[t] ≠ 0 e (vez[t], t) > (vez[j], j) ;     // espera pela sua vez
executa_região_crítica();
vez[t] = 0;                   // não está mais interessado na região crítica
```

Para perceber porque o algoritmo funciona, isto é, porque não existe uma situação em que dois threads executam ao mesmo tempo executa_região_crítica() é preciso entender como funciona o critério de desempate contido no retângulo pontilhado acima. A forma de desempate entre vários threads que por acaso receberam o mesmo número de atendimento se baseia no algoritmo de Petersen (também conhecido como Tie-Breaker), que também é um algoritmo que pode ser usado para o problema da exclusão mútua (já que se quer desempatar entre vários threads que desejam acesso à região crítica). A idéia é um desempate em fases: se existem *n* threads no programa, o algoritmo tem *n* fases. Em uma primeira fase, *n* threads executam vez[t] ≠ 0 e (vez[t], t) > (vez[j], j), mas mesmo que todos estejam empatados no número de atendimento vez[], um deles perde para todos os outros[7], resultando em que no máximo *n-1* threads passam para a segunda fase do algoritmo de desempate. Nesta segunda fase, o mesmo código "enquanto (vez[t] ≠ 0 e (vez[t], t) > (vez[j], j)" é executado, e pelo menos um dos n-1 threads fica preso no loop, resultando em no máximo n-2 threads passando da fase 2 para a fase 3. O mesmo raciocínio se aplica nas outras fases, e temos que na fase n-1 no máximo (n – (n-1) = 1 thread consegue sair do trecho de desempate e entrar na região crítica.

De uma forma geral, as idéias das soluções para o problema da exclusão mútua não fazem parte do arsenal diário de um desenvolvedor de programas concorrentes, já que este se utiliza das primitivas já disponíveis em seu ambiente a fim de proteger recursos compartilhados (regiões críticas) do acesso concorrente. Mas o entendimento de como e porque realmente os algoritmos funcionam é bem útil na formação do desenvolvedor, pois os argumentos exercitam as situações mais comuns de erros na programação concorrente: uma troca de contexto em um momento crucial, ou uma hipótese sobre o estado de uma variável compartilhada

[7] Porque um deles tem o menor identicador de thread.

que não se mantém válida durante toda a execução. Outra razão para não ignorarmos estes algoritmos é que em alguns casos uma forma mais relaxada (e particular para a aplicação) de exclusão mútua é necessária, e dificilmente tal solução ad hoc estará disponível no ambiente de programação. Soluções particulares podem ser obtidas com pequenas alterações nos algoritmos clássicos.

2.3. Semáforos

O conceito de semáforos apareceu em 1968, no trabalho de Dijkstra, como um mecanismo para sincronização de atividades concorrentes. Um semáforo é um *tipo de variável*, ou seja, denota não só como uma determinada informação é armazenada, mas também como esta informação pode ser manipulada, ou seja, que operações estão disponíveis para uso da variável. Ao invés de termos, como nas soluções para o problema de exclusão mútua já descritos, uma variável "normal" sendo usada de forma complexa para expressar sincronismo, teríamos disponível um tipo específico de variável útil apenas para lidar com sincronização. As operações disponíveis para este tipo encapsulariam os detalhes complexos de como o sincronismo é obtido, tornando o projeto de programas mais fácil de ser entendido e de ser verificado como correto. O conceito de semáforo foi uma das primeiras (e até hoje mais importantes) ferramentas para desenvolvimento disciplinado de programas concorrentes. Outro aspecto importante dos semáforos é que sua forma de obter sincronização não se baseia em espera ocupada (como é o caso das soluções nas seções 2.2), e sim em permitir que o thread seja posto de lado, parando de consumir recursos, até que chegue o momento dele sincronizar. O tipo ***semáforo*** é em geral representado nas implementações através um número inteiro e de uma fila que descreve quais threads estão aguardando para sincronizar.

O conceito de semáforo (e obviamete o próprio termo) surgiu da utilização de semáforos no mundo real a fim de evitar colisões de trens. Um semáforo na linha ferroviária é um sinalizador que indica se o trilho a frente está disponível ou ocupado por um outro trem. Nos semáforos de nossas cidades a idéia é semelhante: há um indicador de que a parte do cruzamento está disponível para o usuário ou ocupada por ter sido cedida a usuários do cruzamento que vêm de outra direção. Tanto semáforos de linhas ferroviárias quanto de caminhos viários tem como objetivo coordenar o acesso a um recurso compartilhando, permitindo também sincronização condicional (um trem fica parado até que a condição de linha ocupada mude).

O tipo semáforo oferece os seguintes usos:

- Um semáforo pode ser criado recebendo um valor inicial, que indica o número máximo de threads que o semáforo deve deixar "passar" em direção ao recurso

compartilhado. Por exemplo, quando inicializado com o valor 1 o semáforo pode ser utilizado para prover acesso exclusivo a algum recurso compartilhado;

- Um semáforo pode ser utilizado através de sua operação **wait**, com o seguinte significado: se o valor do inteiro armazenado no semáforo for maior que zero, então este valor é decrementado e o thread que utilizou a operação pode prosseguir normalmente em sua execução. Se o valor é zero, então o thread que chamou a operação é suspenso e a informação de que este thread está bloqueado neste semáforo é armazenada na estrutura de dados do próprio semáforo.
- Um semáforo pode ser utilizado através da operação **signal**, que tem a seguinte funcionalidade: se não há thread algum esperando no semáforo, então incrementa o valor do inteiro armazenado no semáforo. Se há algum thread bloqueado, então libera um thread, que terá sua execução continuando na instrução seguinte ao **wait** que ocasionou a espera. É uma questão de implementação se o thread liberado é o primeiro da fila ou não.

No trabalho original de Dijkstra, a operação **wait** era chamada de **P**, e a operação **signal** de **V**.

O uso de semáforos para proteger uma região crítica de código pode então ser feita através do uso da seguinte variável acessível a todos os threads competidores:

```
Semáforo lock = new Semáforo(1);      // cria variável s do tipo Semáforo,
                                      // inicializando o semáforo com 1
```

Cada thread usaria o semáforo da seguinte forma:

```
<comandos quaisquer que não façam uso de recursos compartilhados>;
lock.wait();     // o thread chama a operação wait do semáforo lock, ou
                 // seja, dependendo do inteiro armazenado em lock (se este
                 // é 0 ou não), o thread pode bloquear.
executa_região_crítica();      // se a execução chegou neste ponto, o semáforo
                       // indicava que a região crítica estava liberada
lock.signal();         // libera a região crítica. Se há outros threads bloqueados
                       // aguardando no semáforo, um deles volta a executar.
<comandos quaisquer que não façam uso de recursos compartilhados>;
```

A escolha do nome *lock* para o semáforo não foi por acaso. É muito comum se referir às variáveis que protegem as regiões críticas como locks, já que funcionam como cadeados protetores que impedem a entrada indesejada. Em alguns ambientes, suporte para exclusão mútua é oferecido através de rotinas lock() e unlock(), em que se obtem permissão de acesso e depois libera-se o acesso para outro thread. Locks são um caso particular de semáforos, pois referem-se a semáforos inicializados com o valor 1.

A implementação dos semáforos (isto é, das operações wait e signal) deve ser feita de forma que a execução de uma dessas operações (onde se consulta e altera o valor de um inteiro, e às vezes se manipula a fila de processos bloqueados) seja atômica, já que um número arbitrário de operações podem ser requisitadas no mesmo semáforo concorrentemente. Ou seja, a própria implementação dos semáforos (que servem para proteger regiões críticas) envolve lidar com regiões críticas, por exemplo através das soluções vistas na seção 2.2.

Textos tradicionais em sistemas operacionais contém muitos exemplos da solução de problemas com o uso de semáforos[Tanenbaum 1992, Silberschatz-Peterson 1988]. Como os ambientes em que desenvolveremos nossos exemplos não incluem diretamente suporte para semáforos, não nos aprofundaremos no assunto.

2.4. Monitores

O uso de semáforos permite diferenciar o que está sendo usado para controlar a concorrência, mas como semáforos são globais a todas os threads que queiram utilizá-los, seu uso a fim de proteger uma única estrutura de dados, por exemplo, pode estar disperso por todo o programa. Para entender como a estrutura é acessada ou verificar que ela está corretamente protegida de acesso concorrente, tem-se que vasculhar o código de todos os threads. Além disso, não há associação sintática entre semáforos e os códigos que estes protegem, portanto nada impede que o programador utilize vários semáforos para proteger a mesma estrutura de dados, tornando difícil entender que todos os vários trechos do programa são relacionados. Analogamente, várias regiões críticas totalmente independentes podem ter sido protegidas com o mesmo semáforo, potencialmente diminuindo desnecessariamente o potencial para concorrência. *Monitores*, inicialmente propostos por Hoare [Hoare, 1974] e Hansen [Hansen 1973a], são módulos de programas que têm a estrutura de um tipo abstrato de dados. Estes módulos contêm dados encapsulados (isto é, disponíveis apenas dentro da implementação do módulo) que são manipulados pelas operações definidas no módulo. Há exclusão mútua entre as operações, isto é, em nenhum momento duas operações poderão estar sendo executadas ao mesmo tempo. A implementação dos monitores deve assegurar a exclusão mútua de execução de operações. Um compilador poderia implementar monitores associando a cada monitor um semáforo; a chamada de

uma operação implicaria em um wait() e o término de execução da operação um signal().

Mas, como vimos desde o início desta seção, exclusão mútua não é suficiente para programação concorrente, pois para muitos problemas a sincronização condicional é essencial para que os threads colaborem. Para este fim, os monitores proveem um novo tipo de variável: **variável de condição**. Estas variáveis podem ser usadas pelo programador para expressar condições de interesse para os threads. Se um thread quer, por exemplo, esperar até que uma determinada variável assuma o valor 0, ele poderia ficar continuamente monitorando o valor da variável até que sua condição desejada seja verdadeira, mas este enfoque tem as desvantagens usuais de uma espera ocupada. Se o thread tem disponível uma variável de condição *cond*, o thread pode bloquear a si mesmo executando a operação *cond.wait()*, e permanecerá assim bloqueado até que algum outro Thread execute *cond.signal()*. Assim, para esperar até que uma determinada variável assuma o valor 0, pode ser criada uma variável de condição, por exemplo, *espera_zero*, e o thread começa a esperar executando *espera_zero.wait()*. Quando outros threads alterarem o valor da variável, eles devem colaborar com o thread bloqueado executando espera_zero.signal(), que acordará o thread bloqueado, que pode então ver se a variável ficou com 0 e até continuar esperando novamente se for o caso. Observe que mesmo este exemplo simples já mostra como o uso de condições é suscetível a erros: se algum thread falha em fazer a sua parte esquecendo de acionar signal(), algum outro thread pode ficar bloqueado indefinidamente. As operações wait/signal apresentam alguma semalhança com o que foi apresentado na seção 2.1, mas como as operações só podem ser utilizadas de dentro de um monitor, sabemos que não seria possível termos vários wait/signal ocorrendo simultaneamente, e portanto a implementação destas primitivas fica simplificada. Mas ainda é possível ter a situação em que um signal() é executado sem haver um correspondente wait() esperando-o, e neste caso o signal() não tem efeito algum. Na seção 2.1 discutimos esta situação no contexto de um thread Procura, responsável pela tarefa de, usando algoritmos matemáticos, encontrar primos, e o thread Informa, que era responsável pelo envio de e-mails quando novos primos eram encontrados, apontando as falhas na seguinte solução:

```
    Thread Procura                          Thread Informa
while ( ) { // executa para sempre      while ( ) { // executa para sempre
    // busca próximo primo              WAIT(Procura);
    acha_próximo_primo();               envia_email_para_chefe();
    // avisa ao outro Thread        }
    SIGNAL (Informa);
}
```

Utilizando monitores, o programa pode ser estruturado de forma a conter os dois threads que realizam o trabalho, mas evidenciado que a colaboração entre os dois threads deve ser sincronizada pelo encontro de novos primos. Uma forma possível de estruturar este programa seria termos um monitor e dois threads, como descrito a seguir:

```
Monitor Primos {
    variável-de-condição primo_encontrado;
    operação procura_próximo() {
        acha_próximo_primo();
        primo_encontrado.signal();
    }
    operação informa_próximo() {
        primo_encontrado.wait();
        envia_email_para_chefe();
    }
} // fim do monitor

Thread Procura {
    while (true) {
        Primos.procura_próximo();
    }
}
Thread Informa {
    while (true) {
        Primos.informa_próximo();
    }
}
```

Com esta solução, nenhuma notificação deixa de ser enviada, mas observe que o potencial de concorrência também foi diminuído: mesmo com dois processadores disponíveis, não seria possível que o trabalho de envio de e-mail fosse feito ao

mesmo tempo que a busca por primos. Em geral, soluções com monitores colocam dentro do monitor apenas a condição de sincronização, e não o acesso aos recursos, a fim de permitir que controlemos acesso concorrente aos recursos compartilhados.

A implementação de variáveis de condição tem um problema potencial. Suponha que o exemplo acima tivesse sido programado da seguinte forma:

```
Monitor PrimosComLoop {
    variável-de-condição primo_encontrado;
    operação procura() {
      while () {
          acha_próximo_primo();
          primo_encontrado.signal();
      }
    }
    operação informa() {
      while (){
          primo_encontrado.wait();
          envia_email_para_chefe();
      }
    }
} // fim do monitor
```

O problema está na operação *procura*: após executar o signal, a operação ainda está ativa, com mais coisas a serem executadas, e a operação informa também estaria ativa por ter sido desbloqueada pelo signal. Mas, por definição, apenas uma operação do monitor pode estar ativa a cada momento ! Uma solução é obrigar o signal a ser a última operação da rotina. Na seção 3 discutiremos como o modelo de concorrência de Java e o do Pthreads abordam a questão.

3. Programação com Threads

Já vimos que uma aplicação concorrente é composta por vários threads de execução, e que um aspecto chave é a colaboração entre os vários threads. O uso de threads requer que o programador lide explicitamente com o gerenciamento de concorrência e sincronização. Como vimos na seção 2, muitos problemas podem estar envolvidos se tivermos que começar do zero, nos apoiando em instruções de hardware atômicas de forma a conseguir algum controle da sincronização. Tradicionalmente estes problemas só interessavam os desenvolvedores de software básico, mas, como argumentado na seção 1, a gama de aplicativos que exigem a utilização de vários threads vem aumentando. Programar com threads é considerado demasiadamento complexo por alguns [Ousterhout 1996], enquanto outros acreditam que na maior parte dos cenários de aplicação, um uso disciplinado de threads pode ser aprendido e empregado [Ruddock-Dasarathy 1996].

As duas formas mais populares hoje em dia de programação com threads são o uso da linguagem Java (que é uma linguagem de programação concorrente!) e o uso uma biblioteca de threads padrão, a *pthreads*.

Pthreads oferecem ao programador primitivas para administração de threads (criação, cancelamento, ajuste de prioridade, etc), para acesso exclusivo a regiões críticas ("mutexes") e variáveis de condição [Bacon 1998]. A definição completa pode ser encontrada no padrão IEEE POSIX 1003.4a, e exemplo de implementação são o Solaris Pthreads e DECthreads (Multi-threading Run-time Library para OSF/1, da Digital). Uma visão geral do uso de bibliotecas de threads é fornecida em [Kleiman 1995, Norton 1996, Butenhof 1997].

Como nossa intenção é criar programas concorrentes para aplicações acessíveis via internet, apresentaremos aqui o modelo de threads da linguagem Java.

A linguagem Java, além de definir uma sintaxe e semântica para especificação de programas, oferece uma boa gama de "pacotes" prontos, isto é, implementa diversos tipos abstratos de dados muito úteis no desenvolvimento de programas.A unidade fundamental de programação em Java é a classe [Arnold-Gosling 1996]. Classes fornecem a estrutura para os objetos (as variáveis interessantes do seu programa, isto é, que não sejam de tipos primitivos como inteiro, real, etc.), a forma de criar objetos a partir das definições das classes e a lista de métodos (operações) disponíveis para manipular os objetos da classe. Do ponto de vista do programador, a essência do suporte para concorrência da linguagem Java é oferecida em duas classes: a *classe java.lang.Thread* e *java.lang.Runnable*. Estas classes permitem que definamos threads para as nossas aplicações e gerenciemos seus estados de execução e a colaboração entre os threads.

A classe java.lang.Runnable é na verdade uma interface, isto é, ela declara um tipo que consiste apenas de métodos abstratos (não implementados) e constantes, permitindo que o tipo possa receber diferentes implementações. A interface java.lang.Runnable contém apenas um método abstrato, o método *run()*, que representa o código a ser executado pelo thread. Qualquer classe que queira implementar a interface *Runnable* deve fornecer o código para *run*(). Ao criarmos um novo thread a partir de uma classe que implementa a interface *Runnable*, quando o thread começar a trabalhar, será este método *run* que será acionado.

A classe java.lang.Thread representa um thread de execução, e oferece métodos que permitem executar ou interromper um thread, agrupar threads, e especificar a prioridade de execução do thread (isto é, que preferência se tem por este thread na hora de escolher um próximo thread para ocupar a CPU).

3.1 Criação de Threads em Java

Essencialmente, a diferença entre a classe Thread e a interface Runnable é que o Thread existe para representar como um thread de execução roda (sua prioridade, seu nome, como ele pode sincronizar com outros threads), enquanto que Runnable define *o que* o thread executa [Farley 1998]. Vejamos como criar, em ambos os casos, um tipo de thread bem simples, que imprime "oi!" e termina, e um programa composto por alguns desses threads.

Com a classe *Thread*, simplesmente utilizamos esta classe já existente alterando o método run a fim de que ele execute o que desejamos, isto é, definimos uma nova classe*NossoThread* como subclasse de *Thread*:

```
public class NossoThread extends Thread {
      public void run ()   {
            System.out.println(" Oi!");
 }
```

Para criar o programa que usa este thread basta definir o programa principal (rotina main) de forma que esta crie um objeto do tipo NossoThread, e inicie sua execução, através do método start() (que simplesmente chama o método run):

```
public class UsaNossoThread {
      public static void main ( String[] args) {
            new NossoThread().start();
}
```

Obviamente as coisas ficam mais interessantes quando o programa é formado por vários threads. Se todos imprimem a mesma mensagem "Oi!", não distinguiríamos qual fez que parte olhando para a saída. Mas como a classe Thread faz com os threads tenham nomes (um nome default atribuído automaticamente, ou o nome escolhido pelo programador, fornecido na hora da criação do objeto), poderíamos melhorar as definições das classes acima da seguinte forma:

```
class NossoThread extends Thread {
      public NossoThread (String nome) {
            // passa o nome recebido pelo construtor da subclasse para o
            // a construtor da superclasse
            super(nome);
      }
      public void run ()   {
            System.out.println("Executando thread "+ getName() + "- Oi!");
 }
```

```
public class UsaNossoThread {
    public static void main ( String[] args) {
        // cria três threads, fornecendo nomes escolhidos para cada um
        new NossoThread("Um").start();
        new NossoThread("Dois").start();
        new NossoThread("Três").start();
}
```

Este mesmo exemplo, utilizando a interface Runnable, seria definido da seguinte forma:

```
class NossoDizOi implements Runnable {
    public void run() {
        System.out.println("Executando thread "
            + Thread.currentThread().getName() + "- Oi!");
    }
}
public class UsaNossoThread {
    public static void main(String[] args) {
        // primeiro, criamos os três objetos que contêm o código a
        // ser rodado
        NossoDizOi a = new NossoDizOi();
        NossoDizOi b = new NossoDizOi();
        NossoDizOi c = new NossoDizOi();
        // agora criamos threads, fornecendo no construtor além do
        // nome que queremos, o objeto que tem o que código a ser rodado.
        new Thread(a, "Um").start();
        new Thread(b, "Dois").start();
        new Thread(c, "Três").start();
    }
}
```

Observe que o código que faz o trabalho do thread mudou um pouco: na versão a partir da classe Thread, para obter o nome do thread que iria imprimir, bastava chamar getNome(), já que o objeto que chama a função herdou a operação de Thread. Já no exemplo com Runnable, primeiro temos que descobrir qual é o thread que está executando *run* (através da rotina Thread.currentThread()), e aí chamar getNome() para este objeto thread.

Há boas razões tanto para escolhermos definir threads a partir de Thread quanto a partir de Runnable. Há uma regra que tem se mostrado útil na prática: *Se a classe que você está criando como código do thread precisa ser derivada de alguma*

outra classe, então utilize Runnable[8]. Para programas que devem ser executados a partir de browsers, precisaremos que o código seja subclasse de Applet, e portanto Runnable é o caminho mais apropriado a ser seguido.

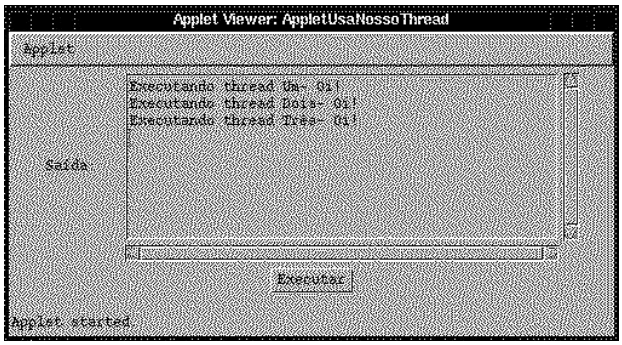
Este mesmo exemplo, na forma de um applet a ser incluído em uma página html, ficaria assim:

```
// preparo para usar applets e elementos gráficos
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class AppletUsaNossoThread extends Applet implements Runnable {
      // Estamos definindo classe de objetos que pode ser incluída em páginas
      // html, e também especifica código a ser executado por thread.

      Label label_saida = new Label("Saída: ");     // elemento gráfico
      TextArea texto_saida = new TextArea("", 10, 50); // área para escrever
      Button botao = new Button("Executar");       // botão para requisitar cria-
                                                   // ção e execução de threads
      public void run() {          // código a ser executado nos threads
            // escreve na área gráfica criada para este fim
            texto_saida.append("Executando thread "
                  + Thread.currentThread().getName() + "- Oi!\n");
      }
      public void init() {
          // coloca itens gráficos
          add(label_saida);
          add(texto_saida);
          add(botao);
          // Define, instancia e registra objeto que trata
          // do botao quando pressionadobotao.
          addActionListener( new ActionListener () {
                  public void actionPerformed(ActionEvent e) {
                  cria_threads();        // rotina definida a seguir
            }});
      }
```

[8] Isto está relacionado ao fato de não haver herança múltipla em Java, ou seja, não seria possível uma classe ser subclasse ao mesmo tempo de Thread e alguma outra classe.

```
    void cria_threads() {
          // agora criamos threads, fornecendo no construtor além do
         // nome que queremos, o objeto que tem o que código a ser rodado.
          new Thread(this, "Um").start();
          new Thread(this, "Dois").start();
          new Thread(this, "Três").start();
     }
}
```

Este applet visto pelo *appletviewer* tem a seguinte apresentação:



Além da execução exemplificada na figura acima, este programa poderia resultar ainda em mais cinco possíveis saídas (duas delas exemplificadas a seguir) na área de texto (TextArea texto_saida), dependendo de qual thread a máquina virtual de java escolheu rodar primeiro

Executando thread Dois - Oi! Executando thread Três - Oi! Executando thread Um - Oi!

Executando thread Três - Oi! Executando thread Um - Oi! Executando thread Dois - Oi!

3.2 Prioridades de Threads em Java

Threads, em Java e em outros ambientes, foram projetados para executar concorrentemente. Mas muitos computadores dispõem apenas de um processador, com threads executando um por vez, de forma a prover a ilusão de concorrência. Os threads são escalonados em alguma ordem, isto é, o gerenciamento (da máquina virtual Java ou do sistema operacional) decide (através de alguma política de escolha) que thread deve ocupar a CPU. Java adota um algoritmo de escalonamento bem clássico: *escalonamento com prioridades fixas*, onde a cada

thread está associada uma prioridade (numérica), e o escalonador escolhe threads com base nas prioridades relativas entre os threads aguardando o uso da CPU.

Quando um thread é criado, ele tem a mesma prioridade do thread que o criou[9].As prioridades podem ser alteradas a qualquer momento através do método *setPriority*. Prioridades são inteiros no intervalo definido pelas constantes disponibilizadas em java.lang.Thread: MIN_PRIORITY e MAX_PRIORITY. Um thread de maior prioridade só cede seu lugar para um thread de menor prioridade se ele terminar sua execução, ceder espontaneamente sua vez (através do método *yield*) ou não puder prosseguir sua execução por estar aguardando algum evento. Threads de mesma prioridade são alternados, formando uma fila em que o thread, após ser servido, vai para o final da fila. Um thread escolhido continuará executando até que uma das seguintes condições se torne verdadeira:

- Algum thread de maior prioridade fica pronto para executar (por ter sido criado ou por não precisar mais esperar por algo). O thread de menor prioridade que estava executando é interrompido em favor do novo thread, de maior prioridade;
- Seu método run() termina, ou ele cede a vez através do yield();
- O ambiente de execução permite que seja implementada a política de "fatia de execução", isto é, entre threads de mesma prioridade cada um recebe um fatia de tempo, e sai da CPU quando sua fatia terminou. Este é o caso, por exemplo, do Windows 95/NT.

Mas não há garantia de que sempre o thread de maior prioridade esteja rodando, pois o escalonador pode decidir escolher um thread e menor prioridade a fim de evitar que este espere para sempre [Tutorial Java].

O uso de prioridades é útil para incentivar que um dos threads consiga continuar progredindo em sua execução. Por exemplo, se sua aplicação executa uma simulação e mostra uma visualização da mesma, é importante que o thread de visualização tenha chance de executar frequentemente, de forma que a visualização esteja atualizada. Por exemplo, para simular o movimento (aleatório) de *n* formigas em uma área retangular podemos usar um thread para cada formiga, com cada um desses threads mantendo disponível a posição atual da formiga a que corresponde. Um thread extra, para visualização da simulação, consultaria a posição corrente de cada formiga e com esta informação alteraria o "display" da área retangular.

---

[9] Observe que a execução sempre começa com um thread, que não é criado de forma explícita pela aplicação. No caso de uma aplicação a ser executada da linha de comando, um thread é automaticamente criado para executar o método public static void main(String[] args).

Simplesmente atribuir a este thread de visualização uma prioridade maior do que as das prioridades dos threads das formigas não leva a uma solução correta, pois este thread teria lugar cativo em um processador, em detrimento dos threads das formigas (ou seja, só seria uma solução aceitável se o número de processadores for pelo menos *n+ 1).* Usualmente o tratamento dado a este thread envolve ganhar uma prioridade maior, mas também deliberadamente desocupar a CPU periodicamente, a fim de que o código de simulação das formigas possa executar. A operação *yield()*, disponível na classe *Thread*, só tem efeito para ceder a vez a outro thread de mesma prioridade. A fim de passar a CPU para um thread de menor prioridade, o thread de maior prioridade pode usar o método sleep(), que faz com que este pare de executar por um intervalo de tempo. Há duas formas disponíveis, uma em que a unidade de tempo é milisegundos e a outra em que se pode também especificar o número de nanosegundos*: static void sleep(long millis) e static void sleep (long millis, int nanos)*[10]. Esta parada temporária de execução pode ser interrompida pelo método interrupt(): se o thread t1 chamou sleep, outro thread t2 que esteja executando pode chamar t1.interrupt() e com isso interromper a "dormida" de t1. Quando isto ocorre, t1 retorna de sleep através do lançamento de uma interrupção; assim, é necessário que as chamadas de sleep lidem com a possibilidade de ser lançada a interrupção InterruptedException (vide o uso de sleep nos exemplos a seguir).

É necessário cuidado também para evitar que cada um dos threads relativos as formigas não atuem de forma "egoísta". Se todos são criados com a mesma prioridade, nos ambientes em que o suporte do sistema operacional para threads não envolva a alocação de fatias limitadas de tempo a cada thread, seria possível que uma formiga monopolizasse a CPU enquanto o thread de visualização está sem executar ("dormindo" por iniciativa própria). Isto pode ser evitado se cada um dos threads, após executar uma parte de sua simulação, cooperar com os outros através da chamada *yield*(), que faz com que a CPU passe para outro thread de mesma prioridade.

3.3 Sincronização em Java

Além do método sleep(), há outras formas de um thread tornar-se não executável: bloquear na execução de I/O (entrada e saída), ou especificar que ele deseja esperar até que uma condição seja satisfeita (um dos cenários desejáveis de sincronização de threads,como vimos na seção 2. O outro cenário de sincronização é a exclusão mútua no acesso a recursos compartilhados).

---

[10] O modificador "static" significa que a rotina pode ser invocada sem a especificação de um objeto específico.

A maior parte das aplicações de interesse exige que threads compartilhem informação e considerem o estado e as atividades sendo exercidas por outros threads antes de progredir com sua própria execução.

Java permite a definição de regiões críticas (seção 2.2) através da palavra-chave ***synchronized***. Regiões críticas, em que um mesmo objeto é acessado por threads concorrentes, podem ser métodos ou blocos de comandos. A implementação de Java associa com todo objeto (estejamos ou não usando threads) um lock, isto é, uma variável para exclusão mútua (como um semáforo inicializado com o valor 1, conforme discutido na seção 2.3). Introduziremos o uso de threads sincronizados em Java com um exemplo onipresente: produtor/consumidor[11]. A aplicação é formada por um thread que periodicamente (com intervalos aleatórios) produz números e dois threads que consomem números produzidos sempre calculando a média dos números já consumidos. O produtor e os consumidores precisam compartilhar um repositório de números produzidos a serem consumidos. O seguinte código descreve a funcionalidade do produtor, consumidor e do repositório, mas não leva em consideração corretamente os aspectos de concorrência:

```
public class ProdutorConsumidor { // classe com programa principal
  private Produtor    prod;       // implementa código do thread para produtor
  private Consumidor  cons1,
                      cons2; // implementa código do thread consumidor
  private Repositorio reposit;   // guarda números a serem consumidor

  // construtor recebe quantidade de elementos a serem prod/cons
  ProdutorConsumidor(int num_elementos) {
    reposit = new Repositorio(num_elementos);
    prod = new Produtor(num_elementos, reposit);
    int num_cada_consumidor = num_elementos/2;
    cons1 = new Consumidor(num_cada_consumidor, reposit);
    if (num_elementos %2 != 0) num_cada_consumidor++;
    cons2 = new Consumidor(num_cada_consumidor, reposit);
    new Thread(prod, "Produtor").start();       // cria thread produtor
    new Thread(cons1, "Consumidor 1").start();  // cria thread consumidor
    new Thread(cons2, "Consumidor 2").start();  // cria thread consumidor
  }
```

[11] Na aula do mini-curso uma versão mais gráfica e interessante deste exemplo é utilizada. Aqui, por brevidade, nos atemos ao aspecto algoritmo do problema. Todos os exemplos utilizados na aula se encontram no sítio deste texto (http://www.ime.usp.br/~dilma/jai99).

```
  public static void main(String[] args) {
    // Programa principal: só cria objeto ProdutorConsumidor, já que
    // o construtor desta classe cria os threads e começa execução.
    // Argumento do construtor – 10 – é a quantidade a ser prod/cons
    ProdutorConsumidor prog = new ProdutorConsumidor(10);
  }
}
// class Produtor: thread que gera número aleatório, adiciona-o no repositório
// compartilhado e fica um período aleatório sem executar.
class Produtor implements Runnable {
  private Repositorio reposit;
  private int     num_elementos;
  // construtor do Produtor recebe quem é o repositório e quantos elementos
  // devem ser produzidos.
  Produtor(int num, Repositorio reposit) {
    num_elementos = num;
    this.reposit = reposit;
  }

  public void run() {
    for (int i=0; i < num_elementos; i++) {
      int numero =(int) (Math.random() * 1000);      // gera número
      reposit.poe(numero);         // coloca na estrutura de dados
                                   // compartilhada
      System.out.println("Produtor colocou " + numero);
      // fica um tempinho sem fazer nada
      try {    // dorme
            Thread.currentThread().sleep ((long) (Math.random() * 100));
      } catch (InterruptedException e) { /* ignora */ }
    }
  }
}
// Consumidor
class Consumidor implements Runnable {
  private Repositorio reposit;
  private int soma = 0;      // mantém média dos elementos consumidos
  private int num_elementos;
  Consumidor(int num, Repositorio reposit) {
    num_elementos = num;
    this.reposit = reposit;
  }
```

```
  public void run() {
    int i = 0;
    while (i < num_elementos) {      // tem que consumir quantidade estipulada
      if (!reposit.vazio()) {
            // repositório tem algo a ser consumido
            int numero = reposit.tira();
            soma += numero;
            i++;
            System.out.println(Thread.currentThread().getName() +
                  " tirou " + numero + ".Media " + (double) soma / i);
      }
    }
  }
}
// Estrutura de dados que armazena números a serem consumidos
class Repositorio {
  int num_elementos;      // tamanho da estrutura
  int in = 0, out = 0;    // indicadores da próxima posição onde serão
                          // colocados/retirados elementos
  int [ ] vetor;          // local de armazenamento
  Repositorio(int num) {
    num_elementos = num;
    vetor = new int[num];
  }
  void poe (int num) {
    vetor[in++] = num;
  }

  boolean vazio () {
    if (in == out)
      return true;
    else
      return false;
  }

  int tira () {
    return(vetor[out++]);
  }
}
```

Um dos problemas com este código é, obviamente, a falta de cuidado com o acesso concorrente ao recurso compartilhado (repositório de números). O thread

consumidor tenta garantir que existam elementos no repositório antes de invocar a retirada, mas notem que o seguinte escalonamento é possível:

Programa principal	prod	cons1	cons2
cria prod,cons1,cons2			
	insere 1 número		
		vazia?não!	
			vazia?não!
		retira 1 número	
			tenta retirar:ERRO

Seria possível ter os dois consumidores retirando o mesmo elemento ao mesmo tempo (e portanto um elemento restaria no final sem ser consumido). Se tivéssemos vários produtores, poderia ocorrer deles tentarem colocar um novo elemento exatamente na mesma posição do vetor de armazenamento, perdendo-se um dos números.

É necessário então delimitar que partes do acesso ao repositório deve ser prevenida de acesso concorrente. A fim de lidar evitar que duas remoções (chamada reposit.tira()) interfiram uma com a outra, este método deve ser executado por no máximo um thread. Se também queremos permitir a possibilidade de termos vários produtores, poe() também deve ser protegido contra acesso concorrente. Quando ao método que checa se o repositório está vazio, também devemos evitar o seguinte cenário:

Thread cons1	Thread cons2
/*chama vazia() no caso	
em que há um elemento: */	
in==out ? NÃO!!	
	/* chama vazia(), ainda há
	um elemento*/
	in == out ? NÃO ∴ ret false
	out++ e retorna elemento
∴ ret false	
out++ e retorna elemento: ERRO !	

Cenários como este indicam que a operação *vazia* deve ser executada com exclusão mútua: enquanto ela executa nenhum outro thread pode estar executando algum métodos do objeto *reposit.*. Da mesma forma, devem ser mutuamente exclusivas as execuções de *retira* e *põe*.

A fim de definir que a estrutura de dados só pode ser manipulada por um de seus métodos de cada vez, basta declarar cada um dos métodos com o modificador *synchronized*:

```
class Repositorio     {
        synchronized void poe (int num)   { … }
        synchronized boolean vazia ()     { … }
        synchronized int tira()           {…}
}
```

Como todos os métodos da classe foram declarados como sincronizados, a classe funciona como um Monitor (seção 2.4).

O objeto *reposit* possui um lock interno, isto é, uma variável que indica se o objeto compartilhado está livre ou ocupado. A execução de um método *synchronized* (por exemplo, reposit.tira() invocada pelo thread cons1) segue o seguinte protocolo: se o lock de reposit está disponível, o lock passa a ficar de posse do thread cons1, e cons1 prossegue executando tira(). Ao final da execução, cons1 libera o lock. Se, por outro lado, o lock estiver ocupado (de posse de algum outro thread, por exemplo cons2), então cons1 fica bloqueado, aguardando a liberação do lock.

A obtenção e liberação de um lock são feitas automaticamente pelo sistema de execução da linguagem Java, e de forma atômica (sem interrupção). A implementação suporta sincronização reentrante, isto é, um thread requisitar um lock que não está disponível por estar exatamente com ele:

```
public class Reentrante {
        public synchronized void a () {
                b();        // chama b, que também é synchronized
                System.out.println("em a()");
        }
        public synchronized void b() {
                System.out.println("em b()");
        }
}
```

A chamada "new Reentrante().a()" resulta na impressão de:

```
em a()
em b()
```

Voltando ao problema do produtor/consumidor, será que agora que tornamos o acesso ao repositório disponível a no máximo um thread por vez, corrigimos o problema de sincronização? Vejamos o código do Consumidor:

```
class Consumidor implements Runnable {
  private Repositorio reposit;
  private int soma = 0,  num_elementos;
  Consumidor(int num, Repositorio reposit) {   . . .   }
   public void run() {
     int i = 0;
     while (i < num_elementos) {      // tem que consumir quantidade estipulada
      if (!reposit.vazio()) {
              // repositório tem algo a ser consumido
              int numero = reposit.tira();
              soma += numero;
              i++;
              System.out.println(Thread.currentThread().getName() +
                    " tirou " + numero + ".Media " + (double) soma / i);
       }
      }
    }
  }
```

O trecho destacado permite o seguinte cenário se o repositório contem apenas um inteiro:

Thread cons1	Thread cons2
if (!reposit.vazio ( ))	
/* vazio( ) retornou falso, já	
que há um elemento */	if (!reposit.vazio( ))
	/* vazio( ) retornou falso, já
int num = reposit.tira ( )	
	int num = reposit.tira ( ); **ERRO !!**

É desejável que as operações vazia ( ) e tira ( ) sejam executadas atomicamente, ou seja, uma vez constatado que há elemento disponível no repositório, o mesmo seja retirado sem que haja chance de intervenção por parte de algum outro thread. Incluíremos o método seDerTira ( ) na classe Repositorio, de forma a permitir que o método infome a disponibilidade e já entregue um valor se for o caso:

```
synchronized Integer seDeTira ( )  {
```

```
        if (vazio ( ) )
              return null;
        else
              return new Integer ( vetor[out++] );
  }
```

Os consumidores usam este novo acesso ao repositório da seguinte forma:

```
  class Consumidor implements Runnable {
    < . . . >
    public void run() {
      int i = 0;
      Integer valor ;
      while (i < num_elementos) {      // tem que consumir quantidade estipulada
       if ( (  valor =  seDerTira ( ) ) != null ) ) {
              // repositório forneceu um valor
              soma += valor.intValue();
              i++;
              System.out.println(Thread.currentThread().getName() +
                    " tirou " + numero + ".Media " + (double) soma / i);
       }
      }
    }
  }
```

Vale a pena observar que se o ambiente em que executarmos o programa não implementa fatias de tempo para cada um dos threads, o seguinte poderia ocorrer: após os três threads (de mesma prioridade) terem sido criados, o escalonador escolhe cons1 para executar; cons1 chama vazia() e verifica que não há elemento a ser consumido, volta para o início do loop e verifica novamente que não há elemento. Como cons1 não sai da CPU, prod nunca conseguiria executar. Isto pode ser evitado se o thread consumidor, ao notar que o repositório está vazio, ceder sua vez para outro thread (trecho destacado com o retângulo):

```
  while (i < num_elementos) { // tem que consumir quantidade estipulada
      if (!reposit.vazio()) {
              // repositório tem algo a ser consumido
              int numero = reposit.tira();
              soma += numero;
              i++;
              System.out.println(Thread.currentThread().getName() +
                    " tirou " + numero + ".Media " + (double) soma / I);
```

```
        }
        else Thread.currentThread().yield();
      }
    }
```

Outra alternativa seria dar mais prioridade ao thread prod.

Além de usar o modificador synchronized nas definições de métodos, é possível proteger trechos de programa, isto é, garantir que um thread só executa o trecho se estiver de posse de um lock específico. Sintaticamente, fica:

```
Object obj = …;      // algum objeto
synchronized (obj) {
      <lista de comandos>
}
```

Se o lock associado ao objeto obj está disponível (isto é, nenhum thread o tem, ou o próprio thread em execução o tem), <lista de comandos> é executado; senão o processo fica bloqueado até que o lock esteja disponível.

3.4 Sincronização condicional em Java

Sincronização condicional é quando desejamos que um thread fique bloqueado até que uma condição fique verdadeira. Em geral, não é viável implementar primitivas do estilo "ESPERE_ATÉ_QUE (condição booleana qualquer)", pois o sistema de execução teria que ficar constantemente monitorando o estado das variáveis utilizadas na condição booleana a fim de ver se esta se tornou verdadeira, aí desbloqueando o thread. Este tipo de sobrecarga no tempo de execução, na maior parte dos casos, é inaceitável. Java (e outros ambientes com suporte para programação concorrente), disponibiliza uma forma mais limitada de "espere até que": os métodos *wait* e *notify/notifyAll*.

O método *wait()* pode ser chamado por qualquer objeto, e faz com que o thread corrente espere até que outro thread chame o método *notify()* ou *notifyAll()* para este mesmo objeto. A restrição é que o método *wait* só pode ser chamado por threads que estejam de posse do monitor do objeto (isto é, do lock associado a este objeto). Se o thread invoca *wait()* sem estar de posse do lock, uma exceção é lançada retratando esta situação de erro. Lembremos que um thread pode obter posse de um lock executando um método ou bloco de código definido como *synchronized*. Pela própria natureza de um lock, no máximo um thread pode estar de posse de um lock a cada momento.

A chamada *obj.notify()* termina com a espera de um thread que tenha executado *obj.wait*(). Se houver mais de um thread na espera, um deles é arbitrariamente escolhido. Já *notifyAll()* acorda todos os threads que estejam esperando.

Estas primitivas de sincronização seriam úteis no problema produtor-consumidor abordado na seção anterior (3.3), pois permitiriam que os threads consumidores, ao invés de executarem uma espera ocupada (como definido na seção 2.2.1) em que seguidamente testam a existência de elementos a serem consumidos, esperassem ser notificados da produção de elementos. Em outras palavras, toda vez que o repositório compartilhado ganhasse algum novo elemento, este notificaria o thread que desejasse executar uma remoção.

O método wait é na verdade ainda mais poderoso, por permitir que o tempo de espera seja limitado: wait(long timeout) aguarda por notificação ou até que o período timeout (especificado em milisegundos) tenha passado. Está disponível também wait(long timeout, int nanos), permitindo especificar com granularidade de nanosegundos. A seguir apresentamos o ProdutorConsumidor utilizando sincronização via wait/notify, de forma que a CPU não fica indevidamente ocupada por consumidores na espera de valores. A grande diferença está no repositório, que não mais necessida do método seDerTira ( ), já que o usuário pode simplesmente requisitar um elemento (método tira ( )), ficando bloqueado até que seu pedido possa ser atendido:

```
class Repositorio     {
        < . . . >
        synchronized void poe ( in t num )  {
              vetor [ in++ ] = num;
              notify ( );
        }
        synchronized boolean vazio ( ) {
              if ( in == out )
                    return true;
              else
                    return false;
        }
        synchronized int tira ( ) {
              if ( vazio ( ) ) {
                    wait ( ) ;
              return vetor [ out++];
        }
 }
```

Outra primitiva de sincronização condicional disponível em Java (e em bibliotecas como POSIX pthreads) é o join: quando um thread t1 invoca t2.join(), t1 fica bloqueado até que a execução de t2 termine. Se t2 foi interrompido, t1 receberia

esta informação via uma exceção. A espera pode, como no caso do wait, ser limitada: t2.join(long timout) faz com que t1 espere por no máximo timeout milisegundos pelo término de t2[12]; join(long timeout, int nanos) permite que nanosegundos sejam especificados. Um exemplo de uso do join no exemplo ProdutorConsumidor seria termos o programa principal esperando pelo término dos threads por ele criados (prod, cons1, cons2) a fim de coletar estatísticas finais sobre os números gerados e consumidos.

3.5 Grupos de Threads em Java

Todo thread em Java é membro de um *grupo de threads*. A utilidade de grupo de threads é tê-los colecionados em um único objeto que pode ser manipulado de forma a afetar a coleção inteira, por exemplo, interrompendo a execução de todos os threads do grupo ou assertindo uma espera até que uma dada fração dos threads do grupo já tenha terminado sua execução. Outro aspecto muito importante dos grupos de threads está relacionado com segurança: o agrupamento de threads permite delimitar o escopo em que threads podem ser controlados por outros threads. Por exemplo, um thread só pode alterar a prioridade de outros threads do mesmo grupo.

No momento de criação de um thread, é possível especificar a que grupo de threads deve ser alocado o novo thread. Se nenhum grupo é especificado (como nos exemplos que vimos até agora), o thread criado nasce em um grupo de threads padrão. Quando uma aplicação começa a executar, o sistema de execução de Java cria um ThreadGroup chamado main. Se o thread é criado em um applet, o grupo do novo thread pode não ser o main (vai depender do browser ou viewer sendo utilizado). Muitos browsers alocam um thread para cada applet na página, usando este thread para executar todos os métodos principais do applet (init, start, stop).

ThreadGroups podem ser membros de ThreadGroups, permitindo uma estruturação hierárquica dos threads.

3.6 Daemons

Daemons são threads que continuam executando enquanto tiver algum thread (que não seja daemon) ativo. Threads podem ser especificados como daemons através do método setDaemon ( )., que deve ser chamado antes do método start ( ) .

3.7 Parando Threads em Java

O suporte para controle da execução de threads não se restringe apenas a start, wait/notify, sleep/interrupt. Até a versão Java 1.1, estavam disponíveis métodos

[12] É possível, através do método isAlive(), descobrir se um dado thread já terminou sua execução ou não.

para suspender um thread (suspend), reativá-lo (resume) e terminá-lo (stop). Estes métodos agora não fazem parte da interface disponível em Java 1.2. Eles foram eliminados por serem inerentemente inseguros. Por exemplo, o método stop() – para parar um thread – fazia com que o thread liberasse todos os locks que estivessem em sua posse, podendo permitir que objetos protegidos por estes locks fossem liberados ainda estando em um estado inconsistente. Este comportamento se manifesta de forma muito sutil e é difícil detectar e corrigir problemas relacionados a locks liberados antes da hora.

É recomendável que programadores que desejam cessar a execução de um thread utilizem uma variável para indicar que o thread deveria parar de rodar (isto é, sair do método run() graciosamente, ao invés de sair como resultado da chamada de stop()). O thread t a ser parado deve checar regularmente esta variável, e o thread que deseja causar o término de t deve modificar o valor da variável no momento desejado. A fim de garantir uma sincronização correta entre o thread que quer terminar t e o próprio thread t a ser terminado, é importante que a variável utilizada para esta comunicação seja declarada como *volatile*. Como na linguagem C, o uso deste modificador proibe que o compilador faça otimizações de código baseadas em consultar o valor da variável através de registradores (o que resultaria em problema, pois o efeito de outro thread alterando a variável não seria percebido até que o código voltasse a checar a variável e não o conteúdo do registrador). A declaração *volatile* indica que pode haver acesso concorrente a variável não sincronizado. O seguinte código, disponível em [Tutorial Java], exemplifica esta forma de causar o término de threads sem utilizar o método stop dentro da definição de um applet com os seguintes métodos *start*, *run* e *pára*::

```
private volatile Thread blinker;       // Thread rodando no applet que faz com
            // que elemento gráfico fique piscando. A idéia é que esta variável
            // guarde a referência do Thread, e que fique nula como indicação
            // de que o thread deve morrer.
public void start() {
      blinker = new Thread (this);
      blinker.start();
}
public void run() {
      Thread thisThread = Thread.currentThread();
      // Uma versão baseada em stop teria while (true) no corpo do run,
      // indicando que o método roda para sempre, até que seja “morto”.
      // Aqui faremos com que ele morra graciosamente, na hora certa.
      while (blinker == thisThread)  {    // enquanto não ficou nulo
      // observe que blinker, por estar em um loop, poderia ter sido alocada
      // a um registrador, não fosse sua alocação declarada como volatile, o que
```

```
        // garante que todo acesso a esta variável causa um fetch da memória.
            try { // temos que usar try porque sleep pode ser interropido
                thisThread.sleep(intervalo);
            } catch (InterruptedException e) { /*ignora interrupção */}
            repaint();      //      chama paint, que causa o efeito visual desejado
    }
    public void pára()    {
        // usar blinker.stop() seria inseguro (além do compilador nos avisar que
        // método está "deprecated". Utilizaremos variável blinker.
        blinker = null;
    }
```

Observe que a chamada ao método sleep em run() indica um intervalo entre checagens. Se o intervalor for grande, bastante tempo poderia passar antes que o thread percebesse que deveria morrer. Para esses casos, o método de término do thread (pára(), no exemplo) poderia, além de alterar o valor da variável de controle, interromper a "dormida" através de interrupt().

Além de Thread.stop(), os métodos Thread.suspend() e Thread.resume ( ) também foram eliminados na Java 1.2. O motivo é que estes métodos eram inerentemente passíveis de deadlock (seção 2.2): se o thread a ser suspendido estava de posse de um lock correspondente a um objeto compartilhado crítico ao sistema no momento em que foi suspenso, nenhum outro thread poderá acessar o recurso até que o thread fosse reativado (através de resume()). Se mostrou comum que o thread que iria chamar o resume() tentasse antes obter o lock do objeto compartilhado, e portanto ficava esperando por um evento (a liberação do lock) que só pode ser gerado pelo thread que espera pelo resume, vivenciando assim uma típica situação de deadlock. Recomenda-se que o efeito de desativação temporária de threads obtido pelo suspend/resume seja implementado via wait/notify, pois com o wait() há a liberação automática do lock, até o retorno à execução. Vejamos um exemplo:

```
    private volatile boolean threadSuspenso;
    public void suspende() {
        threadSuspenso = true;
    }
    public void desuspende() {
        threadSuspenso = false;
    }
    public void run() {
        while (true) {          // suponha que o thread vai executar para sempre
            try {       // vamos chamar sleep, e aí ver se foi suspenso ou não
                Thread.currentThread().sleep(intervalo);
```

```
                    // chamaremos wait se suspenso, o que exige posse do lock
                    synchronized (this) {
                        while (threadSuspenso)
                            wait();
                    }     // fim do trecho sincronizado
                } catch (InterruptedException e) { }
            }
        }
```

Esta solução tem o inconveniente de requerer "synchronized(this)" toda vez que o sleep() termina sua execução. Métodos de sincronização em Java são caros! Uma solução melhor seria requerer o lock do objeto só se percebe-se que a variável threadSuspenso está verdadeira, ou seja, só no caso em que há chance[13] de querermos chamar o wait() e por isso precisamos do lock. Assim, o loop (delimitado por um retângulo) poderia ser substituído por:

```
if (threadSuspenso) {
    synchronized (this) {
        while (threadSuspenso)
            wait();
    }
}
```

4. Desafios: Corretude e Desempenho

Há dois aspectos de prevenção de interferência entre threads e duas propriedades correspondentes desejáveis em software concorrente [Lea 1997]:

- *Safety*: a propriedade de que nada "ruim" ocorre.
- *Liveness*: a propriedade de que alguma hora, algo "bom" acontece.

Falhas de segurança (safety) levam a comportamento indesejável do software, com funcionamento errôneo. Falhas de "vivacidade" (liveness) levam à ausência de comportamento esperado, isto é, algo pedido pelo usuário ou pelo software nunca tendo a chance de ser finalmente executado. É uma prática padrão em engenharia enfatizar primeiramente o aspecto de safety do projeto, evitando que seu comportamento leve a um comportamento aleatório e talvez perigoso. Por outro lado, a maioria do tempo empreendido no ajuste de programas concorrentes está relacionado com aspectos de eficiência e liveness, algumas vezes sacrificando

[13] Note que existe a chance de checarmos a variável threadSuspenso, vermos que está verdadeira, mas durante o tempo que levamos para conseguir executar o synchronized(this), a variável já ter sido alterada pela chamada de desuspende().

segurança em nome da eficiência ou garantia do progresso de execução de forma bem fundamentada. O grande desafio é conseguir ter os dois aspectos funcionamento bem.

Duas propriedades importantes relacionadas à segurança são a ausência de deadlock e a exclusão mútua no acesso a recursos compartilhados críticos. Para o caso do deadlock[14], algo "ruim" significa ter-se threads esperando por eventos que nunca ocorrerão; no caso de exclusão mútua, "ruim" é ter-se dois ou mais threads executando partes de regiões críticas ao mesmo tempo. Exemplos da propriedade de liveness são a garantia de que o requisito por algum serviço será alguma hora atendido, que algum thread aguardando pela entrada na região crítica vai alguma hora ter sucesso.

Do ponto de vista de corretude, provar que propriedades de safety são atendidas significa modelar o programa concorrente em termos de ações baseadas em um estado, e provar que as ações executadas pelo programa preservam as propriedades desejadas, isto é, que predicados desejáveis sobre o estado do programa se mantêm válidos durante a execução. Já provar que a propriedade de liveness é respeitada implica em garantir que a política de escalonamento (isto é, da escolha das ações a serem executadas a cada momento) é justa. Diz-se que uma política de escalonamento é *incondicionalmente justa* se toda ação atômica que não depende de outras (isto é, é incondicionalmente executada) é elegível para ser executada em algum momento. Já *justiça fraca* descreve a situação em que a política é incondicionalmente justa e também toda ação condicional é elegível para execução alguma hora se sua condição se tornar verdadeira e não se tornar falsa novamente a não ser pelo efeito da própria ação. Justiça forte implica em ser incondicionalmente justa e que toda ação condicional atômica consiga executar alguma hora, assumindo que sua condição fica verdadeira um número infinito de vezes.

Na prática, existem estratégias conservadoras de desenvolvimento que inerentemente resultam em projetos seguros:

- Imutabilidade: se evitamos mudanças de estado, evitamos que problemas de exclusão mútua ocorram. Java, por exemplo, fornece meios com os quais especificar que determinadas variáveis não devem nunca mudar de valor. Mas, obviamente, na maior parte das aplicações é impossível obter a funcionalidade desejada lidando apenas com constantes.
- Sincronização: se dinamicamente se garante acesso exclusivo, de forma métodica, o problema de segurança está em parte resolvido. Por exemplo, se

[14] Observe que deadlock está também relacionado a liveness, já que não permite que nada ocorra dali em diante.

metodicamente se usa sincronização total (isto é, todo e qualquer método relacionado a um recurso compartilhado é declarado como synchronized), a chance de conflitos entre acessos é eliminada. Mas também a chance de execução concorrente é eliminada. Com sincronização parcial, poderia-se metodicamente sincronizar somente os trechos de códigos em que variáveis mutáveis são manipuladas. Mas ainda assim pode-se estar limitando em demasiado a concorrência.

- Agregação: se estruturalmente garantimos que para se ter acesso a uma parte de um recurso compartilhado, é necessário ter acesso ao elemento que contém todas as partes, os problemas de exclusão mútua ficam mais raros. Os objetos compartilhados são então estruturados de forma em que um contenha o outro, e seja dono de seus objetos internos. Assim, um método que tenha obtido acesso exclusivo ao recurso "de fora" (o que contém outros), ganha de graça acesso exclusivo aos objetos internos. Em outras palavras, este enfoque limita o compartilhamento de objetos, exigindo que o compartilhamento seja hierárquico.

Os problemas relacionados a liveness são tão sérios quanto os de safety, e em muitas vezes bem mais difíceis de serem detectados. Threads podem falhar em termos de "vivacidade" devido a disputas por recursos compartilhados (em que o thread repetidamente perde a disputa), dormência (um wait em que o correspondente notify nunca é gerado) e deadlock (dois ou mais threads bloqueiam-se mutuamente enquanto tentando obter locks e assim continuar a atividade). Evitar deadlocks é importante durante o projeto de programas concorrentes. Uma forma simples e muito recomendada é prevenir deadlocks através de uma ordenação das variáveis de condição (usadas em sincronização condicional) usadas no programa, isto é, exigir que um thread passe por algumas sincronizações condicionais antes de passar por outras. Este enfoque foi amplamente estudado pela comunidade de banco de dados, gerando protocolos de controle de concorrência como o Two Phase Locking (2PL, locking em duas fases), em que nenhum lock pode ser liberado até que todos os necessários para a atividade tenham sido obtidos.

É necessário que haja um balanceameto entre as medidas adotadas para lidar com safety e as adotadas para lidar com liveness. Em geral, um enfoque estrito para lidar com safety aumenta o potencial para problemas de liveness (além de simplesmente diminuir a concorrência), e um enfoque em que se tente colocar mais threads executando aumenta a chance que eles interfiram de forma indesejada, gerando problemas de safety.

5. Concorrência, Applets e Objetos Distribuídos

Se entendermos o termo "programação concorrente para a internet" como a disponibilidade de aplicações concorrentes na internet, de forma que estas estejam acessíveis e possam ser executadas a partir de browsers, é muito provável que a aplicação lide com recursos que existem antes, durante e depois a utilização da aplicação via um applet. Um objeto derivado da classe applet pode ser utilizado a partir de um browser porque o objeto é enviado pela rede até a máquina local do usuário, e os métodos do objeto são executados pelo browser conforme requisitado pelo usuário do applet. Existem várias restrições sobre que tipo de processamento pode ser feito nos applets, a fim de evitar que o applet atue de forma indesejada no ambiente de seu usuário (por exemplo, deletando arquivos ou aproveitando sua presença no ambiente do usuário a fim de obter informação local). Observe que nas aplicações motivadoras, descritas na seção 1.3, a natureza do processamento desejado não permite que simplesmente a aplicação migre para o ambiente do usuário, pois outros usuários concorrentes precisam utilizar os mesmos recursos. Em outras palavras, o applet não é simplesmente concorrente por ser formado por vários threads que juntos atingem o fim esperado, mas os vários applets concorrentemente carregados por usuários da aplicação concorrem pelos recursos. Por exemplo, no caso da aplicação que ajuda a organizar uma festa (seção 1.3), o usuário da aplicação deseja escolher que comida levará a festa de forma concorrente a outros usuários da aplicação (que tenham, como ele, deixado para visitar a página da festa poucas horas antes da mesma ☺). Proteger um método escolhe_comida() disponível no applet através de um synchronized não resolve o problema, já que em outros applets, executando em outros lugares, o mesmo método pode estar sendo chamado, e o synchronized não tem efeito cascata pela rede de computadores. A fim de que o recurso compartilhado (lista das guloseimas que ninguém ainda escolheu levar) seja protegido de acesso concorrente a partir de qualquer lugar da rede, ele deve estar encapsulado em um único objeto, e todo applet que deseja usar este objeto deve chamar o método do objeto *remotamente*, isto é, o método é chamado como se o objeto ListaGuloseimas fosse local ao applet, mas o efeito é que a chamada seja executada na máquina que hospeda este objeto.

Em http://www.ime.usp.br/~dilma/jai99 você encontrará as aplicações descritas na seção 1.3 detalhadamente documentadas, de forma que você possa desenvolver seus objetos concorrentes com acesso remoto sem necessariamente estudar computação distribuída. Aqui no texto é incluído, por questão de espaço, um exemplo bem mais simples: a aplicação é formada por um objeto cuja funcionalidade é somar um ao inteiro que ele armazena. O objeto pode ser acessado concorrentemente via applets. A implementação descrita a seguir é formada pelos seguintes módulos (arquivos):

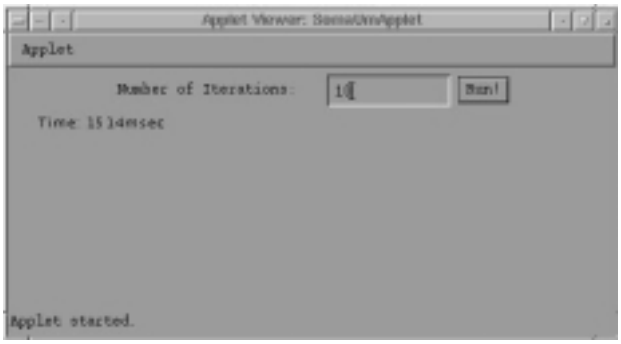
- SomaUm.html, que disponibiliza a aplicação para usuários da Teia
- SomaUmApplet.java, que implementa o applet que está embutido na página SomaUm.html
- SomaUm.java, SomaUmImpl.java e policy, que implementam um objeto servidor, a ser utilizado pelos clientes via applet. O acesso a este objeto, que reside na máquina sushi.ime.usp.br, é feito através suporte fornecido na linguagem Java para chamada de métodos em objetos remotos (RMI, remote method invocation).

Comecemos olhando o fonte SomaUm.html, a partir do qual o usuário pode usar nossa aplicação e solicitar que o inteiro seja incrementado:

```
<html> <head><title>SomaUm</title></head>
<body>
<center><h1>SomaUm</h1> </center>
<applet codebase="executaveis/"
      code="SomaUmApplet"
      width=500 height=200>
</applet>
</body> </html>
```

A inclusão do applet está especificada no trecho delimitado pelo retângulo, indicando que o código do applet está no subdiretório “executaveis”, a partir do diretório que contem o arquivo SomaUm.html. Foram também fornecidos o nome do código a ser carregado e a área que deve ser ocupada pelo applet na página sendo visualizada pelo browser.

A classe SomaUmApplet implementa a funcionalidade do applet, e é descrita a seguir. Através do applet, o usuário pode especificar o número de vezes em que ele deseja invocar o método remoto; para cada chamada é criado um thread. O applet vai mostrando o valor retornado a cada iteração, e no final o tempo total que o processo levou. O applet tem a seguinte aparência:



O código correspondente segue abaixo.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.rmi.Naming;                 // Necessário porque este applet vai
                                        // procurar acessar um objeto em outra
                                        // máquina
import java.rmi.RemoteException;  // Importa definição de exceções geradas
                                        // durante chamada remota de um método


public class SomaUmApplet extends Applet implements Runnable{
  String message = "blank";     // guarda mensagem devolvida pelo obj remoto
  int nb_iterations = 0;              // usuário escolhe quantas chamadas devem ser
                                        // feitas.
  // "obj" é o identificador usado para nos referirmos ao objeto remoto, isto é,
  // ele armazena uma referência para o objeto remoto.
  SomaUm obj = null;
  Label label = new Label("Number of Iterations: ");     // elemento gráfico
  TextField texto = new TextField("10", 12);         // para entrada de num. iter.
  Button b = new Button("Run!");           // elemento gráfico em que usuário
                                        // indica início do trabalho do applet.
  public void run() {         // código a ser executado por um thread
    long delta0 = System.currentTimeMillis();         // pega hora atual e guarda.
    for (int i=0; i < nb_iterations; i++) {   // nb_iterations foi dado pelo usuário
    try {      // chamaremos método remoto, o que pode lançar exceção.
          message = obj.incr();   // aqui está a chamada do método.
          repaint();                    // atualiza a tela (veja paint a seguir).
      } catch (Exception e) {      // se exceção for lançada, trata-a
     System.out.println("SomaUmApplet exception in rmi call incr(): " +
                 e.getMessage());
                e.printStackTrace();
      }
    }
    long total = System.currentTimeMillis() - delta0;      // calcula tempo
                                                           // gasto.
    message = "Time: " + total + "msec";
    repaint();                                             // atualiza tela.
  }
  public void init() {         // método a ser executado na inicialização do
```

```
                        // applet.
final SomaUmApplet thisapplet = this;     // sendo uma variável
                                          // com valor final, podemos
                                          // utilizar dentro do código a
                                          // ser defindo para o botão
                                          // “Run”.
add(label);             // coloca elementos gráficos
add(texto);
add(b);
```

```
  // Define, instancia e registra um objeto (ActionListener) para lidar
  // com a pressão do botão que colocamos na interface.
  b.addActionListener( new ActionListener () {
    public void actionPerformed(ActionEvent e) {       // aqui vai o código
                        // a ser executado quando o botão é pressionado.
       Graphics g = getGraphics();         // precisamos acionar a parte gráfica
        g.drawString("                     ", 30, 60); // limpamos campo em
                                                       // msg é escrita
        String nbstr = texto.getText();                // texto digitado pelo usuário
        int nb = 0;
        boolean gotvalue = true;
        try {      // vamos tentar transformar string em número; temos que
                   // estar preparados para lidar com textos não "traduzíveis".
           nb = Integer.parseInt(nbstr);
       } catch ( NumberFormatException exc) {
            gotvalue = false;            // texto que o usuário digitou não
                                         // corresponde a número.
            g.setColor(Color.red);    // Colomos msg de erro.
            g.drawString("Number is not valid.", 40, 70);
            g.setColor(Color.black);
       }
       if (gotvalue) { // conseguiu número
          thisapplet.nb_iterations = nb;     // acerta valor no objeto.
          new Thread(thisapplet).start();  // cria thread para executar as iterações
       }
    }
  });
  resize(500, 200);         // acerta tamanho do applet na página.
 }
 public void start() {       // parte executada quando o applet começa a trabalhar
    try {
     // Primeiro, acha o objeto remoto, preparado para não dar certo …
     obj = (SomaUm)Naming.lookup("//" + getCodeBase().getHost()
                    + "/SomaUmServer");
   }catch (Exception e) {
         System.out.println("SomaUmApplet exception in Naming.lookup: " +
              e.getMessage());
         e.printStackTrace();
   }
}
```

```
   public void paint(Graphics g) {    // usado para atualizar a imagem do applet
    g.drawString(message, 25, 50);
   }
  }
```

SomaUm.java apenas define a interface oferecida pelo objeto remoto:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface SomaUm extends Remote {
   String incr() throws RemoteException;
}
```

SomaUmImpl.java efetivamente implementa o objeto remoto:

```
// importa classes necessárias por ser objeto remoto.
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;              // lida com entrada e saída via arquivo.
import java.lang.*;
public class SomaUmImpl extends UnicastRemoteObject //
                    // Ele estende a classe UnicastRemoteObject
                    // de forma a comunicar-se via socket e estar
                    // executando durante todo o tempo. A alternativa
                    // é usar Remote Object Activation, mas aqui iremos
                    // pelo jeito mais simples.
        implements SomaUm {// garante que método incr está disponível.
  int num = 1;   // cria variável num com valor inicial um. Este é o valor que
                    // será incremetado a cada execução do método incr().
  public SomaUmImpl() throws RemoteException {
   super();// criação simplesmente delega trabalho para superclasse.
  }
```

```
public String incr() {     // Aqui está o método que exportaremos para
                           // o applet.
        num++;
        // Só para exemplificar como mexer em arquivo aqui no objeto
        // servidor, vamos acrescentar este número a um arquivo de nome
        // ./data. Observe que o “policy” definido para este código estabeleceu
        // que este programa pode ler e escrever neste arquivo.
        String filename = new String("/home/mac/dilma/www/java/testes/) +
                  String(“servidorRemoto/SomaUm/data");
        FileWriter out = null;
        try {
          out = new FileWriter(filename,true);
        }   catch (IOException e) {
              System.out.println("Nao conseguiu abrir para escrita.\n");
            }
            catch (SecurityException e) {
              System.out.println("Problema de seguranca na escrita.\n");
            }             // Abriu data para escrita do tipo “append”.
        String numst = new Integer(num).toString() + "\n";
        try {             // escrita pode gerar exceção
          out.write(numst,0,numst.length());
        } catch (IOException e) {
                 System.out.println("Nao conseguiu escrever o int.\n");
         }
        try {     // fechar o arquivo também pode dar problema.
         out.close();
       }
      catch (IOException e) {
          System.out.println("Nao conseguiu fechar.\n");
      }
 return  "Agora tem " + num;       // Valor retornado para quem ativou
                                   // a execução deste método.
}
public static void main(String [ ] args) {
  // Cria e instancia um gerente de segurança.
  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
  }
```

```
    try {
      SomaUmImpl obj = new SomaUmImpl();
      // Associa este objeto com o nome "SomaUmServer"
      Naming.rebind("//sushi.ime.usp.br/SomaUmServer", obj);
      // acima foi usada porta default 1099
      System.out.println("SomaUmServer bound in registry");
    } catch (Exception e) {
            System.out.println("SomaUmImpl err: " + e.getMessage());
           e.printStackTrace();
    }
}
```

A fim de evitar problemas de segurança, foi criado com a ferramenta *policytool* a seguinte classe que especifica os direitos fornecidos a SomaUmImpl:

```
/* AUTOMATICALLY GENERATED ON Wed May 05 18:36:09 GMT-
03:00 1999*/
/* DO NOT EDIT */
grant {
  permission java.io.FilePermission
      "/home/mac/dilma/www/java/testes/servidorRemoto/SomaUm/data",
      "read, write";
  permission java.net.SocketPermission
     "sushi.ime.usp.br",
     "accept, connect, listen, resolve";
};
```

6. Modelagem e Projeto

Embora a maioria dos programadores lide com threads concorrentes de forma *ad hoc*, sem apoio de métodos ou técnicas sistematizadas, dois enfoques principais para modelagem e projeto de sistemas concorrentes são apresentados na literatura [Nielsen-Shumate 1987, Simpson 1986, Gomaa 1993, Sanden 1997]. O primeiro baseia-se em decompor o sistema em módulos que expressem transformações. A decomposição é expressa em um diagrama de fluxo de dados (DFD) que capture as transformações sucessivas, se desejado em diferentes níveis de abstração. As notações usuais da análise estruturada para DFDs são estendidas de forma a incluir informação de sincronização entre transformações. O segundo enfoque aborda a modelagem concentrando-se nos aspectos inerentemente concorrentes do problema, enfatizando o relacionameto entre o problema e a solução ao invés da série de passos que leva o primeiro ao segundo. Por exemplo, o método ELM (Entiy-life modeling) [Sandem 1997] assume que um sistema deve reagir a eventos (ou mesmo gerar eventos) do ambiente do problema, onde cada evento, embora

não estando necessariamente associado a tempo ou duração, requer uma quantidade finita de processamento. O desenvolvedor fazendo a modelagem deve identificar quais são os eventos do sistema, e atribuir os eventos a diferentes threads. Uma forma de fazer esta atribuição é criar uma linha imaginária de eventos (com várias bifurcações na linha, se for o caso), e particionar o rastreio de eventos representados pelas linhas entre threads, de forma que cada evento pertença a exatamente um thread e os eventos em um mesmo thread estejam suficientemente separados de forma a haver tempo para lidar processar cada um deles. Estes particionamentos de eventos em threads (*threads model*) sugerem a estrutura da solução de software. Um modelo de threads é considerado minimal se há um ponto na execução em que todos os threads têm um evento. O número de threads em tal solução minimal indica o nível de concorrência do problema. Além dos threads, o modelo pode conter objetos compartilhados (ou não) entre os threads. A idéia principal é, sempre que possível, esconder aspectos de exclusão mútua (definição de regiões críticas) dentro dos objetos.

Recentemente, o uso de padrões de projeto (*design patterns*) tem se mostrado útil para ajudar a estruturar o projeto de programas concorrentes. Um padrão descreve uma forma de projeto, usualmente uma estrutura de objetos (também conhecida como micro-arquitetura) consistindo de uma ou mais interfaces, classes ou objetos que obedecem a restrições e relacionamentos particulares (estáticos ou dinâmicos) ao problema. Patterns capturam soluções padrões e suas variações, levando ao reuso de componentes e arcabouços (frameworks). O livro de Doug Lea [Lea 1997] enfatiza o uso de padrões como uma forma de diminuir a distância entre teoria e prática de programação concorrente. Segundo Lea, a pesquisa na área de concorrência baseia-se em modelos e técnicas de difícil utilização no desenvolvimento de software do dia a dia (por exemplo, uso de modelos e notações formais em que problemas reais são de difícil derivação e manipulação incremental no ciclo de desenvolvimento). Muito da terminologia e notação adotada é uma adaptação do trabalho seminal de Gamma, Helm, Johnson e Vlissides [Gang-of-Four 1994]. A utilização de soluções para problemas recorrentes, como os apresentados por Lea, é um bom antídoto contra a tendência de desenvolver programas com threads com uma mentalidade de *hacker*.

A proliferação do uso da linguagem UML [UML 1998] e de métodos baseados em *casos de uso* [Unified 1999] no desenvolvimento de sistemas orientados a objetos também tem potencial para melhorar aspectos de modelagem e projeto de programas concorrentes em Java, já que a notação e os passos recomendados pelo método enfatizam a análise da interação entre os objetos concorrentes que formam o modelo de objetos.

7. Bibliografia

[Arnold-Gosling 1996] K. Arnold, J. Gosling. *The Java Programming Language*. Addison Wesley.

[Andrews 1991] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley.

[Bacon 1998] J. Bacon. *Concurrent Systems - Operating Systems, Database and Distributed Systems: An Integrated Approach*. Addison-Wesley, Segunda Edição.

[Butenhof 1997] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley.

[Eclipse 1998] B. Ozden, A. Silberschatz, J. Bruno e E. Gabber. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains". *Proc. USENIX Annual Technical Conference*, Seattle, Junho.

[Farley 1998] J. Farley. *Java Distributed Computing*. O´Reilly & Associates.

[Gang-of-Four 1994] E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Design Patterns*. Addison-Wesley.

[Gomaa 1993] H. gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Longman.

[Hansen 1973] B. Hansen. *Operating Systems Principles*. Prentice-Hall.

[Hansen 1973a] B. Hansen. Concurrent programming concepts. *ACM Computing Surveys, 5(4).*

[Hoare, 1974] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications da ACM, 17(10).*

[Kleiman 1995] S. Kleiman et al. *Programming with Threads*. Prentice Hall.

[Lea 1997] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley.

[Magee-Kramer 1999] J. Magee e J. Krammer. *Concurrency: State Models & Java Programs*. John Wiley & Sons.

[Milner 1995] R. Milner. *Communication and Concurrency*. Prentice Hall.

[Nielsen-Shumate 1987] K. W. Nielsen e K. Shumate. "Designing Large Real-time Systems with Ada", *Communications da ACM, 30(8),* pg 1073.

[Norton 1996] S. J. Norton et al. Thread Time: *The Multithreaded Programming Guide*. Prentice Hall.

[Ousterhout 1996] J. Ousterhout. Why Threads are a bad idea (for most purposes). Palestra convidada na USENIX Technical Conference, 25/jan/1996. Disponível na WWWem http://www.scriptics.com/people/john.ousterhout em Maio/1999.

[Rosu 1997] D. Rosu, M. B. Joses e M. Catalin Rosu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", *Proc. of the 16th ACM Symposium on Operating Systems Principles*, França.

[Sandem 1997] B. I. Sanden. "Modeling Concurrent Software", IEEE Software.

[Silberschatz-Peterson 1988] A. Silberschatz , I. Peterson. *Operating System Concepts.* Addison Wesley.

[Simpson 1986] H. Simpson. "The MASCOT Method". *IEE/BCS Software Eng. Journal, 1(3),* pg 102-120.

[Smart 1997] J. Nieh e M. Lam. "SMART UNIX SVR4 Support for Multimidia Apolications*", Proc. 16th ACM Symposium on Operating Systems Principles*, França.

[Tanenbaum 1992] A. S. Tanenbaum. *Modern Operating Systems*.Prentice Hall.

[Tutorial Java] Tutorial Java disponibilizado na WWW pela Java SunSoft. Disponível em http://java.sun.com/docs/books/tutorial/essential/threads em Maio/1999.

[UML 1998] G. Booch , J. Rumbaugh e I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley.

[Unified 1999] I. Jacobson, G. Booch e J. Rumbaugh. *Unified Software Development.* Addison-Wesley.