Capítulo IX – Concorrência

Autores:
Jociel Cavalcante Andrade
Mariella Berger
Flávio Miguel Varejão

"Não é a mais forte das espécies que sobrevive, nem a mais inteligente, mas a que melhor responde à mudança"

Charles Darwin

Os programas de computador nada mais são do que seqüências de instruções passíveis de serem executadas por um processador. Eles não apresentam comportamento nem possuem estado. De fato, programas são, em última instância, os conteúdos dos arquivos. Todas as cópias de um mesmo programa são idênticas.

Programas em execução são chamados de processos. Ao contrário dos programas, os processos são entidades ativas e, mesmo quando se referem ao mesmo programa, podem apresentar comportamento e características diferentes em função do contexto. Por exemplo, é possível ter duas instâncias de um jogo de damas em execução, ou seja, dois processos originários de um mesmo programa. Ao longo das jogadas, a disposição das peças nos dois tabuleiros se torna distinta para cada instância do jogo.

Os primeiros sistemas de computação só permitiam a execução de um programa de cada vez. Apenas um processo utilizava o processador até seu término sem sofrer interrupções. Assim, todos os recursos do sistema (impressora, áudio, vídeo, memória, etc.) ficavam disponíveis àquele processo no momento em que ele precisasse. O problema com esse tipo de computação é a ociosidade do processador enquanto espera o acesso a dispositivos externos. Essa espera pode ser longa. Como exemplo, se um processo começa a imprimir um documento, o processador estará ocioso até o término da impressão, embora pudesse estar sendo utilizado para executar instruções de outro processo.

Os sistemas computacionais modernos conseguem executar simultaneamente múltiplos processos. Um processador consegue processar apenas uma instrução por vez. Entretanto, o sistema operacional estabelece fatias de tempo (`time slices`) para que cada processo tenha posse do processador por um certo período, dando a impressão de que eles estão sendo executados ao mesmo tempo. Essa característica pode ocasionar uma situação em que mais de um processo requisite os serviços de um mesmo dispositivo computacional do sistema. Por exemplo, um processo em exe-

cução inicia a impressão de um documento. Em seguida, o sistema operacional alterna a execução para outro processo, que também solicita a impressão de um documento. Isso produziria como resultado a impressão de parte do documento do primeiro processo seguida de parte do documento do segundo processo, e assim sucessivamente[9.1]. Este é um exemplo de concorrência, que é o termo usado em computação para designar situações nas quais diferentes processos competem pela utilização de algum recurso (processador, dispositivos periféricos, etc.) ou cooperam para a realização de uma mesma tarefa.

A competição por um mesmo recurso deve ser feita de forma sincronizada e justa, para que apenas um processo acesse o recurso de cada vez e para que todos tenham esse acesso em algum momento. Caso o acesso não seja mutuamente exclusivo, poderão ocorrer inconsistências nos resultados, como será visto nas seções seguintes.

Uma atividade pode ser realizada através da cooperação de diversos processos, diminuindo assim o seu tempo de execução. Para que essa cooperação ocorra, deve existir uma forma de comunicação entre os processos e um protocolo de comunicação pré-determinado.

Os sistemas operacionais atuais também permitem a um mesmo processo ter mais de um fluxo de execução (`threads`), agravando o problema da concorrência. Esse agravamento ocorre porque muitos fluxos de execução podem manipular o valor de uma mesma variável simultaneamente.

É importante ter ainda em mente que a programação concorrente consiste em construir programas contendo múltiplas atividades que progridem em paralelo. Atividades podem progredir em paralelo sendo realmente executadas em paralelo, cada uma de posse de um processador diferente ou tendo o processador se revezando entre as mesmas, de forma que cada atividade possa fazer uso do processador em um dado instante.

Esse capítulo enfoca os mecanismos oferecidos por linguagens de programação para facilitar a construção de sistemas computacionais concorrentes. Inicialmente, são discutidos vários conceitos envolvidos no estudo da concorrência. Em seguida, são apresentadas algumas técnicas para sincronização de processos e `threads`. Por fim, mostra-se e discute-se os mecanismos oferecidos pelas linguagens de programação para facilitar a programação concorrente.

[9.1] Na realidade, isso não ocorre porque existe um processo responsável por gerenciar a fila de impressão.

9.1 Processos e `Threads`

Para compreender a idéia de concorrência em sistemas computacionais, é importante ter um bom entendimento sobre o que são processos e `threads` e sobre como eles se comunicam e interagem. Essa seção apresenta uma visão geral desses conceitos.

9.1.1 Processos

Programas de computador são seqüências de instruções passíveis de serem executadas por um processador. Quando estes programas estão sendo executados, eles são chamados de processos. Diferentemente dos programas, os processos são entidades ativas cujo estado é alterado durante a sua execução. No processo, além do conjunto de instruções, incluem-se também as informações correntes sobre a execução.

Embora vários processos possam estar associados a um mesmo programa, eles são tratados independentemente. Cada processo possuirá um contexto, isto é, o seu próprio espaço de endereçamento, suas informações de controle, sua identificação, suas variáveis de ambiente, entre outros. Isso torna possível a utilização de um mesmo programa por mais de um usuário simultaneamente, sem que a execução de um processo sofra interferência da execução do outro.

Os processos podem ser classificados em seqüenciais, quando a sua execução for estritamente seqüencial, ou em concorrentes, quando a execução de dois ou mais processos pode ser feita de modo simultâneo.

9.1.2 Estado

Durante a execução de um processo, o seu estado é modificado. O estado de um processo é definido pela atividade que está realizando. Os possíveis estados de um processo são os seguintes:

a. **Novo**: estado no qual o processo está sendo criado;
b. **Executável**: estado no qual o processo está aguardando a liberação de um processador para iniciar a sua execução;
c. **Em execução**: estado no qual o processo ocupa o processador;
d. **Em espera**: estado no qual o processo está esperando a ocorrência de algum evento;
e. **Encerrado**: estado do processo após ter sua execução encerrada.

Apenas um processo pode ocupar o processador a cada instante, ou seja, apenas um processo poderá estar no estado **em execução**. Entretanto, po-

de haver inúmeros processos nos estados **executável** e **em espera**. A figura 9.1 apresenta um diagrama ilustrando as possíveis transições de estados de um processo.

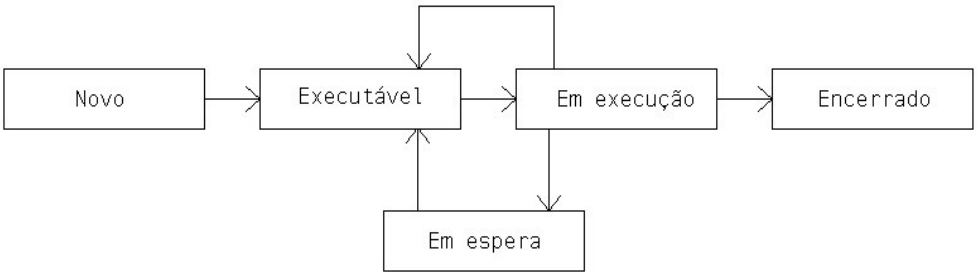

Figura 9. 1 - Diagrama de Transição de Estados de Um Processo

Inicialmente, o processo encontra-se no estado **novo**. A partir desse estado, ele pode somente passar para o estado **executável**, quando está pronto para ser executado. Caso não haja processo no estado **em execução**, um processo passa do estado **executável** para o estado **em execução**.

Quando um processo está no estado **em execução**, ele poderá fazer chamadas de sistema. Enquanto espera esta chamada ser atendida, ele não continuará a sua execução, passando assim para o estado **em espera** e liberando o processador para a execução de outro processo em estado **executável**.

Quando ocorre o evento esperado, o estado daquele processo passa a ser **executável**. O sistema operacional pode também passar o processo do estado **em execução** diretamente para o estado **executável**, caso tenha terminado sua fatia de tempo de posse do processador.

Um processo no estado **executável** voltará ao estado **em execução** quando novamente tomar posse do processador. Terminada a execução do processo, este é colocado no estado **encerrado** para que o sistema operacional libere os recursos ainda não desalocados daquele processo.

Sempre que um processo passa do estado **em execução** para o estado **executável** ou para o estado **em espera**, o sistema operacional salva o contexto de execução daquele processo. O contexto de execução de um processo compreende dados de controle, como, por exemplo, a próxima instrução a ser executada e o estado dos registradores. De forma análoga, quando um processo volta ao estado **em execução**, é necessário recuperar o contexto em que ele se encontra, para que a execução possa prosseguir a partir de onde parou.

9.1.3 Threads

O modelo de concorrência visto até agora envolve a execução de vários processos simultaneamente, sendo cada processo formado por um conjunto de recursos e por um único fluxo de execução.

Os sistemas operacionais atuais permitem a um mesmo processo possuir vários fluxos de execução. Threads são fluxos de execução concorrentes que compartilham recursos do processo do qual são originários. Pode-se pensar em threads como sendo funções ou procedimentos dentro de um mesmo programa, executando concorrentemente e compartilhando o contexto.

Os threads também possuem estados, características próprias e necessidade de gerência como os processos. Eles também são conhecidos como processos leves.

O compartilhamento de recursos de um mesmo processo por vários threads caracteriza a concorrência intraprocessos, e agrava ainda mais a concorrência interprocessos, visto que threads podem compartilhar recursos com outros threads de um mesmo processo ou com threads de processos diferentes.

Por outro lado, threads podem oferecer algumas vantagens para a implementação de sistemas concorrentes. Enquanto processos impõem um custo expressivo para a sua construção e destruição, uma vez que manipulam uma quantidade mais significativa de dados, threads são mais leves, pois utilizam os recursos de um processo criado previamente. Além disso, um thread compartilha memória com o processo que o criou e com os demais threads. Como cada processo possui sua própria memória, torna-se elevado o custo das trocas entre processos.

Os threads também possibilitam a utilização de mais de um método ou função de uma mesma aplicação, simultaneamente. Por exemplo, quando você está utilizando um editor de textos e, enquanto um arquivo é carregado, você consegue alterá-lo, a aplicação está usando threads. Apesar dessa técnica ser possível também com a utilização de processos concorrentes, o uso de threads facilita a implementação e torna a aplicação mais eficiente.

A figura 9.2 representa um sistema operacional servindo como uma camada entre a complexidade do hardware e os processos. Os processos disputam a posse do processador entre si. Além disso, threads em um mesmo processo competem por recursos.

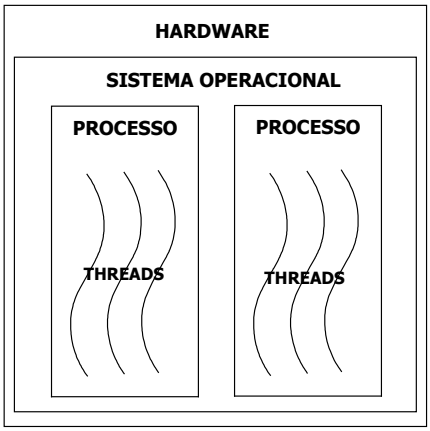

Figura 9. 2 - Processos e `Threads`

9.1.4 Processos Concorrentes

Os programas seqüenciais são determinísticos, isto é, seguem uma seqüência de passos pré-determinada. Essa pré-determinação da ordem de execução dos passos torna o comportamento do programa totalmente previsível. Entretanto, os programas concorrentes são não-determinísticos, ou seja, a ordem na qual as instruções dos processos são executadas não pode ser definida antes da sua execução. Isso acarreta alguns problemas, tal como a indeterminação do valor de uma variável.

O exemplo 9.1 leva em conta que o conteúdo do endereço de memória *i* (*[i]*) é *100*, e *i* é uma variável compartilhada.

```
; Processo 1:     ; Processo 2:
mov ax, [i]       mov ax, [i]  ; guarda o valor da variável i em ax
mul ax, 2         sub ax, 5    ; multiplica/subtrai ax de 2/5
mov [i], ax       mov [i], ax ; guarda o valor do registrador ax
                               ; na variável i
```

Exemplo 9. 1 - Indeterminismo

O processo 1 apenas multiplica o conteúdo da variável *i* por *2* (equivalente em C a *i = i * 2*) e o processo 2 apenas subtrai o conteúdo da variável *i* por *50* (equivalente em C a *i = i - 50*). O conteúdo do endereço de memória *i*, ao final da execução dos dois processos em concorrência, poderia ser *150*, *100*, *200* ou *50*.

A figura 9.3 mostra as quatro possíveis situações. Na primeira situação, representada na figura 9.3(a), os processos 1 e 2 são executados sucessivamente e sem interrupção. Primeiramente, o valor inicial de *i* (*100*) do processo 1 é carregado no registrador *ax*, multiplicado por *2*, e o resultado (*200*) é atribuído de volta a *i*. Em seguida, o processo 2 carrega o valor atual de *i* (*200*) no registrador *ax*, subtrai *50* desse valor e atribui o resultado (*150*) de volta a *i*. Na segunda situação, representada na figura

9.3(b), ocorre o inverso. Assim, o valor inicial de *i* (*100*) é primeiro subtraído de *50* para depois ser multiplicado por *2*, resultando em um valor final de *100* para *i*.

Na terceira situação, representada na figura 9.3(c), o processo 1 começa a ser executado, carregando o valor inicial de *i* (*100*) no registrador *ax*. Após realizar essa operação, ocorre uma interrupção e a posse do processador é passada para o processo 2, que carrega o valor corrente de *i* (o qual continua sendo o valor inicial *100*) no registrador *ax*. Em seguida, esse valor é subtraído de *50* e o resultado (*50*) é atribuído a *i*. Após o término do processo 2, o contexto no momento da interrupção do processo 1 é recuperado (o registrador *ax* continha o valor *100*) e sua execução é continuada multiplicando-se o conteúdo de *ax* por *2* e atribuindo-se o resultado (*200*) a *i*. Observe que nessa situação é como se o processo 2 não tivesse sido executado, uma vez que o valor final de *i* seria o mesmo produzido se o processo 1 tivesse sido o único executado. Na quarta situação, representada na figura 9.3(d), ocorre a situação inversa à representada na figura 9.3(c). Dessa maneira, a execução do processo 2 define sozinha o valor final de *i* (*50*).

Processo1	***Processo2***
mov ax, [i]	
mul ax, 2	
mov [i], ax	
	mov ax, [i]
	sub ax, 50
	mov [i], ax

a. Valor final de *i*: *150*

Processo1	***Processo2***
	mov ax, [i]
	sub ax, 50
	mov [i], ax
mov ax, [i]	
mul ax, 2	
mov [i], ax	

b. Valor final de *i*: *100*

Processo1	***Processo2***
mov ax, [i]	
	mov ax, [i]
	sub ax, 50
	mov [i], ax
mul ax, 2	
mov [i], ax	

c. Valor final de *i*: *200*

Processo1	***Processo2***
	mov ax, [i]
mov ax, [i]	
mul ax, 2	
mov [i], ax	
	sub ax, 50
	mov [i], ax

d. Valor final de *i*: *50*

Figura 9. 3 - Exemplo de Indeterminismo

Outros problemas comuns que podem ocorrer em processos concorrentes são o `lockout` (trancamento), o `deadlock` (impasse) e a `starvation` (inanição).

O `lockout` ocorre quando dois ou mais processos ficam esperando por um evento que nunca acontecerá. O `deadlock` ocorre quando todos os processos estão bloqueados esperando por um evento, o qual só pode ser gerado por outro processo também bloqueado. Isso impede o progresso da execução do sistema. Essa situação só pode ser alterada por iniciativa de um processo não pertencente ao grupo em `deadlock`.

`Starvation` ocorre quando um processo tem a aquisição de um determinado recurso postergada indefinidamente. Como exemplo, em uma política de prioridades, um processo de baixa prioridade pode nunca obter a posse do processador caso existam outros com prioridades maiores. Para resolver este problema podem-se utilizar mecanismos de fila ou técnicas de envelhecimento.

9.1.5 Tipos de Interação

Existem duas maneiras em que processos concorrentes podem interagir: eles podem competir por um mesmo recurso, mas não compartilhar dados, ou podem cooperar pela realização de uma atividade, compartilhando dados ou sendo afetados pela execução de outros processos.

9.1.5.1 Competição

Os processos competitivos são independentes, apenas concorrem com outros processos pela utilização de um mesmo recurso, como exemplo, pela ocupação do processador. A execução concorrente de processos competitivos é efetivamente determinística pois, como não há compartilhamento de dados, a ordem de execução, mesmo não podendo ser determinada antecipadamente, não interfere nos resultados obtidos.

Neste tipo de interação, a sincronização para utilização dos recursos é feita pelo sistema operacional, não provocando qualquer dificuldade para os programadores.

9.1.5.2 Cooperação

Um processo é chamado cooperativo quando afeta ou é afetado pela execução de outro processo em prol da realização de uma atividade. A cooperação entre processos visa aumentar a velocidade de computação de um programa através da modularização das tarefas em vários processos.

Para que ocorra a cooperação entre processos, é necessário que exista uma forma de comunicação entre eles. A comunicação é possível através do uso dos mecanismos de troca de mensagens ou de compartilhamento de memória.

A comunicação entre processos por troca de mensagens tem como objetivo permitir a troca de informações, sem que haja a necessidade de dados compartilhados. Para que essa comunicação ocorra, deve existir um mecanismo bidirecional de passagem de dados, isto é, um canal de comunicação.

Os mecanismos de troca de mensagens são diretamente aplicáveis a processos distribuídos. Os processos estão em máquinas diferentes e, portanto, não compartilham a mesma memória física. Eles também podem ser usados para a comunicação entre processos em uma mesma máquina. Contudo, a troca extensiva de mensagens pode ser um fator de sobrecarga (`overhead`).

A comunicação entre processos por memória compartilhada tem como objetivo prover comunicação através de dados compartilhados. Processos que compartilham memória tendem a ser mais eficientes que processos cuja comunicação é feita através de troca de mensagens. Entretanto, a comunicação por memória compartilhada pode provocar problemas de inconsistência de dados, os quais não ocorrem quando se utiliza troca de mensagens.

Assim, o compartilhamento dos dados deve ser feito de forma sincronizada e padronizada, para que se tenha controle dos recursos compartilhados de modo a não causar inconsistências.

9.1.6. Troca de Mensagens

Geralmente, o mecanismo de troca de mensagens entre processos é implementado pelo sistema operacional, disponibilizando duas chamadas de sistema básicas: `send` (envio) e `receive` (recepção). A chamada `send` permite o envio de uma mensagem de um processo para outro e a chamada `receive` permite a um processo receber uma mensagem enviada para ele.

Existem dois tipos de comunicação por troca de mensagens: direta e indireta. Na comunicação direta, um processo envia mensagens diretamente para outro processo. Neste caso, o processo remetente deve conhecer o identificador do processo para o qual quer enviar a mensagem.

Na comunicação indireta, um processo não envia mensagens para outro processo, mas sim para caixas postais. Cabe a cada processo retirar as mensagens de suas caixas postais. Assim, para enviar uma mensagem, o processo deve conhecer o identificador da caixa postal do destinatário e ter permissão para escrita nessa caixa. Normalmente, o sistema operacional disponibiliza chamadas de sistema para criação, manipulação e destruição de caixas postais.

É importante lembrar que caixas postais permitem uma comunicação de vários processos para vários outros, enquanto que uma comunicação direta é de um processo para outro.

A chamada `send` deve possuir, pelo menos, um identificador do processo ou da caixa postal do destinatário e a mensagem a ser enviada. A chamada `receive` deve possuir, pelo menos, o endereço da variável que receberá o conteúdo da mensagem postada para aquele processo.

A comunicação de processos através de troca de mensagens pode ser bloqueante ou não-bloqueante. A comunicação é dita bloqueante quando, depois de enviar uma mensagem, o processo remetente aguarda (fica bloqueado) até que a mensagem seja confirmada pelo processo destinatário. Na comunicação não-bloqueante, o processo que envia a mensagem não fica esperando confirmação e continua a sua execução.

Pode ocorrer a situação de um processo enviar uma mensagem para outro processo utilizando comunicação direta, e esse último não a retirar imediatamente. Freqüentemente, o sistema operacional possui um `buffer` para armazenar as mensagens que ainda não foram retiradas. Entretanto, quando não há recursos no sistema para tal armazenamento, o sistema operacional normalmente bloqueia o remetente até que o destinatário retire a mensagem. Alguns autores costumam chamar esse tipo de comunicação de *rendezvous* (encontro), porque a comunicação só ocorre quando o remetente e o destinatário se encontram. Nesse esquema, o primeiro processo que chega ao ponto de encontro espera pelo outro.

9.1.7. Compartilhamento de Memória

Quando processos cooperam entre si compartilhando uma mesma memória física, há a necessidade de proteção de acesso a dados compartilhados para evitar inconsistências. Tome como exemplo o problema clássico do produtor e consumidor. Neste problema, há um processo que produz algo e um processo que utiliza o que foi produzido. Há também um `buffer` intermediário que serve para armazenar o que foi produzido pelo produtor e disponibilizar o que está armazenado para o consumidor. A figura 9.4 esquematiza esse exemplo.

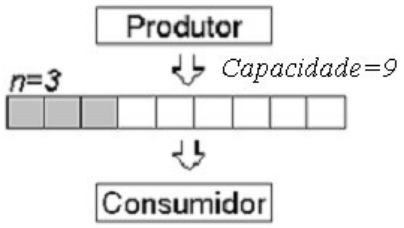

Figura 9.4 - Problema Produtor-Consumidor

Uma possível implementação para esse problema é mostrada no exemplo 9.2. Considere nesse exemplo que o bloco de inicialização é executado antes dos blocos do produtor e do consumidor, os quais são executados de maneira concorrente.

```
// inicializacao
fim = 0;
ini = 0;
n = 0;
// codigo do produtor
for (i=0; i<1000; i++) {
  while (n == capacidade);
  buf[fim] = produzir(i);
  fim = (fim + 1) % capacidade;
  n++;
}
// codigo do consumidor
for (i=0; i<1000; i++) {
  while (n == 0) ;
  consumir(buf[ini]);
  ini = (ini + 1) % capacidade;
  n--;
}
```

Exemplo 9.2 - Implementação do Problema do Produtor e Consumidor.

O produtor apenas produz algo, o coloca na posição *fim* do `buffer` *buf* e incrementa a quantidade de elementos disponíveis. O consumidor, por sua vez, apenas consome o dado da posição *ini* de *buf* e decrementa o valor da quantidade de elementos disponíveis.

Os laços *while* na linha 2 dos códigos do produtor e do consumidor servem apenas para garantir a não extrapolação do tamanho máximo do `buffer`, pelo produtor, e para impedir o consumo de elementos ainda não produzidos, pelo consumidor. Note que os laços *while* são exemplos de `busy wait` (espera ocupada), ou seja, esperam por um evento, mas não liberam o uso do processador. Na seção 9.3 será mostrado um mecanismo usado por algumas linguagens de programação para tratar esse problema. No momento, o importante é mostrar os problemas que podem ocorrer quando o produtor e o consumidor compartilham o `buffer` *buf* e o endereço de memória *n*.

Apesar do código mostrado não estar completo, estas são as partes principais da implementação. É fácil verificar que, se esse programa fosse executado seqüencialmente com tamanho do `buffer` (*capacidade*) maior ou igual a 1000 (para evitar a repetição infinita nos laços *while*), não ha-

veria problema algum. O produtor produziria 1000 elementos consumidos posteriormente pelo consumidor.

No entanto, essa solução torna o início da execução do consumidor dependente do término da execução do produtor. Isso pode não ser a solução mais apropriada. Considere o produtor como sendo um editor de textos produzindo dados para serem impressos e o consumidor como um processo que envia esses dados para a impressora. Com a solução seqüencial, a impressora só conseguiria imprimir após o término do envio de dados pelo produtor. Com a solução concorrente, à medida que os dados são colocados no `buffer` pelo processo produtor, eles já podem ser enviados para a impressora pelo processo consumidor para serem impressos.

Portanto, a utilização de processos concorrentes para resolver esse problema é mais apropriada. Entretanto, surge outro problema: a possibilidade de inconsistência dos dados. Isso pode ocorrer pelo fato de os processos compartilharem a variável *n*, que identifica o número de elementos em *buf*. Suponha, como exemplo, que em um dado instante o valor da variável *n* seja *5*. Vamos supor que em um instante seguinte o processo produtor faça *n++* concorrente com o *n--* do processo consumidor. Ao final do processamento concorrente, poderíamos ter como valores da variável *n* os números *4*, *5* ou *6*, quando o correto seria o valor *5*.

Para entender como esses valores podem ser obtidos, é importante lembrar que o processador só faz operações aritméticas com valores armazenados em registradores. Logo, a tradução para *assembly* das operações *n++* e *n--* envolvem, além da operações de incremento e decremento, operações de cópia de valores entre memória e registrador. Essas instruções poderiam ser representadas como no exemplo 9.3.

```
; n++                    ; n--
mov ax, [n]              mov ax, [n]
inc ax                   dec ax
mov [n], ax              mov [n], ax
```

Exemplo 9.3 - Representação da Codificação *Assembly* das Instruções n++ e n--

A seqüência de operações apresentada na tabela 9.1 produziria o valor *6* em *n* ao final da execução das operações *n++ e n--* dos dois processos. Se a ordem de execução das últimas instruções do produtor e do consumidor fosse invertida, o valor final de *n* seria *4*. Se não existisse intercalação na execução das instruções do produtor e do consumidor, o valor final de *n* seria *5*.

Produtor	Consumidor
mov ax, [n]	
inc ax	
	mov ax, [n]
	dec ax
	mov [n], ax
mov [n], ax	

Tabela 9.1 - Uma Possível Seqüência de Execução do Exemplo 9.3

O indeterminismo ocorre porque ambos processos podem executar simultaneamente os segmentos de código que realizam a alteração do recurso compartilhado. O segmento de código de um programa que manipula recursos compartilhados por processos é denominado de **região crítica**.

Para evitar o problema de indeterminismo deve haver algum tipo de sincronização entre os processos de modo a garantir acesso exclusivo a sua região crítica, ou seja, caso o processo produtor tente modificar o valor da variável *n*, o processo consumidor deverá esperar até o término dessa instrução para só então também poder modificar seu valor. A atribuição à variável *n* nunca deve ser executada de forma concorrente.

9.2 Sincronização

Nessa seção serão vistas duas técnicas de sincronização utilizadas para garantir que programas concorrentes não provoquem inconsistência nos dados compartilhados.

9.2.1. Semáforos

Um mecanismo de sincronização entre processos muito empregado é o semáforo. Um semáforo é um tipo abstrato de dados que possui um valor inteiro, uma lista de processos em espera e duas operações: *P* (do holandês *Proberen*, testar) e *V* (do holandês *Verhogen*, incrementar). Para garantir exclusão mútua a uma determinada região (região crítica), cada processo deve chamar a operação *P* antes de acessar tal região e chamar a operação *V* após sair dessa região.

A operação *P* sobre um semáforo *S* testa se o processo que executou *P(S)* pode ou não entrar na região crítica, isto é, se já existir outro processo na região crítica, ele não poderá entrar e será colocado na fila de espera daquele semáforo.

A operação *V* sobre um semáforo *S* sinaliza ao semáforo que o processo não está mais na região crítica e retira outro processo da fila de espera, colocando-o novamente em execução. A fila de espera pode ser implementada de várias formas diferentes. Para evitar `starvation`, normal-

mente é implementada uma fila simples, ou seja, o processo que executou primeiro a operação *P(S)* é o primeiro a ser colocado de volta em execução quando ocorre a chamada a *V(S)*.

A implementação do semáforo deve ser atômica, ou seja, não pode haver mudança de contexto de um processo para outro no meio de uma operação *P(S)* ou *V(S)*. Um esquema de implementação possível para o semáforo *S* é mostrado no exemplo 9.4.

```
P(S)
        S.valor -= 1;
        Se (S.valor < 0)
                // bloqueia o processo e insere em S.fila
V(S)
        S.valor += 1;
        Se (S.valor <= 0)
                // retira algum processo de S.fila e o coloca em execucao
```

Exemplo 9.4 - Esquema de Implementação de Semáforo

Inicialmente o contador *S.valor* é iniciado com *1*. Quando *P(S)* é chamado, *S.valor* é decrementado de *1*, indicando que há um processo tentando entrar na região crítica. Caso *S.valor* seja um valor negativo, isso indica que há outro processo na região crítica e que o processo corrente deve esperar. Este processo é então inserido em uma fila para ser chamado assim que o outro processo sair da região crítica. Por sua vez, quando *V(S)* é chamado, o contador é incrementado de *1*, indicando que aquele processo saiu da região crítica. Outro processo que esteja na fila de espera é então retirado e colocado em execução. Se *S.valor* possui um valor negativo, seu valor em módulo indica quantos processos estão na fila de espera.

Com esta técnica podemos facilmente resolver o problema de acesso mutuamente exclusivo à região crítica do produtor e consumidor. O exemplo 9.5 é uma modificação do exemplo 9.2, no qual foi adicionado um semáforo delimitando a região crítica. Aqui também se deve considerar que os processos produtor e consumidor são executados concorrentemente.

```
// inicializacao
fim = 0;
ini = 0;
n = 0;
semaforo S;
S.valor = 1;
// codigo do produtor
for (i=0; i<1000; i++) {
  while (n == capacidade) ;
  buf[fim] = produzir(i);
```

```
    fim = (fim + 1) % capacidade;
    P(S);
    n++;
    V(S);
  }
  // codigo do consumidor
  for (i=0; i<1000; i++) {
    while (n == 0) ;
    consumir(buf[ini]);
    ini = (ini + 1) % capacidade;
    P(S);
    n--;
    V(S);
  }
```

Exemplo 9.5 - Alteração do Exemplo 9.2 para Garantir Acesso Exclusivo à Região Crítica

Inicialmente, o valor do semáforo é *1* e os códigos do produtor e do consumidor são executados concorrentemente. Se o processo produtor faz *P(S)*, então o contador, que inicialmente tinha como valor *1*, é decrementado e passa a valer *0*, indicando que há um processo na região crítica.

Se no momento seguinte o processo consumidor fizer também *P(S)* tentando entrar na região crítica, o valor de *S.valor* passa a ser *–1* e, como esse número é negativo, o processo consumidor vai para a fila de espera e permanece lá até que o processo produtor faça a chamada a *V(S)*. Quando o produtor sai da região crítica fazendo a chamada à *V(S)*, *S.valor* é incrementado, passando a valer *0*.

O processo consumidor que estava na fila de espera é liberado, acessando assim a região crítica. Se, novamente, o processo produtor tentar entrar na região crítica chamando *P(S)*, esse irá decrementar *S.valor* tornando-o negativo e, assim, será bloqueado e colocado na fila de espera.

9.2.2. Programação Concorrente Estruturada

Embora semáforos ofereçam um mecanismo simples e eficaz para sincronização entre processos, problemas podem ocorrer devido a uma má programação. Para garantir acesso exclusivo, a região crítica deve estar limitada pelas operações *P* e *V* aplicados a um semáforo, nessa ordem. Agora, suponha que por um erro ou por malícia do programador, essa ordem seja afetada ou que alguma dessas operações não seja feita. As seguintes situações podem ocorrer:

a. Faz-se primeiro a operação *V*, entra-se na região crítica, e após sair da região crítica, faz-se a operação *P*: Nesse caso, vários processos pode-

riam estar executando em sua região crítica violando seu acesso exclusivo. Esse é um erro difícil de ser identificado e reproduzido, pois aparece apenas quando os processos entram simultaneamente na região crítica.

b. Coloca-se a operação *P* no lugar da operação *V* ou omite-se *V*. Nessa situação ocorre `deadlock`, pois o processo, ao entrar na região crítica, nunca vai liberá-la.
c. Coloca-se a operação *V* no lugar da operação *P*, ou omite-se *P*. Nessa situação ocorre violação no acesso exclusivo à região crítica, pois o processo não testa se há algum outro na região crítica.

Para lidar com esses tipos de erros, alguns mecanismos foram propostos para abstrair o conceito de semáforos em LPs. Esses mecanismos deixam para o compilador a responsabilidade de garantir o acesso exclusivo à região crítica.

Regiões críticas condicionais foram propostas com essa finalidade. Toda variável compartilhada necessita ser declarada como tal. Adicionalmente, regiões críticas são demarcadas através de um bloco de instruções especial. Assim, é fácil para o compilador verificar se a variável compartilhada é acessada apenas dentro das regiões críticas. Além disso, o compilador pode inserir automaticamente as operações *P(S)* e *V(S)* nas posições apropriadas.

Monitor é outro mecanismo proposto para esse fim. Ele une as vantagens das regiões críticas condicionais com as da modularização. Por conta disso, LPs que oferecem recursos para a construção de programas concorrentes têm optado por disponibilizar mecanismos para suportar a implementação de monitores.

9.2.2.1 Monitores

Monitores são mecanismos de sincronização compostos por um conjunto de variáveis, procedimentos e estruturas de dados dentro de um módulo cuja finalidade é a implementação automática da sincronização, garantindo exclusão mútua entre seus procedimentos.

Nenhum procedimento dentro do monitor pode ser chamado por mais de um processo simultaneamente. Sempre que um desses procedimentos é chamado, o monitor verifica se já existe outro processo executando algum de seus procedimentos. Caso exista, o processo é colocado em uma fila de espera até que esse procedimento seja liberado. Fica a cargo do compilador a implementação da exclusão mútua dos procedimentos, livrando os programadores dessa complexidade. O esquema básico de um monitor pode ser visto no exemplo 9.7.

```
Monitor nome-do-monitor {
   // declarações de variável
   p1(...) {
         ...
   }
   p2(...) {
         ...
   }
}
```

Exemplo 9.7 - Esquema Básico de um Monitor

No exemplo 9.7 é mostrada a definição de dois procedimentos p1 e p2. Eles possuem acesso exclusivo, ou seja, se um processo chamar um desses procedimentos, outro processo não poderá fazê-lo até que o primeiro termine a chamada. Normalmente, as linguagens que suportam a implementação de monitores também implementam operações que permitem o bloqueio de um processo até que uma condição seja satisfeita.

9.3. Abordagens das LPs

Nesta seção serão mostradas algumas abordagens utilizadas por LPs para permitir a implementação de sistemas concorrentes.

9.3.1. Ausência de Mecanismos

Algumas LPs, tal como C, não oferecem mecanismos próprios para lidar com concorrência. A maneira pela qual é possível criar e manipular processos e `threads`, e ainda resolver os principais problemas de concorrência, é através do uso de chamadas de sistema ou de bibliotecas de funções (específicas da plataforma de execução). C, tal como é freqüentemente utilizada no sistema operacional UNIX, será usada para exemplificar essa abordagem. Será também discutido as possíveis formas de sincronização entre processos e `threads` em C.

9.3.1.1. Chamada de Sistema *fork*

Os sistemas de computação tradicionais utilizavam principalmente chamadas de sistema para criar e manipular processos concorrentes. No UNIX e no LINUX existe a chamada de sistema *fork*, a qual permite a duplicação de um processo.

Quando se faz uma chamada *fork*, o sistema operacional cria um processo idêntico ao que fez a chamada e retorna um número identificador daquele processo. Ambos processos prosseguem a partir do ponto onde ocorreu a chamada de *fork*.

Para diferenciar a execução do processo-filho do processo-pai, o programador necessita testar o número retornado pelo *fork*. Se o retorno for zero, trata-se do processo-filho. Se contiver um valor maior que zero, trata-se do processo-pai e o valor retornado é o identificador do processo-filho. Caso o retorno de *fork* seja negativo, isso indica a ocorrência de algum erro durante a criação do processo.

O exemplo 9.8 mostra a criação de dois processos-filhos a partir do processo-pai através de chamadas *fork*. O primeiro processo criado imprime na tela *1000000* pontos e o segundo processo imprime a mesma quantidade de traços. O comando *wait* faz com que o processo-pai espere até que um processo-filho termine sua execução. A quantidade de traços ou pontos mostrados na tela dependerá do escalonamento dos processos pelo sistema operacional.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void imprime(char *arg) {
  int i;
  for (i=0; i<1000000; i++)
    printf(arg);
  exit(0);
}
int main() {
  int pid;
  int estado_filho;
  pid = fork();
  if (pid < 0) { // erro
   perror("Erro de criação do primeiro filho.");
  } else {
    if (pid == 0) { //filho
      imprime("-");
    }
  }
  pid = fork();
  if (pid < 0) {// erro
    perror("Erro de criação do segundo filho.");
  } else {
    if (pid == 0) { //filho
      imprime(".");
```

```
    }
  }
  wait(&estado_filho);  // aguarda o término de um filho
  wait(&estado_filho); // aguarda o término do outro filho*/
  return 0;
}
```

Exemplo 9.8 - Chamadas *fork*

A chamada de sistema *exit()* na função *imprime* faz com que os processos-filhos terminem. O argumento passado nesta chamada de sistema é capturado pela chamada de sistema *wait* do processo-pai. No exemplo, a variável *estado_filho* receberá o valor *0*. É válido lembrar que os processos-filhos poderiam estar retornando qualquer valor para a variável *estado_filho* capturado pelo *wait*.

Deve-se perceber que estão sendo utilizados dois *waits*, já que existem dois processos-filhos. Cada *wait* utilizado faz com que o processo-pai aguarde o fim de um processo-filho.

Dependendo do sistema operacional, caso o processo-pai termine sua execução antes de seus filhos, todos os outros processos criados pelo processo-pai também têm sua execução encerrada.

É possível implementar semáforos[9.2] e monitores para permitir sincronização entre processos. Contudo, isso necessita ser feito através de chamadas de sistema e não é uma tarefa muito fácil.

Cada sistema operacional pode implementar de modo diferente as chamadas de sistema para lidar com processos e por isso não serão apresentados detalhes sobre os recursos oferecidos por chamadas *fork*.

9.3.1.2. POSIX Threads

Essa seção apresenta uma forma de gestão de processos leves (`threads`) em C através de bibliotecas de funções que obedecem ao padrão POSIX (`Portable Operating System Interface for UNIX`), conhecidas como `Pthreads` (POSIX `Threads`).

POSIX é um padrão definido pela IEEE e pela ISO, o qual define no mundo do Unix como os programas devem se comunicar visando portabilidade[9.3].

O POSIX `threads` é um padrão adotado para manipular `threads` em C. Apesar de ser um padrão para UNIX, existem projetos de definição de

---

9.2 Maiores informações em http://www.cs.uh.edu/~paris/4330/NOTES/Ipsyn.pdf.

9.3 Maiores informações em http://socrates.if.usp.br/~guioc/Library/pthreads_em_C.pdf

APIs (Aplication Programming Interface) para outros ambientes, inclusive para Windows[9.4].

A API padrão do pthreads conta com aproximadamente 60 funções. As principais funções existentes nessa biblioteca são:

- *pthread_create*: criação de um *thread*;
- *pthread_exit*: terminação de um *thread*;
- *pthread_join*: espera pelo término de um thread dependente;
- *pthread_attr_init* e *pthread_attr_destroy*: definição de propriedades de um *thread*;
- *pthread_detach*: terminação de um thread independente;
- *sched_yield*: liberação do processador.

O exemplo 9.9 tem o mesmo propósito do exemplo 9.8, entretanto faz uso de chamadas de procedimentos da biblioteca pthread para manipulação de processos leves (threads).

O comando *pthread_attr_init* inicializa os atributos do thread com os valores padrões na variável *attr*. Esses atributos podem ser o tipo de escalonamento utilizado para o thread ou a definição se o término do thread é dependente ou não do processo que a criou. Na API de *pthreads* existe funções para manipular esses atributos.

```
/* ex1.c                              */
/* Para compilar com gcc, use:        */
/* gcc –lpthread -oex1 ex1.c          */
#include <stdio.h>
#include <string.h>
#include <pthread.h>
void *imprime(void *arg) {
  int i;
  for (i=0; i<1000000; i++)
    printf((char *)arg);
  pthread_exit(0);
}
int main() {
  pthread_t tid1, tid2;
  pthread_attr_t attr;
  char msg1[30];
  char msg2[30];
  strcpy(msg1, ".");
  strcpy(msg2, "-");
```

[9.4] Maiores informações em http://sources.redhat.com/pthreads-win32.

```
    pthread_attr_init(&attr);
    pthread_create(&tid1, &attr, imprime, msg1);
    pthread_create(&tid2, &attr, imprime, msg2);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

Exemplo 9.9 - Uma Implementação Usando `Pthreads`

O comando *pthread_create* cria um novo `thread` tendo como ponto de partida a execução da função *imprime*, recebendo como argumento *msg1* no primeiro caso e *msg2* no segundo caso. O primeiro argumento de *pthread_create* define a variável que conterá o identificador do novo *thread*. Somente através desse identificador, o processo-pai poderá se comunicar com os `threads`-filho. O terceiro parâmetro de *pthread_create* é um endereço de uma função (nesse caso, *imprime*). Esta função deve ter uma assinatura recebendo um argumento *void* * e tendo como tipo de retorno *void* *.

O comando *pthread_join* faz com que o processo-pai espere pelo término dos seus `threads`-filhos. Normalmente, o atributo padrão para os `threads` define o término dos novos `threads` como ligados ao término do processo-pai. Assim, se o processo-pai termina sua execução, o sistema operacional força o término da execução de todos os `threads` criados por ele que possuam essa propriedade.

A API *pthread* inclui também primitivas para sincronização entre `threads`. O mecanismo de sincronização utilizado é um dispositivo de exclusão mútua conhecido como `mutex` (exclusão mútua). O `mutex` é um tipo de semáforo binário, podendo estar nos estados bloqueado ou não-bloqueado.

O semáforo `mutex` possui duas operações: `lock` (bloqueia) e `unlock` (desbloqueia). A operação `lock` determina o bloqueio do processo caso o semáforo esteja em uso e a operação `unlock` permite a liberação de algum processo bloqueado na fila daquele semáforo. Essas operações são equivalentes às operações P e V apresentadas na seção 9.2.1 sobre semáforos.

A biblioteca `Pthreads` oferece ainda um meio de evitar que `threads` fiquem em espera ocupada. Isso é conseguido através do uso de variáveis de condição, às quais devem ser usadas em conjunto com um mecanismo de sincronização para garantir exclusão mútua. Uma variável de condição

suspende a execução de um `thread` ou processo até que uma condição seja satisfeita pela ação de outros `threads` ou processos.

O exemplo 9.10 é uma implementação do exemplo do produtor-consumidor já abordado anteriormente, utilizando um semáforo `mutex` e variáveis condicionais.

```
/* ex2.c                                    */
/* Para compilar com gcc, use:              */
/* gcc –lpthreads -oex2 ex2.c               */
#include<stdio.h>
#include<pthread.h>
#include <stdlib.h>
#define CAPACIDADE 10
typedef struct BufferLim {
   int buf[CAPACIDADE];
   int fim;
   int ini;
   int n;
   pthread_mutex_t mut;
   pthread_cond_t vazio, cheio;
} BufferLimitado;
void *produtor(void *arg) {
   BufferLimitado *bl = (BufferLimitado *)arg;
   int i;
   for (i=0; i<1000; i++) {
      pthread_mutex_lock(&bl->mut);
      while (bl->n == CAPACIDADE)
        pthread_cond_wait(&bl->cheio, &bl->mut);
      bl->buf[bl->fim] = i;
      printf("produzido: %d\n", bl->buf[bl->fim]);
      bl->fim = (bl->fim + 1) % CAPACIDADE;
      bl->n++;
      pthread_mutex_unlock(&bl->mut);
      pthread_cond_signal(&bl->vazio);
   }
   pthread_exit(0);
}
void *consumidor(void *arg) {
   BufferLimitado *bl = (BufferLimitado *)arg;
   int i;
   for (i=0; i<1000; i++) {
      pthread_mutex_lock(&bl->mut);
      while (bl->n == 0)
```

```
        pthread_cond_wait(&bl->vazio, &bl->mut);
      printf("consumido: %d\n",bl->buf[bl->ini]);
      bl->ini = (bl->ini + 1) % CAPACIDADE;
      bl->n--;
      pthread_mutex_unlock(&bl->mut);
      pthread_cond_signal(&bl->cheio);
    }
    pthread_exit(0);
}
int main() {
    BufferLimitado bl;
    Pthread_t tid1, tid2;
    bl.fim = bl.ini = bl.n = 0;
    pthread_mutex_init(&bl.mut, NULL);
    pthread_cond_init(&bl.cheio, NULL);
    pthread_cond_init(&bl.vazio, NULL);
    pthread_create(&tid1, NULL, produtor, &bl);
    pthread_create(&tid2, NULL, consumidor, &bl);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

Exemplo 9.10 Produtor e Consumidor com *mutex* e Variáveis de Condição.

No exemplo 9.10 foi criado o tipo de dados *BufferLimitado*, agregando atributos relacionados ao `buffer`. Valores desse tipo possuem um semáforo *mut*, para sincronizar o acesso à região crítica do produtor e do consumidor, e as variáveis condicionais *vazio* e *cheio*, usadas para sinalizar as condições de `buffer` não vazio e `buffer` não cheio, respectivamente. Um valor do tipo *BufferLimitado* deve ser passado como argumento para o produtor e consumidor.

Inicialmente, durante a execução de *main* o `buffer` *bl* é criado. Em seguida, suas variáveis de controle, o semáforo e as variáveis condicionais são inicializadas e os `threads` produtor e consumidor são criados.

A chamada *pthread_cond_wait(&bl->cheio, &bl->mut)*, no código do produtor, destrava o semáforo *mut* e faz com que ele espere por uma sinalização na variável *bl->cheio*. Essa sinalização indica que o `buffer` compartilhado não está mais cheio, liberando o produtor para continuar produzindo. A sinalização é feita pelo consumidor após o consumo de algum elemento. Já a chamada *pthread_cond_signal(&bl->vazio)* envia um sinal pela variável *bl->vazio*, liberando um dos `threads` em espera por essa condição. Nesse caso, se o consumidor estiver em espera por e-

lementos disponíveis no `buffer`, caso ele esteja vazio, ele só poderá consumir após o envio desse sinal (indicando a produção de algum elemento). O produtor então sinaliza que acabou de produzir. Todos esses passos são realizados atomicamente.

O código do consumidor é análogo, apenas aguardando caso o `buffer` esteja vazio e sinalizando que acabou de consumir algum elemento, ou seja, que o `buffer` não está mais cheio.

O uso do mecanismo de variáveis condicionais torna a implementação muito complexa quando há a necessidade de controlar mais de uma condição. É importante salientar, por fim, que o exemplo 9.10 é uma forma de simular um monitor na linguagem C.

9.3.2. Classes `Threads` e Métodos Sincronizados

Uma outra abordagem, adotada por JAVA, envolve a disponibilização de classes especiais para a criação e manipulação de `threads` e ainda oferece mecanismos de sincronização de métodos. Ao se criar um objeto de uma dessas classes especiais, JAVA cria um novo fluxo de execução. Além disso, é possível especificar métodos de qualquer classe como sendo sincronizados. A seguir, é mostrado como criar `threads` e como é possível trabalhar com sincronização em JAVA.

9.3.2.1 Classes `Threads`

Há duas maneiras de se criar `threads` em Java: herdando da classe *Thread* ou implementando a interface *Runnable*. Em ambas, o corpo de implementação do `thread` está no método *run*. Se uma classe herda de *Thread*, então ela já é um `thread` e pode ser iniciada através de seu método *start*. Se uma classe implementa *Runnable*, então ela deve ser executada a partir de um *thread*. O exemplo 9.11 mostra a criação de `threads` utilizando herança.

```
class Carro extends Thread {
   public Carro(String nome) {
       super(nome);
   }
   public void run() {
       for (int i=0; i<10;i++) {
           try {
             sleep((int)(Math.random() * 1000));
           }
           catch (Exception e) {
           };
```

```
            System.out.print(getName());
            for (int j=0; j<i; j++)
                System.out.print("--");
            System.out.println(">");
        }
        System.out.println(getName() + " completou a prova.");
    }
}
public class Corrida {
    public static void main(String args[]) {
        Carro carroA = new Carro("Barrichelo");
        Carro carroB = new Carro("Schumacher");
        carroA.start();
        carroB.start();
        try {
              carroA.join();
        } catch (Exception e) {
        }
        try {
              carroB.join();
        } catch (Exception e) {
        }
    }
}
```

Exemplo 9.11 – Criação de `Threads` por Herança

O exemplo 9.11 mostra uma corrida de `threads` com dois participantes: Barrichelo e Schumacher. O primeiro carro tem a vantagem de iniciar primeiro através do método *start*. O *thread*-pai espera o término dos dois carros através do método *join*.

O código do método *run* implementa o comportamento de um *thread*, ou seja, é neste método que ocorrerá a definição das tarefas que serão realizadas pelo *thread*. No exemplo o `thread` irá executar um laço 10 vezes. A cada iteração, o `thread` irá dormir por um período de tempo especificado aleatoriamente através do método *sleep*. Quando acordar, irá imprimir o nome do corredor seguido de sua situação, definida pela quantidade de traços a serem impressos.

Um possível resultado da execução do exemplo 9.11 é mostrado na figura 9.7.

```
Barrichelo>
Schumacher>
Schumacher-->
```

```
Barrichelo-->
Schumacher---->
Barrichelo---->
Schumacher------>
Barrichelo------>
Barrichelo-------->
Schumacher-------->
Barrichelo---------->
Schumacher---------->
Barrichelo------------>
Schumacher------------>
Barrichelo-------------->
Schumacher-------------->
Barrichelo---------------->
Barrichelo------------------>
Barrichelo completou a prova.
Schumacher---------------->
Schumacher------------------>
Schumacher completou a prova.
```

Figura 9.7 - Resultado da Execução do Exemplo 9.11

O exemplo 9.12 mostra a classe *Carro2*, agora utilizando *Runnable*.

```
class Carro2 implements Runnable {
   private String nome;
   public Carro2(String nome) {
       this.nome = nome;
   }
   public void run() {
       for (int i=0; i<10;i++) {
           try {
             Thread.sleep((int)(Math.random() * 1000));
           }
           catch (Exception e) {};
           System.out.print(nome);
           for (int j=0; j<i; j++)
               System.out.print("--");
           System.out.println(">");
       }
       System.out.println(nome + " completou a prova.");
   }
}
public class Corrida2 {
   public static void main(String args[]) {
       Carro2 carroA = new Carro2("Barrichelo");
       Carro2 carroB = new Carro2("Schumacher");
       Thread threadA = new Thread(carroA);
```

```
        Thread threadB = new Thread(carroB);
        threadA.start();
        threadB.start();
        try {
          threadA.join();
        } catch (Exception e) {
        }
        try {
          threadB.join();
        } catch (Exception e) {
        }
    }
}
```

Exemplo 9.12 – Criação de `Threads` Através de Runnable.

A classe *Corrida2* não é herdeira de *Thread*. Assim, a criação de um objeto dessa classe não produz mais um novo fluxo de execução. No entanto, JAVA permite a criação de um objeto da classe *Thread* a partir de um objeto que implementa a interface *Runnable*. Como a classe *Corrida2* não é herdeira de *Thread*, para chamar *sleep* no método *run*, é preciso explicitar a classe na qual *sleep* é definida (*Thread*).

Na maioria das vezes, usa-se a interface *Runnable* quando a classe a ser implementada deve ser uma subclasse de uma classe diferente de *Thread*. Caso isso não seja necessário utiliza-se herança da classe *Thread*.

9.3.2.2 Métodos Sincronizados

Todo objeto em JAVA possui um bloco de operações único. Quando métodos de um objeto em JAVA são declarados como *synchronized*, a JVM (*Java Virtual Machine*) faz com que `threads` chamem esses métodos em exclusão mútua.

Se um `thread` inicia a execução de um método *synchronized*, nenhum outro `thread` conseguirá executar qualquer método *synchronized* daquele objeto até que o primeiro libere o bloco de operações do objeto.

Quando um `thread` faz chamada a um método declarado como *synchronized*, é verificado se outro `thread` já está de posse do bloco de operações daquele objeto. Em caso positivo, o `thread` é bloqueado e colocado no conjunto de `threads` de entrada para aquele objeto. Caso contrário, o `thread` toma posse do bloco de operações e inicia a execução do método.

Quando o `thread` libera o bloco de operações do objeto e há `threads` no conjunto de entrada, a JVM seleciona arbitrariamente algum `thread` desse conjunto e dá a ele a posse do bloco de operações do objeto.

Há duas formas de um `thread` liberar o bloco de operações: através do término do método *synchronized* ou através da chamada ao método *wait()*.

Quando um `thread` chama o método *wait()*, ele libera o bloco de operações, é bloqueado e colocado no conjunto de `threads` em espera e a JVM seleciona outro `thread` do conjunto de entrada para tomar seu lugar.

A chamada do método *notify()* faz com que a JVM selecione um `thread` do conjunto de `threads` em espera e o transfira para o conjunto de entrada, fazendo com que esse `thread` também possa competir pela posse do bloco de operações. A chamada a *notifyAll()* faz com que a JVM transfira todos os `threads` do conjunto em espera para o conjunto de entrada.

A figura 9.5 mostra o ciclo de estados possíveis para `threads` que acessam métodos *synchronized* de um objeto.

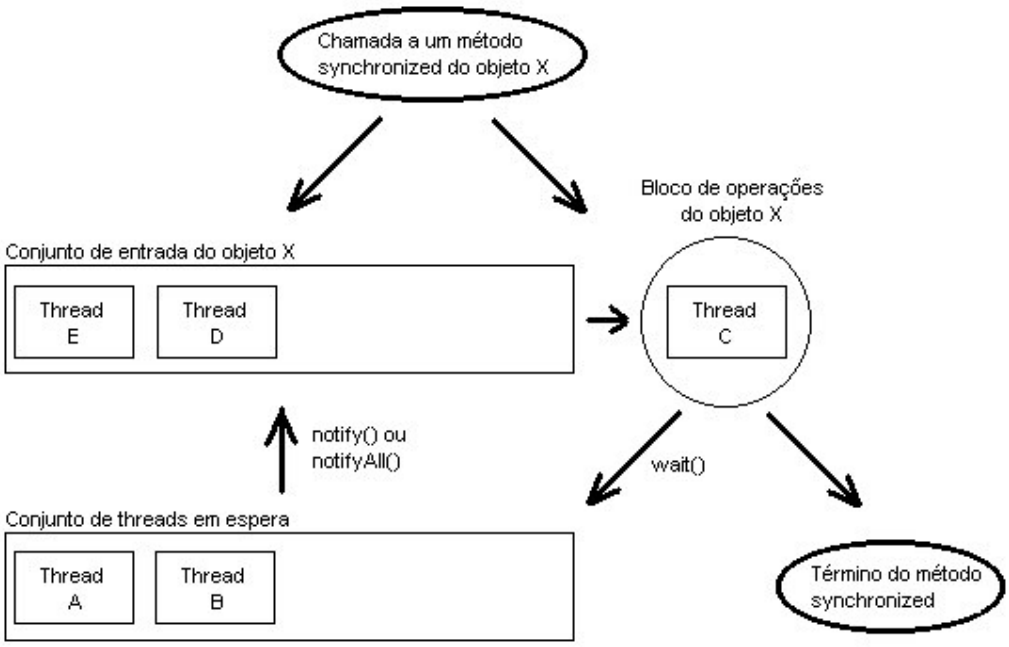

Figura 9.5. Estados de `Threads` no Acesso a Métodos *synchronized*

O `thread` C detém a posse do bloco de operações do objeto X, enquanto os `threads` D e E aguardam a liberação do bloco. Já os `threads` A e B aguardam pela realização de um evento.

Desse modo, somente um `thread` executa um método *synchronized* de um objeto por vez, garantindo exclusão mútua. O exemplo 9.13 mostra a implementação do problema do produtor e consumidor com um `buffer` limitado, isto é, um `buffer` de capacidade finita compartilhado pelos dois `threads`.

```
class BufferLimitado {
   private int capacidade;
   private int n;
   private int buffer[];
   private int fim;
   private int ini;
   public BufferLimitado(int capacidade) {
       this.capacidade = capacidade;
       n = 0;
       fim = 0;
       ini = 0;
       buffer = new int[capacidade];
   }
   public synchronized void inserir(int elemento) {
       while ( n == capacidade) {
           try {
             wait();
           } catch (Exception e) {
           }
       }
       buffer[fim] = elemento;
       fim = (fim + 1) % capacidade;
       n = n + 1;
       notify();
   }
   public synchronized int retirar() {
       while ( n == 0) {
           try {
             wait();
           } catch (Exception e) {
           }
       }
       int elem = buffer[ini];
       ini = (ini + 1) % capacidade;
       n = n – 1;
       notify();
       return elem;
```

```
    }
}
class Produtor extends Thread {
    private BufferLimitado buffer;
    public Produtor(BufferLimitado buffer) {
        this.buffer = buffer;
    }
    public void run() {
        int elem;
        while (true) {
            elem = (int)(Math.random() * 10000);
            buffer.inserir(elem);
            System.out.println("produzido: " + elem);
            Try {
              Thread.sleep((int)(Math.random() * 1000));
            } catch (Exception e) {
            }
        }
    }
}
class Consumidor extends Thread {
    private BufferLimitado buffer;
    public Consumidor(BufferLimitado buffer) {
        this.buffer = buffer;
    }
    public void run() {
        int elem;
        while (true) {
            elem = buffer.retirar();
            System.out.println("consumido: " + elem);
            Try {
              Thread.sleep((int)(Math.random() * 1000));
            }catch (Exception e) {
            }
        }
    }
}
public class Fabrica {
    public static void main(String args[]) {
        BufferLimitado buffer = new BufferLimitado(10);
        Produtor produtor = new Produtor(buffer);
        Consumidor consumidor = new Consumidor(buffer);
        Produtor.start();
```

```
        Consumidor.start();
        Try {
          Produtor.join();
        } catch (Exception e) {
        }
        try {
          consumidor.join();
        } catch (Exception e) {
        }
    }
}
```

Exemplo 9.13 – Produtor e Consumidor com Classe `Thread`

A classe *Fabrica* cria uma nova instância da classe *BufferLimitado* de capacidade igual a 10 e os `threads` *Produtor* e *Consumidor*. A classe *BufferLimitado* possui um `buffer` compartilhado pelos `threads` *Produtor* e *Consumidor* e métodos para inserção e retirada de elementos do `buffer`. Observe que os métodos *inserir* e *retirar* da classe *BufferLimitado* são declarados como *synchronized*, garantindo exclusão mútua no acesso ao `buffer`.

O método *inserir* chama o método *wait()* quando o `buffer` fica cheio, o que faz o `thread` liberar o bloco de operações, ser bloqueado e colocado no conjunto de `threads` em espera. Há também uma chamada ao método *notify()*, que faz com que a JVM transfira um `thread` em espera para o conjunto de entrada, para que ele novamente possa competir pela posse do bloco de operações. Observe que o método *retirar* é análogo, apenas chamando o método *wait()* quando o `buffer` se tornar vazio.

Caso o método *retirar* não tivesse sido declarado como *syncronized*, poderia haver inconsistência dos dados já que um método *syncronized* pode ser executado ao mesmo tempo de outro não declarado como tal.

O exemplo 9.13 mostra uma implementação de monitor em JAVA. À exceção do construtor, todos os métodos da classe *BufferLimitado* são *syncronized*, o que garante a exclusão mútua no acesso aos dados do `buffer`.

9.3.3. Módulos Concorrentes

Outra abordagem para permitir a construção de sistemas concorrentes consiste na definição e uso de módulos concorrentes. A execução de programas na linguagem de programação ADA consiste na execução de uma ou mais tarefas (`tasks`). Em geral, cada tarefa representa um `thread`

de processamento independente e concorrente com pontos de interação com outras tarefas.

ADA disponibiliza dois mecanismos de comunicação entre tarefas: passagem de mensagens (`rendevouz`) e objetos protegidos.

9.3.3.1. Passagem de Mensagens (`Rendezvous`)

O `rendevouz` é o mecanismo básico de sincronização e comunicação de tarefas em ADA. O modelo de ADA é baseado no modelo de interação cliente-servidor.

Uma tarefa, a servidora, declara um conjunto de serviços disponibilizados para outras tarefas (as clientes). Isto pode ser feito declarando um ou mais pontos de entrada na especificação da tarefa.

A troca de mensagem `rendevouz` é feita quando uma tarefa chama uma entrada de outra tarefa. Como dito na seção 9.1.6, o `rendevouz` apenas ocorre quando os dois lados, o cliente e o servidor, se encontram. Quando a tarefa cliente faz chamada a uma entrada, ela espera até sua chamada ser aceita pela tarefa servidora. Quando a tarefa servidora indica que aceita uma determinada entrada, ela espera até alguma tarefa cliente fazer chamada àquela entrada.

Uma tarefa possui uma especificação e um corpo. Na especificação podem-se declarar `entry points` (pontos de entrada), os quais definem interfaces com o mundo exterior à tarefa. No corpo é feita a implementação do restante da descrição da tarefa. Para declaração de um ponto de entrada é utilizado o comando *entry* e o código relacionado à entrada é definido no corpo da tarefa através do co-mando *accept*. A Figura 9.6 mostra o esquema de uma tarefa simples, com destaque para o modo de utilização dos comandos *entry* e *accept*.

```
task <nome-da-task> is
     ...
     entry <nome-da-entrada>
     ...
end <nome-da-task>

task body <nome-da-task> is
     ...
     accept <nome-da-entrada>
     do
          <comandos>
     end <nome-da-entrada>
     ...
end <nome-da-task>
```

Figura 9.6 - Esquema de uma Tarefa Simples em ADA

Os pontos de entrada são mutuamente exclusivos, ou seja, um ponto de entrada nunca pode ser executado concorrentemente com outro ponto de entrada da mesma tarefa.

Quando a tarefa executa o comando *accept* para alguma entrada, ela vai "dormir" até que a entrada seja solicitada por outra rotina. O comando *accept* não é feito na forma de espera ocupada, como já discutido anteriormente.

Uma tarefa pode possuir vários pontos de entrada opcionais, permitindo assim que outras tarefas se comuniquem com ela através deles. Para tornar isso possível, ADA disponibiliza o comando *select*, o qual permite a utilização de vários comandos *accept* opcionais. Isso é demonstrado no exemplo 9.14, que define um tipo *task Carro* que possui três pontos de entrada: *iniciar*, *andar_para_frente* e *andar_para_tras*.

```
with Text_IO; use Text_IO;
procedure teste is
  task type Carro is
    entry iniciar(id: integer);
    entry andar_para_frente;
    entry andar_para_tras;
  end Carro;
  task body Carro is
    posicao: integer;
    meu_id: integer;
  begin
    accept iniciar(id: integer) do
      posicao := 0;
      meu_id := id;
    end iniciar;
    Put_Line("O carro " & Integer'Image(meu_id) &
                  " esta na posicao " & Integer'Image(posicao));
    select
      accept andar_para_frente do
        posicao := posicao + 1;
      end andar_para_frente;
    or
      accept andar_para_tras do
        posicao := posicao - 1;
      end andar_para_tras;
    end select;
   Put_Line("O carro " & Integer'Image(meu_id) &
                  " esta na posicao " & Integer'Image(posicao));
  end Carro;
```

```
   carro1: Carro;
   carro2: Carro;
begin
   carro1.iniciar(1);
   carro2.iniciar(2);
   carro1.andar_para_frente;
   carro2.andar_para_tras;
end teste;
```

Exemplo 9.14 - Utilização de Tarefas e Entradas em ADA

Quando a variável *carro1* é declarada, uma tarefa do tipo *Carro* é iniciada parando no primeiro comando *accept* em espera pela solicitação da entrada *iniciar*. Quando ocorre a chamada ao ponto de entrada *carro1.iniciar(1)*, a tarefa continua a execução imprimindo na tela que o carro 1 está na posição *0,* e é novamente interrompida no comando *select* esperando por uma chamada à entrada *andar_para_frente* ou à *entrada andar_para_tras*. É feita uma chamada à entrada *andar_para_frente* fazendo com que a variável *posição* seja incrementada e seja impresso que o carro 1 está na posição *1*. A execução é análoga para o carro *2*, diferindo apenas em que é feita uma chamada à entrada *andar_para_trás.*

É válido lembrar que as entradas são mutuamente exclusivas e, portanto, nesse exemplo, um mesmo carro não pode andar para frente e para trás. A figura 9.7 mostra um dos possíveis resultados da execução do programa do exemplo 9.14.

```
O carro 1 esta na posicao 0
O carro 1 esta na posicao 1
O carro 2 esta na posicao 0
O carro 2 esta na posicao -1
```

Figura 9.7 – Resultado da execução do Exemplo 9.14

9.3.3.2. Objetos Protegidos

ADA 95 também fornece sincronização através de objetos protegidos. Operações protegidas podem ser procedimentos (`procedures`), funções (`functions`) e entradas (`entries`).

Chamadas a procedimentos e entradas protegidos são executadas em exclusão mútua, ou seja, nenhuma operação do mesmo objeto protegido pode ser executada concorrentemente. As funções podem ser executadas em paralelo, mas não quando um procedimento ou entrada protegidos estão sendo executados. Funções não podem afetar o estado do objeto protegido, ou seja, possuem permissão apenas para leitura dos dados encapsulados.

Um objeto protegido em ADA encapsula dados e provê acesso a esses dados apenas através de entradas protegidas ou subprogramas protegidos. O compilador garante que estes subprogramas e entradas são executados em exclusão mútua.

A unidade protegida pode ser declarada com um tipo ou uma simples instância. No último caso, nós dizemos que a unidade corresponde a um tipo anônimo. A unidade protegida possui uma especificação e um corpo.

A Figura 9.7 mostra o esquema de uma unidade protegida simples. Funções, procedimentos e entradas podem também receber argumentos, apesar de não ser mostrado na figura.

```
protected type <nome-da-unidade> is
     function <nome-da-função> return <tipo-retorno>;
     procedure <nome-do-procedimento>;
     entry <nome-da-entrada>;
     <declaração de variáveis>
end <nome-da-unidade>;

protected body <nome-da-unidade> is
     entry <nome-da-entrada> when <condição> is
     begin
          <comandos>
     end <nome-da-entrada>;
     procedure <nome-do-procedimento> is
     begin
          <comandos>
     end <nome-do-procedimento>;
     function <nome-da-função> return <tipo-retorno> is
     begin
          <comandos>
          return <valor>;
     end <nome-da-função>;
end <nome-da-unidade>;
```

Figura 9.8 - Esquema de uma Unidade Protegida Simples em ADA

Uma entrada protegida é similar a um procedimento protegido, garantindo exclusão mútua e possuindo acesso de leitura e escrita aos dados encapsulados. Entretanto, o acesso a uma entrada protegida é feito através da verificação de uma expressão booleana identificando uma barreira para execução do corpo da entrada. Se a condição da barreira é falsa quando ocorre uma chamada à entrada, a tarefa chamadora é suspensa até que a condição da barreira se torne verdadeira e nenhuma outra tarefa esteja executando dentro do objeto protegido.

Entradas protegidas podem ser usadas para implementar condições de sincronização. O exemplo 9.15 mostra a implementação de um objeto

protegido *Objeto_Sinal* que pode estar em dois estados: aberto (*Aberto = true*) e fechado (*Aberto = false*).

```
protected type Objeto_Sinal is
  entry Espera;
  procedure Sinal;
  function Esta_Aberto return boolean;
private
  aberto: boolean := false;
end Objeto_Sinal;
protected body Objeto_Sinal is
  entry Espera when aberto is
  begin
    aberto := false;
  end Espera;
  procedure Sinal is
  begin
    aberto := true;
  end Sinal;
  function Esta_Aberto return boolean is
  begin
    return aberto;
  end Esta_Aberto;
end Objeto_Sinal;
```

Exemplo 9.15 – Implementação de um Tipo Objeto Protegido

O corpo da chamada à entrada *Espera* somente é executado quando o sinal está aberto (*when Aberto*). Se uma tarefa faz chamada à entrada *Espera* de algum objeto protegido do tipo *Objeto_Sinal* e o estado desse objeto está como fechado, a tarefa vai "dormir" até que o sinal abra e nenhuma outra tarefa esteja executando no objeto protegido.

O Exemplo 9.16 tem o mesmo propósito do exemplo 9.13, entretanto, nesse exemplo são utilizados os recursos da linguagem ADA para trabalhar com concorrência.

```
package Fabrica is
  type Buffers is array(positive range <>) of integer;
  protected type BufferLimitado(capacid : natural) is
    entry Retirar(elem : out Integer);
    entry Inserir(elem : in Integer);
  private
    buf: Buffers(1..capacid);
    n: natural := 0;
    ini, fim: positive := 1;
```

```
    capacidade: natural := capacid;
  end BufferLimitado;
  type ABuffer is access bufferLimitado;
  task type Produtor is
    entry Iniciar(buf : in ABuffer);
  end produtor;
  task type Consumidor is
    entry Iniciar(buf : in ABuffer);
  end consumidor;
end Fabrica;
```

a – Arquivo BufferLimitado.ads (Declarações)

```
with Ada.Text_Io; use Ada.Text_Io;
package body Fabrica is
  protected body BufferLimitado is
    entry Retirar(elem : out integer) when n > 0 is
    begin
      elem := buf(ini);
      if (ini = capacidade) then ini := 1;
      else ini := ini + 1;
      end if;
      n := n - 1;
    end Retirar;
    entry Inserir(elem : in integer) when n < capacidade is
    begin
      buf(fim) := elem;
      if (fim = capacidade) then fim := 1;
      else fim := fim + 1;
      end if;
      n := n + 1;
    end Inserir;
  end BufferLimitado;
  task body Produtor is
    elem: integer;
    pbuf: ABuffer;
    i: integer;
  begin
    accept Iniciar(buf : in ABuffer) do
      pbuf := buf;
    end;
    for i in 0..1000 loop
      elem := i;
      pbuf.inserir(elem);
```

```
      Put_Line("Produzido: " & Integer'Image(elem));
    end loop;
  end Produtor;
  task body Consumidor is
    elem: integer;
    cbuf: ABuffer;
    i: natural;
  begin
    accept Iniciar(buf : in aBuffer) do
      cbuf := buf;
    end;
    for i in 0..1000 loop
      cbuf.retirar(elem);
      Put_Line("Consumido: " & Integer'Image(elem));
    end loop;
  end Consumidor;
end Fabrica;
```

b – Arquivo BufferLimitado.adb (Implementação)

```
with Fabrica; use Fabrica;
with Ada.Text_Io; use Ada.Text_Io;
procedure Teste is
  prod: Produtor;
  cons: Consumidor;
  buf: ABuffer := new BufferLimitado(10);
begin
  prod.Iniciar(buf);
  cons.Iniciar(buf);
end Teste;
```

c – Arquivo teste.adb (Instanciação)

Exemplo 9.16 – BufferLimitado como Objeto protegido.

Na implementação do *BufferLimitado* foi criado um tipo objeto protegido que será compartilhado por tarefas produtoras e consumidoras. Os métodos *inserir* e *retirar* foram declarados como entradas, ou seja, não poderão ter acesso ao *buffer* em um mesmo momento, como visto na parte de declarações (parte a) do exemplo 9.16.

Já na parte de implementação (parte b) do exemplo 9.16 é importante notar no corpo de *BufferLimitado* que as entradas *inserir* e *retirar* possuem condicionais que só permitirão a ação caso o *buffer* não estiver cheio ou vazio, respectivamente.

A implementação das tarefas *Produtor* e *Consumidor* apenas inserem e retiram elementos do *buffer*, respectivamente, e esses métodos foram declarados de forma a garantir exclusão mútua.

Já na parte de instanciação (parte c) do 9.16 foi declarado um produtor, um consumidor e um `buffer` que será compartilhado pelas duas tarefas. As tarefas *Produtor* e *Consumidor* são inicializadas no momento em que são declaradas, mas elas ficam paradas no método *accept* de seus corpos até serem chamadas em *prod.Iniciar(buf)* e *cons.Iniciar(buf)*, respectivamente. Este tipo de implementação caracteriza um `rendezvous`.

É importante notar que o compilador assegura a exclusão mútua. Portanto, o exemplo é uma implementação de monitor na linguagem ADA.

9.3 Considerações Finais

Nesse capítulo foram apresentados os benefícios e os problemas que podem surgir com a programação concorrente. Como solução, foram discutidos mecanismos propostos para apoiar a programação concorrente, tais como os semáforos e os monitores.

Os semáforos são mecanismos de sincronização eficientes que garantem exclusão mútua. Embora simples e eficazes, sua implementação nem sempre é fácil e erros podem facilmente acontecer com pequenos deslizes do programador.

Para lidar com esses tipos de erros, alguns mecanismos foram propostos para abstrair o conceito de semáforos em LPs, com destaque para os monitores. Esses mecanismos deixam para o compilador a responsabilidade de garantir o acesso exclusivo à região crítica.

Também foram apresentadas algumas abordagens utilizadas por LPs para permitir a implementação de sistemas concorrentes.

A linguagem C não oferece mecanismos próprios para lidar com concorrência. A criação, manipulação e sincronização de processos e `threads` é possível através do uso de chamadas de sistema ou de bibliotecas de funções. Um problema da utilização de bibliotecas de funções é que elas são específicas da plataforma de execução. Outro grande problema é que a implementação de alguns mecanismos de sincronização, tal como semáforos, não é trivial.

A linguagem JAVA disponibiliza classes especiais para a criação e manipulação de `threads` e oferece mecanismos de sincronização de métodos. A implementação de sistemas concorrentes se torna simples nesta linguagem.

Já a linguagem ADA permite a construção de sistemas concorrentes através da definição e uso de módulos concorrentes. A execução de programas nesta linguagem consiste na execução de uma ou mais tarefas. A sincronização entre as tarefas pode ser feita através de troca de mensagens ou de objetos protegidos. Enquanto o mecanismo de troca de mensagens é mais elegante, o mecanismo de objetos protegidos é mais eficiente. ADA, portanto, oferece mais recursos e flexibilidade para a implementação de sistemas concorrentes.

Maiores informações sobre concorrência podem ser obtidas nos livros de Watt [WATT, 1992], Silberchatz, Gagne e Galvin [SILBERSCHATZ, GAGNE & GALVIN, 2001], Oliveira, Carissimi e Toscani [OLIVEIRA, CARISSIMI & TOSCANI, 2001] e Burns e Wellings [BURNS & WELLINGS, 2001].

9.4 Exercícios

1. Se a programação concorrente traz dificuldades para a programação, quais vantagens se têm com a sua utilização?

2. Quais são as principais diferenças entre `threads` e processos? Cite as respectivas vantagens e desvantagens de sua utilização.

3. Quais são as principais diferenças entre memória compartilhada e de troca de mensagens? Cite vantagens e desvantagens.

4. Mostre como é possível utilizar semáforos em substituição aos laços *while* dos códigos do produtor e do consumidor no problema mostrado no exemplo 9.2, reduzindo assim o *overhead* do sistema.

5. Faça uma classe *Semaforo* em JAVA que implemente as operações P e V de um semáforo. Utilize para isso os métodos *wait()* e *notify()*. A classe deve possuir métodos P e V em exclusão mútua (*synchronized*).

6. Implemente uma tarefa *Semaforo* em ADA utilizando entradas P e V.

7. Suponha que sejam retiradas as chamadas às entradas *iniciar* de *carro1* e *carro2* no exemplo 9.14. Indique a opção abaixo com o resultado correto da execução.

   a) Aparecerá na tela as seguintes mensagens:

```
O carro 1 esta na posicao 0
O carro 2 esta na posicao 0
```

b) Aparecerá na tela as seguintes mensagens:

```
O carro 1 esta na posicao 0
O carro 1 esta na posicao 1
O carro 2 esta na posicao 0
O carro 2 esta na posicao -1
```

c) Não aparecerá nada na tela.
d) O programa dará erro em tempo de compilação.

8. Implemente o programa do exemplo 9.10 retirando o semáforo, descreva o que acontece e justifique.

9. Quais são as principais características das linguagens C, JAVA e ADA relacionadas à programação concorrente?