Capítulo 13

CONSIDERAÇÕES COMPLEMENTARES

El moreno:
Ya saben que de mi madre
fueron diez los que nacieron;
mas ya no esiste el primero
y más querido de todos:
murió, por injustos modos,
a manos de un pendenciero.

Los nueve hermanos restantes
como güérfanos quedamos;
dende entonces lo lloramos
sin consuelo, creanmeló,
y al hombre que lo mató
nunca jamás lo encontramos.

Y queden en paz los güesos
de aquel hermano querido;
a moverlos no he venido,
mas, si el caso se presienta,
espero en Dios que esta cuenta
se arregle como es debido.

Este capítulo apresenta (velhos) conceitos e notas históricas que complementam a discussão das ferramentas e dos paradigmas de programação apresentados até aqui, bem como faz um resumo da tecnologia atual disponível para o desenvolvimento de aplicações concorrentes.

13.1 O envelhecimento do software

É comum que idéias antigas ressurjam com roupagens novas, dando a impressão que se tratam de “novas descobertas”. Um exemplo é a idéia da programação orientada a objetos, que (por muitos) foi vista como nova, mas cuja fundamentação havia sido dada na linguagem Simula, em 1965 [DAH 66]. Isso ocorre no que diz respeito ao software. Apesar de algumas técnicas de programação terem se tornado obsoletas com o passar do tempo (isto aconteceu, por exemplo, com a técnica de modificar instruções durante a execução), muitas das técnicas de programação serão usadas eternamente (pelo menos enquanto durar o modelo de arquitetura da máquina de von Newman). No hardware, sim, tem-se uma velocidade de desatualisação grande, pois novas tecnologias tornam ultrapassadas até tecnologias recentes. Aliás, na comparação dos avanços do hardware e do software, pode-se dizer a mesma coisa de duas maneiras distintas: (1) que o hardware avança (ou torna-se obsoleto) mais rapidamente, (2) que o software tem uma vida muito mais longa.

13.2 O conceito de secretária

A estruturação com monitor foi usada pela primeira vez por P. B. Hansen no projeto do sistema operacional multiprogramado RC4000, no início da década de 70 [BRI 70]. Mais tarde, em 1974, o conceito foi formalizado por C. A. R. Hoare [HOA 74], num artigo clássico sobre prova de correção de programas concorrentes. Nessa mesma época, E. W. Dijkstra [DIJ 75] apresentou o conceito de *secretária*. Tal como um monitor, uma secretária tinha a finalidade de encapsular dados críticos usados por processos concorrentes.

Diferentemente de um monitor, a secretária era um processo com vida própria que se comunicava com os clientes através de troca de mensagens. Os dados críticos eram representados como dados locais, na secretária, e as operações a serem realizadas sobre os mesmos eram requisitadas através de mensagens enviadas pelos processos clientes. Os resultados de uma operação também eram devolvidos via o mecanismo de comunicação. Como se vê, existem semelhanças entre os conceitos de secretária e de task da linguagem ADA.

13.3 A organização cliente-servidor

A organização cliente-servidor tem se popularizado, sendo anunciada como a base de muitos sistemas operacionais. No fundo, essa organização não passa da conhecida organização baseada em um *kernel*, onde a maior parte do sistema é implementada por um conjunto de processos cooperantes, que utilizam os mecanismos implementados no *kernel*.

Na organização cliente-servidor, os serviços do sistema operacional são implementados por processos (com *multithreads* ou não) os quais, por sua vez, usam mecanismos básicos de

comunicação e sincronização implementados no *kernel*. Os processos do SO desempenham papéis de clientes e/ou servidores, sendo que um servidor (prestador de serviços) pode ser cliente de um outro servidor, evidentemente.

13.4 *Kernel* distribuído

Ninguém tem dúvidas que a tendência atual da computação é no sentido da distribuição do processamento. Para mostrar que a organização baseada em um *kernel* continua válida, será considerada uma possível organização para um sistema operacional distribuído. Considerando o caso geral de um conjunto de nós processadores interligados através de linhas de comunicação, uma organização natural para o software é a seguinte. Em cada nó é implementado um *kernel*, o qual entre outros serviços, oferece facilidades para comunicação com outros nós. Com isto, o restante do sistema pode ser implementado da velha maneira, isto é, por processos que se intercomunicam (localmente ou remotamente) através dos mecanismos do *kernel*. Dependendo da sofisticação desse "*kernel* distribuído", a distribuição (tanto do hardware como do software) pode ser completamente transparente aos usuários.

Na verdade, apesar de problemas novos serem introduzidos pela distribuição do processamento, muitas das soluções para sistemas centralizados continuam a ser usadas. Por exemplo, os algoritmos para detecção de *deadlocks* continuam sendo válidos; a complicação nova é a coleta das informações sobre o estado do sistema, pois estas informações agora estão fisicamente distribuídas.

13.5 Chamada remota de procedimentos

Na área de intercomunicação de processos, referida em inglês pela sigla IPC (*Inter Process Communication*), tem-se uma variedade de mecanismos. Nos sistemas distribuídos, um mecanismo que tem se sobressaído é o denominado *remote procedure call* ou RPC, o qual é discutido a seguir. Ainda aqui não há novidade, pois este mecanismo foi usado pela primeira vez em 1978 na linguagem *Distributed Process*, definida por Brinch Hansen [BRI 78].

Um problema no uso das funções primitivas *send*/*receive* é o baixo nível da interação. O programador se vê obrigado a abandonar as estruturas modulares de comunicação (procedimentos e outros mecanismos de alto nível) e passa a ter que usar operações *send* e *receive*, as quais ficam espalhadas pelos códigos dos processos. As chamadas remotas de procedimentos surgiram como um conceito de alto nível para eliminar esta inconveniência.

Do ponto de vista do programador, uma RPC tem o mesmo efeito de uma chamada normal de procedimento: transferência da execução para outra *procedure* e "suspensão" do processo chamador. Um comando de retorno da *procedure* ocasiona a volta do controle para o chamador, onde a execução continua na instrução seguinte à chamada. A distinção principal

entre um procedimento normal e um procedimento remoto é que o último reside num espaço de endereçamento separado, o que inviabiliza o compartilhamento de variáveis globais (todas as informações precisam ser passadas através de parâmetros).

Do ponto de vista do programador, uma *procedure* remota parece-se e comporta-se exatamente como uma *procedure* tradicional. Ela é chamada usando a sintaxe usual e, idealmente, o chamador nem toma conhecimento que a *procedure* é remota. No nível de implementação, entretanto, os procedimentos remotos são bem diferentes dos procedimentos normais. Como um procedimento remoto é executado em um espaço de endereçamento diferente, possivelmente em outro computador, deve ser criado um processo separado para executar o procedimento. E esse processo pode ser criado dinamicamente ou estaticamente.

No caso da criação dinâmica, um novo processo é criado (e posteriormente destruído) para cada invocação do procedimento. Este *overhead* não existe na abordagem estática, onde se utiliza um processo permanente que encapsula a *procedure* e a executa. Cada chamada do tipo "*call RP*(*param*, *result*)", onde *RP* é um procedimento remoto, é compilada para um código que inclui um par de comandos *send* e *receive*, onde *send* transmite os parâmetros de entrada e *receive* espera pela chegada dos resultados correspondentes. Do outro lado, o processo que encapsula *RP* inicia com um comando *receive* (com nomeação implícita - ver seção 11.1) e termina com um comando *send*. Esta implementação é ilustrada pelo seguinte esqueleto de código, supondo que *pRP* seja a identificação do processo que encapsula *RP*:

```
/*Chamada de RP:*/                    /*Processo que contém RP:*/
---                                   process pRP:
send(pRP, param)                      loop
receive(pRP, result)                      receive(*, param);
---                                       call RP(param, result)
                                          send(*, result)
                                      end loop
```

Pode-se constatar algumas semelhanças entre um procedimento remoto e um procedimento tipo *entry*, usado em um esquema de *rendezvous*. Em ambos os casos a comunicação é síncrona e em ambos os casos o procedimento chamado não é executado pelo processo que o chama. O processo chamador fica bloqueado enquanto o processo remoto executa o procedimento. Após a execução, os parâmetros de saída são passados ao processo chamador e este prossegue sua execução. Devido a estas semelhanças, alguns autores consideram o esquema de *rendezvous* como um esquema de chamada remota de procedimentos.

As RPCs eliminam o vazio entre os esquemas de sincronização orientados a procedimentos e os esquemas baseados em trocas de mensagens. Do ponto de vista do programador, as *procedures* remotas são construções de alto nível que se encaixam bem na filosofia das linguagens bloco-estruturadas. Do ponto de vista do SO, as *procedures* remotas são

adequadas para sistemas distribuídos, pois a implementação é realizada em termos de mecanismos simples de troca de mensagens.

13.6 Sobre o desenvolvimento de aplicações práticas

A linguagem V4 utilizada neste livro possui sintaxe simples, semântica clara, boas facilidades de depuração, permite praticar com os principais mecanismos de controle existentes e, portanto, é ótima para os fins acadêmicos a que se destina. Contudo, ela não é adequada para o desenvolvimento de software (aplicações). Os tipos de dados são limitados (por exemplo, não existem reais, variáveis estruturadas ou ponteiros) e não há preocupação com aspectos de eficiência. Como tal, o desenvolvimento de aplicações práticas terá que ser feito utilizando a tecnologia disponível no mercado.

Este livro ensinou várias ferramentas, todas bem definidas e teoricamente justificadas. Além disso, ensinou as soluções dos problemas de sincronização e comunicação que costumam surgir na prática, os quais são representados pelos quatro problemas clássicos da programação concorrente. Raramente, um problema prático deixará de se enquadrar em um desses quatro modelos (por isso, aliás, eles são considerados problemas clássicos). E esses problemas foram solucionados com todos os principais paradigmas de programação. Isto significa que o bom aproveitamento do conteúdo deste livro dará o preparo necessário para o desenvolvimento de qualquer tipo de aplicação prática. Por outro lado, a visão geral dos problemas, das ferramentas e dos paradigmas de programação, aqui apresentada, permite que o estudante tenha uma opinião crítica sobre a área, o que vai ser muito importante para o seu sucesso quando ele estiver envolvido no projeto de aplicações concorrentes.

É importante ressaltar que o embasamento fornecido por este livro independe do tipo da arquitetura que o computador venha a apresentar. Tanto faz ter-se uma arquitetura simples, tipo máquina de von Newman[68], uma arquitetura distribuída ou uma arquitetura para computação de alto desempenho. Com o embasamento adquirido, não será difícil aprender o uso de novas ferramentas ou se adaptar a novos contextos e ambientes de desenvolvimento.

13.7 Modelos de programas distribuídos

Um programa distribuído (ou paralelo) é composto por processos comunicantes, executados em processadores diferentes, que cooperam entre si para a implementação de uma aplicação (por exemplo, a realização de um cálculo científico, a implementação de um sistema de atendimento *on-line*, etc.).

[68] Máquina com um processador e um módulo de memória.

Os modelos que costumam ser utilizados para o desenvolvimento de programas distribuídos são resumidos a seguir.

Divisão e conquista

Esta técnica consiste em dividir uma tarefa em sub-tarefas menores, do mesmo tipo, e criar processos filhos para executar essas sub-tarefas. Os filhos executam e devolvem os resultados ao processo pai que combina os resultados gerando o resultado da tarefa original.

Pipeline

Um *pipeline* é formado por um conjunto ordenado de processos que trocam informações em um fluxo contínuo (tipo linha de montagem). Para enviar dados a um processo em outro processador, o processo remetente agrupa os dados e os remete. O processo recebedor extrai os dados da mensagem, os processa e envia o resultado para o processo seguinte no *pipeline*.

Mestre/escravo

O programa é organizado de forma a ser constituído por um processo (o mestre) que executa parte da tarefa e divide o restante entre os demais processos (escravos). Cada escravo executa sua tarefa e envia os resultados ao mestre, que envia uma nova tarefa para o escravo.

***Pool* de trabalho**

Um conjunto de tarefas é depositado em uma área acessível aos processos componentes do programa distribuído. Cada processo retira uma tarefa e a executa. Esta fase se repete até que todo o conjunto de tarefas seja executado.

Fases paralelas

O programa distribuído é formado por fases, sendo necessário que todos os processos terminem uma fase para cada um poder passar à sua fase seguinte. Mecanismos de sincronização tipo barreira devem ser usados para que um processo espere pelos demais para então passar à fase seguinte.

No que segue será feito um resumo da tecnologia atual para o desenvolvimento de aplicações concorrentes. Esta tecnologia de programação é utilizada inclusive nas arquiteturas (para processamento) de alto desempenho. Isto permitirá constatar que não existe qualquer novidade em relação aos mecanismos estudados neste livro.

13.8 Concorrência em arquiteturas avançadas

Atualmente, as arquiteturas em voga para o processamento de alto desempenho são os agregados de computadores (*clusters*), arquiteturas compostas por diversos computadores autônomos – não raro dotados de 2 ou 4 processadores, os chamados *Symmetric Multi-Processors* (SMPs) [COS 02]. O sucesso dessas arquiteturas deve-se ao fato delas serem versáteis e apresentarem

baixo custo. Este tipo de configuração de hardware permite explorar a concorrência em dois níveis: dentro de cada nó (intra-nó) e entre os diferentes nós (inter-nó).

Nas seções que seguem serão apresentadas as três principais ferramentas utilizadas para a programação concorrente em agregados de computadores: *Pthreads* (*threads* POSIX), *Parallel Virtual Machine* (PVM) e *Message Passing Interface* (MPI). A primeira é voltada para arquiteturas com memória compartilhada, sendo adequada, portanto, para a concorrência intra-nó. As outras duas são voltadas para arquiteturas distribuídas (ou em rede), sendo adequadas, portanto, para a concorrência entre diferentes nós.

13.9 Formas de implementar *threads*

Recordando, a multiprogramação com *threads* (*multithreading*) ou "multiprogramação leve", permite que os programas sejam executados com desempenho bastante superior ao que seria obtido com uma "multiprogramação pesada" (isto é, multiprogramação através de processos convencionais, nos quais existe uma única linha de execução).

No caso de se ter *multithreading* dentro de processos, estes se tornam um pouco mais pesados. Em compensação, uma aplicação implementada por um único processo com *N threads* (multiprogramação leve) será mais leve que a mesma aplicação implementada por *N* processos convencionais (multiprogramação pesada).

Uma questão interessante diz respeito a granularidade[69] que pode ser obtida com os dois tipos de multiprogramação. Como as *threads* são menos onerosas, elas permitem que o programador descreva sua aplicação com um maior número de atividades concorrentes sem a perda substancial de desempenho que seria verificada caso essa mesma descrição fosse feita através de processos convencionais. Na multiprogramação leve, a granularidade é definida em termos de procedimentos concorrentes, enquanto que na multiprogramação pesada a granularidade é definida em termos de programas (ou módulos) completos. Isto significa que a multiprogramação leve permite maior granularidade que a multiprogramação com processos convencionais. Com o aumento da capacidade de execução paralela das máquinas, é muito conveniente ter-se grande granularidade.

O suporte para *threads* (ferramentas para criação e sincronização de *threads*) pode ser oferecido diretamente pelo hardware (ou *firmware*), pelo sistema operacional (normalmente por seu *kernel*) ou por bibliotecas especializadas, integradas ou não ao sistema operacional. No caso dos sistemas Solaris, AIX e Linux, as *threads* são disponibilizadas por bibliotecas próprias do sistema. No caso dos sistemas Minix e Windows 95/98, a programação com *threads* é possível através de bibliotecas não integradas ao sistema operacional. O fato de serem ou não

[69] Maior granularidade significa mair número de grãos, de menor tamanho.

disponibilizadas diretamente pelo sistema operacional influi diretamente na forma como os programas são executados e, portanto, no desempenho.

As duas maneiras básicas de implementar o conceito de *thread* podem ser caracterizadas pela forma de escalonamento do processador [OLI 01]. Na primeira, o sistema operacional suporta apenas processos convencionais, isto é, processos com uma única *thread*. O conceito de *thread* é então implementado pelo próprio processo a partir de uma biblioteca ligada ao programa do usuário. Devido a essa característica, *threads* implementadas dessa forma são denominadas de *threads* do nível do usuário (*user-level threads*). No segundo caso, o sistema operacional suporta diretamente o conceito de *thread*. A gerência de fluxos de execução pelo sistema operacional não é mais orientada a processos mas sim a *threads*. As *threads* que seguem esse modelo são ditas *threads* do nível do sistema (*kernel threads*).

O primeiro método é conhecido por *N*:1 (*many-to-one*). A principal vantagem é o fato de as *threads* serem implementadas em espaço de usuário, não exigindo nenhuma interação com o sistema operacional. O chaveamento de contexto é mais rápido e o custo para criação e destruição é zero (do ponto de vista do sistema operacional). A biblioteca é responsável pela divisão, entre as *threads*, do tempo alocado ao processo. O sistema operacional preocupa-se apenas em dividir o tempo do processador entre os diferentes processos. A desvantagem desse método é que as *threads* são efetivamente simuladas a partir de um único fluxo de execução pertencente a um processo convencional. Como conseqüência, qualquer paralelismo real disponível no computador não pode ser aproveitado a nível de *thread*. Outra conseqüência é que uma operação de E/S em uma *thread* vai bloquear todas as *threads* do processo.

O segundo método é dito 1:1 (*one-to-one*). Ele resolve os dois problemas mencionados acima: aproveitamento do paralelismo real dentro de um único programa e processamento junto com E/S. Para isso ser possível, o sistema operacional deve ser projetado de forma a considerar a existência de *threads* compartilhando o espaço de endereçamento do processo hospedeiro. As operações relacionadas com as *threads* passam necessariamente por chamadas ao sistema operacional, o que torna *threads* tipo 1:1 "menos leves" (para o sistema operacional) que *threads* do tipo *N*:1. Este modelo é utilizado, por exemplo, no sistema Linux, que também oferece uma biblioteca para o modelo *N*:1.

Existe um modelo misto que combina as duas abordagens, chamado *M*:*N* (*many-to-many*). Neste modelo, o interior de cada processo comporta *N threads* sistema, dentro das quais devem ser “abrigadas” as *M threads* usuário que constituem a aplicação (isto é, o conjunto das *M threads* do programa deve ser particionado e distribuído entre as *N threads* do sistema). Esse método possui escalonamento nos dois níveis. O sistema Solaris oferece este modelo de execução de *threads*, utilizando *ligth weight processes*, as LWPs [POW 91].

Na linguagem V4, pode-se dizer que é utilizado o modelo *N*:1, pois, do ponto de vista do SO, existe um único processo. De fato, por baixo de tudo, o que se tem é um ambiente de

execução Prolog, já que, na versão atual, o sistema foi todo programado em SWI-Prolog [SWI 03]. Os processos, *threads* e *tasks* são representados por estruturas de dados Prolog e o escalonamento da execução desses componentes é implementado pelo programa Prolog. Sobre as *threads* V4, talvez seja interessante referir o seguinte. Conforme foi visto, em V4, a programação *multithread* pode ocorrer dentro de *tasks* ou dentro de processos. Ambas as estruturas permitem a definição de múltiplos fluxos de execução. Contudo, nas versões iniciais de V4, as *threads* só podiam ser criadas dentro de *tasks*[70]. Esta escolha se justificava em aspectos didáticos, pois isto facilitava a apresentação dos modelos de programação (por exemplo, quando se discute o conceito de monitor, as entidades importantes são os monitores e os processos – e não as *threads* ou *tasks*).

13.10 Pthreads

O padrão (ou norma) POSIX 1003.4 da IEEE se preocupa com o desenvolvimento de ferramentas para a programação com *threads*. O padrão especifica uma interface a ser oferecida (pelo fabricante que desejar seguir esse padrão) para o desenvolvimento de aplicações com *threads*. Como esse padrão é utilizado por vários sistemas, as aplicações desenvolvidas segundo o mesmo terão boa portabilidade.

As ferramentas são definidas para uso através das linguagens C e C++. Como tal, a sintaxe não é muito amigável e o uso de ponteiros é abundante. Aqui não haverá preocupação com aspectos sintáticos.

A biblioteca Pthread (onde o P significa POSIX) disponibiliza um conjunto de tipos pré-definidos que devem ser usados para definir as estruturas utilizadas pelo programa. Entre os tipos pré-definidos tem-se *pthread_t*, *pthread_attr_t*, *pthread_mutex_t*, e *pthread_cond_t*. O tipo *pthread_attr_t* corresponde a uma estrutura opaca, isto é, uma estrutura cujos campos não podem ser manipulados diretamente pelo usuário, mas sim através de um conjunto de primitivas do sistema. O tipo *pthread_attr_t* é usado para criar um bloco descritor (de atributos) que irá caracterizar uma *thread* do programa (cada *thread* terá o seu conjunto de atributos).

A seguir são resumidos alguns dos principais serviços definidos pelo padrão POSIX 1003.4.

Manipulação de *threads*

Os mecanismos para criação e sincronização de *threads* são descritos a seguir.

[70] Nas *tasks* as *threads* são realmente necessárias, pois sem elas não se pode implementar não-determinismo (lembre que não existe o comando *select*). Nos processos V4 as *threads* não são necessárias.

Definição do corpo de uma *thread*

O conjunto de instruções a ser executado por uma *thread* é definido por uma função C ou por um método C++. A função ou o método que define o comportamento da *thread* (*função-corpo*) pode receber parâmetros[71] e sempre retorna um valor. O valor de retorno é um ponteiro para o dado que constitui o valor de retorno propriamente dito. Se não há dado de retorno, então o ponteiro é NULL.

Criação de uma *thread*

A criação se dá através da primitiva *pthread_create*, que possui 4 argumentos, os quais são descritos nos parágrafos a seguir. Esta primitiva retorna 0 (zero) se a nova *thread* é criada com sucesso. No final da execução da *thread*, se a sua função-corpo retornar algum dado, esse dado será recuperado pela primitiva *join*, que será vista adiante.

O primeiro argumento de *pthread_create* é o endereço de uma variável que irá receber a identificação única da *thread*[72] criada.

O segundo argumento, é um ponteiro para um descritor de atributos da *thread*. Se este ponteiro é NULL, são utilizados atributos *default*. Um dos atributos diz se a *thread* é do tipo "joinable" (isto é, se essa *thread* se sincroniza com outra através de operação *join*). Se for o caso, a área de dados dessa *thread* só será liberada após a execução da primitiva *join*, ocasião em que o valor retornado pela *thread* será recuperado. Outro atributo indica a política de escalonamento a ser usada, a qual pode ser FIFO, RR (*Round-Robin*) ou OTHER (que corresponde à política *default* da biblioteca).

O terceiro argumento indica a função que constitui o corpo da *thread*. A mesma função-corpo pode ser usada para criar várias *threads*.

Finalmente, o quarto argumento é um ponteiro para a área de memória que contém os parâmetros de entrada para a função-corpo da *thread*.

Término de uma *thread*

O término se dá ao final da execução da função-corpo da *thread*, tipicamente através de um comando *return*. Alternativamente ao *return*, o programador pode usar a primitiva *pthread_exit*, que tem como argumento um ponteiro para o valor de retorno da *thread*.

Sincronização da execução de *threads*

A primitiva *pthread_join* possui dois argumentos. Ela bloqueia a *thread* que a executa, até que termine a *thread* especificada em seu primeiro argumento (este argumento é uma identificação

[71] Na verdade, é um único parâmetro que aponta para uma lista de argumentos.
[72] Caso se faça necessário, uma *thread* pode ter acesso à sua própria identificação através da primitiva *pthread_self*.

de *thread*). O segundo argumento permite receber o dado retornado pela (função-corpo da) *thread*, quando a mesma termina.

Compartilhamento de dados

A seguir são apresentados os mecanismos definidos pelo padrão POSIX para coordenar o compartilhamento de dados entre as *threads* de um processo.

Variáveis tipo *mutex*

Uma variável *mutex* pode ser vista como um semáforo binário cujas operações P e V passam a ser denominadas *lock* e *unlock*, respectivamente. As primitivas para manipulação de variáveis *mutex* são apresentadas a seguir:

A primitiva *pthread_mutex_init* permite definir (isto é, criar e inicializar) uma variável tipo *mutex*. São necessários dois argumentos. O primeiro define o nome da variável e o segundo o seu valor inicial, que pode ser *aberto* ou *fechado*. Se o segundo argumento for NULL, o valor inicial será *aberto* (valor *default*).

As primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* possuem um único argumento, que é o nome da variável *mutex* envolvida na operação. A semântica destas operações é a mesma das operações P e V sobre um semáforo binário. Vale observar que estas operações (*lock*/*unlock*) também são equivalentes às operações *mutexbegin*/*mutexend* parametrizadas.

Variáveis de condição

Estas variáveis foram fortemente inspiradas nas variáveis do tipo *condition* de monitores. Aliás, operações sobre estas variáveis só devem ser executadas dentro de seções críticas protegidas por operações *lock* e *unlock*, sob pena de resultados imprevisíveis serem obtidos.

A primitiva *pthread_cond_init* define e inicializa uma variável do tipo *condition*. São necessários dois argumentos. O primeiro é o nome da variável a ser criada e o segundo é um ponteiro para seus atributos (se é NULL, valores *default* são utilizados).

A primitiva *pthread_cond_wait* possui dois argumentos: o nome da variável *condition* envolvida na operação e o nome de uma variável *mutex* associada, que deve ter sido chaveada (*locked*) anteriormente pela própria *trhead* que está executando o *wait* (isto é, o *mutex* especificado no segundo argumento deve estar de posse – sob o controle – da *thread* que está executando o *wait*). Como nos monitores, esta primitiva bloqueia incondicionalmente a *thread* que a executa (a qual fica à espera de uma sinalização) e libera o *mutex* especificado no segundo argumento. Portanto, a operação *wait* tem uma operação *unlock* implícita, que atua no segundo

argumento. Se a variável *mutex* do segundo argumento não está de posse da *thread* que executa o *wait*, resultados imprevisíveis são obtidos.

Mais tarde, quando a *thread* retornar da chamada *pthread_cond_wait* (isto é, quando ela for desbloqueada), a variável *mutex* do segundo argumento vai estar chaveada (*locked*) e sob o controle da mesma.

As primitivas *pthread_cond_signal* e *pthread_cond_broadcast* servem para sinalizar uma variável *condition*. Ambas têm um único argumento que é a variável condição sinalizada. A diferença entre elas é que o *signal* desbloqueia apenas a primeira *thread* da fila de espera da variável condição, enquanto o *broadcast* desbloqueia todas as *threads* dessa fila.

No caso da primitiva *pthread_cond_broadcast*, cada uma das *threads* desbloqueadas, uma de cada vez, vai ficar de posse do *mutex* especificado na operação *wait* da qual ela está retornando. É como se houvesse uma competição entre as *threads* do conjunto desbloqueado, pela posse desse *mutex*. Dado que a ordem de alocação do *mutex* é desconhecida, isto faz com que cada *thread* tenha que testar a condição sinalizada, a qual pode ter deixado de ser verdadeira pela ação de uma das *threads* do conjunto desbloqueado.

Se a *thread* que executa a operação *signal* ou *broadcast* não está de posse da variável *mutex* que é liberada para a(s) *trhead*(s) desbloqueada(s), resultados imprevisíveis são obtidos. Também são obtidos resultados imprevisíveis se as *threads* bloqueadas em uma variável *condition* especificam variáveis *mutex* diferentes como segundo argumento da operação *wait*.

Tal como a operação *signal* dos monitores, as operações *pthread_cond_signal* e *pthread_cond_broadcast* não têm nenhum efeito se a fila da variável condição está vazia (as operações não são memorizadas).

A semântica confusa e a insegurança das primitivas (*pthread_cond_*) *wait*, *signal* e *broadcast* contrastam com a simplicidade e a segurança das operações *wait* e *signal* dos monitores. Tal como ocorreu na linguagem Java, parece que os projetistas do padrão POSIX não tomaram conhecimento da pesquisa desenvolvida nos últimos 25 anos sobre linguagens de programação paralela [BRI 99].

Na verdade, não se pode dizer que o conceito de monitor tenha sido ignorado, pois ele foi fonte de inspiração para as variáveis *mutex* e *condition*. Mas, sem dúvida, a preocupação com a segurança da programação foi ignorada. Talvez não tenha sido percebido, por exemplo, que a operação *pthread_cond_broadcast* (e sua semântica confusa) é, de fato, desnecessária. O efeito de liberar um conjunto de *threads* bloqueadas em uma variável *condition* pode ser obtido com o uso da operação *signal*, conforme foi visto no exemplo da seção 9.8.5 (sincronização tipo *readers* & *writers* com monitores).

Variáveis tipo semáforo

O padrão 1003.4 não define interface de serviços para utilização de semáforos, mas o padrão Unix POSIX o faz. Portanto, nem toda implementação de Pthreads irá incluir este tipo de variável.

Uma observação final

Se o programa usa outras bibliotecas além de Pthreads, deve-se cuidar que essas bibliotecas sejam do tipo *thread-safe* ou *thread-aware*. Uma biblioteca é *thread-safe* se ela foi desenvolvida de forma que seus serviços sejam executados corretamente, mesmo quando requeridos por mais de uma *thread*. Este tipo de controle pode ser implementado com um mecanismo de exclusão mútua (cada um dos serviços passa a ser uma região crítica). Uma biblioteca é *thread-aware* se seus serviços podem ser executados simultaneamente (e corretamente) quando requeridos por mais de uma *thread*.

13.11 PVM

A biblioteca PVM (*Parallel Virtual Machine*) permite que um ambiente de rede heterogêneo, que pode ser composto por uma variedade de máquinas diferentes (desde PCs até supercomputadores), funcione como se fosse um sistema distribuído formado por processadores homogêneos (em relação aos serviços prestados), com memória distribuída [GEI 94]. Portanto, o PVM transforma uma rede heterogênea numa máquina virtual adequada para o desenvolvimento de aplicações paralelas (ou distribuídas).

O sistema PVM pode ser obtido a partir de sua *home page* [PVM 01]. A instalação do software é simples, sendo muito fácil habilitar um computador da rede a ser usado como *host* PVM. Após a instalação, diferentes usuários podem configurar diferentes máquinas virtuais na mesma rede física, as quais irão funcionar de forma independente. E uma mesma máquina virtual pode suportar diversas aplicações PVM, simultaneamente.

O modelo computacional do PVM se baseia na noção de que uma aplicação consiste de várias *tasks*, cada uma responsável por uma parte da carga de trabalho total dessa aplicação. O PVM permite criar *tasks* (processos) a partir de arquivos executáveis, que podem ser gerados a partir de programas C ou Fortran. Estes programas utilizam as primitivas de comunicação *send* e *receive* para implementar a aplicação distribuída que se deseja. As operações *send* e *receive* são assíncronas, mas é possível usar o *receive* de forma bloqueante[73] (síncrona).

Além das primitivas de comunicação mencionadas, a biblioteca PVM oferece funções para:

[73] O envio de uma mensagem é de uma certa forma bloqueante, pois o emissor espera até que o *buffer* de envio esteja livre.

- Inicialização e término de *tasks* PVM;
- Adição e remoção de *hosts* na máquina virtual;
- Sincronização e envio de sinais entre processos PVM;
- Obtenção de informações sobre a configuração da máquina virtual e processos;
- Empacotamento e desempacotamento de dados;
- Difusão de mensagens (*multicast* e *broadcast*);
- Criação dinâmica de grupos de processos.

O sistema PVM é composto de duas partes. Uma é a biblioteca de funções que fornece os serviços listados acima. A outra é o *daemom*, chamado "pvmd", processo que reside em todos os *hosts* que constituem a máquina virtual. Pode-se dizer que o conjunto dos *daemons*, executando em diferentes máquinas, forma a máquina virtual definida pelo usuário. Os *daemons* gerenciam um console virtual, identificado por um *prompt* próprio, que pode ser utilizado a partir de qualquer *host* da máquina virtual.

Antes de compilar e executar um programa PVM é necessário definir a máquina virtual a ser utilizada. Pode-se fazer isso a partir de qualquer computador que tenha o PVM instalado, escrevendo-se:

```
% pvm
```

Se o PVM estiver instalado nessa máquina, o sistema responderá com o *prompt*:

```
pvm>
```

Pode-se então usar o comando *add hostname* para acrescentar o computador *hostname* na máquina virtual ou usar *delete hostname* para remover o computador identificado por *hostname*. O comando *conf* permite ver a arquitetura atual da máquina virtual.

A forma de compilar e executar programas PVM pode ser encontrada na bibliografia [GEI 94, COS 02], bem como na sua *home page* [PVM 01], e não será aqui abordada. O término de uma execução PVM deve sempre ser feito através do comando *halt*, que termina as *tasks* PVM, destrói a máquina virtual e devolve o *prompt* do console para o sistema operacional.

Primitivas básicas do PVM

Na linguagem C, as primitivas PVM são oferecidas como funções e na linguagem Fortran, como subrotinas (o valor de retorno da função C sempre volta no último argumento da subrotina Fortran correspondente). Na verdade, as primitivas foram escritas em C e as subrotinas Fortran são *stubs* que fazem as conversões necessárias nos argumentos e invocam as correspondentes funções C.

A seguir serão apresentadas as principais primitivas da biblioteca PVM, sem se preocupar com os aspectos sintáticos.

pvm_mytid()

Retorna a identificação da *task* que chama essa função (*tid* significa *task identifier*).

pvm_parent()

Retorna a identificação da mãe (*task* criadora) da *task* que chama essa função.

pvm_config(nhost, narch, hostinfo)

Retorna informações sobre a máquina virtual, inclusive o número de máquinas (*nhost*) e o número de arquiteturas diferentes utilizadas (*narch*).

pvm_spawn(task, argv, flag, where, ntask, tids)

Inicia a execução de várias cópias de um arquivo executável (especificado por *task*) na máquina virtual.

pvm_kill(tid)

Termina a *task* cujo *tid* é fornecido como argumento.

pvm_initsend(cod)

Inicializa um *buffer* para o envio de uma mensagem e retorna a identificação do mesmo. O argumento especifica o esquema de codificação a ser usado.

pvm_send(tid, msgtag)

Envia a mensagem identificada por *msgtag* (qualificador da mensagem) para a *task* identificada por *tid*.

pvm_mcast(tids, ntask, msgtag)

Envia a mensagem identificada por *msgtag* para todas as *tasks* identificadas no vetor *tids*, que tem tamanho *ntask*.

pvm_recv(tid, msgtag)

Bloqueia a *task* que chama esta função até que chegue a mensagem identificada por *msgtag* da *task* identificada por *tid*. O valor –1 para *msgtag* e/ou *tid* funciona como *wildcard* (permite receber qualquer mensagem de qualquer *task*). A função devolve a identificação do *buffer* no qual foi colocada a mensagem.

pvm_nrecv(tid, msgtag)

É a versão não bloqueante da primitiva *receive*. Ela verifica se chegou a mensagem identificada por *msgtag* da *task* identificada por *tid*. Se não chegou, a função devolve o valor zero; se chegou, devolve a identificação do *buffer* no qual a mensagem foi colocada.

pvm_trecv(tid, msgtag, tmout)

Bloqueia a *task* até que chegue a mensagem ou se esgote o tempo especificado por *tmout*.

pvm_bufinfo(bufid, nbytes, msgtag, tid)

Retorna o *msgtag*, o *tid* do emissor e o tamanho (*nbytes*) da mensagem identificada pelo *bufid*. É usada para recuperar informações das mensagens recebidas com *wildcards*.

pvm_exit()

Avisa o *daemon* que esta *task* está deixando o PVM (a *task* pode continuar sua execução, desde que não chame funções do PVM).

13.12 MPI

A biblioteca MPI (*Message Passing Interface*) também permite a programação paralela baseada na troca de mensagens [MPI 94]. Uma execução MPI compreende dois ou mais processos que se comunicam chamando rotinas da biblioteca MPI para enviar e receber mensagens. As rotinas podem ser chamadas a partir de programas C ou Fortran.

Um programa MPI é formado por um conjunto fixo de processos (normalmente, um por processador) criados no momento da inicialização do sistema. Cada um desses processos pode executar um programa diferente, o que caracteriza o modelo MPMD (*Multiple-Program Multiple-Data*). No entanto, a forma mais comum de programar utiliza o modelo SPMD (*Single-Program Multiple-Data*), onde um mesmo programa é executado por cada um dos processadores participantes, mas em cada processador é selecionado um trecho do programa para ser executado. A seguir são apresentadas algumas características do sistema MPI.

A compilação de programas escritos na linguagem C (C++) é feita através do comando *mpicc*, que aceita todos os argumentos de compilação do compilador C. A execução é feita através do comando *mpirun*, cujos argumentos definem os nomes e os tipos das máquinas que constituem o sistema.

O padrão MPI define funções para:

- Comunicação ponto a ponto (transmissor e receptor definidos para cada mensagem);
- Operações coletivas;
- Grupos de processos;
- Contextos de comunicação;
- Ligação para programas ANSI C e Fortran 77;
- Topologia de processos.

Uma mensagem é formada por duas partes: (a) um envelope, que contém a identificação dos processos envolvidos (transmissor e receptor), o rótulo da mensagem e o *communicator* (ver abaixo); (b) um dado, que é especificado por um endereço (localização do dado), número de elementos do dado na mensagem e tipo do dado (*signed int*, *float*, *char*, etc.).

A identificação de um processo é denominada *rank*. Sendo *N* o número de processos do sistema, os *ranks* vão de 0 a *N*-1. O *rank* é usado para identificar o destinatário, na operação *send*, e o remetente, na operação *receive*.

Um grupo (*group*) é um conjunto ordenado de processos. Todo grupo possui um *communicator*[74] associado, que identifica esse grupo. O *communicator default* é o MPI_COMM_WORLD, que é pré-definido no sistema.

Primitivas básicas do MPI

O MPI oferece um conjunto grande de funções (129, na versão 1.1), contudo, com um número reduzido delas (apenas 6) é possível resolver uma grande variedade de problemas. Estas funções básicas são apresentadas a seguir.

MPI_Init(&argc, &argv)

Inicializa uma execução MPI. É responsável por copiar o código do programa em todos os processadores que participam da execução. Os argumentos *argc* e *argv* são variáveis utilizadas em C para recebimento de parâmetros.

MPI_Finalize()

Termina uma execução MPI.

MPI_Comm_Size(communicator, &size)

Determina o número de processos em uma execução. O primeiro argumento indica o grupo de comunicação e o segundo devolve o número de processos no grupo.

MPI_Comm_Rank(communicator, &pid)

Devolve em *pid* a identificação do processo que executa a primitiva.

MPI_Send(&buf, count, datatype, dest, tag, comm)

Permite a um processo enviar uma mensagem para outro.

MPI_Recv(&buf, count, datatype, dest, tag, comm)

Permite que um processo receba uma mensagem. É uma operação bloqueante.

[74] Um *communicator* é uma estrutura de dados que define um contexto de comunicação. Cada processo é identificado pelo seu *rank* dentro desse contexto.

13.13 EXERCÍCIOS

Exercício 13.1 Quais as semelhanças e as diferenças entre as secretárias de Dijkstra e as *tasks* da linguagem ADA?

Exercício 13.2 Por que se pode dizer que as chamadas remotas de procedimentos preencheram a lacuna que havia entre os esquemas de troca de mensagens e os monitores?

Exercício 13.3 Por que as variáveis tipo *condition* da biblioteca Pthreads fazem lembrar os monitores?