UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

DISSERTAÇÃO DE MESTRADO

UM AMBIENTE DE PROGRAMAÇÃO CONCORRENTE NO SISTEMA OPERACIONAL UNIX

JOÃO CARLOS RODRIGUES PEREIRA

CAMPINA GRANDE-PB, BRASIL
AGOSTO DE 1986

P436q Pereira, Joao Carlos Rodrigues
Um ambiente de programacao concorrente no sistema operacional UNIX / Joao Carlos Rodrigues Pereira. - Campina Grande, 1986.
97 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Sistemas Operacionais - UNIX 2. Programacao Concorrente 3. Linguagem de Programacao C 4. Dissertacao - Informatica I. Sauve, Jacques Philippe II. Universidade Federal da Paraiba - Campina Grande (PB)

CDU 004.451(043)

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO E CULTURA
UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
Curso de Pós-Graduação em Sistemas e Computação
AV. APRÍGIO VELOSO, S/N - BODOCONGÓ
58.100 - Campina Grande - Pb.

PARECER FINAL DO JULGAMENTO DA DISSERTAÇÃO DO MESTRANDO

JOÃO CARLOS RODRIGUES PEREIRA

Título: "Um Ambiente de Programação Concorrente no Sistema Operacional UNIX".

COMISSÃO EXAMINADORA	CONCEITO
JACQUES PHILIPPE SAUVÉ - Ph.D - Presidente -	Aprovado
JOSÉ ANTÃO BELTRÃO MOURA - Ph.D	Aprovado
GIUSEPPE MONGIOVI - M.Sc	Aprovado

Campina Grande, 22 de agosto de 1986

JOÃO CARLOS RODRIGUES PEREIRA

UM AMBIENTE DE PROGRAMAÇÃO CONCORRENTE NO SISTEMA OPERACIONAL UNIX.

Dissertação apresentada ao Curso de MESTRADO EM SISTEMAS DA COMPUTAÇÃO da Universidade Federal da Paraíba, como parte dos requisitos para obtenção do titulo de Mestre em Ciências da Computação.

JACQUES PHILIPPE SAUVÉ
Orientador

CAMPINA GRANDE
AGOSTO - 1986

UM AMBIENTE DE PROGRAMAÇÃO CONCORRENTE
NO SISTEMA OPERACIONAL UNIX.

RESUMO

Neste trabalho são abordados os aspectos denotacionais da programação concorrente. Os principais conceitos e notações existentes para a especificação da execução concorrente e da sincronização e/ou comunicação envolvidas são apresentados, e é discutido o uso das primitivas do sistema operacional UNIX neste tipo de aplicações. São propostas e implementadas uma linguagem de programação que estende a linguagem C com novas construções e uma biblioteca com funções que mapeiam primitivas mais poderosas para o conjunto atualmente disponível no UNIX.

ABSTRACT

This thesis deals with the notational aspects of concurrent programming. The principal concepts and notations used to specify concurrent execution, process synchronization and communication are presented. The use of UNIX primitives in such applications is also discussed. A proposal for extending the C programming language with new constructions is presented, along with the implementation of a set of functions which map these new constructions to the available UNIX set.

SUMÁRIO

LISTA DAS ILUSTRAÇÕES

Figura 22: Saídas produzidas no exemplo do usos de algumas macros do pré-compilador.

1. INTRODUÇÃO

O estudo da programação concorrente tem sido tradicionalmente um tópico exclusivo dos domínios da teoria dos sistemas operacionais [2]. Ultimamente, porém, temos assistido à disseminação do seu uso pelos mais diversos tipos de aplicações. Vários fatores contribuíram para isto. Dentre eles destacamos o acelerado crescimento dos conhecimentos sobre a microeletrônica, que fez surgir microprocessadores bastante acessíveis quanto ao preço e ao volume. Estes avanços tecnológicos viabilizaram o projeto de sistemas distribuídos e de arquiteturas de multiprocessamento [22] [24]. Esta nova atitude trouxe mudanças substanciais à programação concorrente: exigiu desenvolvimento de sistemas operacionais que encorajassem o uso de processos concorrentes; e a criação de linguagens de alto nível com notações para expressar a concorrência de modo mais simples e tornar mais explícita a sincronização e a comunicação entre processos [23].

Não obstante os significativos avanços obtidos, a concorrência/cooperação entre processos está longe de ser bem compreendida. Uma das razões para que elas apareçam como uma coleção de técnicas e conceitos díspares são as ligações inexplicáveis entre vários fatores, como por exemplo, a sincronização e a troca de mensagens [6].

Neste trabalho descrevemos os principais resultados que obtivemos dos nossos estudos sobre os aspectos denotacionais da programação concorrente. O nosso objetivo foi o desenvolvimento de mecanismos que permitissem a elaboração sistemática de programas concorrentes estruturados, confiáveis e de fácil manutenção.

Inicialmente, no capítulo 2, fazemos uma retrospectiva dos principais conceitos sobre programação concorrente e descrevemos as mais importantes notações para escrever programas concorrentes. Muito embora esta descrição requeresse uma análise mais detalhada de várias linguagens de programação concorrente, restringimos a nossa atenção às notações que mais influenciaram no estabelecimento de novas metodologias. Fazemos estas descrições

à luz das três questões denotacionais básicas da programação concorrente: como expressar a execução concorrente; como os processos se comunicam; e como os processos se sincronizam.

Estas questões também são tratadas no capítulo 3, porém sob o ponto de vista do sistema operacional UNIX(+). Para fazer uma melhor análise do uso das primitivas deste sistema na programação concorrente, incluímos vários exemplos de programas escritos na linguagem C.

Fundamentamo-nos nestas análises, para propor, no capítulo 4, uma nova linguagem de programação. Esta linguagem foi projetada como uma extensão da linguagem C através da introdução de novos conceitos estruturais e da biblioteca de programação concorrente LIBBIS, que contém funções que fazem o mapeamento de novas primitivas voltadas para a programação concorrente sobre as já existentes no UNIX.

No capítulo 5 apresentamos considerações sobre os detalhes de implementação do pré-compilador desta nova linguagem e das funções de programação concorrente. Discutimos as principais estruturas de dados e algoritmos envolvidos.

No capítulo 6 mostramos as soluções de alguns problemas clássicos codificados com a linguagem implementada.

Finalmente, no capítulo 7, fazemos um sumário dos principais tópicos abordados nesta tese e tecemos considerações sobre os resultados obtidos. Relacionamos as dificuldades encontradas na implementação do pré-compilador e das funções, e identificamos alguns pontos que poderiam ser melhorados. Concluímos apresentando sugestões para estudos posteriores.

+ UNIX é uma marca registrada da Bell Telephone Laboratories.

2. CONCEITOS E NOTAÇÕES PARA A PROGRAMAÇÃO CONCORRENTE

Os programas seqüenciais especificam uma lista de comandos cuja execução, realizada segundo uma ordenação estrita, é chamada de processo [6]. O conceito de processo é essencialmente dinâmico, em contraposição ao de programas que é completamente estático. Nem sempre um problema computacional exige uma ordenação total de todos os passos da computação, na maioria das vezes é necessário apenas uma ordenação parcial no tempo, ou seja, algumas das operações devem ser executadas antes de outras, enquanto que as demais podem ser executadas paralelamente [24]. Por exemplo, para calcular a expressão aritmética

$$(4 - 2) * (3 + 5)$$

podemos computar simultaneamente os resultados de (4 - 2) e de (3 + 5), mas só podemos efetuar a multiplicação após a obtenção deste resultados parciais.

Em um conjunto de processos que estão sendo executados paralelamente, podemos ou não ter interferências de uns sobre outros, quer para competir por um recurso, quer para cooperar entre si. Quando estas interferências acontecem temos a execução concorrente destes processos. Um programa concorrente consiste da especificação de vários programas seqüenciais que são executados concorrentemente [6]. Os programas concorrentes são executados ou nos ambientes de multiprogramação, através do uso de um único processador compartilhado entre vários processos, ou nos ambientes de multiprocessamento, dedicando-se um processador para cada processo quando, então, obtêm-se o verdadeiro paralelamente.

A programação concorrente é o conjunto de notações e técnicas usadas para expressar o paralelismo potencial. Abstraindo-se do fato das operações coincidirem fisicamente no tempo (nos casos de multiprocessamento), ou de entremearem-se (na multiprogramação), a

programação concorrente torna-se um tópico essencialmente dissociado da implementação do paralelismo [21]. Originalmente foi motivada pelos problemas de construção de sistemas operacionais que permitissem um maior aproveitamento de recursos computacionais mais rápidos e poderosos. Atualmente porém as suas técnicas são largamente usadas em vários outros tipos de aplicações, tais como: sistemas de gerência de base de dados; sistemas de controle em tempo real; e programação científica com execução em paralelo em larga escala.

A programação concorrente proporciona uma estrutura conceitual para resolver os problemas de comunicação e sincronização resultantes da concorrência, encorajando o desenvolvimento sistemático de programas bem estruturados, confiáveis, eficientes e de fácil manutenção. Mesmo nos casos em que os processos não são executados simultaneamente é, muitas vezes, mais fácil estruturar um sistema como uma coleção de processos seqüenciais cooperantes do que construí-lo como um único programa seqüencial. Consideremos o seguinte problema:

> Ler cartões de 80 colunas e imprimi-los em linhas de 125 caracteres. Toda série de n=1 a 9 espaços em branco deve ser substituída por um único branco seguido pelo numeral n.

Muito embora a solução não exija paralelismo, não é fácil resolvê-lo usando um programa seqüencial. Existem muitos casos especiais: uma série de brancos sobrepondo-se ao final do cartão, o par branco-n sobrepondo-se ao fim da linha, e assim por diante. Uma maneira de melhorar a clareza seria escrever programas separados: um para ler cartões gravando-os como uma seqüência de caracteres num arquivo temporário; um segundo para ler esta seqüência de caracteres e modificar as séries de brancos gravando a nova seqüência num segundo arquivo temporário; e um terceiro programa para ler este último arquivo e imprimir linhas. Esta solução pode tornar-se inaceitável devido ao grande uso de arquivos temporários. Contudo, se os três programas puderem ser executados concorrentemente, e os arquivos temporários puderem ser substituídos por linhas de comunicação entre eles, teríamos uma solução clara, precisa e eficiente.

Quando processos cooperam, podem se comunicar e devem se sincronizar. A comunicação permite, através da troca de dados, que a execução de um processo influencie na execução de outro. A sincronização possibilita a limitação ou ordenamento na execução de eventos, ou o controle de interferência através da exclusão mútua.

No projeto de uma notação para expressar uma computação concorrente, defrontamo-nos com os seguintes problemas básicos: como especificar a execução concorrente; que modo de comunicação entre processos usar; e quais mecanismos de sincronização empregar [6]. Nas seções seguintes examinaremos cada um destes aspectos isoladamente.

2.1. COMO ESPECIFICAR A EXECUÇÃO CONCORRENTE

Historicamente foram propostas várias notações para especificar a programação concorrente. A maioria delas é demonstravelmente adequada para seus propósitos, mas não há nenhum critério largamente reconhecido par eleger uma delas como a melhor. Uma das mais antigas, as co-rotinas, consistem de tipos especiais de sub-rotinas com as seguintes características: permitem a execução parcial das co-rotinas; eliminam a estruturação hierárquica que as sub-rotinas apresentam; e tornam explícita a comutação de co-rotinas permitindo a transferência simétrica de controle entre si.

As co-rotinas comportam-se como processos sincronizados através do comando RESUME, que é semelhante à chamada de procedimentos, exceto quanto ao ponto de entrada. Uma chamada de procedimentos sempre transferirá o controle para o início destes, enquanto que o ponto de entrada de uma co-rotina é variável: inicialmente é a sua primeira instrução, porém, a cada vez que ela executar um comando RESUME, o novo ponto de entrada fica sendo o endereço seguinte a este comando. Muito embora quando usadas com cuidado as co-rotinas possam organizar aceitavelmente programas concorrentes em ambientes de multiprogramação, não são adequadas nos casos de verdadeiro paralelismo, pois só permitem a execução de uma única co-rotina de cada vez.

A notação COBEGIN/COEND é uma maneira estruturada de representar a execução concorrente de um conjunto de instruções. A notação

S_0;
COBEGIN S_1; S_2; ... ; S_n COEND;
S_{n+1};

indica que as instruções S_1, S_2, ... ; S_n podem ser executadas concorrentemente, sendo precedidas por S_0, e seguidas por S_{n+1}. A execução de um COBEGIN só termina quando encerram as execuções de todas as instruções S_1, S_2, ... , S_n. Embora não seja tão poderoso como o par FORK/JOIN (que será mostrado a seguir), a notação COBEGIN/COEND é suficiente para especificar a maior parte das computações concorrentes. Além disto, a sua sintaxe torna explícita que rotinas são executadas concorrentemente, e proporciona, um único ponto de entrada e um único de saída para a estrutura de controle, facilitando a compreensão da estrutura dos programas onde são usados. O grande problema desta notação é quanto ao seu aspecto estático: o programa fonte deverá especificar claramente quais e como os processos podem concorrer.

Uma outra maneira de denotar concorrência é através do par FORK/JOIN. A instrução FORK, semelhante a CALL ou RESUME, especifica que a rotina designada deve iniciar sua execução procedendo concorrentemente a partir daí. As rotinas estão numa relação de hierarquia, caracterizando-se a que executa a instrução FORK como pai, e a rotina designada no FORK, como filho. Essencialmente o FORK desvia para o filho e continua simultaneamente a seqüência normal de execução do pai. Se for necessário que o pai espere que o filho termine, usamos a instrução JOIN. O pai será bloqueado até que o filho termine a sua execução. Na figura 1 apresentamos esquematicamente o paralelismo do fluxo de execução entre os processos proc_1 e proc_2 usando o par FORK/JOIN. Neste exemplo o processo proc_1 inicia a sua execução e, somente ao encontrar a instrução FORK proc_2 é que o processo proc_2 tem a sua execução iniciada. A partir deste ponto ambos executam concorrentemente até que o processo proc_2 termine ou o processo proc_1 execute a instrução JOIN (neste caso o processo proc_1 fica esperando que o processo proc_2 termine).

Do ponto de vista denotacional, podemos atribuir certas vantagens e desvantagens ao par FORK/JOIN. Uma das vantagens é que o FORK permite a criação dinâmica de processos, originando uma quantidade de processos que pode variar em tempo de execução. Quanto às desvantagens, se o par FORK/JOIN não for criteriosamente usado, a distinção entre as instruções que são executadas seqüencialmente e aquelas que são executadas

concorrentemente ficará prejudicada. Além disto, quando o FORK e o JOIN aparecem no

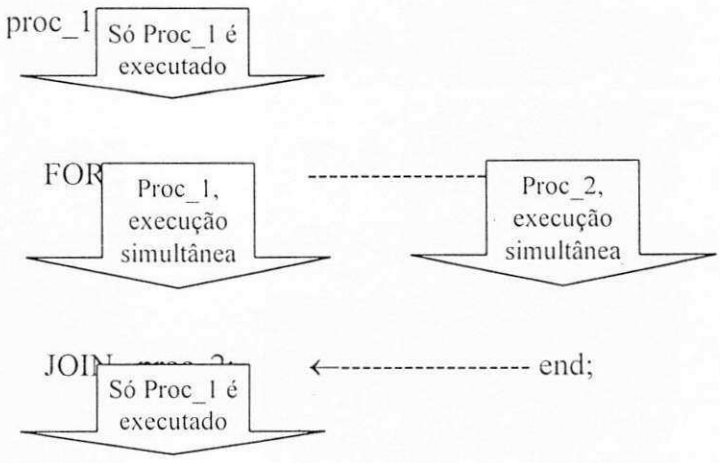

Figura 1: Funcionamento do par FORK/JOIN

interior de laços ou de comandos condicionais, a execução de um programa dependerá do comportamento dinâmico dos processos em tempo de execução. Com isto o esforço necessário para entendermos ou depurarmos um programa aumenta substancialmente com a complexidade das suas construções de controle. Devido a estes problemas de clareza estrutural dizemos que o par FORK/JOIN está para a programação concorrente, assim como o GOTO está para a programação seqüencial. Infelizmente a vantagem e as desvantagens citadas são tão intimamente ligadas que, a princípio, parece-nos impossível dissociá-las.

A estrutura de um programa concorrente fica muito mais clara se a declaração de uma rotina explicitar que a mesma será executada concorrentemente ou não. A declaração de processos introduz restrições sintáticas na especificação de rotinas concorrentes, impondo uma estrutura ao programa. As notações propostas para expressar a programação concorrente podem indicar um número fixo (estático) ou variável (dinâmico) de processos. O dinamismo é obtido combinando-se os mecanismos de criação com os de especificação de processos.

Em algumas linguagens com especificação estática (e. g. Distributed Process e SR), uma coleção de declarações de processos é equivalente a um simples COBEGIN, onde cada

um dos processos declarados é um componente deste COBEGIN, havendo um só exemplar de cada processo declarado.

Dentre as linguagens com especificação dinâmica, algumas (e. g. Pascal e Módula) proporcionam mecanismos de criação que só podem ser usados durante a inicialização do programa. Isto leva a um número fixo de processos durante a execução do programa, mas permite que múltiplos exemplares dos processos declarados sejam criados. Já em linguagens como Plits e Ada os processos podem ser criados a qualquer tempo durante a execução do programa, tornando possível computações com um número variável de processos.

2.2. COMO ESPECIFICAR A SINCRONIZAÇÃO ENTRE PROCESSOS CONCORRENTES

A sincronização entre processos concorrentes é usada para obter-se mútua exclusão, ou uma sincronização de condições. A mútua exclusão assegura que um determinado recurso (e. g. conjunto de declarações, arquivo, região da memória) é usado por um único processo de cada vez. A sincronização de condições impõe aos processos concorrentes uma ordenação (coordenação) na execução de operações, retardando a execução dos processos para satisfazer tais limitações. Podemos implementar a sincronização através do uso de variáveis compartilhadas entre processos, ou através da troca de mensagens entre eles. Nesta seção apresentamos noções de semáforos, regiões críticas condicionais e monitores, que são mecanismos de sincronização baseados no compartilhamento de memória. Na seção seguinte mostraremos alguns conceitos sobre a troca de mensagens entre processos.

Uma maneira de implementar sincronização é fazer com que os processos modifiquem e testem repetidamente variáveis compartilhadas. Esta técnica é conhecida como espera ocupada (+) e tem como desvantagens: os protocolos de sincronização usando somente espera ocupada são difíceis de projetar, compreender e provar a sua correção; e os testes no laço de espera desprendem um grande tempo improdutivo, consumindo ciclos do processador que

---

+ Também conhecida como "busy waitting".

poderiam ser empregados mais produtivamente executando outro processo, até que a condição desejada ocorra.

A primeira tentativa de solução para estes problemas foi proposta por Dijkstra através do uso de semáforos. Um semáforo s é uma variável inteira não negativa com dois estados, ocupado (s = 0) e livre (s > 0), ao qual se pode atribuir um valor inicial e sobre o qual só são permitidas duas operações indivisíveis: wait (s) e signal (s). A semântica das duas operações é a seguinte: wait (s) suspende o processo até que s > 0, quando então ela executa s := s-1, prosseguindo a execução do mesmo. O teste e o decremento são executados como uma operação indivisível. Signal (s) faz s := s+1 e ativa os processo que estão esperando que o semáforo s se desocupe. Apesar da sua simplicidade os semáforos podem ser vistos como ferramentas flexíveis e valiosas para resolver uma grande quantidade de problemas, embora a sua aplicação mais natural seja a troca de sinais de sincronismo. A sua principal desvantagem é permitir o uso não estruturado e indistinto das operações wait e signal, quer para a sincronização, quer par a exclusão mútua. O uso não estruturado possibilita a inversão da ordem em que elas devem aparecer ou a omissão de uma delas, acarretando problemas como o bloqueio perpétuo e/ou a destruição da exclusão mútua no acesso a um recurso. O uso indistinto faz com que, para identificarmos o seu propósito, tenhamos que ler o conjunto completo das operações. As possibilidades das operações virem dispersas ao longo de um processo acentua os problemas acima citados.

Hoare propôs uma solução usando regiões críticas condicionais que proporcionam uma notação mais estruturada para especificar sincronização. Isto foi obtido associando o seguinte ítens: os dados a serem compartilhados; as operações definidas sobre eles; e a declaração direta das condições necessárias à execução destas operações. As variáveis compartilhadas são explicitamente colocadas em grupos, chamados recursos. Um recurso r contendo as variáveis $v_1, v_2, \ldots, v_n$, é declarado como

RESOURCE r : v_1, v_2, ... , v_n

O acesso a uma variável compartilhada em r só pode ser feito em uma região crítica condicional que especifique o recurso r , do tipo:

REGION r WHEN b DO s

onde b é uma expressão boolena e s é uma lista de comandos. A execução do processo é retardada até que os componentes do recurso r satisfaçam à condição b. Só então os comandos da região crítica condicional são executados. A exclusão mútua é obtida garantindo-se a não sobreposição da execução de diferentes regiões críticas condicionais que especificam o mesmo recurso.

As regiões críticas condicionais são onerosas de implementar em um único processador, [6] e seus comandos realizam operações em recursos variáveis dispersos através do processo, de modo que para sabermos qual a finalidade de um recurso temos que realizar um estudo completo de todo o código. Além disto a avaliação repetida da expressão booleana b causa uma considerável ineficiência no tempo de processamento.

Os monitores amenizam estas deficiências agrupando as definições de recursos e das operações que podem ser realizadas sobre eles. São escritos como um conjunto de variáveis e procedimentos, e possuem um corpo que é executado tão logo o programa seja inicializado.

MONITOR : nome
variáveis permanentes
procedimentos comuns
corpo do monitor

As variáveis permanentes só são acessíveis aos procedimentos comuns e podem ser inicializadas pelas operações contidas no corpo do monitor. Os valores destas variáveis são mantidos inalterados entre ativações diferentes dos procedimentos do monitor. A comunicação com um monitor é feita dos parâmetros das chamadas dos seus procedimentos. A execução de procedimentos em um monitor é mutuamente exclusiva, o que assegura que as variáveis permanentes nunca são acessadas concorrentemente. Se, quando um procedimento é chamado, o monitor já estiver atendendo a um pedido anterior, o processo chamante é colocado numa fila de espera.

Os monitores concentram o código que, nas regiões críticas condicionais, estariam dispersos por todo o processo. Desta forma a estruturação é obtida a um custo menor do que o requerido na avaliação repetida das expressões booleanas nas regiões críticas condicionais.

Alguns problemas, entretanto, não foram resolvidos. A centralização do acesso aos recursos através dos procedimentos dos monitores ainda confia no seu uso correto por parte do usuário destes procedimentos. É suficiente que um deles esqueça de liberar algum recurso para que todo o subsistema controlado pelo monitor seja bloqueado. Além disto novos problemas foram introduzidos. A mútua exclusão no acesso a um monitor pode levar a uma excessiva seqüencialização de acesso aos recursos. Isto pode impedir, em alguns casos, uma melhor utilização do tempo de processamento.

2.3. COMO ESPECIFICAR A COMUNICAÇÃO ENTRE PROCESSOS CONCORRENTES

A comunicação entre processos faz com que, através da troca de dados, um processo influencie na execução de outro. Esta comunicação é baseada no uso de variáveis compartilhadas que podem ser referenciadas por mais de um processo, ou na passagem de mensagens entre eles. Quando usamos variáveis compartilhadas os problemas de comunicação prendem-se à sincronização e à garantia da mútua exclusão ao acesso destas variáveis. Se a memória não é compartilhada, a interação entre processos só pode ser feita usando-se a troca de mensagens.

Quando dois processos trocam mensagens podemos ter comunicação e sincronização: obtemos comunicação pois um processo (receptor) recebe valores de outro processo (emissor) ; e obtemos sincronização pois uma mensagem só pode ser recebida após ter sido criada e colocada à disposição do receptor, limitando-se assim a ordem em que estes dois eventos podem ocorrer.

É interessante notar como estão intimamente ligadas a troca de informações e a sincronização até mesmo na vida cotidiana. Pense no que é necessário para uma conversação telefônica. Alguém tem que discar um número e outra pessoa tem que interromper o que está fazendo para poder atender. Depois vem os protocolos simples de sincronização que envolve o alô e a coordenação da troca de mensagens.

O projeto de primitivas para troca de mensagens envolve a escolha sobre a forma e a semântica destes comandos gerais. Dois principais problemas devem ser abordados: como a fonte e o destino serão especificados, e como a comunicação será sincronizada.

Tomados conjuntamente, a fonte e o destino definem um canal de comunicação. Quando um canal só endereça um processo em cada extremidade, temos uma comunicação "um-a-um", e a maneira mais simples de especificar tais canais é através da especificação direta dos processos. O comando

SEND cartão TO executor

envia uma mensagem (cartão) que pode ser recebida somente pelo processo especificado (executor). Do mesmo modo

RECEIVE linha FROM executor

permite unicamente receber uma mensagem (linha) que tenha sido enviada pelo processo denominado executor.

Uma outra maneira de interação entre processos é a relação denominada cliente/servidor: um processo servidor "presta serviços" aos processos clientes. Um cliente solicita um serviço enviando uma mensagem ao servidor. Um servidor recebe repetidamente solicitações de serviços dos clientes, executa-os e, se necessário, retorna uma mensagem ao cliente. Para este tipo de comunicação usa-se um esquema de especificação mais geral denominados nomes globais ou "MAILBOXES". Um MAILBOX pode aparecer como destino em qualquer comando SEND, e como fonte em qualquer RECEIVE. Assim uma mensagem enviada a um dado MAILBOX pode ser recebida por qualquer processo que execute um RECEIVE que especifique este MAILBOX.

A implementação de MAILBOX, entretanto, pode ter um custo muito elevado sem uma rede de comunicação especializada. Para os casos de múltiplos clientes/único servidor a solução corrente é denominada portas. Portas são simples de implementar uma vez que todos

os RECEIVES que especificam uma determinada porta como fonte de mensagens pertencem ao mesmo processo.

Um outro aspecto a considerar é aquele relativo ao tempo de criação do canal: quando a fonte e o destino são fixados em tempo de compilação, denominamos especificação estática de canal e, quando é computado em tempo de execução, denominamos especificação dinâmica de canal. Embora largamente usada, a especificação estática apresenta dois problemas fundamentais: impede as comunicações através de canais não conhecidos em tempo de compilação e, mesmo que um canal seja usado somente por um pequeno lapso de tempo, o acesso deve ser mantido durante toda a existência do processo. Para permitir a especificação dinâmica de processos podemos estender um esquema de especificação estática com variáveis que contenham a fonte e o destino.

Quanto aos aspectos de sincronização na troca de mensagens é importante definir se a sua execução causa uma espera. Os esquemas básicos de protocolos de troca de mensagens incluem três tipos fundamentais: primeiro, o SEND/RECEIVE sem sincronização explícita entre os processos; segundo, o SEND-WAIT/ RECEIVE-REPLY que sincronizam o emissor e o receptor no ponto de troca de mensagem; e em terceiro lugar, o SEND-WAIT/ RECEIVE seguido posteriormente por um REPLY. Neste último esquema o emissor fica bloqueado esperando que o receptor envia uma resposta. Se logo após responder o receptor encerrar a sua execução, este esquema torna-se semelhante às chamadas remotas de procedimentos.

2.4. EXEMPLOS DE LINGUAGENS DE PROGRAMAÇÃO CONCORRENTE

O objetivo desta seção é apresentar de maneira sucinta algumas linguagens de programação concorrente. Nem todas foram muito divulgadas e algumas não passam de esforços experimentais que nem chegaram a ser implementadas. Contudo, elas exerceram influências no desenvolvimento de novas notações. Para exemplificar a sua utilização achamos apropriado considerar as suas aplicações no controle do sistema produtor / consumidor / memória intermediária limitada. A literatura especializada considera necessária a inclusão deste teste numa bateria de avaliação do uso de primitivas da programação concorrente [12]. Este problema envolve três processos que devem executar assincronamente:

o processo buffer que administra uma memória intermediária de tamanho limitado; o processo produtor que produz informações e as envia para a memória; e o processo consumidor que retira informações desta memória. Limitamos, neste exemplo, a capacidade de armazenamento da memória em 10 mensagens. O processo buffer deve impedir o produtor de enviar mensagens quando a memória estiver cheia e o consumidor de retirar mensagens quando ela estiver vazia. Nos exemplos de códigos fonte dos processos mostrados neste trabalho as numerações de linhas fazem parte da sintaxe das respectivas linguagens: são usadas apenas para facilitar as referências nas discussões. Devemos levar em consideração que um único exemplo não é suficiente para inferirmos a superioridade de uma linguagem sobre outras: as soluções para outros exemplos podem não corroborar esta conclusão.

2.4.1. PASCAL CONCORRENTE

Foi desenvolvido por Brinch Hansen para a programação estruturada de sistemas operacionais de computadores, estendendo a linguagem PASCAL com ferramentas de programação concorrente. Um programa em Pascal Concorrente é constituído de três componentes básicos: processos, monitores e classes. Todos definem uma estrutura de dados e os procedimentos que podem operar sobre ela, diferenciando-se apenas pelas maneiras como são ativados e escalonados. Os processos são componentes ativos que podem ser executados concorrentemente e seus objetos não são compartilhados. Os monitores e as classes são passivos: os seus procedimentos só são ativados quando chamados pelos processos. Os monitores podem ser ativados por mais de um componente (não monitor), e o compilador gera o código que provê a mútua exclusão necessária. A comunicação entre os processos é feita através dos monitores que representam desta forma estruturas de dados compartilhadas. As classes só podem ser chamadas privativamente por um único componente e são usadas para tornar os programas mais modulares. A proibição do acesso concorrente é garantida na compilação e torna as chamadas aos procedimentos de uma classe muito mais rápidas do que as chamadas aos monitores.

A sincronização é implementada através de variáveis do tipo fila que só podem ser manipuladas através das operação DELAY e CONTINUE. A operação

DELAY (nome_da_fila)

faz com que o processo executando um procedimento do monitor seja colocado na fila de espera designada em nome_da_fila, perdendo o acesso exclusivo a ele. Quando um outro processo executar a operação

CONTINUE (nome_da_fila)

um dos processos que está na fila de espera será reativado.

A principal contribuição do PASCAL CONCORRENTE é estender o conceito de monitor com uma hierarquia explícita de direitos de acesso a uma estrutura de dados compartilhados que pode ser declarada no texto do programa e verificada pelo compilador. A Implementação do exemplo que usaremos como padrão está na figura 2. A memória intermediária buff (linha 4) é administrada pelo monitor buffer (linha 25) do tipo tbuf (linha 2). Ele possui dois procedimentos que podem ser ativados remotamente: sen (linha 5), que coloca em buf a informação enviada por quem o ativou, e rec (linha 11), que retira informações de buf e a envia para quem o ativou. O procedimento sen se auto bloqueia quando buf estiver vazia (linha 7), só sendo desbloqueado (linha 15) após buf receber alguma informação. Um comportamento análogo ocorre com o procedimento rec: se auto bloqueia quando buf está cheia (linha 13), sendo desbloqueado no final de sen (linha 9). Na linha 25 declaramos também dois procedimentos pro, do tipo tpro

```
1   TYPE
2       tbuf = MONITOR;
3           VAR in, out: INTEGER;
4               buf: ARRAY ( .10. ) OF PORTION;
5           PROCEDURE ENTRY sen ( info: PORTION );
6           BEGIN
7               IF in = out THEN DELAY ( sen );
8               buf ( .in MOD 10. ): = info; in: = in+1;
9               CONTINUE ( rec );
10          END;
11          PROCEDURE ENTRY rec (VAR info: PORTION );
12          BEGIN
13              IF in - out = 10 THEN DELAY ( rec );
14              info: = buf ( .out MOD 10.); out: = out+1;
```

```
15              CONTINUE ( sen );
16          END
17          BEGIN in: = 0; out: = 0; END;
18      tpro = PROCESS ( buffer: tbuf );
19          VAR m: PORTION;
20          CYCLE buffer. sen ( m ); END;
21      tcon = PROCESS ( buffer: tbuf );
22          VAR m: PORTION;
23          CYCLE buffer. Rec ( m ); END;
24  VAR
25      buffer: tbuf; pro: tpro; con: tcon;
26  BEGIN
27      INIT buffer; pro ( buffer ); con ( buffer );
28  END.
```

Figura 2: Implementação do exemplo padrão em Pascal Concorrente.

(linha 18) e con, do tipo tcon (linha 21), que simulam o comportamento do produtor e o consumidor respectivamente. Como maiores detalhes da programação seqüencial não nos interessam, projetamos estes processos como meras repetições infinitas de ativações de procedimentos do monitor. O monitor e os processos são inicializados simultâneamente através do comando INIT (linha 27). O parâmetro nas inicializações de pro e con indica que os mesmos terão direito de acessar o monitor buffer.

2.4.2. PROCESSOS DISTRIBUÍDOS

Apesar de Pascal Concorrente atender aos requisitos para a programação concorrente abstrata, exige memória compartilhada, não sendo portanto, na sua forma atual, ideal para sistemas com memória exclusivamente distribuída.

O conceito de Processos Distribuídos foi proposto por Brinch Hansen para aplicações em tempo real controladas por redes de microcomputadores com memória distribuída, sendo o primeiro fundamentado em chamadas remotas de procedimentos. As idéias propostas são mais atraentes do que tentar adaptar linguagens já existentes, uma vez que unificam os conceitos de monitores e processos, e resultam em programas mais elegantes. Estas idéias tem as seguintes características básicas: primeiro, um programa consiste de um número fixo de processos

concorrentes que são inicializados simultaneamente, e não têm variáveis comuns; segundo, estes processos comunicam-se entre si por meio de chamadas remotas, que são uma chamada de um processo para um procedimento localizado em outro processo. As ativações de um mesmo processo são mutuamente exclusivas. Isto implementa monitores como processos ativos, ao invés de serem procedimentos passivos. Finalmente, os processos são sincronizados por meio de uma variação de regiões críticas condicionais denominados comandos protegidos.

A implementação do nosso exemplo padrão encontra-se na figura 3. O comando inicial do processo buffer (linha 4)

```
1   PROCESS buffer; s: SEQ[10] CHAR
2      PROC sen (c: CHAR) WHEN NOT s. FULL: s. PUT ( C ) END
3      PROC rec ( # v: CHAR ) WHEN NOT s. EMPTY: s. GET ( V ) END
4      s: = [ ]
5   PROCESS con; x: CHAR
6      DO TRUE
7         CALL buffer. sen ( x )
8      END
9   PROCESS prod; y: CHAR
10     DO TRUE
11        CALL buffer. rec ( y )
12     END
```

Figura 3: Implementação do exemplo padrão com Processos Distribuídos

inicializa a memória intermediária como vazia. As operações que podem ser chamadas neste processo são sen (linha 2) que coloca informações na memória e rec (linha 3) que retira informações. O Comando WHEN define uma região protegida do tipo

WHEN b1: s1 | b2 s2 | ... bi | si ... END

que espera até que uma das condições bi seja verdadeira, quando então executará o comando si correspondente. Se diversas condições forem verdadeiras a escolha para a execução é imprevisível. Os processos con e prod chamam os procedimentos sen e rec (linha 7 e 11) enviando e recebendo informações através dos seus parâmetros (x e y).

2.4.3. PROCESSOS SEQUENCIAIS COMUNICANTES

Processos Sequenciais Comunicantes (*) são baseados na passagem síncrona de mensagens e na comunicação seletiva. Os processos são denotados por uma variação do COBEGIN, podem compartilhar variáveis somente para a leitura, e usam comandos de entrada / saída para sincronização e comunicação. A especificação de canais é estática e direta. A comunicação entre processos ocorre quando um processo especifica outro como destino para saída, e o segundo processo especifica o primeiro como fonte para sua entrada. Em CSP os processos são denotados da seguinte maneira:

[X : : i : INTEGER; Y ! i / Y : : j : INTEGER; X ? j]

Esta notação concisa especifica dois processos X e Y que executarão em paralelo. O processo X envia ao processo Y a informação i através do comando Y ! i. O processo Y recebe esta informação na variável j com o comando de entrada X ? j.

CSP adota os comandos protegidos de Dijkstra como estrutura básica de controle do não determinismo, compostos basicamente de pares G → S. A proteção G pode conter condições booleanas seguidas opcionalmente por um comando de entrada, e é verdadeira se as condições booleanas são satisfeitas e o comando de entrada está em condições de ser executado. O comando S só é executado se a proteção G for verdadeira.

A notação

$$* [G_1 \rightarrow S_1 // G_2 \rightarrow S_2]$$

permite a repetição de um conjunto de comandos protegidos, enquanto pelo menos uma das proteções seja verdadeira. O símbolo // indica a escolha não determinística entre os comandos protegidos S_1 e S_2 quando ambas as proteções G_1 e G_2 forem verdadeiras.

---

* "Communicating Sequential Processes (CSP)".

Uma dificuldade surge do fato dos processos deverem explicitar o nome do outro, tornando difícil construir processos servidores, para atender e qualquer processo que o requisitar. Outro problema incômodo é a proibição de colocar comandos de saída dentro da proteção de comandos. CSP introduziu esta limitação para obter uma implementação simples dos mecanismos de comunicação e reduzir os problemas de bloqueio perpétuo. Se por um lado isto pode levar à uma implementação mais eficiente, por outro pode trazer a necessidade de interações múltiplas como no caso do nosso exemplo padrão, mostrado na figura 4. Na linha 5 o processo buffer verifica se a memória buf não está cheia e usa uma primitiva de recepção para verificar se ela pode receber uma mensagem enviada pelo processo pro. Na linha 6 testa se a memória não está vazia e a primitiva de recepção é usada para ver se o pedido de envio de mensagem feito pelo processo con pode ser recebido. Ao receber este pedido de envio buffer manda a mensagem retirada da memória para con. Na linha 11 con manda o seu pedido de envio de mensagem e na linha 12 recebe-a. Na linha 16 pro envia sua mensagem a buffer. Estas operações de envio são síncronas e nenhum dos processos pode continuar até que suas mensagens sejam

```
1   [ buffer: :
2       buf: ( 0 . 9 ) PORTION;
3       in, out: INTEGER;
4       in: = 0, out: = 0;
5       * [ in - out < 10; pro ? buf ( in MOD 10 ) →
            in: = in+1;
6           // in > out; con ? more ( ) →
            con ! buf ( out MOD 10 );
7           out: = out+1;
8       ] /
9   con: :
10      * [  ...
11          buffer ! more ( );
12          buffer ?m;
13          ] /
14  pro: :
15      * [  ...
16          buffer ! m;
17          ] ]
```

Figura 4: Implementação do exemplo padrão em CSP

aceitas por buffer. O sinal more ( ) é necessário para o consumidor indicar que está pronto para receber outro dado de buffer; isto é necessário devido à ausência de proteções de saída em CSP.

2.4.4. PLITS

O projeto PLITS(+) teve como uma das suas premissas básicas mais significativas a dificuldade de se combinar um alto grau de paralelismo com dados compartilhados. Portanto, a passagem de mensagens é o meio apropriado para a interação de processos em um sistema distribuído. Um programa em PLITS consiste de módulos, e os processos são módulos ativos.

A criação de módulos é dinâmica e pode ser efetuada em qualquer ponto do programa. Um módulo envia uma mensagem executando o seguinte comando:

SEND expressões TO destino [ABOUT chave]

A frase ABOUT CHAVE é opcionalmente usada para identificar a mensagem e é a solução para diversos problemas de recepção seletiva, controle de fluxo e servidores multiplexados. Para receber uma mensagem o módulo executa o seguinte comando:

RECEIVE varáveis [FROM fonte] [ABOUT chave]

O módulo que executa o RECEIVE é retardado até a chegada de uma mensagem. As cláusulas FROM e ABOUT implementam seletividade na recepção. FROM fonte exige que a mensagem tenha sido enviada pelo módulo denominado fonte, e ABOUT chave exige que a sua identificação seja igual a chave.

Combinando as opções de SEND e RECEIVE de diferentes maneiras o programador pode exercer uma variedade de controles sobre a comunicação.

+ "Programming Language in The Sky".

A solução de exemplo padrão implementada em PLITS encontra-se na figura 5. Se a variável buf que implementa a memória intermediária limitada estiver vazia (in = out), o processo buffer é retardado através de um comando de recepção (linha 9) até que o processo pro envie uma mensagem. De uma maneira análoga, na linha 12, quando buf está cheia (in-out = 10), buffer espera até que o processo con

```
1   ONST buffer = MOD
2   BEGIN
3      PUBLIC info: PORTION;
4      VAR in, out: INTEGER; buf: ARRAY 0:9 OF PORTION;
5           req, rep: MESSAGE;
6      in: = 0; out: = 0;
7      WHILE TRUE DO BEGIN
8         IF in = out THEN  ( buf vazio )
9           RECEIVE req FROM pro
10          buf ( in MOD 10 ): = req. info; in: = in+1;
11        ELSE IF in-out = 10 THEN        ( buf cheio )
12           RECEIVE req FROM con;
13           req. info: = buf ( out MOD 10 );
14           SEND rep TO con; out: = out+1;
15        ELSE RECEIVE req;
16           CASE req. source OF
17              pro: buf ( in MOD 10 ): = req. info; in: = in+1;
18              con: rep. info: = = buf ( out MOD 10 ); SEND rep
19                   TO con; out: = out+1;
20              END;          {CASE}
21           ENDIF
22        END;        {WHILE}
23     END.  {buffer}
24  CONST con = MOD
25     BEGIN
26        VAR m, rep: MESSAGE;
27        WHILE TRUE DO BEGIN
28           SEND m1 TO buffer;
29           RECEIVE rep FROM buffer; m: = rep. info;
30        END;        {WHILE}
31     END.  {con}
32  CONST pro = MOD
33     BEGIN
34        VAR m1: MESSAGE;
35        WHILE TRUE DO BEGIN
36           m1. info: = m; SEND m1 TO buffer;
37           END;    {WHILE}
38        END.        {con}
```

Figura 5: Implementação do exemplo padrão em PLITS

retire uma mensagem. Nos demais casos, quando a memória não está nem cheia nem vazia, buffer pode atender indistintamente a pro ou a con. Na linha 15 temos uma recepção não seletiva para implementar uma escolha não tendenciosa de processo a ser atendido. Esta solução é imposta pela ausência de construções em PLITS para tomar decisões não determinísticas.

2.4.5. PORTAS DE COMUNICAÇÃO

Portas de comunicação são um conceito de linguagem para escrever programas concorrentes de alta qualidade e eficientes. São semelhantes aos Processos Distribuídos de Brinch Hansen, com duas características adicionais: especificam precisamente que processos podem comunicarem-se com um outro, bem como quando a comunicação entre dois processos deve ser desconectada. A necessidade de sincronização é satisfeita pela introdução da noção de portas de comunicação não determinísticas. Dentro deste esquema um programa consiste unicamente de processos. Componentes globais tais como monitores ou variáveis compartilhadas não são necessários.

Do ponto de vista da sintaxe da linguagem o esquema de portas de comunicação manifesta-se como comandos de comunicação num processo. Três tipos de comandos podem ser usados: comando porta, comando de conexão, e comando de desconexão.

Um comando porta é semelhante a um procedimento e é usado para a comunicação entre processos:

```
PORT nome_porta ( parâmetros );
    BEGIN
        comandos;
        ! !        ... comando de desconexão ...
    ENDPOT
```

Um procedimento executando o comando de conexão

CONECT nome_proc. Nome_porta (parâmetros)

é suspenso temporariamente até que o comando de desconexão seja executado pelo processo conectado.

A diferença entre o processo que contém o comando CONECT (chamado processo servidor) e o processo que contém o comando porta correspondente (chamado processo mestre) é que o mestre tem o direito de desconectar a comunicação no momento em que quiser, determinando um maior grau de paralelismo. Como não existe compartilhamento de memória toda a comunicação é feita pela passagem de parâmetros nas chamadas às portas.

Situações não determinísticas são obtidas precedendo-se a declaração de portas com o símbolo // que permite a seleção arbitrária entre diferentes portas; e da especificação de condições booleanas na declaração CONDITION que, se falsas, evitam a aceitação de mensagens pela porta, semelhantemente aos comandos protegidos de CSP. A implementação da solução do nosso problema encontra-se na figura 6. No processo buffer temos declaradas duas portas sen e rec (linhas 7-12 e 13-19). Estas portas são

```
1    PROCESS buffer;
2    VAR buf: ARRAY [ 0 .. 9 ] OF PORTION
3        in, out: INTEGER
4    BEGIN
5      in: = 0; out: = 0;
6      CYCLE
7        // PORT sen ( c: PORTION );
8          CONDITION in<out+1;
9          SERVANT pro
10         BEGIN ! !
11           buf ( in MOD 10 ): = c;
12           in: = in+1; ENDPORT
13       // PORT rec ( # v: PORTION );
14         CONDITION out<in
15         SERVANT con
16         BEGIN
17           v: = buf ( out MOD 10 ); ! !
18           out: = out+1; ENDPORT
19       ENDCYCLE
```

```
20      END
21  PROCESS con;
22      VAR x: PORTION;
23      BEGIN
24          CYCLE
25              CONNECT buffer. rec ( X );
26                ENDCYCLE
27          END
28  PROCESS pro
29      VAR y: PORTION
30      BEGIN
31          CYCLE
32              CONNECT buffer. sen ( y );
33              ENDCYCLE
34              END
```

Figura 6: Implementação do exemplo padrão com Portas

executadas de maneira não determinísticas e o acesso a elas está protegido pelas condições especificadas nas linhas 8 e 14. As declarações SERVANT (linhas 9 e 15) garantem que só os processos especificados tem acesso às portas. Quando os processos con e pro executam uma conexão com uma porta (linhas 25 e 32) eles são suspensos até que as portas conectadas executem o comando de desconexão (linhas 10 e 17). O uso do comando de desconexão tão logo seja possível tem a vantagem de elevar o nível de paralelismo.

2.4.6. ADA

A linguagem ADA (*) combinou vários conceitos tais como entradas de procedimentos dos monitores e a idéia de comunicação entre processo ao invés de simples sincronização de processos que chamam entradas de procedimentos. A unidade de concorrência em ADA é chamada de "task", cujo corpo pode ser compilado separadamente da sua especificação para facilitar a construção modular de grandes sistemas. A ativação de "tasks" em ADA combina as características do COBEGIN, que é a mais estruturada porém restrita aos casos em que os padrões de execução concorrente podem ser especificados no programa fonte, com as facilidades da inicialização dinâmica.

---

* ADA é marca registrada do Departamento de Defesa dos EUA.

A interação entre "tasks" em ADA é feita através de uma sincronização seguida de comunicação, denominada ponto de encontro (++), implementado pelas chamadas remotas e pelo comando ACCEPT. Um ponto de encontro pode ser visto como um procedimento remoto iniciado com a recepção dos parâmetros de entrada, e encerrado com o retorno dos parâmetros de saída para a "task" chamante.

Um "task" só pode receber uma chamada remota quando está executando um comando ACCEPT. Ambas operações, a chamada remota e o ACCEPT, são bloqueáveis, ou seja: se uma "task" executa uma delas, mas a operação correspondente não foi ainda executada por outra "task", a primeira ficará bloqueada até que esta condição seja satisfeita.

A comunicação seletiva não determinística é obtida usando o comando SELECT:

```
SELECT
   WHEN condição_1 =>
        : : :
   WHEN condição_2 =>
        : : :
   END SELECT
```

que é semelhante aos comandos protegidos de CSP. A solução do exemplo padrão encontra-se na figura 7. Os comandos ACCEPT nas linhas 13 e 19 definem os pontos nos quais a "task" buffer espera ser chamada. Nas linhas 28 e 33 as "tasks" con e pro se comunicam com buffer através de ativações remotas. A execução de ambas é retardada até o ponto da desconexão. Os comandos END que seguem os ACCEPT (linhas 15 e 20) efetuam a desconexão, retornando mensagens às "tasks" chamantes para desfazer a sincronização. Nesta solução con e pro se comunicam de uma maneira simétrica: ambas executam repetidas vezes os comandos de comunicação pedindo ou enviando mensagens a buffer. Na ativação remota, quando uma "task" está esperando por uma informação, ela notifica automaticamente a "task" que deve enviá-la. Desta

```
1   TASK buffer IS
2       ENTRY sen ( p1: IN PORTION );
3       ENTRY rec ( p2: OUT PORTION );
```

++ Também chamada "rendezvous".

```
4   END;
5   TASK BODY buffer IS
6      buf: array ( 0 .. 9 ) OF PORTION;
7      in: INTEGER: = 0;
8      out: INTEGER: = 0;
9   BEGIN
10     LOOP
11        SELECT    - não determinismo
12           WHEN in - out<10 = >- não cheio
13                 ACCEPT sen ( p1: IN PORTION ) DO
14                    buf ( in MOD 10 ): = p1;
15                 END;        - desconexão
16                 in: = in+1;
17        OR
18           WHEN in>out =>        - não vazio
19                 ACCEPT rec ( p2: OUT PORTION ) DO
20                    p2: = buf ( out MOD 10 );
21                 END;        - desconexão
22                 out: = out+1;
23        END SELECT;
24     END LOOP;
25  END buffer;
26  TASK BODY con IS
27     LOOP
28        buffer. rec ( m );
29     END LOOP;
30  END con;
31  TASK BODY pro IS
32     LOOP
33        buffer. sen ( m );
34     END LOOP;
35  END pro;
```

Figura 7: Implementação do exemplo padrão em ADA

maneira evita-se o pedido de envio de mensagens que tem de ser explicitamente colocado em algumas linguagens.

3. UMA AVALIAÇÃO DOS ASPECTOS DENOTACIONAIS DA PROGRAMAÇÃO CONCORRENTE NO SISTEMA OPERACIONAL UNIX

O UNIX é uma família de sistemas operacionais para computadores, desenvolvida pelos Laboratórios Bell. Escolhemos este sistema operacional para desenvolvermos este trabalho pois além de apresentar uma simplicidade, generalidade e, acima de tudo, uma inteligibidade não usuais, oferece características raramente encontradas mesmo em sistemas operacionais maiores, destacando-se:

(i) um sistema de arquivos hierárquicos incorporando volumes desmontáveis;

(ii) compatibilidade entre arquivos e dispositivos de entrada/saída;

(iii) programação concorrente através de processos assíncronos;

(iv) um poderoso interpretador de comandos, que usa uma linguagem de alto nível para discorrer sobre programas e arquivos;

(v) cerca de 100 subsistemas, incluindo uma dúzia de linguagens;

(vi) um alto grau de portabilidade, constituindo uma base estável de "software" em relação a mudanças drásticas de "hardware"; e

(vii) uma filosofia de "ferramentas" que, pela facilidade de combinar programas, encoraja o uso de módulos de software pequenos e coerentes.

Neste capítulo fazemos uma avaliação sobre o ambiente de programação concorrente oferecido pelo UNIX sob o ponto de vista relativo aos aspectos denotacionais, ou seja, como o programador especifica uma computação concorrente/cooperante. As restrições de desempenho na execução de tarefas cooperantes que são impostas pelos mecanismos de sincronização e comunicação entre processos, embora sejam importantes, não serão tratados neste trabalho.

Os resultados desta avaliação não devem ser considerados como falhas do projeto original do UNIX uma vez que, segundo Ritchie [20], uma grande versatilidade de programação concorrente não era o objetivo fundamental do UNIX. Além do mais, a

arquitetura do PDP-11, no qual o UNIX foi desenvolvido, impõe sérias limitações ao tamanho do núcleo, reduzindo assim as facilidades possíveis de serem implementadas.

Na próxima seção, apresentamos certos aspectos denotacionais de processos, tecendo considerações de como são especificados os processos concorrentes (cooperantes ou não). A seção 3.2 traz os aspectos denotacionais da comunicação entre processos no UNIX, quer eles cooperem para trocar dados, quer cooperem para se sincronizarem. Quanto à sincronização, na seção 3.3 analisamos o seu uso para atender quer às necessidades de exclusão mútua para o compartilhamento de recursos, quer para a sincronização de condições.

3.1. COMO ESPECIFICAR A EXECUÇÃO CONCORRENTE NO UNIX

Exceto no caso das rotinas de autocarregamento do sistema, um novo processo no UNIX só pode ser criado pela duplicação de um outro processo já existente, usando-se para isto a chamada fork( ):

```
id_processo = fork( );
```

O novo processo (filho) herda uma cópia completa da imagem do processo mais antigo (pai). A partir daí são executados independentemente, diferindo apenas nos aspectos seguintes. Primeiro cada processo tem seu próprio número de identificação. Segundo, a chamada fork( ), retorna neles valores diferentes: no processo pai em caso de sucesso da duplicação, é retornada a identificação do processo filho, e em caso de insucesso, o valor -1; no processo filho, sempre é retornado o valor 0.

Com o intuito de se esclarecer o funcionamento da chamada fork( ), apresentamos na Figura 8 um exemplo de seu uso (*).

---

* Todos os exemplos deste capítulo estão codificados na linguagem de programação C.

```
main ( ) {

    int id_proc;

    if ‘( id_proc = fork( ) ) < 0 ) {
        fprintf ( stderr, “Não pode criar processo\n” );
        exit ( 1 );
    }
    if ( id_proc = = 0 )
        /* código executável só pelo filho */
        write ( 1, “filho\n”, 6 );

    else
        /* código executável só pelo pai */
        write ( 1, “pai\n”, 4 );

    /* código executável por ambos */
    write ( 1, “ambos\n”, 6 );
}
```

Figura 8: Exemplo do uso da primitiva fork( )

A execução com sucesso deste programa gerará uma saída comuma vez os “strings” “pai” e “filho”, e duas vezes o “string” “ambos”.

Além das vantagens e desvantagens inerentes à filosofia do FORK já mostradas na seção 2.1, versão do FORK usada pelo UNIX apresenta outras desvantagens no que diz respeito aos aspectos denotacionais. Citemos quatro.

A primeira desvantagem é que, para caracterizarmos quem é pai, e quem é filho, devemos testar o valor retornado pelo fork( ).

A segunda é que o fork( ) duplica todo o código do processo que o ativou. Caso queiramos que o filho execute um código diferente, temos que usar a chamada exec( ), que faz com que o sitema leia e execute o programa contido no arquivo especificado como parâmetro da chamada.

A terceira, é que não há uma boa separação entre a declaração e a ativação do processo, ou seja, os comandos dos dois processos ficam estruturalmente misturados. Uma possível solução para este problema é apresentada na Figura 9.

```
main ( ) {
   • • •
   if ( fork( ) = = 0 )
      proc_a ( );
   else
      proc_ _b ( );
   • • •
}

proc_a ( ) {
   • • •
}

proc_b {
   • • •
}
```

Figura 9: Exemplo da declaração de processos usando procedimentos

Porém como proc_a ( ) e proc_b ( ) são procedimentos, e não declarações da processos, pode ser usados e ativados em qualquer outra parte do programa e, para vermos se estão funcionando como processos paralelos, ou como simples procedimentos, temos que procurar localizar exatamente onde são ativados.

A quarta e última desvantagem, como vimos na Figura 8, é que pode existir uma parte do programa executável tanto pelo pai, quanto pelos filhos. Como é raro querer-se fazer isso, torna-se necessário uma grande atenção do programador em encerrar o processo após o mesmo ter executado o código a ele destinado, sob pena da parte comum do programa ser executada indesejavelmente.

A necessidade de usar comandos que não estão diretamente ligados à especificação de processos cooperantes, e a mistura entre a ativação e a declaração de processos, infringem todas as regras da programação estruturada no tocante à clareza local prejudicando, conseqüentemente, a legibilidade do texto. Com isto podemos concluir que, apesar de ser uma ferramenta poderosa sob o UNIX, a utilização da chamada fork( ) é meticulosa, exigindo um programador com boa disciplina de programação.

3.2. COMUNICAÇÃO ENTRE PROCESSOS NO UNIX: ASPECTOS DENOTACIONAIS

Processos cooperantes podem trocar dados, e devem se sincronizar, quer para trocar dados, quer para funcionar corretamente, mesmo sem trocar dados. Nesta seção examinamos os mecanismos disponíveis no UNIX para a troca de dados entre processos (sub-seção 3.2.1) e para a sincronização de processos cooperantes (sub-seção 3.2.2). Finalmente fazemos uma avaliação crítica destes mecanismos, no que diz respeito às suas desvantagens denotacionais (sub-seção 3.2.3).

3.2.1. Troca de Dados

No UNIX os processos não compartilham dados na memória ([+]). Existem, porém, três maneiras dos processos trocarem informações: usando arquivos normais, sinais, ou dutos ([++]).

3.2.1.1. Troca de Dados Usando Arquivos Normais

A troca de dados usando arquivos normais exige uma rigorosa disciplina de acesso aos mesmos, requerendo para isto, do usuário, a implementação de exclusão mútua, uma vez que a mesma não é oferecida automaticamente pelo UNIX. Infelizmente, devido aos problemas de troca de dados inerentes às outras técnicas (que apresentaremos na sub-seção 3.2.3), a troca de dados usando arquivos normais é freqüentemente a única maneira indicada.

3.2.1.2. Troca de Dados Usando Sinais

Um sinal é uma maneira assíncrona de um processo sinalizar um evento para outro processo. Como a informação enviada pelo sinal consta de um só bit (o evento ocorreu ou não), torna-se uma técnica mais útil à sincronização entre processos do que à troca de dados.

3.2.1.3. Troca de Dados Usando Dutos

Um duto é um arquivo do tipo lista "FIFO", com capacidade de armazenamento limitada e possuindo um controle de fluxo automático, com a sincronização feita pelo sistema. Uma vez que o sistema operacional oferece todas as facilidades básicas de controle e

---

[+] No novo UNIX/SYSTEM V, esta troca é permitida.

[++] Também conhecidos por "pipes".

sincronização, os dutos apresentam-se como a técnica fundamental de troca de dados sob o UNIX.

Diversos processos podem compartilhar a mesma extremidade de um duto. Quando o duto é criado, o processo aparenta ter dois novos arquivos, um aberto para leitura e o outro aberto para gravação. Se o processo é duplicado, seu filho receberá na sua cópia da imagem a tabela dos arquivos abertos para o pai no momento da duplicação. A partir deste momento estes arquivos estarão também abertos para o filho. Como o duto só pode ser usado como um canal de uma só direção, antes de usá-lo os processos que vão ler no duto devem fechar o descritor de gravação, e os que vão gravar devem fechar o descritor de leitura. Isto se torna necessário para que um processo lendo (gravando) um duto para o qual não existem mais escritores (leitores) seja avisado do fato. Se um processo quer ler um duto que está vazio, ele esperará até que apareçam dados. Se um processo quer escrever em um duto cheio, ele esperará até que surja espaço suficiente.

Na Figura 10 damos um exemplo da troca de dados usandodutos, através da implementação do seguinte algorítmo:

> Dada uma matriz nxn de elementos binários, construir uma outra onde o valor de cada elemento é 1, se a maioria dos seus vizinhos for 1, ou 0 em caso contrário.

Esta implementação não é prática uma vez que a criação de processos é dispendiosa. Contudo, é bastante útil como exemplo. A função media_local utiliza-se do fato dos processos não compartilharem dados na meméria, e que cada filho tem sua própria cópia da matriz original (inalterável pelos outros), apresentando as seguintes características:

(i) A matriz dada está inserida numa matriz (n+2) x (n+2), com zeros nos limites para evitar a necessidade de testes;

```
media_local ( a, n )
unsigned shrot a [ ] [ ], n;  {

   unsigned short media ( );
   unsigned short i, j;
   int pid, pip [ 2 ];
   struct  {
      unsigned short r, i, j;
   }  y;

   if ( pipe ( pip ) < 0 )
     return ( 0 );

     for ( j = 1; j <= n; j + + )
        for ( i = 1; i <= n; i + + )  {
           pid = fork ( );
           if ( pid < 0 )
              continue;
           if ( pid = = 0 )  {
              close ( pep [ 0 ] );
              y. r = media ( a, i, j );
              y. i = i;
              y. j = j;
              write ( pip [ 1 ], εy, sizeof y );
              exit ( 0 );
           }
        }

     close ( pip [ 1 ] );
     for ( j = o; read (pip [ 0 ], εy, sizeof y ) = = sizeof y;
              j + + )
        a [ y.i ] [ y.j ] = y.r;
     return ( j = = n*n );
}

  unsigned short media ( a, i, j )
  unsigned short a [ ] [ ], i, j;  {
     int u, v, x;

     x = 0;
     for ( u = -1; u <= 1; u + + )
        for ( v = -1; v <= 1; v + + )
           x + = a [ i+u ] [ j+v ];
     x - = a [ i ] [ j ];
     return ( x > 4 );
  }
```

Figura 10: Exemplo da troca de dados usando dutos

(ii) Para cada elemento da matriz, é criado um processo separado, que é executado concorrentemente com os outros. Este processo computa o novo valor de sua célula e coloca-o num duto juntamente com as coordenadas do elemento na matriz;

(iii) O pai 10 o duto e insere os novos valores na matriz original; e

(iv) Quando todos os filhos forem executados, o pai recebe um valor zero na chamada read (cada processo, ao se encerrar, fecha seus arquivos, incluindo-se suas extremidades de dutos).

Tendo apresentado as técnicas básicas de troca de dados no UNIX, voltemos a nossa atenção às técnicas de sincronização. Lembremos que a crítica dessas técnicas de programação será abordada na seção 3.2.3.

3.2.2. Como Especificar a Sincronização no UNIX

Processos cooperantes necessitam sincronizarem-se para a realização de uma exclusão mútua entre processos compartilhando recursos, e para a sincronização de condições, como por exemplo: "o processo A deve fazer X antes do processo B fazer Y".

3.2.2.1. Exclusão Mútua

Quando processos querem trocar dados usando arquivos normais, é necessário que o usuário trate do problema da exclusão mútua. Para implementar a exclusão mútua, o usuário pode utilizar qualquer primitiva indivisível do UNIX que tenha um efeito permanente. A primitiva normalmente usada é a de criar um arquivo normal vazio. Se a troca de dados for feita usando dutos, o próprio operacional se encarrega de manter as operações de leitura e gravação indivisíveis, não tendo o usuário que se preocupar com os problemas de exclusão mútua.

3.2.2.2. Sincronização de Condições

A sincronização de condições pode ser feita através da chamada wait( ), de sinais, ou de dutos, como veremos nas sub-seções seguintes.

3.2.2.2.1. Sincronização com Wait

A chamada wait( ) é equivalente, no UNIX, ao JOIN do par FORK/JOIN:

```
id_proc = wait( status);
```

faz com que o processo que a ativou seja bloqueado até que um dos seus filhos termine a sua execução. A chamada wait( ) retorna a identificação do processo que terminou. Se o processo não tem filhos, é retornado o valor -1.

P programa da Figura 11 é um exemplo do uso da chamada wait( ) para obter uma sincronização entre pai e filho: o "string" "filho" será escrito antes do "string" "pai de ...". Na Figura 8 tivemos que usar a primitiva write( ) para escrever as mensagens porque, como os processos não estavam sincronizados, era necessário usar uma primitiva indivisível para não termos problemas de exclusão mútua.

```
main ( ) {

    int id_proc;

    if ( ( id _porc = fork( ) ) < 0 ) {
        fprintf (stderr, "Não pode criar processo\n" );
        exit ( 1 );
    }
    if ( id_proc = = 0 )
        printf ( "filho\n" );
    else {
        wait ( );
        printf ( "pai de %d\n", id_proc );
    }
}
```

Figura 11: Exemplo de sincronização com a chamada wait ( )

Neste exemplo podemos usar a chamada printf ( ), que é a maneira normal de imprimir mensagens, porque a sincronização obtida com a chamada wait ( ) garante a exclusão mútua no acesso ao arquivo de saída.

3.2.2.2.2. Sincronização com Sinais

Os sinais são uma ferramenta que o UNIX reserva para fins específicos, tais como interrupções de terminais e condições anormais. Como existe um pequeno número fixo (normalmente 16) de sinais possíveis no UNI, toda vez que queremos usar um deles para fins de sincronização, temos que alterar a ação programada para ele, "roubando" o sinal do seu uso normal, como mostramos na Figura 12. Neste exemplo o "string" "pai de ..." será impresso antes do "string" "filho".

```
#include <signal.h>
usigned short ok;
main() {

    int id_proc;
    int interrupcao();

    /* associa SIGINT à rotina interrupcao() */
    signal(SIGINT, interrupção);

    ok = 0;
    if((id_proc = fork()) < 0) {
        fprintf(stderr,"Nao pode criar processo\n");
        exit(1);
    }
    if(id_proc == 0) {
        while(!ok)
            pause();
        printf("filho\n");
    } else {
        printf("pai de %\n", id_proc);
        kill(id_proc,SIGINT);    /* manda o sinal */
    }
}

interrupcao() {
    ok = 1;
}
```

Figura 12: Exemplo do uso de sinais para sincronização

3.2.2.2.3. Sincronização com Dutos

Já que o controle do fluxo num duto é automaticamente feito pelo UNIX, o seu uso é bastante simples e geral, como mostramos na Figura 13. Este código garante que "filho" será sempre escrito antes de "pai de ...", independentemente de como o UNIX comuta os processos.

Um outro exemplo de sincronização como dutos é o exemplo clássico do produtor/consumidor, que mostramos na Figura 14.

```
main() {
    int id_proc pip[2];
    unsigned short lock;

    lock = 0;
    if (pipe(pip) < 0) {
        fprintf(stderr,"Nao pode criar pipe\n");
        exit(1);
    }

    if ((id_proc = fork()) < 0) {
        fprintf(stderr,"Nao pode criar processo\n");
        exit(1);
    }

    if (id_proc == 0) {
        /* fecha o descritor de leitura */
        close(pip[0]);
        printf("filho\n");
        /* escreva ao pai, permitindo-o continuar */
        write(pip[1], &lock, sizeof lock);

    } else {
        /* fecha o descritor de gravação */
        close(pip[1])
        /* espera que o filho escreva */
        read(pip[0], &lock, sizeof lock);
        printf("pai de %d\n", id_proc);
    }
}
```

Figura 13: Exemplo do uso de dutos para sincronização

3.2.3. Avaliação dos Mecanismos de Troca de Dados d Sincronização

Após examinarmos, nas sub-seções anteriores, os mecanismos para troca de dados e para sincronização de processos cooperantes disponíveis no UNIX, passamos agora a fazer a avaliação crítica relativa a seus aspectos denotacionais. Na sub-seção 3.2.3.1 avaliaremos o uso da chamada wait(); na sub-seção 3.2.3.2 avaliaremos o uso de sinais; e, finalmente, na sub-seção 3.2.3.3, avaliaremos o uso de dutos.

```
main () {

   int id_proc pip[2];
   char mensagem[TAM_MSG];

   if (pipe(pip) < 0) {
      fprintf(stderr,"Nao pode criar pipe\n");
      exit(1);
   }
   if ((id_proc = fork ()) < 0)) {
      fprintf(stderr,"Nao pode criar processo\n");
      exit(1);
   }
   if(id_proc == 0) {                    */ consumidor */
      close(pip[1]);
      while(TRUE){
         read(pip[0], mensagem, size of mensagem);
         /* consumir a mensagem */
                  . . .
      }
   } else {      /* produtor */
      close(pip[0];
      while(TRUE) {
         /* produzir mensagem */
                  . . .
         write(pip[1], mensagem, size of mensagem);
      }
   }
}
```

Figura 14: Exemplo de processos produtor/consumidor com dutos.

3.2.3.1. Avaliação do Uso do Wait

A chamada wait( ) é uma ferramenta voltada para casos muito especiais de sincronização. Esta falta de generalidade limita sua utilização. A chamada wait( ) tem

algumas características próprias. Primeiro, a sincronização só é feita quando o filho termina, ou seja, uma vez sincronizados não poderá mais existir qualquer interação entre eles. Segundo, quando existem diversos filhos temos que usar várias vezes a chamada wait( ) e, como ela não especifica por qual filho estamos esperando, o seu uso neste caso tende a tornar-se complexo.

3.2.3.2. Avaliação do Uso de Sinais

O uso de sinais como ferramenta de sincronização apresenta os seguintes aspectos denotacionais negativos. Primeiro, seu uso sempre implica no "roubo" de sua atividade normal de sinalizar condições anormais, exigindo um cuidado e atenção redobrados. Segundo, quase todos os sinais após serem recebidos, são novamente associados à ação básica automática ([+]). Assim, se é desejável receber e processar repetidamente esses sinais, devemos usar várias vezes a chamada signal(), que associa uma nova ação ao sinal desejado. Se repetidos sinais são recebidos antes que o anterior possa ser re-assinalado, eles serão processados pela ação básica. Por fim, quando um sinal é recebido durante certas chamadas ao supervisor, elas são abortadas pelo usuário, a quem cabe decidir reexecutá-las, ficando ao seu encargo a construção da rotina necessária para essas reexecuções.

3.2.3.3. Avaliação do Uso de Dutos

O uso de dutos apresenta vários aspectos denotacionais negativos que descrevemos a seguir.

Denominam-se "Processos servidores", aqueles que efetuam serviços para vários outros ([++]). Um exemplo de um processos servidor é o spooler de saída que administra o acesso à impressora, recebendo dados ou comandos de outros processos. A primeira desvantagem do uso de dutos no UNIX é que um duto tem que ser criado por um ancestral comum aos processos que o compartilham. Isto impede que os dutos sejam usados para a comunicação com processos servidores, que são normalmente criados durante a inicialização do sistema. Essa restrição não se aplica somente a processos servidores, mas esses fornecem

---

[+] Estas ações também são conhecidas como "default".

[++] Tais processos rambém são chamados de "daemon".

um exemplo comum de processos para os quais o mecanismo de dutos não pode ser usado. A única maneira disponível para se comunicar com tais processos é através da utilização de arquivos comuns, ficando ao encargo do programador todas as tarefas inerentes à sincronização.

Para exemplificarmos uma segunda desvantagem consideremos vários processos que queiram se comunicar entre si através de dutos, onde a decisão final sobre quais comunicações realmente se efetivarão só será tomada durante a execução dos mesmos. No UNIX, a criação de dutos não é dinâmica no sentido de que, após o(s) filho(s) ter(em) sido criado(s), não há meios do pai estabelecer novos dutos com ele(s). Portanto todos os possíveis dutos deverão estar previamente criados antes do pai executar a chamada fork(), mesmo que os processos não utilizem vários dos dutos criados.

Como terceira desvantagem, uma vez que o processos tem que fechar um dos descritores do duto, introduz-se no código do programa instruções que não estão diretamente ligadas ao envio de mensagens, dificultando a clareza local do programa.

Quarta, é difícil associar um duto a um arquivo já existente. Por exemplo, um processo sempre começa a rodar sob o UNIX com três arquivos abertos: a entrada padrão, a saída padrão e a saída padrão de erro. Esses arquivos são normalmente associados a um dispositivo (o terminal do usuário). A associação de um duto a um desses arquivos tem uma implementação pouco clara [16].

Quinta, a operação de leitura (gravação) em um duto é bloqueável: se o duto estiver vazio (cheio), o processo espera até que apareça algum dado disponível para p\leitura (um espaço disponível para gravar). Porém, não é infreqüente a necessidade de operações não bloqueáveis.

Finalmente, a sexta desvantagem é que um processo não manda uma mensagem endereçando-a a um outro processo, mas sim colocando-a num canal de comunicação (duto). Isto faz com que a implementação de comunicações do tipo "um-a-um" e "vários-a-um" (onde se deseja enviar uma mensagem a um determinado processo) fique pouco clara. pela mesma razão, temos a situação em que qualquer processo acessando este duto pode apanhar uma mensagem não destinada a ele.

4. PROPOSTA DE SOLUÇÃO: A LINGUAGEM DE PROGRAMAÇÃO BIS

Vimos no capítulo anterior que, apesar de ter várias características importantes, o sistema operacional UNIX não oferece um ambiente adequado para a programação concorrente. Os mecanismos disponíveis para a especificação, ativação, comunicação e sincronização de processos apresentem características denotacionais que dificultam e limitam a sua utilização pelos programadores.

O melhoramento deste ambiente pode ser atingido de duas maneiras básicas. A primeira seria uma mudança radical no núcleo do sistema operacional UNIX para prover o usuário com chamadas ao supervisor que permitiriam uma programação concorrente mais estruturada do que a disponível com as chamadas atuais (fork, exec, pipe, signal, kill, etc). Esta forma de melhorar o ambiente de programação tem várias desvantagens. Destaquemos duas. Primeiro, a substituição da antigas chamadas ao supervisor por outras mais estruturadas implicaria em reescrever quase todos os comandos e utilitários existentes no UNIX, o que representaria um esforço considerável. Para eliminar essa desvantagem, poderíamos incluir novas chamadas ao supervisor, sem eliminar as antigas. Infelizmente existe no PDP-11 uma séria restrição quanto ao tamanho do espaço de endereçamento lógico de um processos. Como o núcleo do sistema é implementado como um processo (permanentemente alocado na memória), essa restrição impossibilita a inclusão de várias novas chamadas.

Uma segunda maneira de melhorar o ambiente de programação concorrente no UNIX seria mapear um conjunto de primitivas mais poderosas para o conjunto oferecido pelo UNIX. Isso proveria o programador com a ferramenta desejada sem implicações no tamanho do núcleo do UNIX e sem o esforço de recodificar os comandos e utilitários do sistema. Essa maneira, entretanto, apresenta como desvantagem a falta de rigor na sintaxe de programação, como citamos na seção 3.2.3.3. Em alguns casos (menos numerosos), a causa do problema é a semântica de mapeamento é que ela não resolve os problemas semânticos apontados.

Adotamos a segunda maneira de proceder. Além de melhorar o ambiente de programação, a escolha do conjunto de primitivas incluídas atendeu a duas restrições importantes: o mapeamento (usando-se um pré-processador sintático) deve ser possível; e a execução do programa resultante deve ser razoavelmente eficiente, quando comparada à execução de um programa semelhante usando u conjunto normal de primitivas do UNIX.

Muito embora as escolhas do sistema hospedeiro e da linguagem de programação sejam conceitualmente independentes, deve haver significativas evidências de harmonia entre eles. Uma vez fixado o sistema UNIX a linguagem de programação C tornou-se uma escolha natural por vários motivos. Destacamos a possibilidade de exercer controle sobre os recursos de baixo nível, de vital importância para o desenvolvimento de inúmeras aplicações, e o oferecimento das seguintes características de programação de alto nível, que generalizam o seu uso nas mais diversas áreas:

i. economia de expressão através de um rico conjunto de operadores que permitem uma notação versátil, concisa e clara;

ii. uma hierarquia conceitualmente ilimitada de tipos de dados, derivados dos tipos fundamentais (caracteres, inteiros de diversos tamanhos e números de ponto flutuante), através do uso de apontadores, arrays, estruturas, uniões e funções;

iii. construções fundamentais de controle de fluxo necessárias para uma programação bem estruturada tais como: agrupamento de comandos; tomadas de decisão (if); laços com teste de término no início (while, for), ou no final (do); seleção de uma entre de diversas opções possíveis (switch);

iv. apontadores e habilidades para a aritmética de endereços, e a possibilidade da passagem explícita de apontadores como parâmetros, proporcionando a chamada por referência;

v. uso de funções com bastante flexibilidade: podem ser compiladas juntas ou separadamente e ativadas recursivamente; e

vi. capacidade de expressar diversos escopos de variáveis: internas a uma função; externas a elas, mas conhecidas somente dentro de um único arquivo fonte; ou completamente globais.

Além disto, devido principalmente aos fatos de C ter sido desenvolvida no UNIX e de ter servido como linguagem básica para escrever as versões atuais do sistema, é possível uma grande interface entre elas.

Para melhorar o ambiente de programação concorrente no UNIX sem perder as vantagens existentes em C, desenvolvemos a linguagem BIS. Ela substitui o conceito de programa pelo de módulo, engloba sem restrições todas as construções de C, e acrescenta outras que possibilitam a declaração e ativação de módulos, a criação e destruição de portas de comunicação, a sincronização de processos, e a comunicação entre eles por intermédio de primitivas de troca de mensagens. A sintaxe usada é compatível com a de C dando uma homogeneidade aos programas resultantes.

4.1. ESTRUTURAS DA LINGUAGEM BIS

Um módulo é a maior unidade que pode ser construída na linguagem BIS. Consiste de declarações de variáveis globais e dos procedimentos que podem compartilhar estas variáveis.

A sintaxe de um módulo é:

```
MODULE nome_módulo
{{
        variáveis globais
        procedimentos
}}
```

O nome_módulo é um identificador usado para referenciar o módulo durante as ativações de suas instâncias. Cada ativação de um módulo vai constituir um processo que, por sua vez, é a unidade de concorrência da linguagem. As variáveis globais declaradas no início de um módulo serão compartilhadas por todos os procedimentos deste módulo, e podem ser de qualquer um dos tipos básicos da linguagem C. Módulos diferentes, ou diferentes instâncias do mesmo módulo não podem compartilhar dados na memória principal. Um objeto

declarado num arquivo onde estão definidos vários módulos, porém fora das suas estruturas, é considerado global a todos os módulos subsequentes à sua declaração.

A declaração de módulos apresenta as seguintes vantagens: permite o controle de acesso a variáveis; torna facilmente visíveis os trachos de um programa que podem ser executados concorrentemente; e introduz restrições sintáticas que impõem uma estrutura ao programa.

O uso de procedimentos proporciona um meio conveniente para encapsular computações, permitindo a sua utilização sem a necessidade de se desvendar os seus interiores. Com procedimentos realmente bem projetados é possível ignorar como um trabalho é feito, sendo suficiente apenas saber o que é feito. O seu uso correto evita a complexibilidade dos grandes módulos.

A sintaxe e semântica dos procedimentos semelhantes às das funções na linguagem C(+):

```
tipo nome_do_procedimento([lista_de_argumentos])
[declaração_de_argumentos]
{
declarações_de_variáveis
comandos_do_procedimento
}
```

Em tipo declaramos o tipo do objeto a ser retornado pelo procedimento. Ele deve ser especificado em todas as chamadas a este procedimento. Quando não é informado, assume-se como sendo inteiro. O nome_do_procedimento é um identificador usado para referenciá-lo nas chamadas. A lista_de_argumentos é opcional e fornece os parâmetros formais para onde serão passados os valores dos parâmetros seguindo o nome_do_procedimento é obrigatório. Os argumentos especificados na lista deverão ser declarados imediatamente a seguir. É permitida a chamada recursiva de procedimentos, porém um procedimento só pode ser chamado por outro declarado no mesmo módulo a que ele pertence, não sendo, portanto, permitidas chamadas remotas.

As chamadas aos procedimentos também são idênticas às chamadas de funções em C:

· Usamos a notação [X] para indicar que X é opcional.

tipo nome_do_procedimento([lista_de_argumentos])

A declaração do tipo é opcional para procedimentos que retornam variáveis do tipo inteiro, e desnecessária para os procedimentos que não retornam valores.

Todo módulo deve conter obrigatoriamente um único procedimento com o nome MAIN. A chamada ao procedimento MAIN ocorrerá no momento em que uma instância do módulo é ativada. Nesta chamada são passados automaticamente dois parâmetros. O primeiro, convencionalmente chamado argc, é a quantidade de argumentos contidos no comando de ativação do módulo. O segundo é um apontador para um vetor de "strings" que contém os argumentos propriamente ditos, armazenados um em cada "string". Por convenção argv[0] sempre conterá o nome do módulo chamado, consequentemente o menor valor que argc pode assumir é 1.

Para a criação de processos através da ativação de instâncias de módulos criamos o comanedo RUN. Nele procuramos manter a vantagem do dinamismo da primitiva fork, acrescentando como características estruturais a obrigatoriedade da designação do módulo aser ativado e a possibilidade da passagem de parâmetros no momento da ativação. Com isto a chamada RUN ficou com uma sintaxe bem definida, mantendo uma grande flexibilidade ao permitir que se mude, em tempo de execução, o nome e a quantidade de instâncias ativadas, bem como a quantidade e o conteúdo dos parâmetros a serem passados. A sua sintaxe é:

id_procesos = RUN (nome_do_módulo,[end_param])

Os parâmetros endereçados por end_param são passados para o procedimento MAIN da instância récem-criada do módulo nome_do_módulo. End_param é um apontador para um vetor de "strings" onde os parâmetros estão armazenados. A rotina de armazenamento dos parâmetros estão armazenados. A rotina de armazenamento dos parâmetros estão armazenados. A rotina de armazenamento dos parâmetros fica ao encargo do programador. É retornado um inteiro usado para designar o novo processo nas chamadas de sincronização e comunicação.

A chamada

estado=HALT(id_processo)

permite a um processos (pai) destruir um processo (filho) anteriormente criado por ele através da chamada RUN. Quando a destruição é completada com sucesso é retornado o valor 0. Em caso contrário é retornado –1.

4.2. MECANISMOS DE SINCRONIZAÇÃO ENTRE PROCESSOS CONCORRENTES

Quando um processo precisa esperar pelo término de um dos seus filhos usa a chamada

estado = SYNCHRONIZE(id_processo)

O processo que ativou esta chamada é suspenso atém que o filho especificado por id_processo tenha completado sua execução. SYNCHRONIZE retorna um inteiro contendo informações sobre o filho. Se o processo pai não tem descendentes, ou nenhum deles tem a identificação id_processo, é retornado valor –1.

Esta sincronização poderia ser feita com um par SEND/RECEIVE. Porém, como o envio da mensagem ficaria ao encargo do programador, podem advir daí, por descuido, duas situações indesejáveis: ou a mensagem de término é enviada e o processo continua a existir, ou o processo é destruído antes de ser enviada a mensagem. O uso de SYNCHRONIZE evita esta situação uma vez que a informação de término gerada automaticamente pelo próprio sistema operacional.

4.3. MECANISMOS DE COMUNICAÇÃO ENTRE PROCESSOS CONCORRENTES

Na linguagem BIS a comunicação entre processos ocorre em dois níveis distintos. O primeiro corresponde à passagem de parâmetros na criação de um processo. Como mostramos na seção 4.1 ao se usar a chamada RUN para ativar uma instância de um módulo podemos ter a passagem de parâmetros para o procedimentos MAIN do processo criado. O retorno de um valor na chamada SUNCHRONIZE também corresponde a uma comunicação neste nível.

Num segundo nível temos a comunicação usando a porta de entrada que pode ser associada a cada processo.

Uma porta de entrada funciona como uma memória de armazenamento temporário de mensagens, estruturada em forma de fila, que não pode ser compartilhada para leitura. Quanto ao compartilhamento para gravação, uma porta pode se de dois tipos: geral, quando permite o compartilhamento entre vários processos, ou privada, quando o compartilhamento não é possível. Para que um processos tenha acesso a uma porta ele deve executar a chamada

descr_porta=CONNECT(nome_porta, pares, parcomp)

que retorna um descritor inteiro para referenciar a porta nos comandos de leitura e desconexão. NO parâmetro pares especificamos se o processo está criando esta porta para leitura (pares=0) ou gravação (pares=1), e no parâmetro parcomp dizemos se a porta é geral (parcomp=0), ou privada (parcomp=1). A execução do CONNECT realizará consistências do pedido de criação quanto a estas permissões de compartilhamento.

Quando um processo não precisa mais do acesso a uma porta ele pode desconectá-la. A chamada

estado=DISCON(descr_porta, partes)

é usada para esta finalidade. Ela retorna o inteiro 0 quando a operação for completada com sucesso. Se o descritor da porta não for conhecido é retornado o valor –1. Quando o parâmetro partes=1 a porta só será desconectada se estiver vazia. Caso existam mensagens pendentes de recepção será retornado o valor 1. Um valor nulo em partes indica que a porta será desconectada independentemente do seu conteúdo.

Um processo envia mensagens a uma porta conectada a ele para gravação através da chamada

q_env=SEND(descr_porta, &msg, tamanho, prioridade)

que envia a mensagem endereçada pelo apontador &msg à porta descrita por descr_porta. A quantidade de bytes a ser enviada é informada pelo parâmetro tamanho e cada mensagem é

assinalada com o inteiro armazenado na variável positiva prioridade que permite a implementação de seletividade na recepção. Esta chamada retorna um valor inteiro informando sobre o resultado da sua execução. Normalmente o valor retornado é a quantidade de bytes que foi realmente colocada na porta. Uma diferença entre esta quantidade e o valor contido em tamanho indica uma provável falha na execução. O valor –1 revela que ocorreu um erro do tipo: tamanho maior que o permitido, descritor de porta não conhecido, endereço da mensagem inválido, erro físico de gravação, etc.

A chamada RECEIVE permite ao módulo receber uma mensagem através da sua porta de entrada:

msg = RECEIVE(lim1, lim2, parbloq)

que retorna um apontador para uma mensagem lida na porta de entrada do processo. A leitura é feita normalmente obedecendo-se à ordem de chegada das mensagens à porta. Esta ordem pode ser alterada pelos critérios de recepção seletiva especificados pelas variáveis inteiras lim1 e lim2. Quando ambas forem nulas o RECEIVE retornará a primeira mensagem disponível. Quando forem positivas será retornada a primeira mensagem que obedeça à condição lim1<=prioridade<=lim2. Finalmente, quando forem negativas, retornará a mensagem cuja prioridade obedeça aos limites, pesquisada de modo que a ordem crescente das prioridades sobreponha-se à ordem cronológica de chegada. O último parâmetro diz se o RECEIVE será bloqueado (parblog=0) ou não (parblog=1). A recepção bloqueada retarda o processo enquanto não surgirem mensagens que atendam aos critérios de seletividade. Na recepção não bloqueada este atraso não ociosso: quando nenhuma mensagem corresponde às especificações é retornado um apontador nulo.

4.4. O PRÉ-COMPILADOR BIS

Os códigos fonte de módulos escritos em BIS deverão estar armazenados em arquivos cujos nomes terminam em ".b", como por exemplo prog1.b e prog2.b. Submetemos estes arquivos ao pré-compilador BIS com o comando ([+]):

BIS prog1.b prog2.b

[+] A sintaxe completa co comando BIS encontra-se no anexo I.

Inicialmente o arquivo é pré-processado realizando-se as substituições de macros, e outras operações determinadas pelos comandos de pré-processamento. O pré-compilador BIS dispões das mesmas funções que o pré-processador do compilador C, com sintaxe e semântica idênticas, diferenciando-se apenas pelo fato de que as linhas que irão se comunicarem com o pré-compilador BIS iniciarem-se com o símbolo ##. Estas linhas obedecem às mesmas regras de escopo das variáveis: as definidas fora dos módulos são globais aos módulos subsequentes, e as declaradas no interior de um módulo só tem validade dentro dele.

Em seguida o Pré-Compilador BIS traduz as estruturas do código fonte para formas conhecidas pelo compilador C, e faz algumas consistências sobre estas construções com o objetivo de detectar erros na programação dos módulos.

Para apresentarmos os resultados deste processamento utilizaremos o exemplo mostrado na Figura 15. Neste exemplo supomos que o arquivo prog1.b contém o código fonte do módulo b e de parte do módulo a, e que prog2.b contém o fonte do módulo c e do restante do módulo a. A pré-compilação conjunta destes dois arquivos irá produzir às seguintes saídas:

i. para cada arquivo fonte são gerados tantos arquivos objeto quantos forem os módulos declarados nele. Os nomes dos arquivos objeto relacionam os arquivos fonte aos módulos nele contidos, terminando com o sufixo ".o": prog1.a.0, prog1.b.o, prog2.a.o e prog2.c.o.

ii. para cada módulo é criado um arquivo executável gerado pela edição e ligação dos arquivos objeto correspondentes ao módulo. Os nomes destes arquivos são gerados apondo-se o sufixo ".o" aos nomes dos módulos: a.o, b.o e c.o.

iii. para cada arquivo fonte é gerado um arquivo contendo os nomes dos módulos declarados nele. Estes arquivos agilizam as recompilações parciais. Os seus nomes são obtidos substituindo-se o sufixo ".b" dos nomes dos arquivos fonte pelo sufixo ".B": prog1.B e prog2.B.

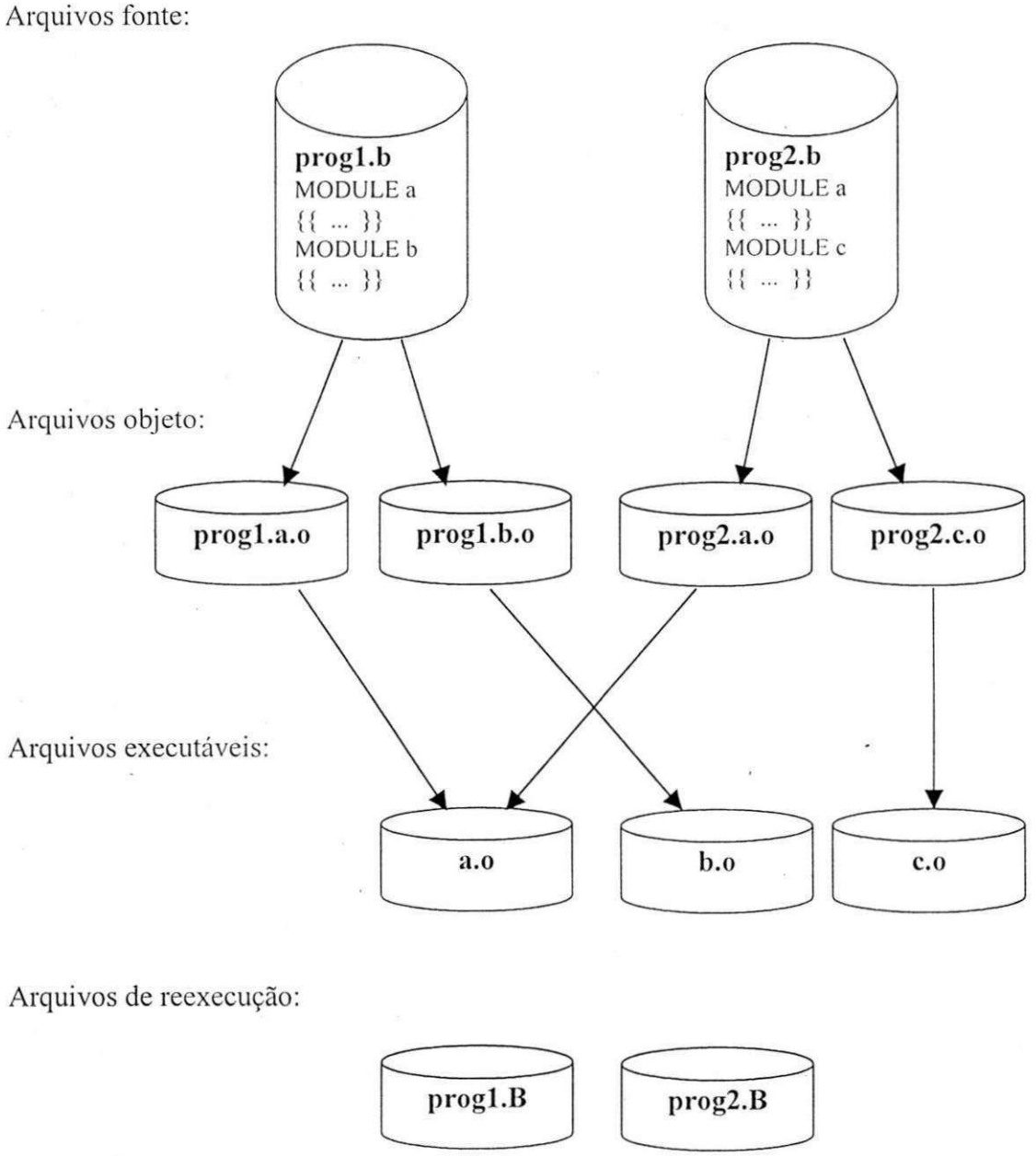

Figura 15: Exemplo de saídas geradas pela pré-compilação.

Se houver um erro na pré-compilação, digamos, em prog2.b este arquivo pode ser pré-compilado novamente, e o resultado editado e ligado com os arquivos objeto previamente gerados para o arquivo prog1.b, através do comando:

BIS prog1.B prog2.b

Os nomes dos módulos (a e b) armazenados no arquivo prog1.B permitem a geração dos nomes dos arquivos objeto já existentes (prog1.a.o e prog1.b.o) que serão editados e ligados com os arquivos resultantes de pré-compilação de prog2.b (prog2.a.o e prog2.c.o).

5. CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Neste capítulo tecemos considerações sobre a implementação do pré-compilador BIS, e fazemos referências a sua arquitetura e aos principais algoritmos e estruturas de dados envolvidos. Usamos a linguagem de programação C, nesta implementação, para tirarmos o máximo proveito das facilidades advindas da compatibilidade entre BIS, C e o UNIX. Na seção 5.1 tratamos do pré-compilador BIS e seus diversos módulos e, na seção 5.0, referimo-nos às funções incluídas na biblioteca de programação concorrente LIBBIS.

5.1 IMPLEMENTAÇÃO DO PRÉ-COMPILADOR BIS

Para uma melhor apresentação fizemos um agrupamento das principais atividades do pré-compilador em módulos. A Figura 16 mostra esta estrutura funcional e dá, entre parêntesis, uma indicação das sub-seções onde estão definidas. Nas sub-seções seguintes descrevemos cada um destes módulos isoladamente.

5.1.1 MÓDULO DE INICIALIZAÇÃO

Este módulo inicializa as variáveis e as tabelas com dados internos., abre os arquivos de uso global, e processa os par6ametros passados na chamada BIS. Os parâmetros precedidos pelo símbolo "-", que elegem as opções de pré-compilação,

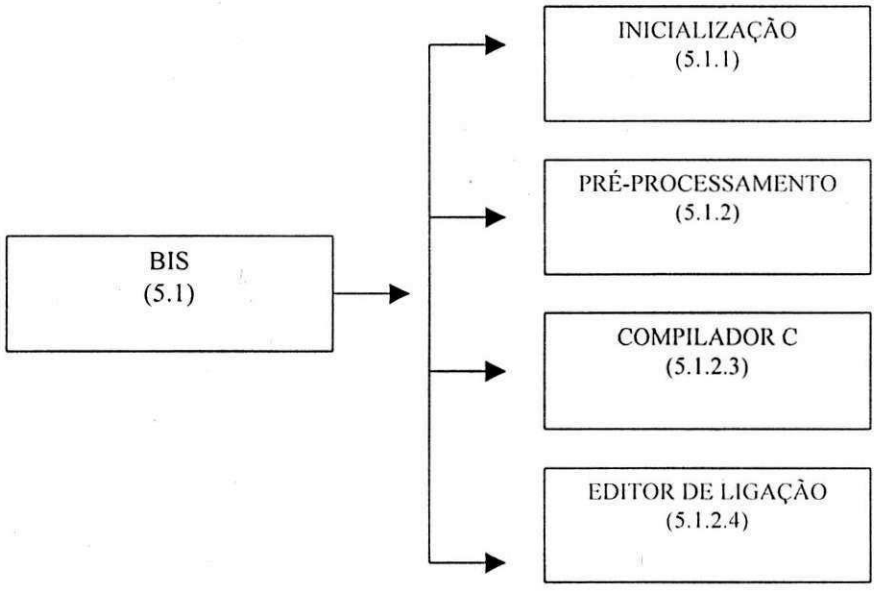

Figura 16: Estrutura funcional do pré-compilador BIS

são testados quanto à sua validade e são processados posicionando as variáveis que indicam aos demais módulos qual o processamento a ser efetuado. Os demais parâmetros são sos nomes dos arquivos a serem processados pelo pré-compilador, e são passados ao módulo seguinte.

5.1.2. MÓDULO DE PRÉ-PROCESSAMENTO

Processa, de acordo com o seu conteúdo, os arquivos cujos nomes foram passados como parâmetros da chamada. A identificação do conteúdo do arquivo é feita pelo sufixo do seu nome. Qualquer sufixo diferente dos descritos a seguir não é reconhecido, sendo geradas mensagens de erro.

5.1.2.1. PRÉ-PROCESSAMENTO DOS ARQUIVOS DE REEXECUÇÃO

Estes arquivos são identificados pelos nomes terminados em ".B". As informações contidas neles vão ser armazenadas na tabela modtab que relaciona cada módulo com os arquivos nos quais o seu código está contido. A estrutura usada para implementar esta tabela foi uma lista encadeada de nomes de módulos, onde cada elemento desta lista contém um apontador para uma lista encadeada de nomes de arquivos relacionados a este módulo.

5.1.2.2. PRÉ-PROCESSAMENTO DOS ARQUIVOS FONTE

Os arquivos cujos nomes tem o sufixo ".b" contém o código fonte dos módulos escritos na linguagem BIS. São lidos caractere por caractere, identificando-se os comandos do pré-processador, os corpos dos módulos e o s objetos de uso global.

Os comandos do pré-processador são executados no memento em que são identificados. Os nomes das macros usadas no pré-processamento são armazenados numa tabela que os relacionam com apontadores para a sua definição. Para obtermos uma melhor performance utilizamos técnicas de "hash" nas pesquisas. Implementamos o controle sobre o escopo na definição de macros usando uma tabela das definições globais que é copiada no início do processamento de cada módulo. No interior de um módulo as operações são realizadas sobre esta cópia, que é abandonada no final do módulo, quando então voltamos a trabalhar com a tabela de definições globais.

Os objetos globais aos módulos são incluídos em todos os módulos subsequentes às suas declarações. usamos um arquivos temporário para armazenar estes objetos.

Para aproveitarmos as construções de controle de acesso a variáveis e de passagem de parâmetros, e as facilidades para a programação e compilação concorrente já disponíveis, fizemos um mapeamento das estruturas de BIS para as existentes em C. Desta forma módulos correspondem a programas e procedimentos a funções.

O código correspondente a cada módulo é colocado num arquivo separado juntamente com os objetos que lhe são globais. Neste arquivo incluímos também informações que permitem ao compilador C fazer referências corretas às linhas de cada comando numerando-as de acordo com a sua posição no arquivo original. Para cada arquivo criado é gerada uma entrada na tabela modtab citada na sub-seção anterior (5.1.2.1).

O processamento de cada arquivo fonte gera também um arquivo com os nomes dos módulos que o compõem. Estes arquivos agilizam futuras reexecuções parciais da pré-compilação.

5.1.2.3. COMPILADOR C

Após cada arquivo do código fonte de um módulo ser gerado chamamos o compilador C para processá-lo, fornecendo a ele os parâmetros informados na chamada BIS que irão influenciar nesta compilação. O nome do arquivo objeto produzido é composto pelo nome do módulo antecedido do nome do arquivo fonte a que pertence, seguido pelo sufixo ".o".

5.1.2.4. EDITOR DE LIGAÇÃO

Após o processamento correto de todos os arquivos cujos nomes foram informados como parâmetros de chamada BIS percorremos a tabela modtab. Com as informações contidas nelas geramos os nomes dos arquivos objeto que deverão ser editados e ligados conjuntamente para compor o arquivo executável de cada módulo.

5.2 IMPLEMENTAÇÃO DA BIBLIOTECA DE FUNÇÕES LIBBIS

Esta biblioteca é usada para fornecer o código fonte das funções que fazem o mapeamento das primitivas de programação concorrente de BIS para o conjunto oferecido pelo UNIX. Nesta seção comentamos os aspectos mais importantes da sua implementação.

5.2.1. A FUNÇÃO RUN

O mapeamento da estrutura de cada módulo BIS para um programa em C, possibilitou implementar a sua ativação usando as primitivas fork e excv do UNIX. Se por um lado esta solução limitou a passagem de parâmetros aos do tipo "string", por outro trouxe os seguintes benefícios: uma grande flexibilidade ao permitir a programação em arquivos separados; e uma exclusividade bastante confiável no acesso a variáveis de um módulo.

5.2.2. A FUNÇÃO SYNCHRONIZE

Como na primitiva wait não podemos especificar por qual filho estamos esperando, e em SYNCHRONIZE devemos fazer esta discriminação, para implementar o mapeamento entre elas tivemos que alocar, para cada processo, uma tabela que relaciona a identificação dos seus filhos já terminados com os estados retornados pelos mesmos. A função

SYNCHRONIZE primeiramente verifica se o filho já está nesta tabela retornando, em caso positivo, o estado correspondente. Em caso contrário ela executa repetidamente as seguintes operações: bloqueia-se até receber o sinal de que um dos seus filhos terminou, inclui este filho e o seu estado na tabela, e verifica se ele era o esperado. A função permanece neste laço até que o filho esperado termina, ou o processo não tenha mais filhos, quando então retorna um erro.

5.2.3. A FUNÇÃO CONNECT

Implementamos a comunicação entre processos usando os dutos especiais desenvolvidos por Ken Thompson. Semelhantemente aos dutos comuns eles possuem estruturas que administram automaticamente as tarefas de armazenamento temporário, sincronização e escalonamento dos acessos. A diferença mais visível é que eles podem ser referenciados através dos seus nomes. Desta forma processos que não descendem de um ancestral comum podem abrir o mesmo duto usando primitivas idênticas às de criação dos arquivos normais. Esta possibilidade de compartilhamento remove a principal barreira no uso de dutos para a comunicação com processos servidores citada na sub-seção 3.2.3.3. Na nossa linguagem denominamos estes dutos de portas de comunicação.

A função CONNECT associa os processos a estes dutos, retornando descritores de arquivos que são usados para referenciar as portas nas chamadas SEND e DISCON.

5.2.4. A FUNÇÃO SEND

Antes de enviar uma mensagem a função SEND testa se a prioridade é positiva e se o tamanho informado não ultrapassa o maior valor permitido. Este valor máximo é dado pela constante simbólica MAXMEN encontrada no código fonte da biblioteca LIBBIS. Podemos alterar este valor no arquivo LIBBIS.c. Para os testes realizados neste trabalho adotamos o valor 508. Se o tamanho não excede ao limite, são colocadas na porta designada as seguintes informações: a prioridade e o tamanho da mensagem e, por fim, a mensagem propriamente dita. As mensagens são colocadas na porta obedecendo à ordem cronológica das suas criações.

5.2.5. A FUNÇÃO RECEIVE

Na implementação desta função um aspecto muito importante que tivemos de considerar foi a seleção de prioridades na recepção de mensagens, que pode fazer com que elas sejam colhidas numa seqüência diferente daquela em que foram enviadas. Isto pode acontecer quando um RECEIVE encontra uma fila de mensagens esperando ser recebidas: a fila será perscrutada em busca da primeira que atenda às especificações desejadas. Consequentemente poderá ser extraída uma mensagem que esteja localizada numa posição qualquer da fila, desobedecendo assim à ordem cronológica de sua geração.

Como as portas de comunicação são mapeadas para dutos, a implementação de um algoritmo para realizar uma busca deste tipo diretamente na fila seria potencialmente ineficiente e complexa. A ineficiência viria da obrigatoriedade de relermos as mensagens que antecedem àquelas que estão sendo selecionadas, cada vez que a função fosse chamada. Como esta leitura seria feita em discos magnéticos a degradação de tempo poderia ser inaceitável. A complexibilidade decorreria das modificações nas primitivas de leitura do duto e das operações com apontadores que fatalmente deveriam ser introduzidas. Para conferir versatilidade à recepção criamos uma lista encadeada em cada processo para armazenar as mensagens que chegam à sua porta de entrada.

A função RECEIVE administra esta lista abastecendo-a com as informações contidas na porta de entrada correspondente e atendendo aos pedidos de recepção e de seleção de mensagens.

5.2.6. AS FUNÇÕES DISCON E HALT

A implementação destas duas funções não apresentam características especiais, constando de mapeamentos triviais para chamadas ao UNIX. Apenas em DISCON, quando o parâmetro partes=1, verificamos se ainda existem mensagens na porta.

6. ALGUNS TESTES UTILIZANDO A LINGUAGEM BIS

Para uma apreciação do uso da linguagem BIS mostramos neste capítulo alguns dos problemas mais citados nos textos sobre programação concorrente. Estes exemplos foram incluídos numa pequena coleção de programas usados para a testar a implementação do pré-compilador e a biblioteca de funções.

6.1. PROBLEMA DO PRODUTOR/CONSUMIDOR /MEMÓRIA COMPARTILHADA LIMITADA

Inicialmente na figura 17 mostramos o exemplo padrão definido no capitulo 3. A solução deste problema usando BIS é bastante simplificada pelo fato dos dutos usados na implementação das portas de comunicação comportarem-se como o processo buffer deste problema: armazenam mensagens, bloqueiam a leitura/gravação quando ele está vazio/cheio. Para das uma maior destaque aos aspectos de comunicação consideramos que a mensagem é uma constante.

```
#include "funções.c"
MODULE pro {{
   char mensagem[]="...mensagem...";
   main() {
          int p_com,i;
          p_con=CONNECT("con", 1, 1);
          for ( ; ; )
                SEND(p_con, mensagem, sizeof mensagem, 0);
   }
}}
MODULE con {{
   main() {
          int i, len;
          char *m;
          CONNECT("con", 0, 1);
          for ( ; ; )
                m=RECEIVE(0, 0, 1);
   }
}}
```

Figura 17: Problemas do produtor /Consumidor/Memória Intermediária Limitada.

6.2. O JANTAR DOS FILÓSOFOS

Para a solução deste interessante problema proposto por DiJkstra usamos o módulo mesa para administrar os recursos (cadeiras) e uma instância do módulo fil para cada filósofo. As únicas atividades de um processo do tipo fil são pedir acesso à mesa através de chamadas SEND e esperar através de um RECEIVE que a concessão deste acesso lhe seja comunicado. Após obter acesso o processo espera um intervalo de tempo e pede para retirar –se da mesa, voltando a repetir o pedido inicial de acesso. Este exemplo encontra-se na figura 18.

```
#include "funcoes.c"
#define numfil 5
char *P_F[NUMFIL]={"p_f1","p-f2","p-f3","p-f4","p-f5"};
MODULE mesa {{
   main() {
         int cad[NUMFIL],p_fil[NUMFIL], msgn, p_mesa, i;
         char *msg, *arg[2];
         for (i=0; i<NUMFIL; i++) {
               p_fil[i]=CONNECT(P_F[i], 1, 1);
               sprinti(arg[0], "%s", i);
               RUN("fil", arg);
         }
      CONNECT("P_MESA", 0, 0);
      for ( ; ; ) {
         msg=RECEIVE( 0, 0, 1)
         msgn=msg[0]-"1";
         if (cad[msgn]==0 && cad(msgn-1)%NUMFIL]==0
               && cad [(msgn+1)%NUMFIL]==0{
               cad [msgn]=1;
               SEND (p_fil[msgn], "s", 2, 0);
         } else
               cad[msgn]=0;
          }
      }
      }}
      MODULE fil {{
         main(arg, argv)
         int argc;
         char *argv[];
         {
               int p_mesa, nfil;
               char *msg;
               nfil=argv[1][0]-"0";
```

```
            p_mesa=CONNECT("P_MESA", 1, 1);
            CONNECT("P_F[n_fil]", 0, 0);
            for( ; ; ) {
                msg="n";
                do{
                    SEND(p_mesa, argv[1], 2, 0;
                    msg=RECEIVE( 0, 0, 0);
                }while (msg[0]i="s");
                sleep(1);
                SEND(p_mesa, argv[1], 2, 0);
            }
        }
    }}
```

Figura 18: Problemas do "Jantar dos Filósofos".

6.3. PROBLEMA DE DEFORMAÇÃO E FORMATAÇÃO

Este problema foi descrito no inicio do capitulo 2- Na figura 19 mostramos sua implementação. Para não entrarmos em detalhes que dizem respeito a periféricos consideraremos como entrada para os registro de 80 caracteres a estrada padrão do sistema. A função getline(line, LENCARD) coloca no máximo LENCARD caracteres no string line, e retorna a quantidade de caracteres realmente lidos. Quando não existem caracteres disponíveis para leitura é retornado o valor –1.

```
#include <stdio.h>
#include "funcoes.c"
#define LENCARD 80
#define LENLINE 125
MODULE def {{ /* módulo de deformatação */
    main(){
        char line[LENCARD], null[2];
        int i, p_cod, tam;
        p_cod=CONNECT("P_COD", 1, 1);
        while ((tam=getline(line, LENCARD))>0) {
            for(i= 0; i< LENCARD; 1++)
                SEND( p_cod, &line[i], 1, 0);
        }
        sprintf(null, "%s", ' ');
        SEND(p_cod, null, 1, 0);
    }
}}
```

```
MODULE cod {{ /* módulo de cod. brancos */
        int i, p_for;
        main(){
                char *c, par[2]
                CONNECT(*P_COD, 0, 1);
                p_for = CONNECT("P_FOR", 1, 1);
                c=RECEIVE( 0, 0, 1);
                while ((c= RECEIVE(0, 0, 1)) I= eof) {
                        i = 0;
                        while ( c [0] == " " ) {
                                i++;
                                if (i=9)
                                        env();
                                c=RECEIVE( 0, 0, 1);
                        }
                        if (i1=0)
                                env();
                                SEND(p_for, c, 1, 0);
                                if (c==eof)
                                        break;
                }
        }
        env()
        {
                int c1;
                send(p_for, "", 2, 0);
                sprintf(c1, "%s", i+ "0");
                SEND(p_for, "", c1, 2, 0);
                i=0;
        }
}}

MODULE for {{ /* módulo de formatar linhas */
        main() {
                int i;
                char *c;
                CONNECT("P_FOR", 0, 1);
                I=1;
                while ((c=RECEIVE( 0, 0, 1)) I= EOF){
                        putchar(c[0]);
                        i=0;
                }
                i++;
        }
        putchar("0");
}}
```

Figura 19: Poblema de Deformatação e Formatação

6.4. EXEMPLO DO USO DE ALGUMAS MACROS DO PRE-COMPILADOR

Na figura 20 mostramos um exemplo do uso de algumas

```
##define      glob2
#include      <stdio.h>
##include     "arqx"
MODULE m_1 {{
##define      GLOB1       2
        proc_1(){
##ifdef       GLOB2
        wglob1=GLOB1;
##else
        wglob1=wglob1+GLOB1;
##endif
        }
}}
MODULE m_2 {{
##undef       GLOB2
proc_2(){
##ifdef       GLOB2
        wglob=GLOB1;
##else
        wglob1=wglob+GLOB1;
##endif
        }
}}
```

Figura 20: Exemplo do uso de algumas macros do pré-compilador

macros do pré-compilador, e do comportamento dos objetos em relação al local onde estão definidos(escopo). As linhas iniciadas com ## vão se comunicar com o processador de macros do pré-compilador. O conjunto de macros pré-definidas é idêntico ao do pré-processador do compilador C. Na figura 21 mostramos o conteúdo do arquivo arqx usado neste exemplo pela

```
##define GLOB1
int wglob1;
main(){
        wglob1=GLOB1;
        }
```

Figura 21: Conteúdo do arquivo arqx usado no exemplo do uso de algumas macros do pré-compilador.

macro ##include. O resultado do processamento de macros e do mapeamento das estruturas de BIS para C é mostrado na figura 22. São dois arquivos contendo o código fonte dos módulos m_1 e m_2 convertido para linguagem C, e o arquivo de reexecução.

código do módulo m_1

```
#line 2
#include        <stdio.h>
#line 1 "arqx"
#line 2
int vglob1;
#line 3
main(){
#line 4
vglob1= 1 ;
#line 5
}
#line 4 "ex6.5.b"
#line 6
proc_1(){
#line 8
vglob1= 2 ;
#line 12
}
```

código do módulo m_2

```
#line 2
#include        <stdio.h>
#line 1 "arqx"
#line 2
int vglob1;
#line 3
```

```
main(){
#line 4
vglob1=1 ;
#line 5
}
#line 4 "ex6.8.b"
#line 16
proc_2(){
#line 20
vglob1=vglob1+ 1 ;
#line 22
}
```

arquivo de reexecução

```
m_1
m_2
```

Figura 22: Saídas produzidas no exemplo do uso de algumas macros de pré-processador.

7. CONCLUSÃO

Neste capítulo fazemos um resumo dos principais tópicos abordados neste trabalho, tecemos comentários sobre os resultados obtidos, e apresentamos algumas sugestões. O principal objetivo desta tese foi melhorar os aspectos denotacionais do ambiente da programação concorrente no sistema operacional UNIX. Iniciamos nosso trabalho apresentando, no capítulo 2, os conceitos de processos, paralelismo e concorrência, incluindo as questões semânticas que envolvem a escolha de primitivas usadas para especificar a computação concorrente e a sincronização e/ou comunicação resultantes. Tecemos também considerações sobre várias notações propostas e mostramos a implementação em várias linguagens, do problema do produtor / consumidor / memória intermediária limitada.

No capítulo 3 justificamos a escolha do sistema operacional UNIX e fizemos uma avaliação dos aspectos denotacionais do seu ambiente de programação concorrente. Um dos principais motivos que fundamentou esta seleção foi o fato dele ser um dos mais versáteis, eficientes difundidos sistemas operacionais de tempo compartilhado, disponível para uso em mini e microcomputadores. Para demonstrar o uso de primitivas do UNIX na programação concorrente incluímos nessa avaliação diversos exemplos de programas escritos na linguagem C. Os maiores problemas encontrados estão relacionados à falta de estrutura em algumas primitivas e/ou à limitação das suas aplicações à situações específicas, obrigando-nos a usar comandos que não estão diretamente ligados à computação que queremos descrever. Por causa destas deficiências a criação de probrams corretosx e legíveis exige uma meticulosa disciplina de programação.

Fundamentando-nos nesta análise sugerimos no capítulo 4 uma maneira de melhorar os aspectos denotacionais deste ambiente. Inicialmente analisamos duas soluções básicas: uma mudança radical no núcleo do sistema para provê-lo com chamadas adequadas à programação concorrente estruturada; ou um mapeamento de primitivas mais poderosas para um conjunto já existente. Descartamos a possibilidade de modificar o núcleo devido principalmente às limitações de espaço do mesmo. Ale'm disto ele foi projetado como um grande programa

monolítico, tornando-o mais vulnerável a possíveis erros que seriam propagados de uma maneira desastrosa por todo o sistema. Para evitar estes problemas adotamos o mapeamento como solução. O código fonte destes mapeamentos encontram-se na biblioteca LIBBIS.c. O fato de não modificarmos o núcleo do sistema operacional permite o fácil transporte desta biblioteca entre instalações diferentes, sem a necessidade de recompilar o código fonte deste núcleo.

A escolha do uso de mecanismos de troca de mensagens decorreu do fato do UNIX não permitir o compartilhamento de memória([+]). Esta escolha entretanto atende a objetivos mais globais uma vez que, do ponto de vista da implementação, a troca de mensagens é mais desejável pois pode ser obtida com ou sem memória compartilhada. O mesmo não acontece com os mecanismos dependentes deste compartilhamento: eles são difíceis de implementar quando o compartilhamento não existe [17]. A troca de dados através de mensagens é conceitualmente uma idéia simples que assegura uma boa modularidade e independência entre os processos, propriedades estas que acentuam a clareza e a facilidade de manutenção [26].

A seleção das primitivas de troca de mensagens envolveu questões semânticas relativas à sincronização e à especificação e criação de canais. Quanto à sincronização usamos chamadas não bloqueadas para permitir um maior paralelismo do sistema. Colocamos a opção de bloqueio na recepção para atender às aplicações nas quais o processo receptor quer sincronizar-se com o emissor. A implementação desta sincronização usando primitivas de recepção não bloqueadas requer um esquema de espera ocupada com todas as desvantagens associadas.

O esquema mais geral para a especificação de canais é a designação global ou caixa postal ([++]), pois permite os compartilhamentos tanto para leitura como para gravação, oferecendo uma solução simples para interações do tipo múltiplos_clientes/múltiplos_servidores. Infelizmente caixas postais são difíceis de implementar num ambiente que não possua uma rede de comunicação especializada [1]. Optamos então pelo uso de portas, que são um caso especial de caixas postais onde o compartilhamento para leitura não é permitido. Apesar desta limitação elas oferecem uma solução direta para a maioria dos problemas existentes, incluindo os que envolvem interações

---

[+] Neste trabalho usamos a versão 6 do UNIX.

Não obstante todas as precauções adotadas para que o código produzido não gerasse situações indesejadas, identificamos dois casos onde elas podem aparecer. Primeiro existe a possibilidade de surgirem problemas causados por condições dependentes de velocidades de execução([+]), principalmente na função RECEIVE. A situação mais vulnerável é quando usada com a seleção de recepção que sobrepõe a ordem crescente das prioridades à ordem cronológica de chegada: enquanto selecionamos e retiramos uma mensagem da lista, uma outra, com prioridade menor porém dentro dos limites de seleção, pode estar sendo recebida pela porta. A Segunda é que, apesar da chamada CONNECT impedir o compartilhamento de portas para a leitura, um processo filho herdará aquelas já abertas pelo pai antes do momento da sua criação. Na versão atual da função CONNECT não está implementado o controle de portas gerais ou privadas.

Estudos adicionais podem ser realizados para otimizar estes resultados. Um dos mais importantes seria examinar os mapeamentos sob os critérios da eficiência de execução. Certos conflitos entre as modificações necessárias para melhorar a performance, e as limitações impostas pelo UNIX, provavelmente só poderão ser resolvidas com a introdução de algumas primitivas no núcleo. Outros aspectos a serem analisados são possíveis inclusões de novas alternativas de processamento nas chamadas como, por exemplo, permitir à chamada RECEIVE o desbloqueio ou saída por tempo([++]).

Achamos que a importância do nosso trabalho ficou demonstrada ao longo desta tese, e os objetivos propostos foram atingidos: as novas primitivas introduzidas e a linguagem desenvolvida permitem a elaboração sob o UNIX de programas concorrentes mais estruturados, mais confiáveis e de manutenção mais fácil do que os escritos em C. Os testes que realizamos corroboram esta afirmação.

[+] Também conhecidos como "race condition".

[++] Também conhecida por "time-out".

8. BIBLIOGRAFIA

1. ANDREWS, G. R. & SCHNEIDER, F. B. – Concepts and Notations for Concurrent Programming, Computing Survevs, Vol. 15, N°. 1, Mar 1983, pp. 3-42.

2. BEN-ARI, M. – Principles of Concurrent Programming, Prentice-Hall, Englewwod Cliffs, N. J., 1982.

3. BLAIR, S. B. et alii - A Practical Extension to UNIX for Interprocess Communication, Soft.-Pratice & Experience, Vol. 13, N°, 1983, pp. 45-58.

4. BRINCH HANSEN, P. – Operating Systems Pronciples, Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 1973.

5. BRINCH HANSEN, P. – Distributed Processes: A Concurrent Programming Concept, Communications of the ACM, Vol. 21, N° 11, 1978, pp. 934-941.

6. CASHIN, P. – Inter Process Communication, Bell-Northern Research, 19813.

7. CONWAY, M. E. – Design of a Separable Transition – Diagram Compiler, Communications of the ACM, Vol. 6, N° 7, Jul 1963, pp. 396-408.

8. DEITEL, H. M. – An Introduction to Operating Systems, Addison-Wesley, Reading, 1984.

9. DENNIS, J. B. & VAN HORN, E. C. – Programming Semantics for Multiprogrammed Computations, Communications of the ACM, Vol. 9, N° 3, Mar 1966, pp. 143-155.

10. DIJKSTRA, E. W. – Cooperating Sequential Processes, Programming Languages, Academic Press, New York, 1968.

11. FELDMAN, J. A. – High Level Programming for Distributed Computing, Communications of the ACM, Vol. 22, N° 6, Jun 1979, pp. 353-368.

12. GUIMARAES, C. C. – Princípios de Sistemas Operacionais, 3ª Ed. Campus, R. Janeiro, 1983.

13. HARTMANN, A. C. – A Concurrent Pascal Compiler for Minicomputers, Springer-Verlag, N. York, 1977.

14. HOARE, C. A. R. – Communicating Sequential Process, Communications of the ACM, Vol. 21, Nº 8, Aug 1978, pp 666-677.

15. KERNIGHAN, B. W. & RITCHIE, D. M. – The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, 1973.

16. KERNIGHAN, B. W. & RITCHIE, D. M. – UNIX Programming, Unix Programmer's Manual, Vol. 2, Bell Laboratories, New Jersey, 1978.

17. MAO, T. E. & YEH, R. T. – Communication Port: A Language Concept for Concurrent Programming, TEFE Trans. Soft. Eng., Vol. SE-6, Nº 2, Mar 1980, pp.

18. PEREIRA, J. C. R. & SAUVE, J. P. – Uma Avaliação dos Aspectos Denotacionais da Programação Concorrente no Sistema Operacional UNIX, RT 03/84, Grupo de Redes da UFPB, C. Grande, Jan 1984.

19. PRATT, T. W. – Programming Languages: Design e Implentation, Prentice-Hall, Englewwod Cliffs, N. J., 1975.

20. RITCHIE, D. M. – UNIX Time-Sharing System: A Retrospective, Bell System Technical Journal, 57, Jul-Aug 1978, pp. 1947-1969.

21. RITCHIE, D. M. & THOMPSON, K. - - The UNIX Time-Sharing System, Bell System Technical Journal, 57, Jul-Aug 1978, pp. 1905-1929.

22. SANTOS, S. M. – Programação Concorrente e Mecanismos de Comunicação e Sincronização de Processos, Quarta Estória de Computação, I. M. E., U. S. P., São Paulo, 1984.

23. SHATZ, S. M. – Communication Mechanisms for Programming Distributed Systems, Computer, Vol., Nº, Jun 1984, pp. 1984, pp. 21-28.

24. SILBERSHATZ, A. – Communication and Syncronization in Distributed Systems, IEEF Trans. Soft. Eng., Vol. SE-5, Nº 6, Nov 1979, pp. 542-546.

25. STANKIVC, J. A. – Software Communication Mechanisms: Procedures Calls Versus Messages, Computer, Vol. 15, Nº 4, Apr 1982, pp. 19-25.

26. THOMPSON, K. – UNIX Time-Sharing System: UNIX Implementation, Bell System Technical Journal, 57, Jul-Aug 1978, pp. 1931-1946.

ANEXO I – DESCRIÇÃO DO COMANDO BIS

Apresentamos neste anexo uma descrição concisa do uso do comando BIS. Este comando reside atualmente no diretório /u/jocarlos e pode ser chamado diretamente por qualquer usuário.

Sinopse:

BIS [opções] ... arquivos ...

Descrição:

BIS é o pré-compilador da linguagem de programação BIS. Ele aceita diversos tipos de argumentos na sua chamada. O sinal "-" deve preceder cada um dos argumentos que informam sobre as opções de pré-compilação. Basicamente estes argumentos são os mesmos usados no comando cc, além do seguinte, específico do comando BIS:

-b Indica que os arquivos intermediários temporários com o código fonte na linguagem C (correspondente a cada módulo contido nos arquivos fonte na linguagem BIS) não serão removidos no final da pré-compilação.

Outros argumentos, que normalmente seguem as opções, são os nomes dos arquivos. O pré-compilador decide qual o processamento a ser realizado sobre estes arquivos verificando os sufixos dos seus nomes. Um argumento terminando em ".b" é assumido como sendo o nome de um arquivo que contém código fonte escrito na linguagem BIS. Este arquivo será pré-compilado gerando as seguintes saídas: um arquivo contendo informações usadas em reexecuções parciais da pré-compilação; um arquivo de código objeto para cada módulo declarado no arquivo fonte; e, na saída padrão, uma relação dos erros encontrados na pré-compilação. O nome de um arquivo de reexecução é obtido trocando-se, no nome do arquivo fonte, o sufixo ".b" por ".B". Obtém-se o nome de um arquivo objeto trocando-se a letra "b", do sufixo do nome do arquivo fonte, pelo nome do módulo correspondente seguido do sufixo ".o".

Em uma reexecução parcial da pré-compilação os arquivos de reexecução podem ser informados como argumentos na ativação do comando BIS. As informações neles contidas vão informar ao editor de ligação quais os arquivos objeto correspondentes aos arquivos fonte já pré-compilados corretamente.

ARQUIVOS:

Além daqueles usados pelos comandos cc e ld, são envolvidos numa pré-compilação os seguintes arquivos:

Nome Arquivo	Especificação do Arquivo
arquivo.b	arquivo fonte
arquivo.módulo.c	arquivo temporário usado por BIS
arquivo.módulo.o	arquivo objeto
módulo.c	arquivo executável
arquivo.B	arquivo de reexecução
/tmp/ctm?	arquivo temporário usado por BIS

DIAGNÓSTICOS:

As mensagens de diagnóstico produzidas por BIS são facilmente entendíveis. Os arquivos intermediários temporários contendo código na linguagem C são gerados de modo que as mensagens de erro emitidas pelo compilador C refiram-se corretamente à linha em que o comando errado encontra-se no arquivo fonte original.

ANEXO II - LISTAGEM DO CÓDIGO FONTE DO PRÉ-COMPILADOR BIS

```
1    :/*
2    : * bis -- Bis macro pre-processor
3    : */
4    :
5    :#include <stdio.h>
6    :#include <sys/types.h>
7    :#include <sys/mode.h>
8    :#include <sys/stat.h>
9    :#include <sys/sig.h>
10   :
11   :#define PMODE  644
12   :#define NP 30
13   :#define NDIUE  20
14   :#define NFILES 50
15   :#define SBSIZE 10000
16   :#define NCPS   8
17   :#define MAXINC NFILES+10
18   :#define LNSIZE 1000
19   :#define SYMSIZ 400
20   :#define FORGET 0
21   :#define DEFINE 1
22   :#define INCLUDE    2
23   :#define ENDIF  3
24   :#define IFDEF  4
25   :#define IFNDEF 5
26   :#define LINE   6
27   :#define ELSE   7
28   :#define UNDEF  8
29   :#define IF 9
30   :#define OTHER  10
31   :#define MODULE 11
32   :
33   :char   *ilist1[10];
34   :char   **ilistp = ilist1;
35   :char   *ilist2[ ] = {
36   :   "/usr/source/include",
37   :   "/compool",
38   :   0,
39   :};
40   :char *pref = "/lib/crt0.o";
41   :char *preff = "/lib/fcrt0.o";
42   :char *prefi = "/lib/fjcrt0.o";
43   :char *prefj = "/lib/fjcrt0.o";
44   :char *prefp = "/lib/mcrt0.o";
45   :char *pref2 = "/lib/crt2.o";
46   :char *pref20 = "/lib/crt20.o";
```

```
47   :
48   :struct modtab {
49   :    char *modnam;
50   :    struct modtab *pmtab;
51   :    struct flist *pmtab;
52   :} * hdtab;
53   :
54   :struct flist {
55   :    char *filnam;
56   :    struct flist *pflist;
57   :} ;
58   :
59   :struct symtab {
60   :    char    name[NCPS+1];
61   :    char    keyv;
62   :    char    *value;
63   :} stab0[SYMSIZ], stab1[SYMSIZ], *tabsym = stab0;
64   :
65   :struct prep {
66   :    char    *p_name;
67   :    char    p_val;
68   : preplist[ ] = {
69        "define",       DEFINE,
70        "include",      INCLUDE,
71        "endif",        ENDIF,
72        "ifdef",        IFDEF,
73        "ifndef",       IFNDEF,
74        "line",     LINE,
75        "else",     ELSE,
76        "undef",        UNDEF,
77        "if",       IF,
78        "MODULE",   MODULE,
79        "unix",     OTHER,
80        0,
81   :};
82   :
83   : struct include {
84   :    FILE    *file;
85   :    int lineno;
86   :    char    *fnames;
87   :} incl[MAXINC],    *inclev;
88   :
89   : int   modname;
90   : int   keyv;
91   : int   ndiue;
92   : int   chalevel;
93   : int   modlevel;
94   : int   cpargc = 2;
95   : int   instring;
96   : int   depth;
97   : FILE  *obuf;
98   : FILE  * 0buf;
99   : FILE  * tbuf;
100  : char  * fnam;
101  : char  line[LNSIZE];
```

```
102  : char  * modnam;
103  : char  * lp;
104  : int   lineno[MAXINC];
105  : int   extail;
106  : int   Eflag, Fflag, Pflag, bflag, cflag, eflag, fflag,
             iflag;
107  : int   trulvl;
108  : int   flslvl;
109  : char  *cpargv[NP+3] = {"cc", "-c"};
110  : char  *tname = "/tmp/ctmXXXXXX";
111  : char  sbf[SBSIZE];
112  : char  *stringbuf = sbf;
113  : char  pushbuff[500];
114  : char  *pushp = pushbuff;
115  : char  *calloc( );
116  : char  *strsave( );
117  : char  *mktemp ( );
118  : int   ndiue;
119  : char  *list[3*NFILES];
120  : char  fname[60], oname[60], *onam;
121  : char  pushbuf[LNSIZE], *pushp;
122  : char  but[60];
123  :
124  :define ungetc(c) (*++pushp = (c))
125  :
126  :main(argc, argv)
127  :char *argv[ ];
128  :{
129  :   register struct prep *pp;
130  :   register c;
131  :   register char *rlp;
132  :   struct symtab *np;
133  :   struct modtab *pmtab;
134  :   struct flist *plist;;
135  :   char *t;
136  :   int f20, dexit( ), i,j;
137  :   char *av[NFILES+3];
138  :   char *av1[7];
139  :
140  :#if DO
141  :   printf("\nComecei main %d\n", argc);
142  :#endif
143  :   setbuf(stdout, NULL);
144  :   inclev = incl-1;
145  :   for (pp = preplist; pp ->p_name; pp++)
146  :       insym(pp->p_name, pp->p_val);
147  :   for (; argc > 1 && argv[1][0]=='-'; argc--, argv++)
148  :       switch (argv[1][1]) {
149  :       case '0':
150  :           if (argc > 2) {
151  :               if ((t = getsuf(argv[2])) == 'c' ||
152  :                    t == 'o' || t == '0')
153  :                   terror("Wold overwhite %s", argv[2]);
154  :               guard(argv[1]);
155  :               guard(argv[2]);
```

```
156 :               argc--; argv++;
157 :           }
158 :           break;
159 :       case 'E':
160 :           Eflag++;
161 :           gdiue(argv[1]);
162 :       case 'p':
163 :           Pflag++;
164 :       case 'c':
165 :           cflag++;
166 :           break;
167 :       case 'b':
168 :           bflag++;
169 :           break;
170 :
171 :       case 'U':
172 :           np = lookup(argv[1]+2, -1);
173 :           if (np->name[0])
174 :               np->keyv = FORGET;
175 :           gdiue(argv[1]);
176 :           break;
177 :
178 :   case 'D':
179 :       for (rlp = argv[1]+2; *rlp; rlp++)
180 :           if (*rlp == '=')
181 :               break;
182 :       if (rlp == argv[1]+2)
183 :           break;
184 :       if (*rlp == '=')
185 :           *rlp++ = 0;
186 :       insym(argv[1]+2, OTHER);
187 :       if (!rlp[-1]) {
188 :           np= lookup(argv[1]+2, 1);
189 :           np->value = rlp;
190 :       }
191 :       gdiue(argv[1]);
192 :       break;
193 :
194 :   case 'I':
195 :       *ilistp++ = argv[1]+2;
196 :       gdiue(argv[1]);
197 :       break;
198 :
199 :   case '2':
200 :       if(argv[1][2] = = '\0')
201 :           pref = pref2;
202 :
203 :       else {
204 :           pref = pref20;
205 :           f20 = 1;
206 :       }
207 :       break;
208 :   case 'f':
209 :       fflag ++;
210 :       if(argv[1][2] = = 'i')
```

```
211  :            iflag++;
212  :        pref = prefj;
213  :        guard(argv[1]);
214  :        break;
215  :    case 'F':
216  :        Fflag++;
217  :        pref = preff;
218  :        if(argv[1][2] == 'i')
219  :            iflag++;
220  :        guard(argv[1]);
221  :        break;
222  :    case 'i':
223  :        iflag++;
224  :        guard(argv[1]);
225  :        break;
226  :    case 'p':
227  :        pref = prefp;
228  :        guard(argv[1]);
229  :        break;
230  :    case 'B':
231  :    case 'O':
232  :    case 't':
233  :    case 'S':
234  :    guard(argv[1]);
235  :    break;
236  :
237  :    default:
238  :        error(Unlnow flag %s, argv[1]);
239  :    }
240  :
241  :    cpargv[cpargc+1] = 0;
242  :    if(iflag)
243  :        if(Fflag) {
244  :            fflag = 0;
245  :            pref = prefi;
246  :        } else
247  :            iflag = 0;
248  :
249  :    if((SIGINT, SIG_IGN) != SIG_IGN)
250  :        SIGNAL(sigint, &dexit);
251  :    if(!mktemp(tname)[0])
252  :        terror("Can't create temp file");
253  :    if (argc <= 1)
254  :        include(NULL, 0);
255  :    else
256  :        for(; argc>1; argc--, argv++) {
257  :            rlp = argv[1];
258  :            if((c=getsuf(rlp)) == 'b') {
259  :                include(rlp, 0);
260  :                exfail = 0;
261  :            if(tbuf = fopen(tnane, "w") == NULL)
262  :                error("Can't create temp file");
263  :            fnam = setsuf(rlp, ' ');
264  :            setini ( )
265  :            if((Obuf=fopen((t=setsuf(rlp, '0')), "w"))
```

```
266  :              == NULL)
267  :                  error("Can't create %s", t);
268  :              while(getline ( )) {
269  :                  putline ( );
270  :              }
271  :              putline ( );
272  :              if(chalevel)
273  :                  error("Unbalanced { { } }");
274  :              if(modlevel)
275  :                  error("Module incomplete");
276  :              bclose( );
277  :          } else if (c == '0') {
278  :              if((obuf = fopen(rlp, "r")) == NULL)
279  :                  error("Can't open %s", rlp);
280  :              fnam = setsuf(rlp, ' ');
281  :              while(fscanf(obuf, "%s", buf) != EOF)
282  :                  incmod(buf, fnan);
283  :              fclose(obuf);
284  :          } else
285  :              error("Unknow sufix %s", rlp);
286  :          }
287  :
288  :
289  :          if(hdtab && !cflag && !exfail) {
290  :              av[0] = "ld";

291  :              av[1] = "-x";
292  :              av[2] = pref;
293  :              av[3] = "-v";
294  :              if(f20)
295  :                  av1[i++] = "-12";
296  :              if(iflag)
297  :                  av1[i++] = "-lfi";
298  :              else if(fflag)
299  :                  av1[i++] = "-lfj";
300  :              av1[i++] = "-ls";
301  :              if(iflag)
302  :                  av1[i++] = "-lfi";
303  :              else if(fflag)
304  :                  av1[i++] = "-lfj";
305  :              av1[i++] = "-lc";
306  :              av1[i++] = "-l";
307  :              av1[i++] = "\0";
308  :              for(pmtab=httab; pmtap; pmtap = pmtab ->pmtab)
{
309  :                  i = 4;
310  :                  for (pflist=pmtab->pflist; pflist;
                           pflist=pflist->pflist)
311  :                      sprintf(av[i++], "%s.%s.o", p flist-
                             >filnam,
312  :                              pmtab->modnam);
313  :                  j = 0;
314  :                  do {
315  :                      av[i++] = av1[j];
316  :                  } while (av1[j++]);
```

```
317  :                exfail | = callsys("bin/ld", av);
318  :            }
319  : if NLOAD | | D1 || DO
320  :    } else
321  :        error("Load not realized");
322  :#else
323  :    }
324  :#endif
325  :    dexit( );
326  :}
327  :
328  :getline( )
329  :{
330  :    register int c, sc, state;
331  :    struct symtab *np;
332  :    char *namep *filname, *cp, *rlp;
333  :    int search, t, n, r, status;
334  :
335  :#if DO
336  :    printf("\nComecei getline\n");
337  :#endif
338  :    lp = line;
339  :    *lp = '\0';
340  :    state = 0;
341  :    if ((c = getch( )) == '#') {
342  :        sch (c);
343  :        if ((c = getch( )) == '#') {
344  :            state = 1;
345  :            sch(c);
346  :            c = getch( );
347  :        }
348  :    }
349  :    while (c!='\n'&& c!='\0') {
350  :        if (letter(c)) {
351  :            namep = lp;
352  :            sch(c);
353  :            while (letnum(c = gatch( )))
354  :                sch(c);
355  :            sch('\0');
356  :            lp--;
357  :            if (state > 3 && state < 10) {
358  :                if (!flslvl &&
359  :                   (state + (lookup(nanep, -1) -> keyv
                          ==FORGET))==5)
360  :                    trulvl++;
361  :                else
362  :                    flslvl++;
363  :        out:
364  :                while (c!='\n' && c!= '\0')
365  :                    c = getch ( );
366  :                return(c);
367  :            }
368  :            if (state!=2 | | !flslvl) {
369  :                unget(c);
370  :                np = lookup(namep, state);
```

```
371  :                keyv = np->keyv;
372  :                c= getch( );
373  :            }
374  :            if(modlevel && ! state) {
375  :                if(modname)
376  :                    error("MODULE syntax");
377  :                else {
378  :                    lp = namep;
379  :                    sch('\0');
380  :                    modname = 1;
381  :                    if(0buf)
382  :                        fprintf(0buf, "%s", namep);
383  :                    sprintf(oname, "%s.%s.c", fnan,namep);
384  :                    if((obuf=fopen(oname,"w")) == NULL)
385  :                        terror(Can't create %s", oname);
386  :                    incmod(namep, fnam);
387  :                    cpargv[cpargc] = oname;
388  :                    closop("r");
389  :                    if(tbut)
390  :                        while((c = getc(tbuf)) != EOF)
391  :                            ptc(c);
392  :                        closop("a");
393  :                    }
394  :            } else if(!state && keyv==MODULE && !FLSLVL){
395  :                if(CHALEVEL | | MODLEVEL)
396  :                    if(chalevel)
397  :                        error("Unbalanced{{}}");
398  :                    if(modlevel)
399  :                        error("Nested modules");
400  :                    {
401  :                else
402  :                    for(t=0; t<SYMSIZ; t++) {
403  :                        strcpy(stabl[t].name,
404  :                            stab0[t].nane);
405  :                        stab1[t].keyv=stab0[t].<eyv;
406  :                        stab1[t].value=stab0[t].value;
407 :      modlevel = 1;
408 :      modname = 0;
409 :      lp = namep;
410 :      sch("\0");
411 :      tabsym = stab1;
412 :   }
413 :} else if (state == 10) {
414 :  if (!flslvl && np->kevy == OTHER)
415 :      np->kevy = FORGET;
416 :  goto out;
417 :} else if (state == 1) {
418 :switch (kevy) {
419 :case DEFINE: state = 2; break ;
420 :case INCLUDE: state = 3; break ;
421 :case IFNDEF: state = 4; break ;
422 :case IFDEF: state = 5; break ;
423 :case UNDEF: state = 6; break ;
424 :case endif:
425 :  if (flslvl)
```

```
426 :     flslvl--;
427 :  else if (trulvl)
428 :     trulvl--;
429 :  else
430 :     errback("if-less endif")
431 :  goto out;
432 :case ELSE:
433 :  if (trulvl) {
434 :     trulvl--;
435 :     flslvl++;
436 :  } else if (flslvl) {
437 :     if (--flslvl==0)
438 :     trulvl++; else
439 :     flslvl++;
440 :  } else
441 :     errback("if-less else")
442 :  goto out;
443 :case IF:
444 :  if (!flslvl && yyparse())
445 :     trulvl++;
446 :  else
447 :     flslvl++;
448 :  return("\n");
449 :case LINE;
450 :  fputs("#line ");
451 :  ip=line;
452 :  for(; c !="\n" && c != "\0"; c=getch())
453 :    if (!Pflag || Eflag)
454 :        shc(c);
455 :  shc("\0");
456 :  return(c);
457 :default:
458 :errback("Undefined control");
459 :while (c!="\n" && c!="\0")
460 :  c= getch();
461 :return (c) ;
462 :}
463 :} else if (state==2){
464 :if (flslvl)
465 :        goto out;
466 :     np->value = stringgbuf;
467 :     np->kevy = OTHER;
468 :     for(; c!="\n" && C!=0; c = tegch()){
469 :        if (c == "\\" )
470 :            if ((C = getch)) ! = "\n"){
471 :               ungetec( c );
472 :               c = "\\";
473 :            else
474 :               c = "";
475 :        savch( C );
476 :     }
477 :     savch("\0");
478 :     return(1);
479 :  }
480   continue
```

```
481 :} else if ((cs=c)=="\"" || sc=="'" || (state==3 &&
                sc=="<")){
482 :   sch(sc)
483 :   filname = ip;
484 :   search = 1;
485 :   if (sc == "<"){
486 :      sc="<";
487 :      search = -1;
488 :   }
489 :   instring++;
490 :   while ((C = getch())!=sc && c!="\n" && c!="\0"){
491 :      sch(c);
492 :      if (C=="||")
493 :          shc(getch());
494 :   }
495 :   instring = 0;
496 :   if (flslvl)
497 :      goto out;
498 :   if (state==3){
499 :      if (flslvl)
500 :          goto out;
501 :      *lp = "\0";
502 :      while ((C=getch())!="\n" && c!="\0")
503 :          ;
504 :      include(filname, search);
505 :      return(c);
506 :   }
507 :} else if(c=="{" && !flslvl){
508 :   if((C = getch ()) == "{"){
509 :      if(chalevel || !modlevel || !modname){
510 :          if(chalevel)
511 :              error("nested {{ }}");
512 :          if(!modlevel)
513 :              error("Missino MODULE");
514 :          if (!modlevel)
515 :              error("Missino MODULE name");
516 :          }
517 :      } else
518 :          c= getch();
519 :      chalevel = 1;
520 :   { else
521 :      sch("{");
522 :      continue
523 :      } else if(!flslvl && c == "}"){
524 :          if ((C == getch())=="}"){
525 :              if(!chalevel)
526 :                 error("Unbalancede {{ }}");
527 :              else if(modlevel && modname){
528 :                 sch("\0");
529 :                 if(state > 1)
530 :                  errback("control syntax");
531 :                 putline();
532 :                 exfail i= callsys("bin/cc", cpargv);
533 :                 if(!bflag && obut ! = stdout &&
534 :                  obuf != tbuf)
```

```
535 :                 unlink(cname)
536 :                modinit();
537 :                c = getch();
538 :             }
539 :         } else
540 :             sch("}");
541 :         continue
542 :     }
543 :     shc( c );
544 :     c = getch();
545 :  }
546 :  sch("\0");
547 :  if (state>1)
548 :     errback("Control Systax");
549 :  return(c);
550 :}
551 :
552 :insym(named, val)
553 :char *named;
554 :}
555 :  register struct symtab *np;
556 :
557 :#if D
558 :  printf("\nComecei insym com named = %s e val = %d\n",
          named, val");
559 :#endif
560 :  np = lookup(namep, 1);
561 :  np->value = named = namep
562 :  if (np->kevy == FORGET)
563 :     np->kevy = val;
564 :}
565 :
566 :error(s, x)
567 :char *s;
568 :{
569 :  FILE *ef;
570 :
571 :  ef = Eflag? Stderr: stdou;
572 :  if (inclev > incl - 1)
573 :     fprintf(ef, "%s: %d: ", inclev->fnames, inclev-
             >lineno):
574 :  fprintf(ef, s, x);
575 :  putc("\n", ef;)
576 :  exfail++;
577 :}
578 :
579 :errback("s, x");
580 :char *s;
581 :{
582 : #if D0
583 :     printf("\nComecei  errback  com s = s% e x = %d\n", s,
             x);
584 :#endif
585 :     inclev->lineno--;
586 :     error( s, x);
```

```
587 :     inclev->lineno++;
588 : }
589 :
590 : sch(c)
591 : {
592 :     register  char  *rlp;
593 :
594 : #if D0
595 :     printf("\nComecei  sch  com  c = %d\n", c);
596 : #endif
597 :     rlp = lp;
598 :     if (rlp == line+LNSIZE-2)
599 :         error("Line overflow");
600 :     *rlp++  =  c;
601 :     if (rlp>line+LNESIZE-1)
602 :         rlp = line+LNSIZE-1;
603 :     lp = rlp;
604 : }
605 :
606 : savch(c)
607: {
608 : #if D0
609 :     printf("\nComecei  savch  com  c = %d\n", c);
610 : #endif
611 :     *stringbuff++ = c;
612 :     if (stringbuf-sbf < SBSIZE)
613 :         return;
614 :    error("Too much definning");
615 : exit(1);
616 : }
617 :
618 : getch()
619 : {
620 :     register  int c;
621 :
622 : #if D
623 :     printf("\nComecei  getch  \n");
624 : #endif
625 : loop:
626 :     if (( c=getc1())=='/'&& !instring) {
627 :         if ((c=getc1( ))!='*')  {
628 :          ungetc(c);
629 :          return('/');
630 :         }
631 :         for ( ; ; )  {
632 :          c = getc1();
633 :         cloop:
634 :          switch (c)  {
635 :
636 :          case '\0':
637 :              return('\0');
638 :
639 :          case '*':
640 :              if ((c=getc1(b))=='/')
641 :                  goto loop;
```

```
642 :            goto loop;
643 :
644 :         case '\n':
645 :            ptc('\n');
646 :            continue;
647 :         }
648 :        }
649 :     }
650 :     return(c);
651 : }
652 :
653 : getc1( )
654 : {
655 :     register c;
656 :     char  intbuf[20];
657 : #if D
658 :     printf("\nComecei  getc1\n"0;
659 : #endif
660 :
661 :     if (*push != 0)
662 :         return(*pushp--);
663 :     depth=0;
664 :     while ((c=getc(inclev->file))==EOF && inclev > incl) {
665 :         fclose(inclev->file);
666 :         inclev--;
667 :  fprintf(obuf, ?"\n#  %d  \"%s\"\n", inclev->lineno,
          inclev->fnames);
668 :     }
669 :     if (c == EOF)
670 :         return(0);
671 :     if (c == '\n')
672 :         inclev->lineno++;
673 :     if (c <= 0  ||  c >= 0200)  {
674 :         error(Illegal character %0, replaced by blank",
              c);
675 :         c = '  ';
676 :     }
677 :     return(c);
768 : }
679 :
680 : lookup( namep, enterf)
681 : char *namep;
682 : {
683 :     register  char  *np, *snp;
684 :     register  strucdbf  symtab  *sp;
685 :     int  I, c, around;
686 :
687 : #if D
688 :  printf("\nComecei lookup com namep = %s e enterf =
         %d\n", namep, enterf);
689 : #endif
690 :     np = namep;
691 :     around = i = 0;
692 :     while (c = *np++)
693 :         I += c;
```

```
694 :     i %= SYMSIZ;
695 :     sp = tabsym + I;
696 :     while (sp->name[0])  {
697 :         snp = sp;
698 :         np = namep;
699 :         while (*snp++ == *np)
700 :         if (!snp++  ||  np ==nemp+NCPS){
701 :             if (!enterf && sp->keyv!=FORGET)
702 :                 subst(namep, sp);
703 :             return(sp);
704 :         }
705 :         if(++sp >= tabsym + SYMSIZ)
706 :         if (around++) {
707 :             error("too many defines");
708 :             exit(1);
709 :         } else
710 :             sp = tabsym;
711 :     }
712 :     if (enterf > 0) {
713 :         snp = namep;
714 :         for ( np = &sp->name[0]; np < &sp->name[NCPS];)
715 :         if (*np++ = *snp)
716 :             snp++;
717 :     }
718 :     return(sp);
719 : }
720 :
721 : char revbuf[200], *bp;
722 :
723 : backsch(c)
724 : {
725 : #if D0
726 :     printf("\nComecei backsch com c = %d\n", c);
727 : #endif
728 :     if (bp->revbuff > sizeof revbuff)
729 :         error("Excessive define looping", bp--);
730 :     *bp++ = c;
731 : }
732 :
733 : subst( np, sp)
734 : char *np;
735 : struc symtab *sp;
736 : {
737 :     register char *vp;
738 :     int callf;
739 :
740 : #if D0
741 :     printf("\nComecei subst com np = %s, sp = %d\n",  np,
          sp);
742 : #endif
743 :     lp = np;
744 :     bp = revbuff;
745 :     if (depth++  >  100)  {
746 :         error("define recursion loop on %s", np);
747 :         return;
```

```
748 :    }
749 :    if (!(vp = sp->value))
750 :        return;
751 :    /* arrange that define unix still has
752 :        no effect avoinding rescanning    */
753 :    for (callf = *vp == '('; isspace( *vp);)
754 :        vp++;
755 :    if (!strcmp( sp->name, vp)) {
756 :        while (*vp)
757 :         sch(*vp++);
758 :        return;
759 :    }
760 :    backsch(' ');
761:   if (callf)
762 :        expdef(vp);
763 :    else
764 :        while (*vp)
765 :         backsch(*vp++);
766 :    backsch(' ');
767 :    while (bp > revbuff)
768 :        ungetc(*--bp);
769 :}
770 :
771 :#define PROTLIM  600
772 :
773 :expdef( proto)
774 :char *proto;
775 :{
776 :    char buffer[PROTLIM], *parg[20], *pval[20], name[20],
                *cspace, *wp;
777 :    char protcop[PROTLIM], *pr;
778 :    int narg, k, i, c, inquote;
779 :
780 :#if D0
781 :    printf("\nComecei expdef com proto = %s\n", proto);
782 :#endif
783 :    for (pr = protcop; *pr++ = *proto++; )
784 :        if (pr >= protcop+PROTLIM) {
785 :         error("define prototype too long");
786 :         return;
787 :        }
788 :    proto = protcop;
789 :    for (narg = 0; (parg[narg] = token(&proto)) != 0;
              narg++)
790 :        ;
791 :    /* now scan input */
792 :    cspace = buffer;
793 :    while ((c=getch()) == ' ' ||  c == '\t')
794 :        ;
795 :    if (c != '(' ) {
796 :        error("defined function requires arguments");
797 :        return;
798 :    }
799 :    ungetc(c);
```

```
800 :     for(k=0; pval[k] = coptok(&cspace, buffer+sizeof
                buffer-8); k++)
801 :         ;
802 :     if (k != narg)  {
803 :         error("define argument mismatch");
804 :         return;
805 :     }
806 :     inquote = 0;
807 :     while (c = *proto++) {
808 :         if (inquote || !letter(c)) {
809 :         if (c == inquote || c == '\\' &&  *proto ==
                  inquote)
810 :             inquote = 0;
811 :         else if( !inquote && (c == '"' || c == '\''))
812 :             inquote = c;
813 :         backsch(c);
814 :         } else {
815 :         wp = name;
816 :         *wp++ = c;
817 :         while (letnum(*proto))
818 :             *wp++ = *proto));
819 :         *wp = 0;
820 :         for (k=0; k < narg; k++)
821 :             if(!strcmp(name,parg[k]))
822 :                 break;
823 :         wp = k < narg? Pval[k]: name;
824 :         while (*wp)
825 :             backsch( *wp++);
826 :         }
827 :     }
828 : }
829 :
830 : token(cpp)
831 : register char **cpp;
832 : {
833 :     register char *val;
834 :     register stc;
835 :
836 : #if D0
837 :     printf("\nComecei  token com cpp = %s\n", token);
838 : #endif
839 :     stc = **cpp;
840 :     *(*cpp)++ = '\0';
841 :     if (stc == ')')
842 :         return(0);
843 :     while (**cpp == ' ')
844 :         (*cop)++;
845 :     for (val = *cpp; (stc = **cpp) != ','&& stc != ')';
              (*cpp)++)
846 :         if (!letnum(stc) || (val==*cpp && !letter(stc))) {
847 :         error("define prototype argument error");
848 :         return(0);
849 :         }
850 :     return(val);
851 : }
```

```
852 :
853 : coptok(cpp, ep)
854 : register char **cpp;
855 : char *ep;
856 : {
857 :     char *val;
858 :     register stc, stop;
859 :     int paren;
860 :
861 : #if D0
862 :     printf("\nComecei coptok com cpp = %s e ep = %d\n",
            *cpp, ep);
863 : #endif
864 :     paren = 0;
865 :     val = *cpp;
866 :     if (getch( ) == ')' )
867 :         return(0);
868 :     while (((stc = getch( )) != ','&& stc != ')' ) ||
                  paren > 0) {
869 :         if (*cpp > ep) {
870 :         error("macro argument string too long");
871 :         val = 0;
872 :         break;
873 :         }
874 :         if (stc == '\0') {
875 :         error("non terminated macro call");
876 :         val = 0;
877 :         break;
878 :         }
879 :         if (stc == '"' ||  stc == '\'')  {
880 :         stop = stc;
881 :         *(*cpp)++  = stc;
882 :         while ((stc = getch( )) != stop)  {
883 :             if (stc == "\n" || stc == '\0') {
884 :                 error("non-terminated string");
885 :                 break;
886 ;             }
887 :             if (stc == '\\') {
888 :                 stc = getch( );
889 :                 *(*cpp)++ = '\\";
890 :             }
891 :             *(*cpp)++ = stc;
892 :         }
893 :         *(*cpp)++ = stop;
894 :         } else if (stc == '\\') {
895 :         stc = getch( );
896 :         if (stc != ','&& stc != '"'&& stc != '\\')
897 :             *(*cpp)++ = '\';
898 :         *(*cpp)++ = stc;
899 :         } else {
900 :         *(*cpp)++ = sct;
901 :         if (stc == '(')
902 :             paren++;
903 :         else if (stc == ')' )
904 :             paren--;
```

```
905 :         }
906 :     }
907 :     *(*cpp)++ = 0;
908 :     ungetc(stc);
909 :     return(val);
910 : }
911 :
912 : letter(c)
913 : register c;
914 : {
915 : #if D
916 : printf("\nComecei letter com c = %c\n", c);
917 : #endif
918 :     return( ( c >= 'a' && c <= 'z') ||
919 :         ( c >= 'A' && c <= 'Z') ||
920 :         ( c == '_'));
921 : }
922 :
923 : letnum(c)
924 : register c;
925 : {
926 : #if D
927 :     printf("\nComecei letnum com c = %c\n", c);
928 : #endif
929 :   return(letter( c ) || c>="0" && c<="9");
930 : }
931 :
932 : fputs(s)
933 : register char *s;
934 : {
935 :   register c;
936 :
937 : #if D
938 :   printf("\nComecei fputs com s = %s\n, s);")
939 : #endif
940 :   if (P flag && !Eflag)
941 :     return;
942 :   while (c = *s++)
943 :     ptc( C );
944 : }
945 :
946 : char *
947 : strsave(s)
948 : register char *s;
949 : {
950 :   register i;
951 :   register char *cp;
952 :   char *scp;
953 :
954 : #if DO
955 :   printf("\nComecei strsave com s = %s\n, s);")
956 : #endif
957 :   for (i=0; s[i];
958 :     i++;
959 :   scp = cp = calloc(i+1, sizeof(*cp));
```

```
960 :   if (cp == NULL){
961 :     error("cpp: Out of core for names");
962 :     exit(1);
963 :   }
964 :   while (*c++ = *s++)
965 :     ;
966 :   return(scp);
967 : }
968 :
969 : include(name, search)
970 : char *name;
971 : {
972 :   register struct include *ip;
973 :   register char **sp;
974 :   char fname[60];
975 :
976 : #if DO
977 :   printf("\nComecei include com name = %s, search = %d\n"
          name, search);
978 : #endif
979 :   if ((ip = ++inclev)> = & incl[MAXINC]){
980 :     error("Unreasonable include nesting",0);
981 :     exit(1);
982 :   }
983 :   ip->file = NULL;
984 :   if (name) {
985 :     if (search < 0 ||
986 :         (ip->file = fopen(name, "r")) == NULL && search) {
987 :     for (sp = ilist1; *sp; sp++) {
988 :         sprint (fname, "%s%s%s", *sp, *sp? "/": "", name);
989 :         if (ip->file = fopen(fname, "r "))
990 :          break;
991 :     }
992 :     if (ip->file == NULL)
993 :         for (sp = list2; *sp; sp++) {
994 :          sprint (fname, "%s%s, *sp, name);
995 :          if (ip->file = fopen(fname, "r "))
996 :          break;
997 :         }
998 :     if (ip->file)
999 :         name = fname;
1000 :   }
1001 : } else {
1002 :   ip->file = stdin;
1003 :   name = "";
1004 : }
1005 : ip->fnames = strsave(name);
1006 : ip->lineno = 1;
1007 : if (ip->file == NULL) {
1008 :   inclev --;
1009 :   errback(search? "Missing file %s": "Can't find %s",
            name);
1010 :   exit(1);
1011 : }
1012 : fputs("\n# 1 \"");
```

```
1013 : fputs(name);
1014 : fputs("\"");
1015 :#if D1
1016 : printf("\nTerminei incluse\n");
1017 :
1018 :}
1019 :
1020 :/*
1021 :#include "y.tab.c"
1022 :/*
1023 :yyerror(s)
1024 :{
1025 :#if DO
1026 : printf("\nComecei yyerror com s = %s\n", s);
1027 :#endif
1028 : error("##if %r", &s);
1029 :}
1030 :
1031 :gdiue(s)
1032 :char *s;
1033 :{
1034 :#if D
1035 : printf("\nComecei gdiuem com s = %s\n", s);
1036 :#endif
1037 : if (++ndiue >= NDIUE)
1038 :    terror("Too many D, I, U, E flags");
1039 : guard(s);
1040 :}
1041 :
1042 :guard(s)
1043 :char *s;
1044 :{
1045 :#if DO
1046 : printf("\nComecei guard com s = %s\n", s);
1047 :#endif
1048 : if (cpargc >= NP)
1049 :    terror("Too many flags");
1050 : cpargv[cpargc++] = s;
1051 :}
1052 :
1053 :struct modtab *
1054 :mktab(mod)
1055 :char *mod;
1056 :{
1057 : struct modtab *pmtab;
1058 :#if DO
1059 : printf("\nComecei mktab com mod = %s\n", mod);
1060 :#endif
1061 : pmtab = (struct modtab *) calloc(1, sizeof(*pmtab));
1062 : if(!pmtab)
1063 :    error(Can't allocate memory");
1064 : else{
1065 :    pmtab ->modnam= strsave(mos);
1066 :    pmtab ->pmtab = NULL;
1067 :    pmtab ->pflis = NULL;
```

```
1068 :    return(pmtab);
1069 : }
1070 :}
1071 :
1072 :
1073 :struct flist *
1074 :mklist(file)
1075 :char *file;
1076 :{
1077 : struct flist *pflist;
1078 :
1079 :#if DO
1080 : printf("\nComecei mklist com file = %s\n", file);
1081 :#endif
1082 : pflist = (struct flist *) calloc(1, sizeof(*pflist));
1083 : if(!pflist)
1084 :    error(Can't allocate memory");
1085 : else{
1086 :    pflist->filnam = strsave(file);
1087 :    pflist->pflist = NULL;
1088 : }
1089 : return(pflist);
1090 :}
1091 :
1092 :terror(s, x)
1093 :char *s;
1094 :{
1095 : error(s, x);
1096 : dexit( );
1097 :}
1098 :incmod(mod, file)
1099 :char *mod, *file;
1100 :{
1101 : int comp;
1102 : struct flist *pflist, *.mklist( );
1103 :    struct modtab *pmtab1, *mktab();
1104 :
1105 :#if DO
1106 :    printf("\ncomecei incmod com mod= = %s, file = %s\n",
            mod, file);)
1107 :#endif
1108 :    sprintf(fname, "%, out", mod);
1109 :    if(!hdtab)
1110 :        hdtab = mktab(mod);
1111 :    else {
1112 :        pmtab = hdtab;
1113 :        while((pmtab1=pmtab->pmtab) != NULL &&
1114 :            (comp=strcmp(mod, pmtab, pmtab1->modnam)>0))
1115 :            pmtab = pmtab1;
1116 :        if(comp != 0){
1117 :            pmtab = mktab(mod);
1118 :            pmtab-pmtab = pmtab1;
1119 :        }
1120 :    }
1121 :    sprintf(fname, "%s.%s.o", file, mod);
```

```
1122 :    file = fname;
1123 :    if (pmtab->pflis == NULL)
1124 :        pmtab->pflis = mklist(file)
1125 :    else {
1126 :        pflist = pmtab->pflis;
1127 :        while(pflist->pflist !=NULL &&
1128 :            (comp=strcmp(file, pflist->filname) !=0))
1129 :            pflist = pflist->pflist;
1130 :        if(comp != 0)
1131 :            pflist = mkflist(file);
1132 :    }
1133 :}
1134 :
1135 :
1136 :dexit()
1137 :{
1138 :#if DO
1139 :    printf("\ncomecei dexit\n");
1140 :#endif
1141 :    bclose();
1142 :    exit(exfail);
1143 :}
1144 :getsuf(s)
1145 :register char *s;
1146 :{
1147 :    register c, t;
1148 :
1149 :#if DO
1150 :    printf("\nComecei getsuf com s= %s\n", s);
1151 :#endif
1152 :    c= 0;
1153 :    while (t = *s++)
1154 :        if (t=="/")
1155 :            c=0; else
1156 :            c++;
1157 :    s -=3;
1158 :    if (c<=14 && c>2 && *s++==".")
1159 :        return(*s);
1160 :    return(0);
1161 :}
1162 :
1163 :char *
1164 :setsu(s, suf)
1165 :register char *s;
1166 :{
1167 :    register char *s1;
1168 :
1169  #if DO
1170 :    printf("\nComecei Setsuf com s = %s, suf =
          %c\n",s,suf);
1171 :#endif
1172 :    for (s1 = s; *s;)
1173 :        if (*s++ == "/")
1174 :            s1 = s;
1175 :    s = s1 = copy(s1);
```

```
1176 :    while (*s)
1177 :        s++;
1178 :    if(suf == "")
1179 :        s[-2] = "\0";
1180 :    else
1181 :        s[-1] = suf;
1182 :    return(s1);
1183 : }
1184 : callsys(f,v)
1185 : char *f, **v;
1186 : {
1187 :    int t, status;
1188 :
1189 : #if DO
1190 :    printf("\nComecei callsys com f = %s\n", f);
1191 : #endif
1192 :    if ((=fork())==0{
1193 :        execv(f, v);
1194 :        printf("Can't find %s\n", f);
1195 :        exit(100);
1196 :    } else if (t == -1)[
1197 :        printf("Try again\n");
1198 :        return(100);
1199 :    }
1200 :    while (t!=wait(&status))
1201 :        ;
1202 :    if ((t=(status&0377)) != 0 && t!=14){
1203 :        if (t != SIGINT)
1204 :            error ("fatal error in %s\n", f);
1205 :        dexit();
1206 :    }
1207 :    return ((status>>8) & 0377);
1208 : }
1209 :
1210 : modinit()
1211 : {
1212 :    tabsvm = stab0;
1213 :    obuf = tbuf;
1214 :    strcpy(oname, tname);
1215 :    modname = 0;
1216 : }
1217 :
1218 : setinit()
1219 : {
1220 :    modinit();
1221 :    chalevel = modlevel = 0;
1222 : }
1223 : pct( c )
1224 : {
1225 :    if (obuf)
1226 :        putc(c, obuf);
1227 : }
1228 :
1229 : closop(mod)
1230 : char *mod;
```

```
1231 :{
1232 :   if(tbuf){
1233 :       fclose(tbuf);
1234 :       tbuf = 0;
1235 :   }
1236 :   if((tbuf = fopen(tname, mod)) == NULL)
1237 :       error("can't open temp file");
1238 :}
1239 :
1240 :putline()
1241 :{
1242 :   char *rpl;
1243 :   int c;
1244 :
1245 :   if (keyv !=LINE)
1246 :       fprintf(obuf, "#line %d", inclev->lineno);
1247 :   else
1248 :       keyv = 0;
1249 :   if (line [0] != "#" && line[1] != != "#" && !flslvl)
1250 :       for(rlp = line; c=  *rlp++;)
1251 :           ptc("\n");
1252 :       ptc("\n");
1253 :}
1254 :
1255 :bclose()
1256 :{
1257 :   if ( tbuf)
1258 :       inlink(tname);
1259 :   if( obuf != tbuf && obuf)
1260 :       fclose(obuf);
1261 :   if(obuf)
1262 :       fclose(obuf);
1263 :   obuf = tbuf =tbuf = 0;
1264 :}
1265 :
1266 :yyparse( )
1267 :{ }
1268 :EOF
```