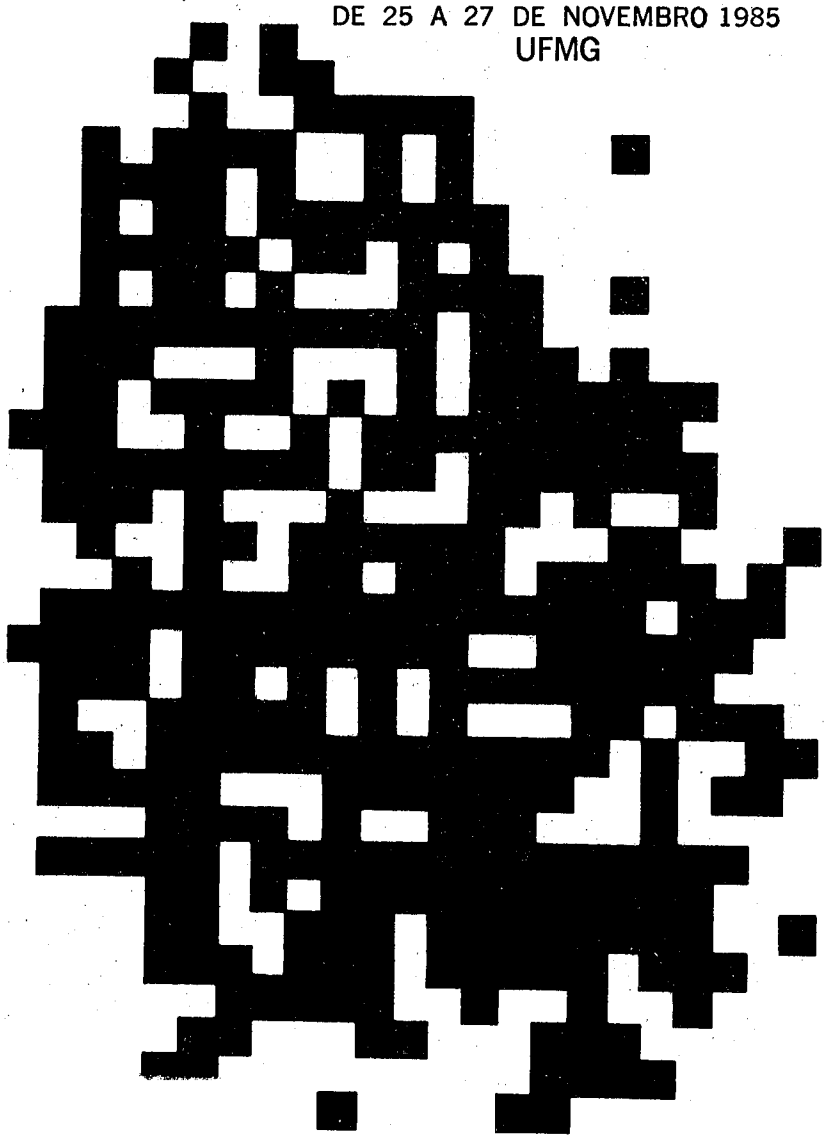
V SIMPÓSIO

SOBRE DESENVOLVIMENTO DE SOFTWARE BÁSICO

DE 25 A 27 DE NOVEMBRO 1985
UFMG



ANAIS

5º SIMPÓSIO SOBRE DESENVOLVIMENTO DE SOFTWARE BÁSICO

BELO HORIZONTE 25 A 27 DE NOVEMBRO DE 1985

A N A I S

PROMOÇÃO: SOCIEDADE BRASILEIRA DE COMPUTAÇÃO - SBC
. COMISSÃO ESPECIAL PARA LINGUAGENS E SISTEMAS DE PROGRAMAÇÃO
UNIVERSIDADE FEDERAL DE MINAS GERAIS - UFMG

PATROCÍNIO: CONSELHO NACIONAL DE DESENVOLVIMENTO CIENTÍFICO E TECNOLÓGICO - CNPq

A implementação de Modula-2 em microcomputadores nacionais - um relatório de progresso.

Noemi Rodriguez

Michael Stanton

Departamento de Informática, PUC/RJ,
22453 Rio de Janeiro - RJ

Resumo

Descreve-se o andamento de um projeto para implementar um ambiente de suporte de programação em Modula-2 num microcomputador de oito bits. Atenção especial é dada á implementação do compilador utilizado, e ao suporte básico para execução de programas.

1 - Introdução

Em [Seg84a] foi apresentada uma proposta para a implementação da linguagem Modula-2 [Wir82] em microcomputadores baseados no processador Z80 da Zilog. Este projeto, que deverá durar dois anos, teve início em dezembro de 1984 e está sendo conduzido por dois professores, uma programadora e vários alunos de pós-graduação da PUC/RJ e da COPPE/UFRJ. Aqui apresentamos um relatório interino relatando resultados parciais e dando maiores detalhes sobre aspectos selecionados do projeto.

A proposta inicial identificou quatro metas a serem cumpridas :

Registradores:
- L base do registro de ativação corrente
- G base do segmento de dados globais
- S topo da pilha
- H limite da pilha
- F base do segmento de código corrente
- PC endereço da instrução corrente
- P descritor do processo corrente
- M máscara de interrupções

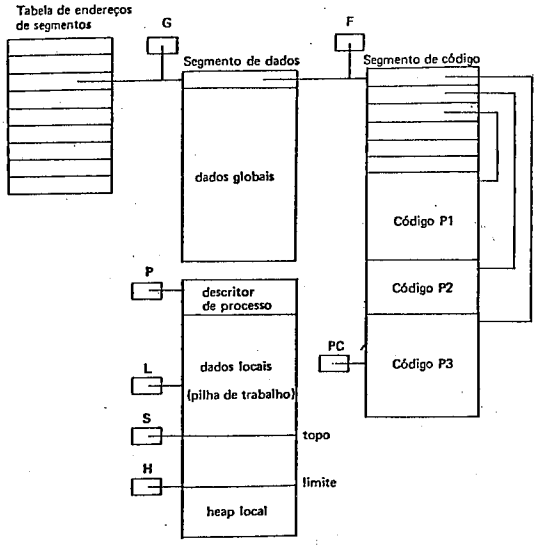


Figura 1 : A arquitetura da máquina M

Procedimentos também são endereçados através de índices. O início do segmento de código de cada módulo contém uma tabela dos endereços dos procedimentos locais. Este segmento é apontado pela primeira palavra no segmento de dados do módulo. O registrador F aponta o segmento de código do módulo que contém o procedimento a ser executado. Procedimentos externos são endereçados através de (índice de módulo, índice de procedimento).

Finalmente, dados imediatos são representados em quatro, oito ou dezesseis bits de acordo com o valor usado, e todos os desvios utilizam endereços relativos.

O resultado desta forma de representação é que as instruções do código M ocupam de um a três bytes (ver a figura 2), sendo que aproximadamente 70% são de um byte, 15% de dois bytes e 15% de três bytes.

um programa em linguagem de máquina é 3,9 vezes maior que o mesmo programa em código M. Deve-se notar que a arquitetura do PDP-11 é considerada adequada para a implementação de linguagens de alto nível.

Esta densidade muito alta do código M é resultado de duas características importantes da arquitetura que o interpreta :

1) utiliza-se uma pilha de expressões - operadores aritmético-lógicos não usam endereços explícitos para seus operandos;

2) existem formas compactas de endereçar a grande maioria das variáveis e procedimentos manuseados por um programa.

Ilustraremos este último ponto a seguir.

Um programa em Modula-2 consiste de uma coleção de módulos e cada módulo corresponde estaticamente a um segmento de dados, onde estão guardadas suas variáveis globais, e um segmento de código, contendo os procedimentos do módulo em código M.

Adicionalmente, cada co-rotina possui uma área de trabalho própria que contém a pilha de registros de ativação de procedimentos (ou, simplesmente, pilha de procedimentos) - com um registro de ativação por procedimento ativo - e o heap, onde são guardadas as variáveis dinâmicas do programa.

O código M permite endereçar a maioria das variáveis de interesse usando no máximo oito, e muitas vezes somente quatro bits. Isto é feito usando registradores de base que apontam as áreas de dados mais frequentemente usadas da seguinte maneira (ver a figura 1).

1) modo global G + n
O registrador L aponta o início do segmento de dados do módulo que contém o procedimento em execução.

2) modo local L + n
O registrador L aponta o início do registro de ativação do procedimento em execução.

3) modo pilha S + n
O registrador S aponta o final do registro de ativação do procedimento em execução.

4) modo externo TM(m) + n
Para se ter acesso a variáveis globais de outros módulos, mantém-se um Tabela de Módulos (TM) contendo o endereço do segmento de código para cada módulo carregado. Módulos externos são endereçados através do seu índice nesta tabela.

a. implementação de Modula-2 num microcomputador de oito bits;

b. implementação do núcleo do ambiente de suporte de programação de Modula-2;

c. implementação de ferramentas para o suporte de programação de Modula-2;

d. implementação de suporte de comunicação entre nós fracamente acoplados num sistema distribuido;

Neste trabalho consideraremos somente as primeiras três metas; outro trabalho apresentado neste simpósio considera em maiores detalhes o uso de Modula-2 em sistemas distribuidos [Seg85].

A implementação de Modula-2 descrita aqui é baseada naquela utilizada no computador Lilith [Wir81], projetado e construido no Instituto Federal de Tecnologia (ETH) em Zurique, Suiça, pela equipe de Niklaus Wirth. O compilador do Lilith, chamado M2M, é distribuido em fonte (escrito em Modula-2); uma cópia deste fonte foi obtida em 1983. Além da descrição do Lilith [Wir81], outras fontes importantes de informação sobre a implementação de Zurique são as dissertações de doutorado de membros da equipe de Wirth [Gei83, Jac82, Knu83]. Em particular, [Gei83] descreve detalhadamente a estrutura do compilador M2M, e [Knu83] fornece pormenores sobre o sistema operacional MEDOS-2 que provê o ambiente de suporte de programação em Modula-2 no Lilith.

A estrutura deste artigo é a seguinte: na seção 2 é apresentada a máquina abstrata que executa o código gerado pelo compilador M2M; a seção 3 trata do transporte propriamente dito do compilador; e a seção 4 discute aspectos relacionados ao ambiente de suporte.

2 - O interpretador de código M

O compilador M2M gera código objeto para uma máquina ideal, otimizado para executar programas escritos em Modula-2. Este código objeto é chamado código M e, no computador Lilith, que foi construido com a tecnologia de "bit-slice", sua interpretação é feita por firmware. Uma decisão fundamental neste projeto foi a adoção deste compilador, e a subsequente necessidade da interpretação do código M por software. Esta escolha foi fortemente motivada pelo pequeno espaço de endereçamento (64 K bytes) disponível com microprocessadores de oito bits, que força a economia de memória. Evidentemente esta ecónomia de memória será feita ás custas de tempo de execução, que aumenta por um fator da ordem de 10 para as implementações que usamos nos microcomputadores de dezesseis bits [Log84, MRI83].

Programas em código M geralmente são muito mais compactos do que os mesmos programas em linguagem de máquinas mais convencionais. No caso do PDP-11, por exemplo, foi citado em [Wir81] que

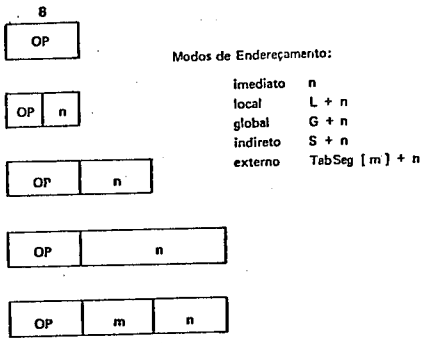


Figura 2 : Formato de Instruções do código M

Além dos registradores G, L, S e F já mencionados, a arquitetura proposta em [Wir81] para a máquina M tem mais três registradores:

PC - Contém o endereço da próxima instrução relativo ao início do segmento de código.

P - Aponta o início da área de trabalho da co-rotina de execução. Este é também o início do seu descritor, que corresponde ao tipo PROCESS da linguagem.

H - Aponta a última posição já alocada no heap. O heap cresce em direção oposta á pilha de procedimentos. Se H < S em algum instante, ocorre estouro desta pilha.

Os registradores da máquina M são de dezesseis bits, e o endereçamento de dados é feito por palavra de dezesseis bits. Portanto a memória de dados endereçável é de 64 K palavras. O endereçamento de instrução é por byte de oito bits, e são usados somente quinze bits do registrador PC dando um limite máximo de 32 K bytes por módulo. Observar que a cada acesso á memória para trazer um byte de instrução é necssário somar ao valor do PC o endereço do início do segmento de código.

Implementação

O interpretador é um programa que executa no processador Z80 com até 64 K bytes de memória.

O interpretador carrega para sua memória simulada um programa em código M, já pronto para execução (ver seção 4.1.2), inicializa os registradores e começa a executar as instruções carre-

gadas.

A estrutura interna do interpretador segue quase exatamente a arquitetura descrita acima, com uma exceção, que afeta o registrador PC, o contador de instruções. Ao invès deste registrador conter o endereço da instrução apontada *relativo* ao início do segmento de código, ele contèm seu endereço *absoluto*. Com isto evitamos ter que calcular o endereço efetivo a cada acesso ao segmento de código, o que diminuiria sensivelmente a velocidade de execução. Felizmente esta alteração tem poucas consequências adversas, principalmente porque todos os desvios são relativos ao atual conteúdo do PC. Somente foi necessàrio alterar a implementação das instruções de chamada e retorno de procedimento, porque o endereço de retorno guardado no registro de ativação utiliza somente quinze bits, como no projeto original. No momento de guardar o conteúdo antigo do PC, ele è convertido em endereço relativo ao início do segmento de código.

A primeira versão do interpretador foi baseada na especificação dada em [Wir81], e programada na linguagem C [Ker78]. A escolha de C se deveu ás seguintes razões:

- A linguagem è especialmente adequada para operações sobre bits (extração e deslocamento) que são muito usadas no interpretador;

- Existe suporte para todos os tipos usados em Modula-2, inclusive REAL;

- Existe um compilador disponível para o computador alvo [AZT83];

- facilita o transporte do interpretador para outros equipamentos, devido á transportabilidade de C.

Esta escolha de implementação teve uma consequência adicional para o interpretador. No computador Lilith, a representação do tipo REAL è a mesma usada no PDP-11. Para evitar conversões de representação pouco producentes em tempo de execução, resolvemos adotar a implementação de REAL usada pelo compilador de C [AZT83]. Esta mudança è praticamente transparente para o projeto. Uma descrição detalhada do interpretador se encontra em [Hir85].

Numa segunda etapa, è provàvel que seja feita uma segunda versão do interpretador em linguagem de montagem, visando economizar memòria e aumentar a velocidade de execução. A decisão final dependerà da anàlise, ainda a ser feita, do desempenho da atual versão.

3 - Transporte do compilador M2M para o ambiente alvo

O compilador M2M desenvolvido pela equipe de N. Wirth [Gei83] è escrito em Modula-2. Para transportà-lo para o ambiente alvo, è necessàrio obter uma versão em código M. Dispùnhamos no

início do ano de duas implementações de Modula-2 para microcomputadores de dezesseis bits da classe IBM PC, desenvolvidas pela Logitech [Log84] e pelo MRI [MRI83]. O software do MRI, que produz código M, funciona sob o sistema PC-DOS, enquanto que a Logitech configura seu compilador, que gera código nativo para o 8086, para o PC-DOS e para o CP/M-86. Como o primeiro objetivo é a criação de um ambiente Modula-2 sob o CP/M do Z80, optamos pela utilização do software da Logitech para evitar problemas de compatibilidade de formatos de arquivos. Esta escolha impôs a necessidade de duas compilações do compilador M2M, a primeira para obter um programa, em código do 8086, que transforma Modula-2 em código M, e a segunda para obter o mesmo programa em código M (figura 3).

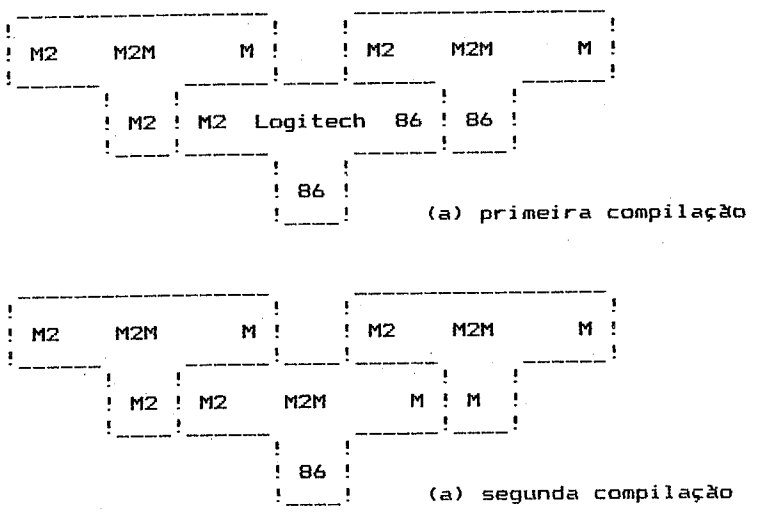


Figura 3 : Processo de produção do compilador para o ambiente alvo

Para realizar a primeira compilação, foram necessárias diversas pequenas alterações no fonte do compilador M2M. A principal causa de alterações foi a diferença de endereçamento existente entre o compilador da Logitech e o compilador M2M. No compilador da Logitech, o tipo ponteiro (ou endereço) ocupa duas palavras de memória, enquanto o tipo inteiro ocupa apenas uma. No compilador M2M, os dois tipos ocupam uma palavra de memória. O programa fonte que compilamos utilizava incontáveis vezes esta equivalência, armazenando e lendo valores nos arquivos intermediários sem se preocupar se estes eram inteiros ou endereços.

Não esperamos encontrar problemas na segunda compilação, uma vez que estaremos usando como entrada do compilador M2M o fonte do mesmo compilador. A única alteração prevista para esta fase é a da representação dos números reais; o compilador ficará com representação de reais idêntica á do compilador C que estamos usando para desenvolver o interpretador no ambiente alvo [AZT84], o que deve facilitar a implementação de instruções envolvendo operações com reais no código M.

4 - Ambiente de Suporte

Concluída a primeira fase de trabalho com o compilador, passamos ao estudo do ambiente de programação dentro do qual este funcionará. Como previsto, o desenvolvimento deste ambiente está sendo realizado em duas etapas. Na primeira, já em andamento, pretende-se criar interfaces com o CP/M que forneçam o suporte necessário á execução de programas escritos em Modula-2. Esta estapa deve ser concluida atè o final deste ano. Paralelamente, estamos estudando quais as características e a estrutura de sistema operacional desejáveis para a versão final do ambiente de suporte, que não incluirá utilização do CP/M. A implementação desta versão está prevista para 1986.

4.1 - Estrutura necessária para dar suporte á execução

O ambiente de programação deve fornecer os mecanismos oferecidos pelo MEDOS-2 [Knu83] e que são supostos para o funcionamento do compilador. Estes incluem a ligação e carga de programas, o controle de execução de programas e a administração da comunicação de programas com arquivos e com o terminal.

Segue-se uma descrição resumida destes mecanismos, juntamente com a exposição de algumas das dificuldades e soluções que encontramos para fornecê-los.

4.1.1 - Organização geral da memória no MEDOS-2

Um único processo executa todos os programas de usuário. Isto é possivel graças ao conceito de "chamadas de programas". Qualquer programa pode conter um comando que causa a ativação de um outro programa, como uma chamada de procedimento.

Cada vez que um programa è chamado, um registro de ativação para ele è colocado na pilha de execução; este registro è retirado quando o programa termina. O primeiro registro de ativação na pilha corresponde á parte residente do sistema operacional. O conceito è semelhante ao de registro de ativação de um procedimento, sendo que no caso de programas o registro de ati-

vação contêm, além das variáveis globais e de uma pilha de trabalho para execução do programa, o código dos módulos carregados para executar o programa.

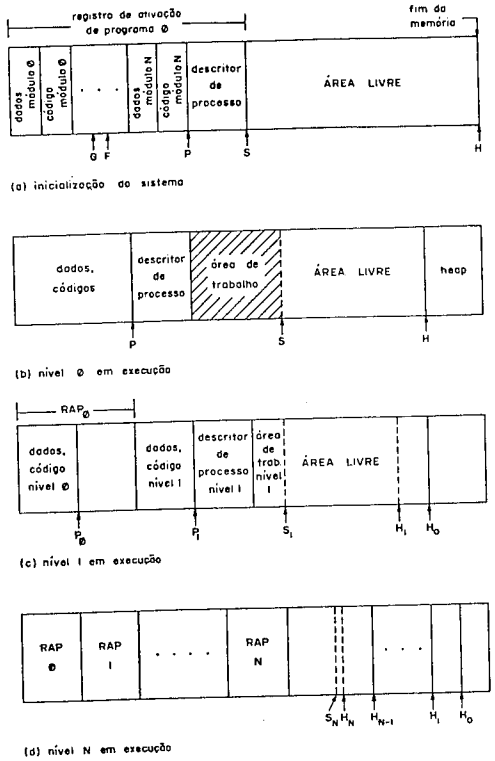


Figura 4 : Organização da memória durante execução

Variáveis locais são semi-dinâmicas, isto é, são alocadas na pilha de trabalho somente durante a execução de um procedimento, mas seus endereços são determinados pelo compilador como deslocamentos relativos ao endereço de base do seu registro de ativação. Esta é a forma usada convencionalmente para linguagens que definem procedimentos potencialmente recursivos, como Pascal, Algol, etc.

Quando o sistema é inicializado, encontram-se na memória os módulos pertencentes ao sistema operacional. O registro de ativação de nível 0 corresponde ao programa que contém estes módulos. Excluindo-se a área ocupada pelas variáveis globais e código destes módulos, toda a memória está disponível como área de trabalho deste programa (figura 4(a)).

A medida que o programa começa a ser executado, ele vai em geral requerer área do heap. Este será alocado no final da memória, diminuindo a área da pilha de execução (figura 4(b)).

Quando o primeiro programa de usuário é chamado, é criado um novo registro de ativação para ele; a área de memória da qual ele dispõe é limitada pelo fim do heap do programa anterior (figura 4(c)).

Chamadas de programas podem ser aninhadas enquanto houver espaço suficiente na memória para a pilha de execução do programa corrente. Se em algum nível N o registrador S atingir o valor do registrador H, não será possível continuar a execução. Neste caso, ocorre um "trap" (como será explicado mais adiante), cancelando o nível de ativação corrente, e o controle é devolvido ao programa de nível (N - 1) (figura 4(d)).

4.1.2 - Ligação e carga de programas

Os conceitos da arquitetura da máquina M simplificam o trabalho do ligador. A referência a módulos importados é feita indiretamente com ajuda da Tabela de Módulos (TM). A ligação dos módulos é feita pelo mesmo programa que faz a carga, que passamos a chamar de ligador-carregador (L-C).

Quando o compilador gera código para um módulo com importações, ele atribui a cada módulo importado um número local, segundo a ordem em que seu nome aparece. Existe uma seção no arquivo objeto, chamada tabela de importações (TI), que contém uma lista dos módulos importados. Nesta lista, a posição onde cada módulo aparece corresponde ao seu número local. No código gerado, as referências aos módulos são feitas através destes números locais.

O ligador-carregador utiliza uma lista de importações, local a ele; esta descreve os módulos que devem sera carregados. Quando o módulo principal de um programa começa a ser carregado, esta lista não contém nenhum elemento.

Quando o ligador-carregador lê a tabela de importações , ele verifica se cada um dos módulos importados já foi carregado na memória ou já está na lista de importações. Em qualquer um destes casos, já existe na TM uma posição correspondente a este módulo. Este índice é o chamado número real do módulo. O conteúdo de cada posição na TM é o endereço do início do segmento de dados do módulo correspondente. O nome do módulo está carregado na memória imediatamente antes do segmento de código (figura 5). Quando o módulo correspondente a um índice ainda não foi carregado, o conteúdo de sua posição na TM é NIL. Neste caso, existe um descritor de "importação a fazer" na lista de importações do ligador-carregador, que contém o nome deste módulo e o seu índice na TM.

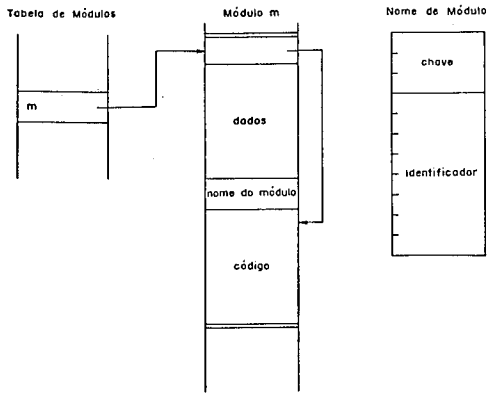


Figura 5 : Um módulo carregado na memória

Se o ligador-carregador não encontra o módulo nem na TM (através da comparação com o nome carregado) nem na lista de importações, ele cria um descritor de "importação a fazer" para ele, insere-o na última posição da lista de importações, e reserva uma nova posição na TM (posição esta armazenada no descritor).

A medida que o ligador-carregador trata a tabela de importações, ele também constrói uma tabela de tradução, associando a cada número local de módulo (que é o índice nesta tabela) o seu número real (índice na TM).

Sempre que o ligador-carregador acaba de carregar um módulo, ele retira o primeiro descritor da lista de importações e passa a carregar o módulo correspondente a a ele. O trabalho do ligador-carregador termina quando a lista de importações fica vazia.

A interface de módulo que exporta objetos é representada pelo seu módulo de definição. O compilador transforma os módulos de definição nos chamados arquivos simbólicos. Quando um módulo de implementação é compilado, os arquivos simbólicos de todos os módulos que ele importa são usados pelo compilador para fornecer as declarações dos identificadores importados. Para poder verificar se todas as referências a um módulo separado são baseadas na mesma interface (isto é, no mesmo arquivo simbólico), o compilador gera um número associado á hora de compilação, chamado chave do módulo, quando compila um módulo de definição, e inclui este número no arquivo simbólico. O par (nome,chave) identifica unicamente uma determinada versão de um módulo separado. Na geração do arquivo objeto de um módulo de implementação, o compilador copia o nome e a chave do módulo para o arquivo objeto, na seção cabeçalho, assim como os pares (nome,chave) de todos os módulos referenciados por este módulo, na tabela de importações.

É responsabilidade do ligador-carregador determinar se a interface de um módulo exportador é a mesma que é referenciada pelo módulo que a importa. Para isto, o L-C verifica se a chave do módulo é a mesma que foi referenciada pelo outro módulo.

A chave de cada módulo carregado é armazenada na memória juntamente com o seu nome, imediatamente antes do segmento de código (figura 5).

A medida que o ligador-carregador lê seções de código de um arquivo objeto, ele substitui todas as referências a números locais de módulos externos pelos seus números reais, guiando-se pela tabela de tradução. As posições onde as substituições devem ser feitas são indicadas por seções do arquivo objeto chamadas tabelas de referenciamento externo.

Cada módulo deve ser inicializado uma única vez, e módulos importados devem ser inicializados antes do módulo que os importa. Para isto, o compilador insere, no código de módulos que importam outros, antes das instruções de inicialização, uma chamada ao procedimento 0 de cada um dos módulos importados (o procedimento 0 de qualquer módulo é o seu corpo). Para garantir a unicidade da inicialização, é reservada uma palavra em cada segmento de dados que inidica se o módulo foi ou não inicializado, e no início do corpo de cada módulo é inserido código para o controle retornar ao importador se esta palavra já houver sido alterada (e para alterá-la caso contrário). Desta forma, o problema da inicialização de módulos é transparente para o ligador-carregador.

Implementação

No caso do computador Lilith, o módulo que contém o ligador-carregador (módulo Program) é residente na memória, fazendo parte do sistema operacional. O módulo principal do MEDOS-2, chamado SEK, é o módulo que recebe o controle na inicialização do sistema, estando assim o ambiente necessário para execução de programas em código M sempre montado.

Em nosso primeiro ambiente, precisamos estabelecer um procedimento de inicialização que coloque o sistema nesta situação. A solução adotada foi a criação de um ligador em separado que faz a ligação do módulo principal do ambiente Modula-2 com todos os módulos que ele importa, criando em um arquivo uma imagem de memória. O interpretador, ao iniciar execução, chama uma rotina simples que copia este arquivo para a memória. A seguir, o interpretador passa a executar as instruções do segmento de código do módulo principal do ambiente de suporte, correspondente ao módulo SEK.

4.1.3 - Controle de Execução

Apesar da semelhança entre os conceitos de chamada de procedimento e chamada de programa, existem algumas diferenças importantes. Não existem instâncias de um programa, isto é, ele só pode estar ativado uma vez, não existindo recursividade. Além disto, o código de um procedimento está sempre na memória. No caso de uma chamada de programa, podem haver módulos entre os que o compõem já na memória, e outros que ainda não hajam sido carregados. Os módulos neste último caso são colocados na memória no momento da chamada e retirados quando termina a execução de programas, fazendo parte do registro de ativação descrito anteriormente.

O módulo Program é responsável pela criação de uma co-rotina para execução do ligador-carregador quando um programa é chamado. Este módulo exporta o procedimento Call, que deve ser importado por quaisquer módulos que desejem realizar chamadas de outros programas. Program também contém o procedimento correspondente ao ligador-carregador.

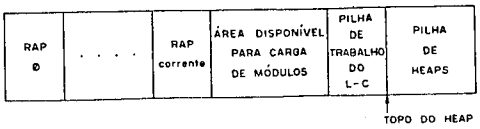


Figura 6 : Organização da memória depois da ativação do ligador-carregador

O procedimento Call determina o espaço de memória disponível para carga de módulos, passando esta informação para o ligador-carregador através de variáveis globais. O número de módulos já carregados é armazenado. A seguir, Call cria uma co-rotina para executar o ligador-carregador e transfere o controle para ela. O espaço de trabalho desta co-rotina é alocado antes do limite do heap, no final da área da pilha de execução (figura 6), para não afetar a pilha de registros de ativação de programas, pois a co-rotina do ligador-carregador será destruida antes que o novo programa comece a ser executado. Call transfere o controle para esta nova co-rotina. O ligador-carregador é executado, seguindo o algoritmo descrito na seção anterior, atribuindo a uma variável global ProcPrograma o procedimento de inicialização (corpo do módulo principal do programa carregado). A última instrução do ligador-carregador transfere o controle de volta para o procedimento Call. Este poderia executar diretamente uma chamada do procedimento contido em ProcPrograma, mas isto criaria a possibilidade de uma queda total do sistema no caso de erro no novo programa. Para evitar tal acontecimento, Call cria uma nova co-rotina associada a ProcPrograma. A execução desta co-rotina é responsabilidade do módulo Monitor, que é importado por Program.

Para gerenciar o final da execução de programas, é utilizado o conceito de "trap". Um trap é chamado quando uma condição errônea é detetada pelo processador (interpretador de código M), tal como estouro de memória (erro irrecuperável) ou uma divisão por zero (erro recuperável). Um trap também pode ser chamado incondicionalmente por uma instrução do código M. Quando um trap é invocado, um código indicando a condição de erro é passado como parâmetro. Este código é armazenado no descritor da co-rotina corrente; esta é suspensa e o controle é passado para um procedimento localizado em uma posição fixa da memória, o tratador de traps.

O módulo Monitor contém um módulo local chamado GerentedeExceções, com dois procedimentos, Call e TrapHandler. O procedimento Call é exportado, constituindo a interface com o módulo Program; o Call de Program chama o Call de Monitor para que este inicie a execução da co-rotina associada ao novo programa. O procedimento TrapHandler é definido na posição de memória fixa para tratamento de traps, já mencionada acima. A inicialização do módulo Monitor cria um processo para TrapHandler, deixando-o pronto para receber o controle através de um trap. O procedimento Call de Monitor armazena um descritor do processo anterior e transfere o controle para o novo processo (programa chamado). Se ocorrer um erro de execução, automaticamente é invocado um trap. Se a execução ocorrer normalmente, o programa termina com a execução de um trap incondicional. Em qualquer caso, TrapHandler recebe o controle , copia a co-rotina interrompida para uma variável global, e transfere o controle para a co-rotina associada a Call (de Monitor). Esta restaura ao status de "corrente" o programa anterior, e termina, retornando o controle para Call de Program.

Neste ponto o procedimento Call providencia o restabelecimento da situação de memória anterior, isto è, restaura o antigo limite do heap e também o antigo número de módulos carregados. Além disto, Call examina o descritor do processo terminado para retornar uma indicação de status do programa chamado em um de seus paraâmetros.

4.1.4 - Comunicação com arquivos e terminais

No MEDOS-2 existem dois módulos residentes responsáveis por estes serviços de comunicação, FileSystem e Terminal.

O módulo FileSystem exporta o tipo Arquivo e uma coleção de operações sobre ele, permitindo criação, destruição, mudança de nome, e acesso sequencial e randômico a arquivos. O acesso è sempre a um caractere ou a um valor que ocupe uma palavra (dois bytes). Outros módulos na biblioteca exportam operações de acesso a reais e strings.

Os arquivos são todos definidos em um mesmo nível, isto è, não existe nenhuma hierarquia de diretórios.

O módulo Terminal exporta operações simples de acesso ao terminal, como leitura e escrita de caractere e de string, e as operações ReadLn e WriteLn.

Implementação

Como já dito, nosso primeiro ambiente é baseado na utilização do CP/M. Para fornecer os serviços descritos acima, foram criados dois módulos cuja implementação é baseada em chamadas ao BDOS. Estas chamadas são feitas sob a forma de chamadas de procedimentos exportados por um outro módulo, BDOSCalls. Estes procedimentos são definidos de acordo com número e tipo dos parâmetros. A implementação do módulo BDOSCalls é feita através da utilização de procedimentos em código M [Wir81]. Um código especial de instrução ordena ao interpretador a execução de uma chamada ao BDOS. A simplicidade desta chamada é garantida pelo fato da implementação da linguagem C que estamos usando conter em sua biblioteca funções para este fim.

4.2 - Propostas para a versão definitiva do sistema

4.2.1 - Multiprogramação

Não existe apoio explícito para multiprogramação no MEDOS-2. Pode-se utilizar o sistema para programação concorrente, através da manipulação de co-rotinas [Seg84b]. Entretanto, em nosso caso, gostaríamos que o próprio desenvolvimento do sistema fizesse uso de processos paralelos. O conceito se torna especialmente importante dado que pensamos na futura ampliação do ambiente de programação mono-usuário em desenvolvimento para uso em sistemas distribuídos [Seg85].

Neste esquema é prevista a necessidade de acesso a recursos nas diferentes estações através de chamadas remotas de procedimentos. A implementação deste acesso requer a utilização de processos servidores, cujo gerenciamento pede por sua vez algum esquema de multiprogramação. A proposta atual inclui o uso de monitores para proteção do acesso a variáveis compartilhadas.

4.2.1 - Gerência de configuração e o sistema de arquivos

Uma vez que estamos considerando um sistema dedicado exclusivamente a Modula-2, torna-se possível e também quase obrigatório o desenvolvimento de ferramentas de apoio ao programador. Entre estas, destacamos uma cuja importância consideramos tão imediata que optamos por entrelaçar seu desenvolvimento com o do próprio sistema operacional. Esta ferramenta é o gerente de configuração, cuja função será definida a seguir.

A linguagem Modula-2 permite que um programa seja formado por diversas unidades compiladas separadamente. Esta característica implica em diversas vantagens [Ich79], entre as quais : facilidade de desenvolvimento de grandes sistemas por um grupo de programadores, criação de bibliotecas de rotinas já compiladas e economia em tempo de compilação. No entanto, alguns problemas são introduzidos por este mecanismo :

- Como garantir que a mesma versão de um módulo de definição está sendo usada para compilar todos os módulos que dele dependem?

- Como saber em que arquivo está armazenado o módulo de definição de que precisamos?

Um dos possíveis mecanismos de controle consiste de uma base de dados que armazena todas as formas dos módulos componentes de um programa junto com informações sobre a história e composição destes. O compilador e ligador-carregador devem gerar e atualizar estes dados. Devem existir ainda utilitários que trabalhem sobre esta base fornecendo vários tipos de informações como, por exemplo, quais módulos devem ser recompilados devido a alterações em um determinado módulo de definição.

Este sistema seria o gerente de configuração.

Para a construção de um sistema como este é importante que se disponha de um sistema de arquivos adequado, no que diz respeito á representação de relações entre arquivos. O sistema de arquivos do PC-DOS inclui uma hierarquia de diretórios, que acreditamos ser adequada ao desenvolvimento do gerente de configuração. Além disto, gostaríamos de adotar o formato de disco de algum sistema de grande divulgação, para não acrescentar mais um capítulo ao problema já generalizado de incompatibilidades. Optamos então pela adoção do formato do PC-DOS. Esta escolha tem ainda outra vantagem, que é a existência de um compilador Modula-2, nos moldes do que estamos desenvolvendo, para o sistema PC-DOS (compilador este já citado anteriormente). Desta forma, oferece-se a possibilidade de transferência de programas desenvolvidos em um dos sistemas para o outro, em qualquer uma das direções.

Algumas operações envolvendo o manuseio de diretórios serão incluídas no módulo correspondente a FileSystem. De resto, a definição deste módulo será de acordo com a biblioteca padrão definida em [Mod85]. O módulo FileSystem desta biblioteca apresenta algumas evoluções em relação aos definidospelo MEDOS-2 e pela Logitech. Em primeiro lugar, é mais simples, dispensando as operações de definição de estado (SetRead, SetWrite) existentes anteriormente. Em segundo, já oferece acesso a valores de qualquer tamanho, podendo-se ter acesso a um arquivo lendo ou gravando uma estrutura do tipo record, array, ou um tipo simples. Esta segunda característica já era encontrada na implementação feita pela Volition.

A idèia da biblioteca padrão surgiu a partir das primeiras implementações de Modula-2. Cada uma destas definiu sua prôpria biblioteca, e como Modula-2 não possui comandos de entrada e saída nem de gerenciamento de memòria, entre outros, um programa escrito para uma destas implementações deve ser parcialmente reescrito para poder funcionar em outra. Isto aumenta os custos de programação e cria relutância em usar a linguagem.

A partir do final de 1983, um grupo formado por pessoas que haviam participado de alguns dos projetos de implementação (ETH-Zürich, Logitech, Volition e Diser) foi criado para discutir e especificar uma biblioteca padrão. A primeira versão do resultado foi publicada em dezembro de 1984.

5 - Conclusões

Apresentamos acima os principais aspectos da implementação do ambiente de suporte de programação em Modula-2 num microcomputador isolado. O atual projeto pretende implantar este ambiente em microcomputadores baseados no processador Z80 e unidades de disquete; algumas das caracteristicas do projeto são claramente influenciadas pelas limitações impostas por esta escolha de equipamento, particularmente o pequeno espaço de memòria endereçável pelo processador. Porèm, em grande parte, o software não depende da escolha do processador, e, a princípio, poderia ser transportado para outro tipo de equipamento com maiores recursos (Neste caso, porèm, as limitações da máquina M, em particular a memòria de dadosde 64K palavras, poderiam ter uma influência restritiva).

O cronograma do projeto prevè um funcionamento precário do compilador no computador alvo atè dezembro de 1985 e, no momento de escrever este artigo, temos razoável confiança no cumprimento deste prazo. Previmos também concluir a instalação do ambiente definitivo do ambiente de suporte atè meados de 1986.

Os recursos usados no trabalho descrito aqui foram modestos. Um dos autores (NR) se dedica quase integralmente ao projeto. O outro (MS) tem dedicação parcial, como também tem uma aluna de mestrado desenvolvendo sua dissertação de mestrado sobre o interpretador. Os recursos materiais incluem dois microcomputadores CP-500 da Prològica, dedicados ao projeto, e acesso eventual aos microcomputadores de dezesseis bits do Departamento de Informàtica da PUC.

6 - Agradecimentos

Acesso a informação sobre a implementação de Modula-2 no computador Lilith foi conseguido graças a Hermann Seiler do Centro de Computação do Instituto Federal de Tecnologia (RZETH) em Zurique. Albrecht von Plehwe trouxe o compilador M2M de Zurique para a PUC, e também instalou CP/M nos equipamentos

usados pelo projeto. Emiko Hiraga implementou o interpretador em C.

Um dos autores (MS) agradece o apoio financeiro do CNPq.

O trabalho descrito aqui è apoiado parcialmente pela EMBRATEL, sob o contrato C. DCD - 014/84.

Bibliografia

[AZT84] "AZTEC C II User Manual", Manx Software Systems, Release 1.06, março de 1984.

[Gei83] L. Geissman, "Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith", DISS7286, ETH Zürich, 1983.

[Hir85] E. Hiraga, M. Stanton, "Especificação do còdigo interpretâvel gerado pelo compilador de Modula-2 (còdigo M)", Nota Tècnica 01/85, Projeto Modula-2, Departamento de Informàtica, PUC/RJ, junho de 1985.

[Ich79] J. Ichbiach, J. Barnes, J. Heliard, B. Krieg-Brueckner, O. Roubine, B. Wichmann, "Rationale for the Design of the Ada Programming Language", SigPlan Notices, vol 14, num 6, parte B, junho de 1979.

[Jac82] C. Jacobi, "Code Generation and the Lilith Architecture", DISS7195, ETH Zürich, 1982.

[Ker78] B. Kernigham, D. Ritchie, "The C Programming Language", Prentice-Hall, New Jersey, 1978.

[Knu83] S. Knudsen, "MEDOS-2 : A Modula-2 Oriented Operating System for the Personal Computer Lilith", DISS7346, ETH Zürich, 1983.

[Log84] "Modula-2/86 User's Manual", Logitech, Inc, Release 1.02, junho de 1984.

[Mod85] "Modula-2 Standard Library Definition Modules", Modula-2 News, Issue #1, janeiro de 1985, 22-37.

[MRI83] L. Bingham, L. Geissman, C. Jacobi, R. Riggs, N. Wirth, "Modula-2 Handbook", Modula Research Institute, 1983.

[Seg84a] L. Segre, M. Stanton, "O Desenvolvimento da Linguagem Modula-2 para Microcomputadores Nacionais", Anais do IV Simpòsio sobre Desenvolvimento de Software Bàsico, São Josè dos Campos, outubro de 1984, 61-65.

Desenvolvimento de Software Concorrente", Anais do XVII CNI, Rio de Janeiro, novembro de 1984.

[Seg85] L. Segre, M. Stanton, "Sobre o uso de Modula-2 para programação em ambientes distribuidos", submetido para apresentação no V Simpòsio sobre Desenvolvimento de Software Bàsico, a se realizar em Belo Horizonte, novembro de 1985.

[Wir81] N. Wirth, "The Personal Computer Lilith", A. I. Wasserman Ed., Software Development Environments, IEEE Computer Society Press, 1981.

[Wir82] N. Wirth, "Programming in Modula-2", Springer Verlag Berlin e Heidelberg, 1982.