ISTVAN SIMON

FUNDAMENTOS DE SISTEMAS OPERACIONAIS

Dissertação de mestrado submetida à Comissão de Pós-Graduação do Instituto de Matemática e Estatística da Universidade de São Paulo afim de obter o grau de Mestre em Matemática Aplicada.

A meus Pais e Irmãos .

AGRADECIMENTOS

No decorrer deste trabalho recebi a ajuda valiosa de diversas pessoas.

A todas gostaria de expressar aqui a minha gratidão.

Devo agradecimentos especiais:

Ao prof. Guy Augier, da Universidade de Grenoble, França, cujo estímulo teve influência decisiva para que este trabalho fôsse feito; ao Dr. Valdemar W. Setzer pela sua segura orientação e paciente leitura do manuscrito que foram muito apreciadas; aos membros da banca Dr. Ivan de Queiroz Barros e Dr. Max Hehl que aceitaram o encargo de julgar este trabalho apesar da época inconveniente; aos meus colegas do departamento de Matemática Aplicada, professores e funcionários que muito me aliviaram de parte de minhas responsabilidades usuais; ao Sr. Jorge Stolfi que leu parte do manuscrito, pelas suas valiosas observações; ao Dr. Adolpho Ribeiro Netto, ao Sr. José Antonio Lacerda Duarte e ao Sr. Elzo Sigueta da Fundação Carlos Chagas pela sua colaboração na elaboração das matrizes deste trabalho, e em particular à D. Marianina Malvezzi, à Srta. Vera Helena Cardoso Dias, à Srta. Marilda Colferai e à Srta. Rita de Cássia Nazário, pelo excelente trabalho de datilografia e ao Sr. José Antonio Lacerda Duarte pelos desenhos de ótima qualidade; e ao Sr. Armando Garcia Segura pelo esmerado trabalho de impressão.

SUMÁRIO

Este trabalho trata das técnicas utilizadas em montadores e sistemas operacionais. O capítulo 1 trata de conceitos fundamentais de programação. A organização básica de um computador digital é descrito; um computador hipotético, o HIPO, é definido; é dada uma introdução rigorosa ao conceito de funções computáveis; o conceito de subrotina é precisamente definido e os métodos de ligação usuais com subrotinas fechadas é tratado; o conceito de subrotina recursiva é abordado com detalhes; as técnicas fundamentais de busca em tabelas são revistas; e finalmente é apresentada uma evolução geral cronológica dos sistemas operacionais. No capítulo 2 são vistas as técnicas e conceitos fundamentais usados em montadores. A linguagem de montagem do HIPO, HAL, é introduzida; o montador de dois passos é examinado em detalhes; e o montador de um passo só é descrito sucintamente. O capítulo 3 contém as técnicas fundamentais de ligação entre módulos. O conceito de módulos independentes é definido; o problema da relocação é examinado; símbolos externos e públicos relativamente a um módulo são introduzidos; a declaração de símbolos externos e públicos em FORTRAN e HAL é exposta; o "link-editor"|carregador é estudado detalhadamente; finalmente o problema de superposição é tratado e um esquema geral automático de superposição, para o caso em que módulos são subrotinas fechadas não recursivas, é proposto; O capítulo 4 versa sobre o sistema de contrôle de entrada/saída. Os conceitos de canal e interrupção são introduzidos; a técnica de "buffers" múltiplos é estudada; as rotinas e tabelas básicas do sistema de contrôle de entrada/saída são expostas. O capítulo 5 trata dos conceitos e técnicas utilizadas em sistemas de multiprogramação e tempo compartilhado; o conceito de programa reentrante é introduzido; métodos de relocação dinâmica com registradores-base, e através de paginação e segmentação são expostos; o conceito de processo e processadores virtuais é definido e o problema de sincronização de processos paralelos é examinado; o problema da alocação de recursos é tratado e finalmente o trabalho se encerra com considerações a respeito de problemas de proteção por "hardware" e "software".

E R R A T A

Página	linha	onde se lê	leia-se
8	9 de baixo	...se agora $l_2 = 0$ ...	...se agora $I_2 = 0$...
11	15 de cima	...se usar ou & no...	...se usar b ou & no...
18	7 de cima	...tais que $0 \leq x \leq 9$,...	...tais que $0 \leq X \leq 9$,...
19	13 de baixo	...por exemplo "$\frac{n}{n}$" no...	...por exemplo "$\frac{m}{n}$" no...
25	12 de baixo	...Descrevemos agora...	...Descreveremos agora
29	13 de cima	...S_1 X T, ié um ...	...$S_1 \times T$, ié um ...
35	4 de cima	...por volta de 1965,...	...por volta de 1956,...
40	11 de cima	... b ou O ...	... b ou 0 ...
40	13 de cima	... b ou O ...	... b ou 0 ...
43	2 de baixo	... ser colocada ...	... ser alocada ...
45	4 de cima	... seqüencialmen-...	...seqüencialmen-...
46	1 de cima	... tomar cetos ...	... tomar certos ...
46	17 de cima	...um símbolo s ...	...um símbolo s ...
46	18 de cima	...por end-simb[s];...	...por end-simb[s];...
46	16 de baixo	... ; of ← b ; ...	... ; of ← b
48	12 de cima	...(símbolo superdefini-do); senão:	...(símbolo superdefini-do) e vá para PI 12; senão:
49	20 de baixo	...e o conteúdo da ...	... é o conteúdo da ...
49	15 de baixo	... e p ← 0 ; ...	... e p ← 0 ; ...
49	7 de baixo	...codop com[op...	...codop-com[op...
55	3 de baixo	...ser ajustadas, ié re-locadas, ...	...ser ajustados, ié re-locados,...
63	14 de baixo	... n ← 0; ...	... n ← 0; ...
68	na figura - nó 12	... ↓ [5 / nil]	↓ [5 / •] → ↓ [13 / nil]
70	7 de baixo	... para uma ...	... para ...
70	6 de baixo	... unidades rápida...	...unidades rápidas...
77	25 de cima	...tanto pela unidades...	...tanto pela unidade...

Página	linha	onde se lê	leia-se
85	14 de cima	...capítulo examinaremos	...capítulo examinamos
85	17 de cima	...processador central;...	...processador central,
86	15 de baixo	... tradas em 1. ...	... tradas em [1]. ...
98	8 de cima	...PBR, DBR ou ...	...PBR, BPR ou ...
99	7 de baixo	...PBR, DBR, ou ...	...PBR, BPR, ou ...
101	3 de cima	...nas memória ...	... nas memórias ...
101A	8 de cima	...ligação de A. ...	... ligação de <A>. ...
103	16 de baixo	...denominado de vetor...	...denominada de vetor
114	11 de cima	... este pode pode ...	... este pode ...
115	3 de baixo	...ser fetivamente ...	... ser efetivamente ...
117	22 de baixo	... cessário em ...	... cessários em ...
118	3 de cima	...controlado pela sistema ...	...controlado pelo sistema ...
119	10 de baixo	...necessário neste...	...necessária neste...
120	16 de baixo	...Waterloo, Out. ...	...Waterloo, Ont. ...
120	10 de baixo	..."Supervisury and...	.."Supervisory and...
120	8 de baixo	..."The Struchue of...	..."The Structure of...
120	3 de baixo	... ting Sinveys ...	... ting Surveys ...
121	2 de baixo	... [30] Holt, ...	... [25] Holt, ...

I N D I C E

Título I

Agradecimentos II

Sumário III

Indice IV

CAPÍTULOS

1. Introdução e Prérequisitos 1

2. A linguagem de montagem e o programa montador 39

3. O processo de "link-edição" e a carga do programa objeto na memória 54

4. Entrada e Saída e seu contrôle 70

5. Conceitos fundamentais de sistemas multiprogramados e de tempo compartilhado 88

Referências 120

Capítulo 1

Introdução e Prérequisitos

1.1. Organização e funcionamento básico dos computadores digitais

Um computador eletrônico se compõe de um certo número de subsistemas, cada um com uma função específica executada independentemente dos outros subsistemas. Um subsistema, por sua vez, se compõe de uma ou mais partes funcionais menores, que se chamam unidades.

Os principais subsistemas do computador são:

A - MEMÓRIA

B - PROCESSADOR CENTRAL

C - EQUIPAMENTOS DE ENTRADA E DE SAÍDA

D - SUBSISTEMA DE CONTRÔLE

Estes subsistemas estão esquematicamente representados na fig. 1.

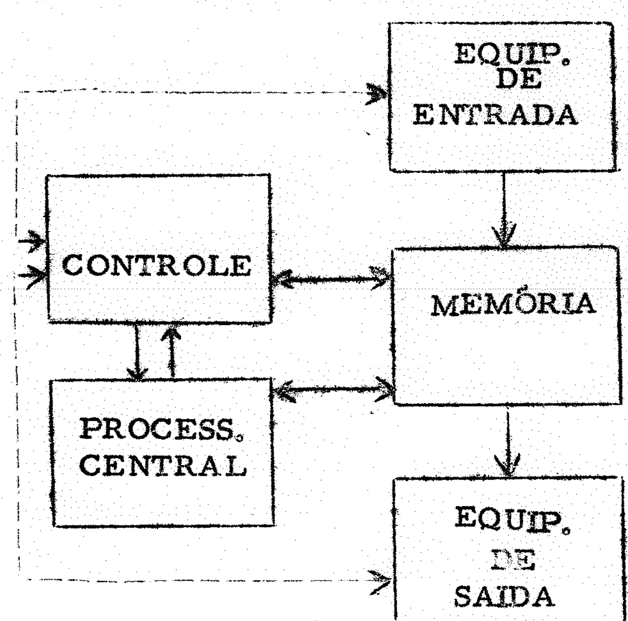


Fig. 1: Subsistemas principais do computador digital.

A - A memória tem três funções:

a) armazenamento de informações por tempo indeterminado

b) recuperação de informação armazenada (leitura da memória)

c) memorização de informação (escrita na memória)

Para executar estas funções a memória se utiliza de quatro Unidades:

I. A unidade de armazenamento, que se compõe de um grande número de registradores (usualmente de dezenas a centenas de milhares), acessíveis individualmente, chamados de células de memória. A cada célula é associado um número natural denominado de endereço da célula. Células cujos endereços diferem de 1 são denominadas de células contíguas.

II. Um registrador denominado de MAR (de "memory address register") que armazena o endereço da célula que vai ser utilizada em uma operação de leitura ou escrita executada em seguida.

III. Um registrador denominado de MDR (de "memory data register") em que é copiado a informação da célula lida numa operação de leitura, ou da qual é copiada a informação na célula escrita numa operação de escrita.

IV. Unidade de contrôle da memória, que controla e sincroniza as outras três unidades descritas.

A fig. 2 ilustra a organização da memória.

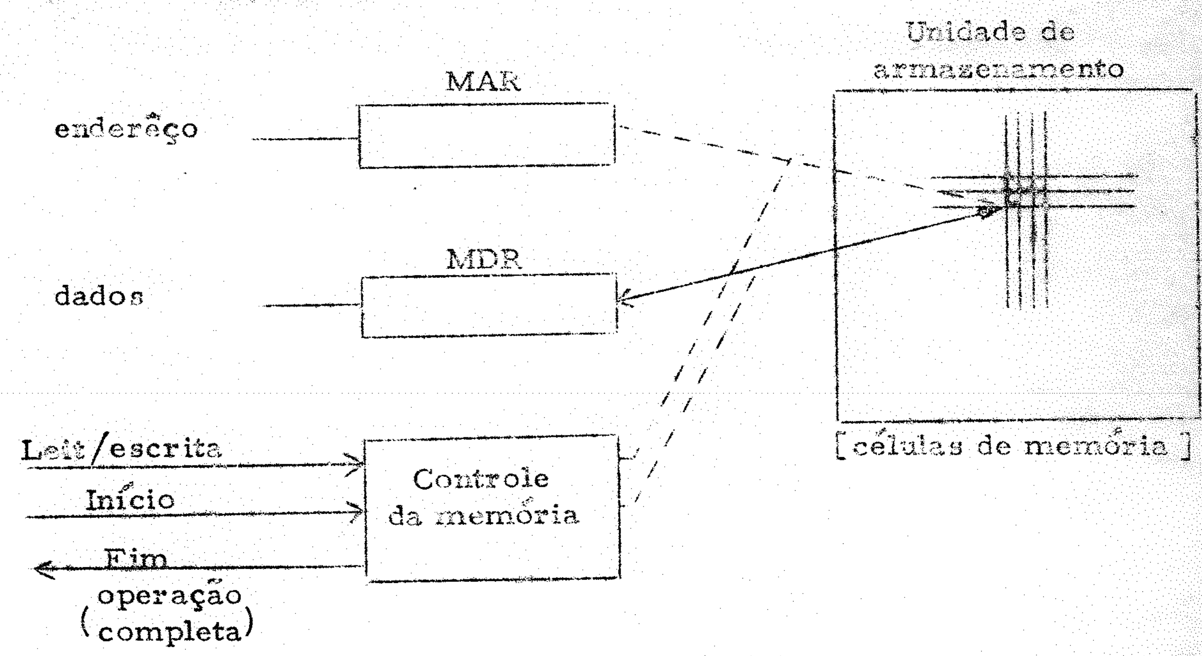


Fig. 2: A Memória

A função de armazenamento consiste em manter as informações memorizadas para eventual consulta posterior. As células são construidas de tal modo que uma vez contendo uma certa informação, são capazes de guardá-la por tempo indeterminado sem modificação até que eventualmente a informação seja alterada por uma operação de escrita nesta célula. Isto normalmente é conseguido através de núcleos magnéticos que uma vez magnetizados num sentido, ficam magnetizados neste sentido até que eventualmente se modifique o sentido de magnetização ao ser aplicado ao núcleo um campo magnético de sentido contrário.

A função de leitura da memória consiste em se obter a informação armazenada numa célula. Em cada operação de leitura pode-se obter a informação de exatamente uma célula. Esta célula é especificada através de seu enderêço. A informação armazenada numa célula se diz o conteúdo da mesma. A função de leitura não destrói o conteúdo da célula, isto é, a célula continua com o mesmo conteúdo após executada uma leitura. Para se executar uma leitura, armazena-se o enderêço da célula que se quer ler no MAR; em seguida a unidade de contrôle da memória é requisitada para uma leitura através de um pulso de leitura; é dado um pulso que inicia o processo; a unidade de contrôle localiza a célula pedida e copia o seu conteúdo no MDR; finalmente a unidade de contrôle indica que a operação foi executada através de um pulso de operação completa. A informação lida está então disponível no MDR.

A função de escrita na memória é inteiramente análoga. Consiste que se modifique o conteúdo de exatamente uma célula, especificada através de seu endereço. A informação que se quer armazenar, isto é, o que será o novo conteúdo da célula, é colocada no MDR; o endereço cujo conteúdo se quer modificar é colocado no MAR; requisita-se ao subsistema da memória uma operação de escrita através de um pulso de escrita; inicia-se o processo através de um pulso de início; a unidade de contrôle localiza a célula especificada e nela copia o conteúdo de MDR; a unidade de contrôle gera um pulso indicando que a operação foi completada. A função de escrita, como se vê, é destrutiva, no sentido de que a informação contida na célula afetada antes da operação é perdida.

B - O processador central consiste de uma unidade de contrôle e de um certo número de registradores especiais sôbre as quais se executam operações elementares. As informações que vão sofrer transformações são armazenadas nos registradores do processador; a unidade de contrôle é informada da função que deve executar; um pulso de início é dado; a unidade de contrôle do processador central executa a operação requisitada passando as informações através de blocos operadores que fazem passo a passo as transformações necessárias para a execução da função, armazenando as informações intermediárias ou o resultado nos registradores do processador; quando a função foi executada indica o fim do processo por um sinal de função completa.

C - Os equipamentos de entrada / saída tem por função a comunicação de informações entre o computador e o meio exterior através de algum meio intermediário como cartão, fita magnética, disco magnético, tambor, fita de papel, formulário contínuo, etc. Equipamentos de entrada executam transferência de informação de algum dos meios intermediários para a memória. Uma transferência assim é referida como uma <u>operação de leitura</u>, e normalmente envolve a execução de várias funções de <u>escrita</u> em células contíguas de memória. (Não confundir com a função de leitura da memória). Analogamente, equipamentos de saída executam transferência de informação da memória para algum meio intermediário.

D - Cada processador central possui um conjunto de transformações que é capaz de executar, chamadas de instruções. Cada instrução é usualmente uma operação relativamente simples, como por exemplo uma soma entre dois números inteiros. Combinando-se, porém, convenientemente estas instruções pode-se conseguir transformações mais complexas. Caracterizar

que tipo de transformações se pode conseguir através destas combinações de instruções é um assunto da teoria matemática da computação e sôbre a qual lançaremos alguma luz na secção 1.3.

Instruções são codificadas de tal modo que podem ser armazenadas em células da memória. Um conjunto ordenado de instruções armazenadas em células (em geral contíguas) da memória se chama um programa. A execução de um programa consiste na execução ordenada de suas instruções, uma atrás de outra. A ordem em que as instruções são executadas é de máxima importância. Assim é importante especificar após a execução de cada instrução qual é a instrução seguinte. Veremos mais adiante como se faz esta especificação.

A função do subsistema de contrôle é exatamente controlar a execução de um programa armazenado na memória. O subsistema de contrôle dispõe para isso dos outros subsistemas aos quais da ordens de uma maneira ordenada. Este contrôle é feito com o auxílio de quatro unidades deste subsistema:

a) Um registrador que contém a instrução a ser ou sendo executada, denominado de IR (de "Instruction Register")

b) Um registrador que contém o endereço da instrução que está sendo ou vai ser executada, denominado de IAR (de "Instruction address register")

c) O controlador, que distribui as informações para os outros subsistemas, sua sequência apropriada e supervisiona as funções executadas por outros subsistemas (iniciando estas funções com pulsos de início e recebendo pulsos de função completa)

d) O analisador, que decodifica qual a instrução que deve ser executada, isto é que determina o que deve ser feito de acôrdo com a instrução que está no IR.

A execução de cada instrução é feita em duas fases. Na primeira fase obtém-se a instrução a ser executada. ("Fetch"). Na segunda fase analisa-se a instrução que em seguida é executada.

Na primeira fase ("fetch") o conteúdo de IAR é transferido para o MAR; suponhamos para simplificar que cada instrução seja armazenada em exatamente uma célula de memória. O controlador inicia então uma função de leitura da memória; o subsistema da memória obtém então o conteúdo da célula de endereço em MAR e copia este conteúdo em MDR, indicando que a função foi completada por um pulso de função completa; o conteúdo do MDR é então transferido para o IR, terminando assim a primeira fase.

Na segunda fase o analisador manda um sinal apropriado por uma linha particular de cada campo da instrução conforme o conteúdo desta parte do IR. Um dos campos mais importantes é o código de operação da instrução, que especifica a natureza da operação a ser executada. Um decodificador ligado ao campo do IR que contém o código de operação gera um sinal característico na linha correspondente, caracterizando esta operação. Este sinal de

sencadeia as operações elementares na sequência necessária executando assim a instrução. As operações executadas dependem da particular instrução considerada. Na maioria dos casos um dos campos do IR contém um endereço, que após determinadas transformações que podem depender de outros campos do IR (tais como indexação, que veremos na secção 1.2, por exemplo) vão produzir um endereço de uma célula que no momento oportuno será lida ou escrita na memória no decorrer da execução da instrução. Os pulsos produzidos pelo analisador são coordenados pelo controlador que delega as subtarefas necessárias à execução da instrução aos demais subsistemas na sequência apropriada. Ao terminar a segunda fase da execução de uma instrução que não modifica o IAR, este é incrementado de uma célula iniciando-se então novo ciclo de "fetch" causando a execução da instrução seguinte. As instruções que não modificam durante a sua execução o conteúdo de IAR são seguidas, portanto, pela execução da instrução contida na célula imediatamente contígua. Existem, porém, algumas instruções que modificam durante sua execução o conteúdo do IAR, alterando-se, assim, a ordem de execução das instruções do programa.

1.2. O computador hipotético HIPO

Para fixar idéias vamos, a seguir, definir um computador examinando assim um conjunto de instruções típico. Apenas o essencial é descrito aqui. Informações completas podem ser obtidas em [12].

O HIPO é um computador hipotético definido com finalidades didáticas. O seu simulador foi implantado no B-3500 do CCEUSP.

A. <u>Generalidades</u> - A memória do HIPO é constituída por 1000 células de memória de tamanho fixo. Esta memória pode ser expandida se necessário até 10000 células. A célula de memória do HIPO se denomina de palavra, e é constituída de um sinal algébrico (+ ou -) e 10 algarismos decimais. As palavras são numeradas de 0000 até 0999. O número associado a uma palavra como já se viu, é o seu endereço. Leitura e escrita na memória são feitas com o auxílio dos registradores MAR e MDR, exatamente como foi descrito na secção 1.1.

O HIPO está atualmente ligado a apenas uma leitora de cartões como equipamento de entrada e a uma impressora de linha, como equipamento de saída. Nada impede, no entanto, que outros equipamentos de entrada/saída lhe sejam ligados.

O HIPO possui ainda 3 registradores, fora da memória, que são acessíveis por programa:

a) O Acumulador (indicado por A) - que é um registrador do tamanho igual ao de uma palavra da memória. Como veremos, a maioria das instruções usam o conteúdo de A podendo ou não modificá-lo.

b) A extensão do acumulador (indicado por Q) - que é um registrador de mesmo tamanho que A (isto é 10 dígitos com sinal) e que é usado independentemente ou em conjunto com A por algumas instruções.

c) O IAR, que tem a mesma função que foi explicada na secção 1.1.

B. Endereço Efetivo - As instruções do HIPO quando executadas operam sôbre dados que estão ou na memória ou em algum de seus registradores como o acumulador por exemplo. Quando a instrução se refere a um dado na memória é necessário em geral especificar qual a palavra da memória que contém o dado em questão. Evidentemente isto se faz através do endereço da palavra. Este endereço é achado por algum dos processos de endereçamento que serão descritos a seguir e se chama de endereço efetivo. O endereço efetivo é indicado por EA.

C. Formato das Instruções - Cada instrução do HIPO ocupa 1 palavra e tem o seguinte formato:

```
|+ 00CC  X IEEEE|
 S 0123  4 56789
```

onde:

CC - representa o código da instrução;
X - é a especificação de indexação;
I - é o campo de endereçamento indireto; e
EEEE - é o campo de endereço da instrução

Os campos X, I e EEEE determinam o endereço efetivo da instrução.

D. Representação de números e símbolos - O conteúdo de uma palavra do HIPO pode ser interpretado, de três maneiras diferentes: como um número em ponto fixo, como um número em ponto flutuante e como uma palavra alfanumérica. Cada instrução interpreta os dados sôbre as quais opera de uma e só uma destas maneiras.

Quando uma instrução interpreta o conteúdo da palavra como um número em ponto fixo, a palavra representa um número inteiro de dez dígitos decimais com sinal. Assim -329 seria representado na memória do HIPO como -0000000329. O número 0 (zero) pode ser representado como +0000000000 ou -0000000000, embora o resultado de uma operação aritmética em ponto fixo que dê zero seja sempre representado com sinal +.

Quando uma instrução interpreta o conteúdo da palavra como um número em ponto flutuante, a palavra representa um número fracionário decimal da forma $\pm .\, m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8 \times 10^{e_1 e_2 - 50}$. Esta notação se chama excesso cinquenta. m_1 deve ser sempre diferente de zero para números não nulos, isto é, o número deve ser normalizado. O sinal da palavra é o sinal da mantissa. O ponto é subentendido à esquerda da mantissa de 8 dígitos decimais representado pelas posições de 0 a 7 da palavra. As posições 8 e 9 representam o expoente "$e_1 e_2$" excesso 50. Os exemplos abaixo esclarecem a notação:

VALOR	Representação na memória
$2.5 = .25 \times 10^1 = .25 \times 10^{51-50}$	+ 2500000051
$-0.87 = -.87 \times 10^0 = -.87 \times 10^{50-50}$	- 8700000050
$0.0 = .0 \times 10^{0-50}$	+ 0000000000

O maior número representável é $+.99999999 \times 10^{49}$; o menor número é $-.99999999 \times 10^{49}$; e o menor número positivo representável é $+.10000000 \times 10^{-50}$.

Informação não numérica é representada no HIPO por meio de códigos numéricos. Cada caráter alfanumérico é codificado num código de dois dígitos decimais. A conversão para os códigos internos é feita automaticamente pelas instruções de entrada alfanumérica, bem como a conversão dos códigos internos para os caractéres correspondentes, pelas instruções de saída.

Fora destas intruções porém, informação codificada alfanumérica pode ser processada como se fôsse informação numérica. Os valores numéricos dos códigos dos caractéres são crescentes na ordem alfabética para as letras, o que permite por exemplo ordenar textos em ordem alfabética. (A SEQUÊNCIA DOS CÓDIGOS é a mesma do código EBCDIC). Damos a seguir a tabela de códigos alfanuméricos do HIPO.

00	(branco)	61	A	71	J	82	S	90	0
21	.	62	B	72	K	83	T	91	1
22	(	63	C	73	L	84	U	92	2
23	+	64	D	74	M	85	V	93	3
24	&	65	E	75	N	86	W	94	4
25	$	66	F	76	O	87	X	95	5
26	*	67	G	77	P	88	Y	96	6
27	)	68	H	78	Q	89	Z	97	7
28	/	69	I	79	R			98	8
29	,							99	9
31	-								
32	"								
33	=								

E. Formas de Endereçamento - Já foi dito que o enderêço efetivo para cada instrução é determinado pelos campos X I e EEEE da instrução. O HIPO utiliza cinco formas de endereçamento. Cada instrução interpreta estes campos por um dos modos abaixo descritos.

a Endereçamento direto

É o tipo mais comum de endereçamento. Neste caso os campos X e I da instrução devem conter zeros. O endereço efetivo será igual ao próprio conteúdo do campo EEEE.

b Endereçamento indexado

As palavras de endereço 0001 a 0009 da memória do HIPO têm função especial e são chamadas de indexadores. Estas palavras podem ser usadas normalmente, como as outras palavras, mas têm algumas propriedades adicionais. A mais importante destas propriedades é o seu papel no endereçamento indexado. Tôdas as instruções que permitem endereçamento direto, permitem também endereçamento indexado.

Haverá endereçamento indexado se o campo X fôr diferente de zero. Se $X \neq 0$ (e $I = 0$) o endereço efetivo EA será a soma algébrica do conteúdo do indexador X com EEEE. Assim se $X = 1$ EA = EEEE + IX1 onde IX1 é o conteúdo da palavra 0001; se $X = 2$ EA = EEEE + IX2 é o conteúdo da palavra 0002; e assim por diante. Doravante indicaremos o conteúdo da palavra n com $1 \leq n \leq 9$, como o indexador n, ou simplesmente IXn.

O endereçamento indexado é muito útil em operações com matrizes.

c Endereçamento indireto

Tôdas as instruções que permitem endereçamento direto permitem também endereçamento indireto. Haverá endereçamento indireto se o campo $I = 1$. Neste caso (supondo $X = 0$) o endereço EEEE indica não o endereço do parâmetro da instrução, isto é, o endereço efetivo EA, mas é o endereço de uma palavra cujos últimos 6 algarismos são interpretados como os novos valores de X, I e EEEE, que indicaremos para evitar confusão por X_1 I_1 e $E_1E_1E_1E_1$, com os quais se repetem os cálculos de endereçamento.

Assim se $I_1 = 0$ dizemos que o endereçamento indireto é de primeiro nível. Se $I_1 = 1$ repete-se o procedimento obtendo-se os valores de X_2 I_2 $E_2E_2E_2E_2$; se agora $I_2 = 0$ dizemos que o endereçamento indireto é de segundo nível. Como se vê teòricamente pelo menos podemos ter infinitos níveis de endereçamento indireto. Em muitos computadores isto e realmente possível. No HIPO porém, tendo em vista a sua finalidade didática o nível de endereçamentos indiretos foi limitado em 10. De qualquer modo é muito raro ser necessário mais de 2 níveis de endereçamento indireto.

Vamos resumir os tipos de endereçamento vistos até agora.

Instruções que permitem endereçamento direto calculam EA através dos campos X, I e EEEE da instrução do seguinte modo:

1. Se X = 0 fazemos A ← EEEE; senão p/ X = n ≠ 0 fazemos A ← EEEE + I Xn.

2. Se I = 0 o endereço efetivo EA é A.

3. Se I = 1 os últimos 6 digitos da palavra de endereço A são interpretados como novos valores de X, I, e EEEE voltando-se ao passo 1. para repetir os cálculos de endereçamento com os novos valores de X, I e EEEE.

d Outros tipos de endereçamento do HIPO

A maioria das instruções do HIPO utilizam um parâmetro endereçado através do procedimento acima descrito. Algumas das instruções do HIPO interpretam os campos X, I e EEEE de outra forma. Assim as instruções 29, 40, 50, e 70 ignoram o conteúdo destes campos (vide descrição das instruções - item F); e as instruções 61 62 63 64 e 65 são imediatas. Dizemos que uma instrução é imediata se o campo EEEE não é um endereço, mas já é o valor do parâmetro utilizado pela instrução. Nestas instruções I é ignorado. Nas instruções 61 a 64 se X ≠ 0 o valor do parâmetro é dado pelos 2 últimos algarismos do indexador X.

F. Instruções do HIPO

Para completar nossa discussão vamos descrever as instruções do HIPO. Na descrição das instruções indicaremos o conteúdo da palavra de endereço EA por E; o acumulador por A; a extensão do acumulador por Q. Quando a palavra de endereço EA é interpretado como número em ponto flutuante, indicaremos o seu conteúdo flutuante como Fl (E); analogamente Fl (A) indica conteúdo flutuante de A. A flecha deve ser lida "é substituido por". Nas instruções em que A e Q formam um só registrador (com os algarismos de Q à direita dos de A, e prevalecendo o sinal de A) indicamos este registrador de tamanho duplo por AQ. Sempre que não é especificado o contrário, A, Q e E são interpretados como números inteiros.

1. Instruções de transferência

Código da operação	Código mnemônico	(significado)	Operação
11	LDA	(Load Acumulador)	A ← E
12	STA	(Store Acumulador)	EA ← A
13	LDQ	(Load Extension)	Q ← E
14	STQ	(Store Extension)	EA ← Q

2. Instruções de máscara

Nas instruções de máscara E é interpretado como uma "máscara" de 10 algarismos que são processados dígito a dígito com o algarismo correspondente (ie de mesma posição relativa) de A. Na descrição das instruções de máscara indicaremos por D um dígito do acumulador e por M o dígito correspondente da máscara.

15 LZR (Load zeros) carrega zeros no acumulador nas posições em que M é zero os outros dígitos de A não são alterados:

$$\begin{cases} \text{se } M \neq 0 & D \text{ não é alterado} \\ \text{se } M = 0 & D \leftarrow 0 \end{cases}$$

16 LDG (Load digits) carrega os dígitos não nulos da máscara nas posições correspondentes de A:

$$\begin{cases} \text{se } M \neq 0 & D \leftarrow M \\ \text{se } M = 0 & D \text{ não é alterado} \end{cases}$$

3. Instruções aritméticas

21	ADD (Add)	$A \leftarrow A + E$
22	SUB (Subtract)	$A \leftarrow A - E$
23	MPY (Multiply)	$AQ \leftarrow A \times E$; os sinais de A e Q ficam iguais ao sinal do produto.
24	DIV (Divide)	$A \leftarrow AQ/E$ $Q \leftarrow$ resto da divisão
25	FAD (Floating add)	$A \leftarrow Fl(A) + Fl(E)$
26	FSU (Floating Subtract)	$A \leftarrow Fl(A) - Fl(E)$
27	FMP (Floating Multiply)	$A \leftarrow Fl(A) \times Fl(E)$
28	FDV (Floating Divide)	$A \leftarrow Fl(A) / Fl(E)$
29	RVS (Reverse Sign)	$A \leftarrow -A$

Nas instruções aritméticas é possível acontecer que o resultado de uma operação aritmética não possa ser representado numa palavra do HIPO, porque o resultado seja maior do que o maior número representável (+ 9999999999), ou menor do que o menor número representável (-9999999999), inteiro ou de ponto flutuante.

Neste caso dizemos que ocorreu uma sobrecarga ("Overflow"), e é armazenado respectivamente o maior ou menor número representável, inteiro ou flutuante conforme o caso. No caso de operações flutuantes, é possível também que o resultado da operação seja menor, (em valor absoluto), do que o menor número positivo representável. Dizemos então que ocorreu um "underflow", e é armazenado + 0000000000.

O Hipo possui um indicador especial indicado por OF, que é ligado em qualquer dos casos acima. OF é desligado apenas pela instrução 58 (vide abaixo).

3. Instruções de entrada

31.	RNW	(Read numerical word)	Transmite a palavra perfurada nas primeiras 11 colunas de um cartão para EA. O sinal deve vir perfurado na coluna 1 podendo se usar ou & no lugar de +. Brancos nas colunas 2 a 11 são lidos como se fôssem zeros.
32.	RNC	(Read numerical card)	Lê em EA, EA + 1, ..., EA + 6. 7 palavras perfuradas nas colunas 1 a 77. A primeira palavra é perfurada nas colunas 1 a 11, a segunda de 2 a 22, ... a sétima de 71 a 77. Valem as convenções descritas na instrução 31.
35.	RAW	(Read alphanumeric word)	Transmite o conteúdo alfanumérico perfurado nas colunas 1 a 5 de um cartão, para EA. Cada caráter é automaticamente convertido para o código correspondente descrito no item D. O sinal de EA se torna +.
36	RAC	(Read alphanumeric card)	Lê em EA, EA + 1, ..., EA + 15, 16 palavras perfuradas alfanumèricamente nas colunas 1 a 80 de um cartão. Valem as convenções descritas na instrução 35.

4. Instruções de saída

40.	DMP	(Dump)	Imprime pela impressora o conteúdo de tôda a memória e encerra o processamento deste programa entrando em estado de carga. (vide abaixo)
41	PNW	(Print numerical word)	Imprime numèricamente a palavra em EA, nas posições 1 a 11 da impressora.

42	PNL	(Print numerical line)	Imprime numèricamente o conteúdo do numérico de 10 palavras contíguas (EA, EA + 1, ..., EA + 9) nas posições 1 a 119 da impressora, deixando um branco entre 2 palavras consecutivas.
45	PAW	(Print alphanumeric word)	Imprime alfanumèricamente a palavra de enderêço EA nas posições 1 a 5 da impressora. O sinal da palavra é ignorado. Cada caráter deve estar na memória no código do item D.
46.	PAL	(Print alphanumeric line)	Imprime alfanumèricamente 24 palavras de endereços EA, EA + + 1, ..., EA + 23, nas posições 1 a 120 da impressora valem as convenções descritas na instrução 45.

5. **Instruções de desvio**

São as instruções que alteram o IAR modificando, assim, eventualmente, a ordem de execução das instruções.

50.	NOP	(No operation)	IAR ← IAR + 1
51	BRN	(Branch)	IAR ← EA
52	BNP	(Branch no positive)	Se A ≤ 0, IAR ← EA; senão IAR ← IAR + 1
53	BNZ	(Branch no zero)	se A ≠ 0, IAR ← EA, senão IAR ← IAR + 1
54	BPS	(Branch positive)	se A > 0, IAR ← EA; senão IAR ← IAR + 1
55	BZR	(Branch on zero)	se A = 0, IAR ← EA; senão IAR ← IAR + 1
56	BNG	(Branch negative)	se A < 0, IAR ← EA; senão IAR ← IAR + 1
57	BNN	(Branch no negative)	se A ≥ 0, IAR ← EA; senão IAR ← IAR + 1.
58	BOF	(Branch on overflow)	se OF = 1, IAR ← EA; senão IAR ← IAR + 1. OF ← 0.
59	BST	(Branch and stone)	EA ← IAR + 1 (palavra seguinte à instrução BST sendo executada) IAR ← EA + 1.

6. Instruções de deslocamento

Nestas instruções o número de deslocamentos de dígitos do acumulador se X = 0 é dado por EE (duas últimas posições de EEEE), ou se X ≠ 0 pelos 2 últimos algarismos do indexador X. I é ignorado. O preenchimento de casas é feito com zeros. Os sinais de A e de Q não é alterado.

61	SLA	(Shift left acumulator)	Desloca A para a esquerda
62	SRA	(Shift right acumulator)	Desloca A para a direita
63	SLQ	(Shift left acumulador and extension)	Desloca AQ para a esquerda
64	SRQ	(Shift right acumulator and extension)	Desloca AQ para a direita

7. Instruções misceláneas

65	MDX	(Modify index and skip)	Soma em ponto fixo o número EEEE (positivo) ao indexador X e pula a próxima instrução (IAR ← IAR + 2) se o resultado da soma fôr nulo ou com sinal oposto ao do conteúdo original do indexador. I é ignorado.
70	STP	(Stop)	O HIPO entra em estado de carga. (veja item G)

G. Estado de carga e estado normal

Todos os computadores têm projetado algum recurso para poder inicializar a sua operação, isto é carregar algumas instruções na memória para poder iniciar a execução de um programa. Em geral estas instruções carregam o resto do programa na memória para depois iniciar a sua execução. No HIPO a carga de programas é altamente facilitada. O HIPO possui dois estados de funcionamento. No estado normal o HIPO está sob controle do programa em sua memória, conforme já foi descrito nas secções anteriores. No estado de carga, porém, o HIPO carrega automàticamente em sua memória o programa em cartões em determinado formato.

Para maiores informações veja [12]

1.3. Algorítmos e funções computáveis.

Nas secções precedentes é descrito o funcionamento básico dos computadores e foi definido um computador específico (o HIPO).

O objetivo principal dêste trabalho é estudar certos programas que são referidos normalmente por sistema operacional. Ao explicar as técnicas e funcionamento dêstes programas precisaremos descrevê-los de algum modo. Seguiremos nesta descrição, em geral, a notação de algorítmos proposta por Knuth em [10]. Nesta secção explicaremos esta notação. Contudo, esta secção tem ainda uma outra finalidade. Mesmo quem já tenha certa experiência no uso e programação de vários computadores achará natural indagar certas questões sôbre a natureza destas máquinas como:

1 - Será que todo procedimento descrito na notação mencionada pode ser programado em qualquer computador?

2 - Qual é a classe de problemas que podem ser programados? Esta classe de problemas varia de computador para computador?

3 - Tendo em vista as perguntas 1 e 2, tem sentido afirmar que as instruções do HIPO são "típicas"? Isto significa de algum modo que as diferenças entre os diversos computadores não são essenciais no que concerne aos problemas que neles podem ser processados? etc.

Estas questões levam naturalmente a um exame mais minucioso de o que é um programa e é isto que discutiremos resumidamente nesta secção.

Uma discussão excelente destas noções pode ser encontrada em [10]. Nós repetiremos aqui os pontos mais importantes daquela discussão por conveniência. Começaremos por examinar a noção de procedimento, ou método computacional. Na secção anterior examinamos o procedimento que o HIPO segue para achar o endereço efetivo do parâmetro da maioria de suas instruções:

Dada uma instrução do HIPO |+ 0 0 C C X I E E E E| já sabemos
S 0 1 2 3 4 5 6 7 8 9

que entram no cálculo do endereço efetivo EA, os valores de três variáveis de nomes X, I e EEEE. O valor inicial destas variáveis encontra-se na palavra de endereço IAR. X é o valor do quarto algarismo, I o valor do quinto, e EEEE, o valor do campo que vai do sexto algarismo até o nono algarismo da palavra em questão. Quando queremos definir um campo de uma palavra do HIPO usaremos a notação: endereço [a_1 : a_2], onde "endereço" é o endereço da palavra em questão; a_1 é o número do primeiro algarismo do campo; e a_2 é o número de algarismos do campo. Assim 0032 [2: 5] significa o campo que vai do segundo algarismo até o sexto algarismo da palavra 0032. Com esta notação X, I e EEEE são os campos IAR [4: 1], IAR [5: 1], e IAR [6: 4], CONT (endereço) significa o conteúdo da palavra de endereço "endereço". Assim CONT (0318) é o conteúdo algébrico da palavra 0318. Tendo em vista estas convenções a descrição do procedimento fica:

Procedimento A.

A1. [Inicialize X, I, e EEEE] X ← IAR [4: 1], I ← IAR [5: 1],
EEEE ← IAR [6: 4].

A2. [Ache EA inicial.] Faça EA ← EEEE.

A3. [É indexada?] Se X ≠ 0 seja EA ← EA + CONT (X)

A4. [Não é indireto?] Se I = 0 o procedimento termina. EA é o endereço efetivo.

A5. [Ache novos X, I, EEEE] (É indireto) X ← EA [4 : 1], I ← EA [5: 1], EEEE ← EA [6: 4] e vá para A2.

Examinemos a notação empregada para a descrição do procedimento. Para a descrição de um procedimento daremos uma letra de identificação (A no caso acima), e cada passo do procedimento é identificado por esta letra seguida do número do passo (A1, A2, etc.)

Cada passo começa por uma frase curta entre colchetes. Esta frase resume o conteúdo do passo. Se a descrição do procedimento vier acompanhada de um "diagrama de blocos" (v. fig. 3), esta frase aparece no bloco correspondente ao passo. Seguindo a frase entre colchetes, vem a descrição precisa do passo. Às vêzes há também um comentário entre parênteses, que não faz parte da descrição pròpriamente dita, mas é incluído como uma explicação adicional sôbre o passo. (v. no passo A5.) A flecha "←" indica uma operação importante, que denominaremos de operação de designação ou de substituição. "←" deve ser lido "é substituido por", e significa que o valor da variável à esquerda da flecha deve ser substituído pelo valor da expressão à direita da flecha, calculada com os valores presentes das variáveis que nela comparecem. Assim EA ← EA + CONT (X) por exemplo (passo A3) significa que o valor de EA depois da execução da operação de substituição será o valor presente de EA (ié antes da substituição) somado ao valor presente do indexador X. Quando um passo (como o passo A3.) começa com uma condição, o restante do passo só é aplicável se a condição fôr satisfeita. Assim, no caso do passo A3, se X = 0 não é somado a EA o valor do indexador X mas continua-se o procedimento no passo A4. O procedimento começa no passo de menor número e continua em ordem crescente dos passos, a não ser que o passo especifique claramente a ordem da computação como no passo A5. por "vá para A2."

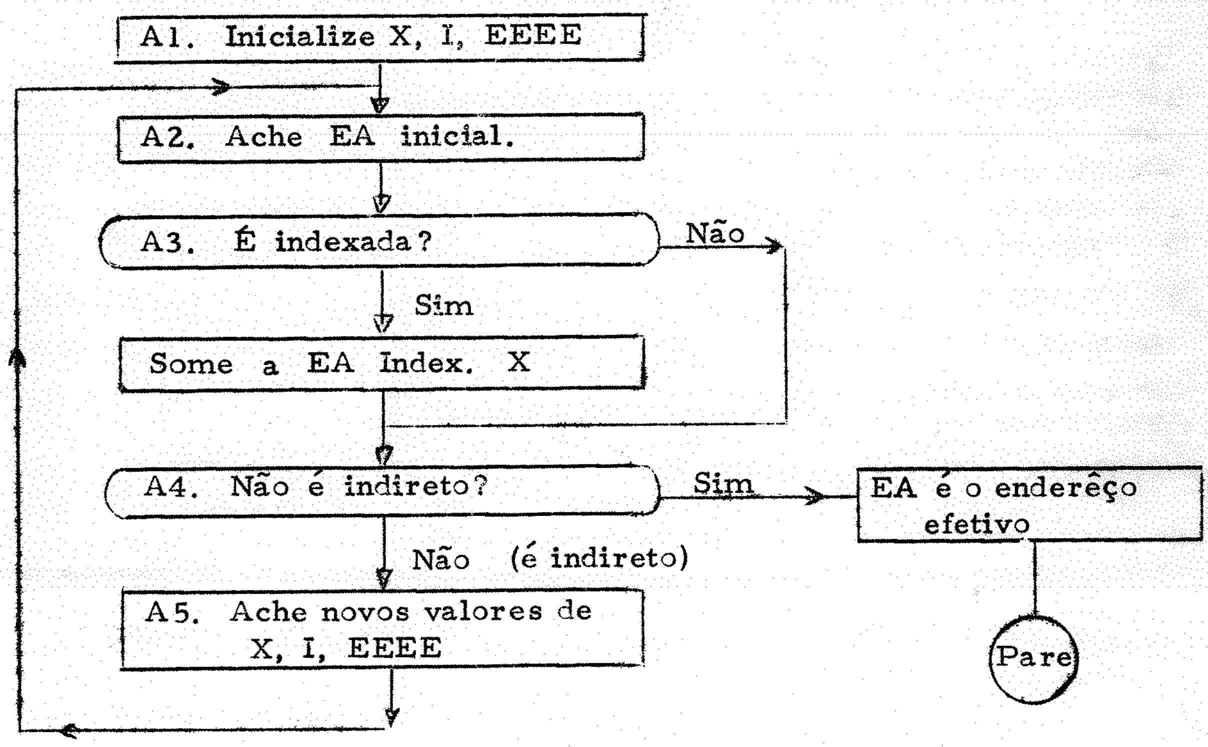


Fig. 3. Diagrama de blocos do procedimento A.

Um procedimento é então um conjunto finito de passos que descrevem uma sequência de operações que resolvem um tipo específico de problema. Estes passos devem porém satisfazer às quatro condições seguintes.

i - <u>Definição precisa</u> - Cada passo do procedimento deve estar precisamente definido, de modo que as operações possam ser executadas rigorosamente e sem ambiguidade

ii - <u>Entrada</u> - Um procedimento tem zero ou mais entradas, i.é valores iniciais que são dadas a algumas ou todas as suas variáveis, antes de começar a execução do procedimento. Estas entradas devem ser tomadas de um conjunto bem especificado de elementos. Por exemplo no procedimento A. a entrada é IAR, que deve ser tomado do conjunto de endereços do HIPO, (portanto um número inteiro não negativo menor do que 1000 no nosso caso), e, implicitamente o conjunto dos conteúdos de todas as palavras do HIPO

iii - <u>Saída</u> - Um procedimento tem uma ou mais saídas, i.é valores com uma relação especificada com as entradas, que o procedimento fornece quando termina. O procedimento A. tem por saída EA, que é o endereço efetivo da instrução que está na palavra de endereço IAR.

iv - <u>Efetividade</u> - Um procedimento deve ser efetivo, no sentido de que todas as operações do procedimento devem ser suficientemente básicas para que em princípio possam ser executadas em tempo finito e de maneira exata por um homem usando apenas lápis e papel. As operações do procedimento A. são: a operação de substituição, comparação de um número inteiro com zero e soma de números inteiros. Todas estas operações são efetivas. Consideremos algumas operações não efetivas. Por exemplo "Expresse, por meio de funções elementares, a integral $\int_0^X e^{-z^2} dz$ " não é efetiva porque não se conhece um método finito para exprimir esta integral por funções elementares. Em geral o processo de integração de uma função contínua não é efetiva, pois embora demonstre-se que a integral de qualquer função contínua exista, não conhecemos, no caso geral, um método para acharmos uma expressão explícita, por meio de funções elementares por exemplo, mesmo que tal expressão exista. (em contraste, o processo de integração de funções racionais em forma fatorada, é efetivo, bem como o processo de derivação das funções elementares) É fácil dar, portanto, exemplos de operações não efetivas. É possível dar mesmo, operações bem especificadas, mas que não sabemos sequer se são ou não efetivas. Por exemplo: " Se n não é o maior número perfeito então vá para o passo 5." É fácil construir um procedimento que após um número finito de passos determine se um número inteiro positivo é ou não um número perfeito. Portanto, se testarmos sucessivamente os números inteiros n + 1, n + 2, ... podemos determinar se existe um número perfeito maior que n. Como não sabemos, porém, se existem infinitos números perfeitos ou não, não sabemos se este passo é ou não efetivo.

Um algorítmo, é um procedimento que satisfaz mais uma condição, além das quatro condições já estipuladas:

v - Finitude - Um algorítmo sempre termina após um número finito de passos. O procedimento A. não é, portanto, um algorítmo pois para certas entradas, i.e. para certos valores de IAR, e certas configurações de memória, o procedimento nunca termina. (São as configurações que corresponderiam a infinitos níveis de endereçamento indireto) Por exemplo para IAR = 0100 e CONT (0100) = + 0011010100 o procedimento A. nunca termina. (Embora no caso real, mais de 10 níveis de endereçamento indireto, no HIPO, causariam uma mensagem de erro; mas isto acontece porque o procedimento seguido pelo HIPO para a determinação do endereço efetivo não é exatamente o procedimento A.)

A definição acima de procedimento e algorítmo não é evidentemente uma definição matemática precisa, mas foi não obstante até muito recentemente a única definição disponível destes conceitos. Ela é suficientemente precisa sempre que exibirmos um algorítmo, (ou procedimento), que resolva um determinado problema, mas não é suficientemente precisa para demonstrar, por exemplo, que não existe um algorítmo para um certo problema, ou demonstrar a existência de um algorítmo, mas sem exibí-lo. É óbvio, portanto, que para se chegar a resultados deste tipo é essencial formular uma definição mais precisa, uma definição, por assim dizer, mais "manejável" matematicamente, e que no entanto conservasse as propriedades enunciadas.

Não se pode esperar no entanto uma demonstração rigorosa de que o conceito formalmente definido seja equivalente ao conceito um tanto vago mas "intuitivo" de procedimento no sentido visto acima. Dedicaremos o restante desta secção à exposição de uma definição precisa de procedimento e algorítmo.

Formalmente um método computacional ou procedimento, é uma quádrupla ordenada (Q, E, Σ, f), em que Q é um conjunto, que representa os estados da computação, E é o conjunto das entradas, sendo $E \subset Q$, e Σ o conjunto das saídas, sendo $\Sigma \subset Q$, e f é uma função de Q em Q $(f: Q \rightarrow Q)$ tal que se $q \in \Sigma$ $f(q) = q$, chamada de regra computacional. Cada entrada $X \in E$ define uma sequência computacional $X_0, X_1 \ldots X_n \ldots$ por:

$$X_0 = X$$

$$X_{n+1} = f(X_n) \quad \text{para} \quad n \geq 0$$

Dizemos que a sequência computacional termina em k passos sse (se e somente se) k é o menor inteiro tal que $X_k \in \Sigma$, (e neste caso será evidentemente $X_j = X_k$ para todo $j \geq (k)$; em tal caso diremos que X produz a saída X_k pelo procedimento. Se para qualquer X em E sempre existe um k tal que $X_k \in \Sigma$ então o procedimento é um algorítmo.

Para formalizar nestes termos o procedimento A. por exemplo, podemos tomar

Σ = conjunto de endereços admissíveis do HIPO
(ié no nosso caso, o conjunto dos inteiros não negativos menores que 1000);

M = conjunto de configurações da memória do HIPO, (ié o conjunto das 1000-uplas $(Z_0 \quad Z_1 \ldots Z_{999})$ onde Z_i é um número inteiro tal que $|Z_i| \leq 9999999999$);

$E = \Sigma \times M$;

C = conjunto das sétuplas da forma

(IAR, Z, EA, X, I, EEEE, n) onde IAR, EA $\in \Sigma$, $Z \in M$, e X, I, EEEE, n são inteiros tais que $0 \leq X \leq 9$, $0 \leq I \leq 9$, $0 \leq EEEE \leq 9999$ e $1 \leq n \leq 5$;

$Q = E \cup \Sigma \cup C$.

Sejam $g_X: E \to Z^+$ (Z^+ = conjunto dos inteiros não negativos)

$g_I: E \to Z^+$

$g_{EEEE}: E \to Z^+$ definidos por

g_X (END, Z) = X $\quad g_I$ (END, Z) = I e g_{EEEE} (END, Z) = EEEE

onde $Z = (Z_0, Z_1, \ldots Z_{END}, \ldots Z_{999}) \in M$; END $\in \Sigma$ e

$Z_{END} = s \times (P \times 10^6 + X \times 10^5 + I \times 10^4 + EEEE)$ com $s = \pm 1$ e

P, X, I, EEEE $\in Z^+$ sendo P, EEEE ≤ 9999; X, I ≤ 9.

(portanto, as funções g_X g_I e g_{EEEE} definem respectivamente os campos END [4:1], END [5:1] e END [6:4]) definamos finalmente a regra computacional por:

f (EA) = EA para EA $\in \Sigma$;

f (IAR, Z) = (IAR, Z, 0, 0, 0, 0, 1) para (IAR, Z) $\in$ E;

f (IAR, Z, EA, X, I, EEEE, 1) = (IAR, Z, EA, g_X (IAR, Z), g_I (IAR, Z), g_{EEEE}(IAR, Z), 2)

veja correspondência com o passo A1.)

f (IAR, Z, EA, X, I, EEEE, 2) = (IAR, Z, EEEE, X, I, EEEE, 3)

f (IAR, Z, EA, 0, I, EEEE, 3) = (IAR, Z, EA, 0, I, EEEE, 4)

f (IAR, Z, EA, X, I, EEEE, 3) = (IAR, Z, EA + Z_X, X, I, EEEE,4) para $X \neq 0$

f (IAR, Z, EA, X, 0, EEEE, 4) = E A

f (IAR, Z, EA, X, I, EEEE, 4) = (IAR, Z, EA, X, I, EEEE, 5) para $I \neq 0$

$$f\,(IAR,\ Z,\ EA,\ X,\ I,\ EEEE,\ 5) = (IAR,\ Z,\ EA,\ g_X\,(EA,\ Z),$$
$$g_I\,(EA,\ Z),\ g_{EEEE}(EA,\ Z),\ 2)$$

A correspondência entre o procedimento formal acima e o procedimento A. é evidente.

A definição formal acima equivale às condições i, ii e iii pois a regra computacional é de definição precisa, e é razoável supor que qualquer passo de procedimento precisamente definido possa ser formalizado em têrmos de função e, portanto, incluido numa regra computacional conveniente; a relação entre ii e E e entre iii e Σ é óbvia. Observemos apenas que em iii impusemos pelo menos uma saída, o que é uma condição conveniente do ponto de vista prático mas não essencial. De qualquer modo, podemos incluir esta condição na definição formal impondo $\Sigma \neq \emptyset$. A condição de finitude, (v) para algotítmos claramente é equivalente à condição da existência de um número inteiro K tal que $X_K \in \Sigma$ para qualquer entrada. A condição de efetividade (iv) não está incluída na definição formal acima, pois nada impede que os elementos de E não possam ser objetos não representáveis de modo finito no papel, como por exemplo números reais, e a definição de regra computacional é também geral demais, de modo que nada impede que inclua transformações não efetivas. Para incluir, portanto, também a noção de efetividade, é necessário restringir Q, E, Σ e f. Esta é a condição cuja formalização é a mais difícil, pois que em última análise é preciso interpretar e formalizar a noção um tanto vaga de o que pode um homem fazer com lápis e papel. As restrições que iremos fazer estão de acôrdo com [13].

Quando executamos um procedimento usando lápis e papel registramos os valores das variáveis no decorrer da computação no papel. Para isto utilizamos símbolos com significados previamente convencionados como letras, algarismos, símbolos de operações, ou outros caractéres. Um procedimento utiliza usualmente variáveis que podem ser de natureza muito variada, como números inteiros, frações, polinômios, fórmulas do cálculo de predicados, tabuleiro e peças de xadrez, etc. Quando os valores destas variáveis são representadas os símbolos são utilizados frequentemente em arranjos bidimensionais no papel como por exemplo " $\frac{n}{n}$ " no caso de frações, ou representando uma potência como em " x^n ", ou no caso de uma matriz como em $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

É porém fácil ver que com convenções apropriadas podemos representar os mesmos objetos em arranjos lineares unidimensionais. Assim por exemplo nos casos mencionados podemos utilizar "(m/n)" no caso de frações, ou "(x ↑ n)" no caso de potências, ou "a, b ## c, d " no caso de matrizes. Com separadores apropriados podemos representar um número arbitrário de objetos de natureza diversa. É portanto razoável admitir, que não se perde generalidade alguma se restringirmos os objetos de um procedimento a cadeias unidimensionais de caractéres. Assim seja A um conjunto finito de símbolos, que denominaremos de alfabeto e seja A^* o conjunto de sentenças de A que definiremos como o conjunto das sequências finitas de símbo-

los de A : $A^* = \{a_1 a_2 \ldots a_n \mid , a_j \in A$ para $1 \leq j \leq n, n \in Z^+\}$. Um procedimento transforma, portanto, em passos sucessivos, uma sentença de A^* em outra que constitui a saída do procedimento.

Seja $\alpha \in A^*$. Nós dizemos que θ ocorre em α sse existirem $\gamma, \delta \in A^*$ tal que $\alpha = \gamma\, \theta\, \delta$. Por exemplo se $A = \{a, b, c\}$ então "ac" ocorre em "bbcbacb", ocorre 2 vêzes em "acbacb" e não ocorre em "cabc". Cada passo do procedimento transforma uma sentença em outra através de certas regras. Nós vamos chamar de um passo elementar, que indicaremos por $\theta \rightarrow \sigma$, onde $\theta, \sigma \in A^*$, um passo que aplicado a $\alpha \in A^*$, substitui a primeira ocorrência de θ por σ. (ie que transforma $\gamma\, \theta\, \delta$ em $\gamma\, \sigma\, \delta$, sendo que θ não ocorre em γ). Se θ não ocorre em α dizemos que o passo elementar $\theta \rightarrow \sigma$ não é aplicável a α. Claramente um passo elementar é efetivo. Nós vamos admitir porém que qualquer passo efetivo possa ser reduzido a aplicações sucessivas de passos elementares. Imporemos, portanto, que cada passo do procedimento seja um passo elementar. Mais ainda, admitiremos que sempre seja possível a especificar um procedimento através de uma lista finita ordenada de passos elementares aplicados do seguinte modo:

Em cada estágio da computação aplicamos o primeiro passo elementar aplicável à sentença atual da computação. Uma execução de um procedimento especificado nesta forma significa aplicar repetidas vêzes a regra precedente até que eventualmente nenhum dos passos elementares da lista possa ser aplicada, ou até que apliquemos algum passo elementar "final". Passos elementares finais são passos elementares prèviamente designados como "finais", o que significa simplesmente que se aplicados encerram a computação. Estas idéias podem ser formalizadas em têrmos da quádrupla (Q, E, Σ, f) do seguinte modo: Dado um alfabeto A tomaremos Q o conjunto de pares ordenados da forma (α, j) onde $\alpha \in A^*$ e $0 \leq j \leq n$, onde $n, j \in Z^+$ (inteiros não negativos). E será o conjunto de pares da forma $(\alpha, 0)$ e Σ da forma (α, n). Dados ainda os passos elementares:

$$\begin{array}{ccc} \theta_0 & \rightarrow & \phi_0 \\ \theta_1 & \rightarrow & \phi_1 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \theta_{n-1} & \rightarrow & \phi_{n-1} \end{array}$$

definiremos a regra computacional f por:

$f(\alpha, n) = (\alpha, n)$; e p/ $j < n$ $f(\alpha, j) = (\alpha, j+1)$ se θ_j não ocorre em α; e se $\alpha = \gamma\, \theta_j\, \delta$ e θ_j não ocorre em γ, $f(\alpha, j) = (\gamma\, \phi_j\, \delta, n)$ se $\theta_j \rightarrow \phi_j$ é um passo elementar final, ou

$$f(\alpha, j) = (\gamma\, \phi_j\, \delta, 0) \text{ em caso contrário.}$$

Fica assim definido procedimento (algorítmo) efetivo. Mais uma vez enfatizamos que a argumentação acima não constitui uma prova da equivalência do conceito formal com o conceito "intuitivo", mesmo porque já dissemos que tal prova é impossível. A experiência entretanto mostra que os algorítmos conhecidos podem ser formalizados nos têrmos propostos. Mais ainda, vários autores como Church, Post, Turing, definiram por caminhos independentes o conceito de procedimento efetivo, pretendendo cada um caracterizar a classe de funções mais geral possível, computáveis através de um procedimento efetivo. Estas definições no entanto se mostraram equivalentes.

Para encerrar esta secção vamos voltar agora a uma rápida discussão das questões enunciadas no início desta secção, à luz do que foi exposto até aqui.

Cada instrução do HIPO foi descrita na notação vista nesta secção e é claramente efetiva. Qualquer programa HIPO portanto é um procedimento efetivo. Abstraindo-se da limitação de memória do HIPO, (que, em princípio, pode ser eliminada "ligando-se" ao HIPO unidades de disco magnético por exemplo), qualquer procedimento efetivo formal pode ser programado no HIPO. Admitida portanto a equivalência do conceito formal com o conceito "intuitivo", qualquer procedimento pode ser programado no HIPO. Uma instrução qualquer de um computador digital qualquer nos moldes descritos na secção 1.1 é uma função cujo valor depende apenas do conteúdo de um certo número de células de memória, isto é, o domínio da função é finito. Isto significa que qualquer instrução é efetiva e que portanto qualquer programa em qualquer computador é um procedimento. Se as instruções do computador são suficientemente poderosas para que passos elementares possam ser executados no sentido visto, então qualquer procedimento pode ser programado neste computador. Em particular, cada instrução de um computador A pode ser encarada como um procedimento e portanto pode ser programada num computador B, o que atesta a possibilidade de um computador A qualquer poder ser simulado num computador B. Neste sentido, portanto, computadores são máquinas universais.

1.4. Subrotinas

Frequentemente é conveniente construir um novo algorítmo combinando dois algorítmos no seguinte sentido: para uma certa entrada aplicamos inicialmente o primeiro algorítmo, e então tomamos a saída dêste como entrada do segundo, e aplicamos o segundo algorítmo. Anàlogamente poderíamos considerar a combinação de n algorítmos $a_1\ a_2 \ldots a_n$. Esta idéia corresponde à composição de funções, e o algorítmo resultante da combinação de algorítmos no sentido descrito se chama composição dos algorítmos dados. Consideremos a composição de $a_1\ a_2 \ldots a_n$. Se m dêstes algorítmos $(1 \leq m \leq n)$ forem iguais, ié $a_{i_1} = a_{i_2} = a_{i_m} = \beta$ então diremos que β é uma subrotina da composição a de $a_1, a_2 \ldots a_n$. Imaginemos agora o programa que corresponde ao algorítmo composição a. Se as instruções que correspondem a β são repetidas m vêzes no programa, correspondendo respectivamente a $a_{i_1}, a_{i_2}, \ldots a_{i_m}$, então diremos que β é uma subrotina aberta. Se as instruções que correspondem a β apare-

cem uma só vez no programa, e cada vez que β deve ser executado desviamos para β retornando depois da execução de β ao passo seguinte em que chamamos β, então diremos que β é uma subrotina fechada. Quando dissermos simplesmente <u>subrotina</u>, entenderemos implicitamente <u>subrotina</u> fechada. Especialmente quando a β correspondem muitas instruções o uso de subrotinas fechadas é vantajoso porque em cada ponto do programa em que é necessária a execução de β só são incluidas algumas poucas instruções necessárias para transferir o contrôle do programa a β o que acarreta considerável economia de memória.

A vantagem do uso de subrotinas não reside apenas na economia de memória. Elas possibilitam a visualização fácil da estrutura de um algorítmo complexo em função de algorítmos mais simples o que facilita em geral o projeto de programas complexos; reduzem a tarefa de programação pois programando-se uma vez as subrotinas "básicas" de um certo processo complexo, podemos programar o processo compondo convenientemente as subrotinas já programadas; facilitam o teste de tais programas complexos pois cada subrotina pode ser testada individualmente independentemente das demais; possibilitam a formação de bibliotecas de subrotinas, etc. O emprêgo de subrotinas é na realidade tão importante que a maioria dos computadores possuem instruções especialmente projetadas para facilitar o seu uso.

Como subrotinas são executadas em geral várias vêzes num programa e em pontos independentes, a entrada e a saída da subrotina, em cada execução podem ser distintas. É necessário portanto, em cada caso especificar quais as variáveis que serão a entrada da subrotina em cada execução, e quais as variáveis em que queremos armazenar as saídas da subrotina. Entradas e saídas da subrotina serão por nós designadas por parâmetros. Quando queremos portanto desviar para uma subrotina, (diz-se: chamar a subrotina) devemos especificar quais os valores presentes de seus parâmetros. Além de especificar os parâmetros é necessário transmitir à subrotina qual o ponto do programa para onde a subrotina deve desviar após a sua execução, já que se a subrotina fôr fechada êste ponto dependerá de onde ela foi chamada. O método pelo qual se desvia para a subrotina chama-se ligação com a subrotina.

Uma subrotina pode chamar por sua vez outras subrotinas. Isto se chama de encadeamento de subrotinas. Chamamos de nível do encadeamento o número máximo de subrotinas encadeadas. O nível de encadeamento pode em geral ser arbitràriamente grande, mas frequentemente exige-se que tôdas as subrotinas encadeadas seja distintas, isto é, em qualquer ponto do encadeamento só podemos chamar uma subrotina que ainda não foi encadeada. Se esta restrição fôr levantada, isto é, se fôr permitido que uma subrotina se chame a si própria, (diretamente ou indiretamente através do encadeamento com outras subrotinas) diz-se que a subrotina em questão é recursiva. A ligação com subrotinas recursivas exige técnicas especiais como veremos logo.

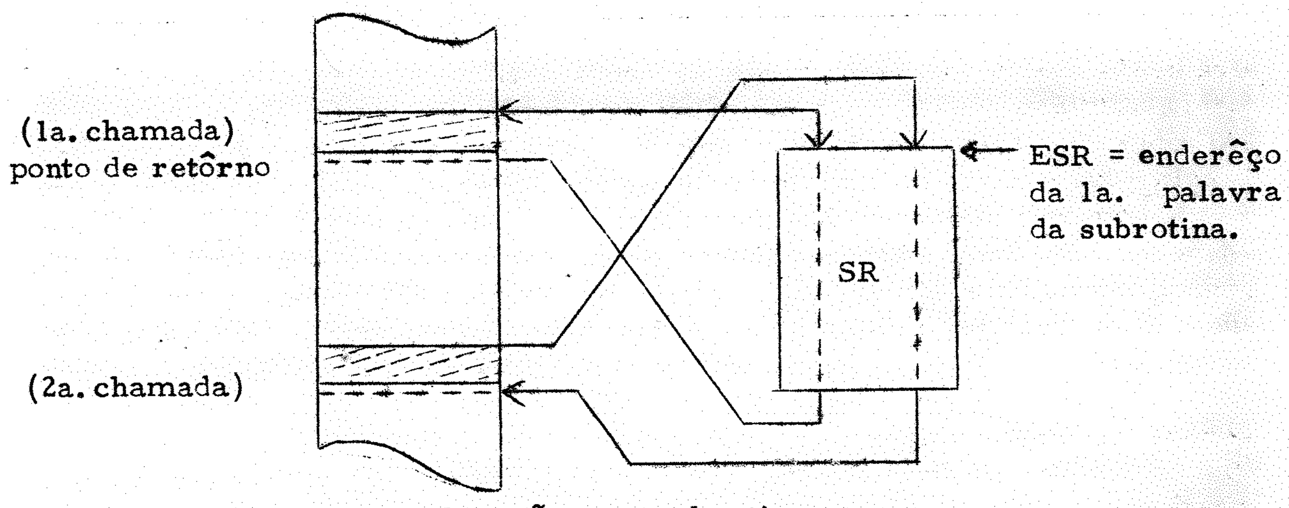


Fig. 4. Ligação com subrotinas.

A fig. 4. esquematiza a ligação com uma subrotina. Veremos agora alguns métodos comuns de ligação.

No HIPO a instrução BST de código 59 pode ser utilizada para a ligação com subrotinas. Esta instrução armazena na palavra de endereço EA o conteúdo + 1 de IAR, e desvia para EA + 1. Assim a primeira instrução da subrotina deve ser colocada em EA + 1, sendo a palavra de endereço EA reservada para, quando a subrotina fôr chamada, conter o endereço de retôrno. Na fig. 4 êste endereço está representado por ESR. As palavras seguintes à instrução BST são usadas para armazenar os endereços dos parâmetros nesta particular chamada da subrotina, uma palavra por parâmetro, numa ordem préestabelecida para cada subrotina. Assim a subrotina poderá ter acesso aos endereços dos parâmetros através de referências indiretas ao conteúdo de ESR. Obter êstes endereços é com êste método de ligação um tanto trabalhoso no entanto pois para se obter o endereço do n-ésimo parâmetro é preciso somar n-1 ao conteúdo original de ESR, o que significa a execução de algumas instruções adicionais para cada parâmetro a ser obtido. O endereço de retôrno da subrotina será o endereço original contido em ESR somado a m, onde m é o número de parâmetros da subrotina. Depois de fazer êste ajuste, portanto, podemos voltar ao programa que chamou a subrotina através de um desvio indireto para ESR. Observemos ainda que êste método de ligação permite sem nenhuma dificuldade o encadeamento de subrotinas já que o endereço de retôrno fica armazenado na própria subrotina, mas não permite, (sem técnicas adicionais), o emprêgo de subrotinas recursivas, pois se num determinado encadeamento uma subrotina fôr chamada mais de uma vez, o endereço de retôrno da primeira chamada se perde.

Tendo em vista facilitar o retôrno e o acesso aos parâmetros alguns computadores utilizam uma técnica de ligação diferente que passaremos a discutir agora. Nêste método tira-se proveito da possibilidade de endereçar os parâmetros através de endereçamento relativo a um indexador, (ié endereçamento indexado). Vamos chamar uma instrução dêste tipo de BSX, mnemônico de "Branch and store in index register". Esta instrução desvia para EA, e armazena no indexador especificado pela instrução o conteúdo + 1 de IAR. (ié: X ← IAR + 1 (endereço da palavra seguinte à instrução BSX sendo executada) IAR ← EA)

Usando esta instrução o acesso aos parâmetros fica muito facilitado. Por exemplo se usarmos o indexador 4 para a ligação com a subrotina e se na palavra seguinte à instrução BSX guardarmos o endereço do primeiro parâmetro então a instrução LDA 4, I, 0000(+0011410000 no HIPO) carrega o valor (não o endereço) do primeiro parâmetro da subrotina. Anàlogamente; LDA 4, I, 0002 carrega o valor do terceiro parâmetro, etc. É portanto muito mais fácil obter os parâmetros usando a instrução BSX para a ligação com subrotinas do que a instrução BST. O retôrno também fica facilitado. Com efeito podemos voltar ao programa que chamou a subrotina através de uma instrução de desvio incondicional indexada. Por exemplo se a subrotina tiver 5 parâmetros a instrução BRN 4, 0005 ié + 0051400005, volta ao ponto de retôrno. Se a instrução BSX facilita em relação à instrução BST o acesso aos parâmetros e o retôrno da subrotina, em compensação dificulta o encadeamento. Com efeito para podermos encadear várias subrotinas precisamos salvar o conteúdo do indexador que usamos para a ligação com subrotinas, pois a instrução BSX destrói o conteúdo prévio dêste indexador.

Assim antes da instrução BSX é necessário salvar o conteúdo dêste indexador que deve ser restaurado depois do retôrno da subrotina. Êste método de ligação também não permite (sem técnicas adicionais) o emprêgo de subrotinas recursivas.

O terceiro método de ligação com subrotinas incorpora as vantagens vistas nos dois métodos já vistos e permite outrossim fàcilmente empregar subrotinas recursivas. Subrotinas recursivas são muitas vêzes de grande utilidade e frequentemente podem facilitar muito a tarefa de programação.

Muitos conceitos são definidos recursivamente em matemática. Por exemplo, uma expressão aritmética envolvendo apenas variáveis e as quatro operações elementares pode ser definida recursivamente da seguinte maneira:

"Uma expressão aritmética é uma fórmula formada:

1. por uma só variável; ou
2. por (A + B); ou
3. por (A - B); ou
4. por (A x B); ou
5. por (A ÷ B);

onde de 2 a 5, A e B são expressões aritméticas".

Assim, se representarmos variáveis por letras minúsculas temos:

x - é uma expressão aritmética pela regra 1.

(x + y) - é uma expressão aritmética pois x é uma expressão aritmética (regra 1); y é uma expressão aritmética (regra 1) e portanto (x + y) é uma fórmula do tipo da gra 2.

Anàlogamente $((x \times y) + (z \div (w - t)))$ é uma expressão aritmética. Por outro lado, pela definição acima, $x + y -$ não é uma expressão aritmética pois não é formado por uma só variável e nem é uma fórmula do tipo da regra 2, 3, 4, ou 5.

Se quisermos fazer uma subrotina que reconheça se uma dada expressão é ou não uma expressão aritmética, a tarefa de programação será muito facilitada se a subrotina puder chamar-se a si própria, isto é, se fôr recursiva, devido à estrutura recursiva das expressões aritméticas, evidenciada pela definição acima.

Sabemos que em cada chamada da subrotina, esta precisa conhecer os seus parâmetros atuais e o ponto de retôrno para onde a subrotina deve desviar após terminar sua execução. Chamaremos ao par ponto de retôrno parâmetros, de par de ligação. As técnicas de ligação vistas até aqui não podem ser usados diretamente para a ligação com subrotinas recursivas pois com estas técnicas só há um registrador reservado para o par de ligação desta chamada. Assim se esta subrotina se chamar a si própria o ponto de retôrno original é destruido antes da subrotina utilizá-la. É portanto uma idéia natural resolver êste problema provendo a subrotina não com um registrador, mas com vários, para conter os pares de ligação, um para cada chamada. Chamaremos êstes registradores de pilha de ligação. A razão dêste nome se tornará aparente logo a seguir. Em princípio o número de vêzes que a subrotina pode chamar-se sucessivamente, chamado de nível de recursão, é arbitrário, mas na prática êste número é restringido a algum nível máximo, compatível com o espaço que podemos reservar para a pilha de ligação. Não é necessário reservar uma pilha de ligação para cada subrotina. Basta que reservemos uma pilha de ligação para tôdas as subrotinas, o que significa uma economia de memória, e ao mesmo tempo permite que o nível de encadeamento e de recursão máximo seja o maior possível para tôdas as subrotinas. Isto é possível porque uma subrotina recursiva pode ser vista como o encadeamento de uma subrotina com si mesma. Em outras palavras, quando várias subrotinas são encadeadas, eventualmente com si mesmas, os pares de ligação são utilizados disciplinadamente, em ordem tal que a última subrotina encadeada é sempre a primeira a terminar e retornar ao seu ponto de retôrno. Podemos portanto armazenar os pares de ligação numa lista, única para tôdas as subrotinas, do tipo "o último que entra na lista é o primeiro que sai", ié uma lista com estrutura de pilha. Esta é a razão do nome de pilha de ligação.

Descrevemos agora o terceiro método de ligação com subrotinas e em seguida estudaremos um exemplo detalhado de como êste método pode ser utilizado para implementar uma subrotina recursiva. Vamos chamar uma instrução dêste tipo de BSS, mnemônico de "branch and store in stack". Esta instrução utiliza o indexador 9 e a palavra de enderêço 0000 (zero) da memória. A palavra 0000 deve conter antes da execução da instrução o enderêço da primeira palavra livre da pilha de ligação. A instrução BSS armazena a partir do enderêço apontado pela palavra 0000, o conteúdo + (n + 1) de IAR, o conteúdo de IX9 e o conteúdo das n palavras seguintes à instrução BSS sendo executada (que deverão conter os parâmetros, ou os enderêços dos parâmetros da subrotina nesta chamada), em seguida transfere o conteúdo da palavra 0000 para IX9; e soma n + 2

ao conteúdo da palavra 0000 (que portanto conterá assim o endereço da próxima palavra livre na pilha), desviando finalmente para EA. O valor de n é especificado na própria instrução. A fig. 5 ilustra a execução da instrução BSS.

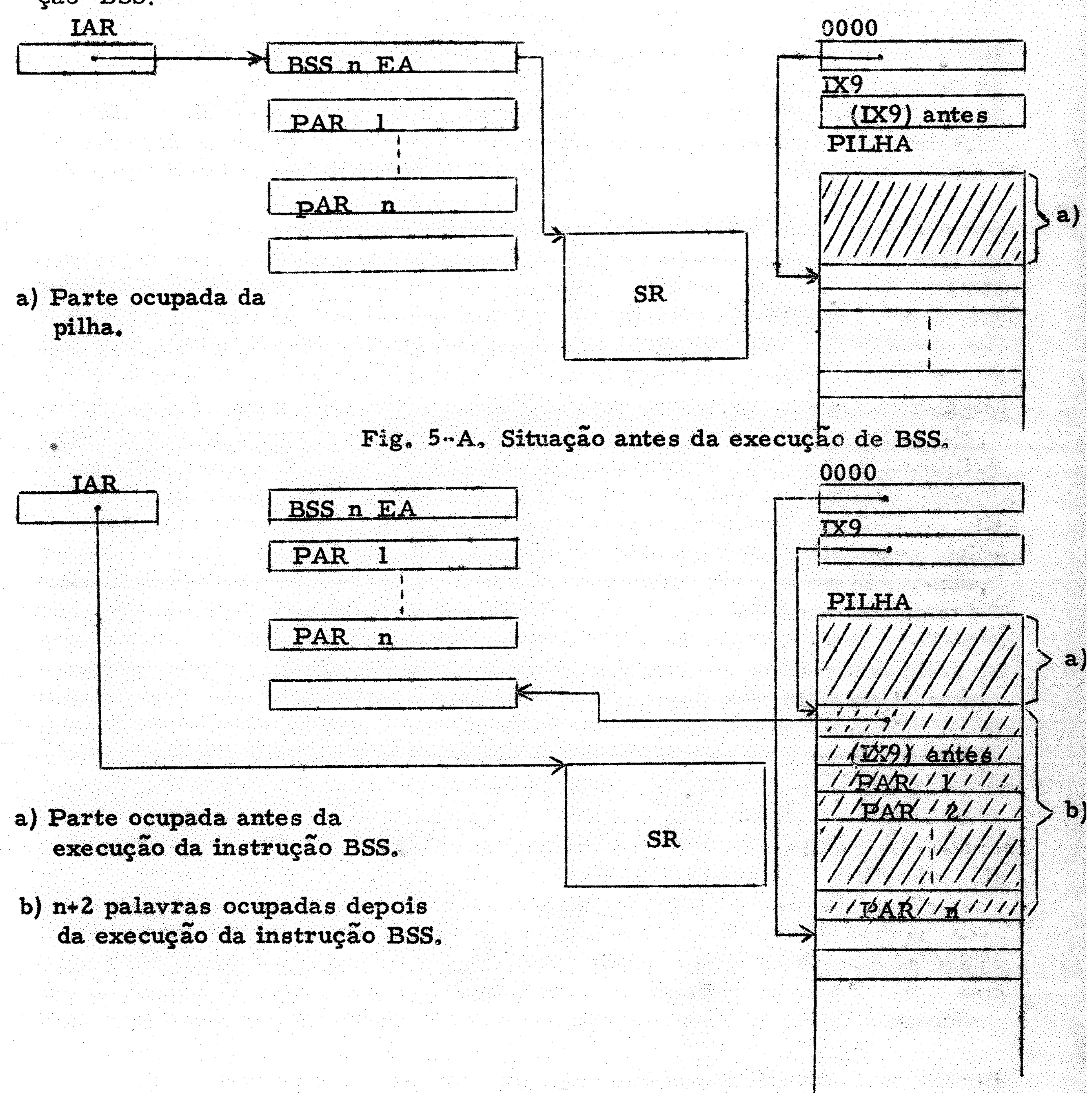


Fig. 5-A. Situação antes da execução de BSS.

Fig. 5- B Situação depois da execução de BSS.

Para retornar da subrotina existe uma instrução usada em conjunção com BSS, que chamaremos de RET, mnemônico de "return". A instrução RET, transfere o conteúdo de IX9 para a palavra 0000, em seguida o conteúdo da palavra de endereço igual ao conteúdo + 1 da palavra 0000 para IX9, e finalmente desvia para o ponto de retôrno, (contido na palavra da pilha apontada pela palavra 0000).

Estudemos agora como êste método pode ser utilizado para a implementação de uma subrotina recursiva. Examinaremos a implementação recursiva da função inteira fatorial. Êste exemplo foi escolhido devido a sua simplicidade. Como se sabe fatorial pode ser definido recursivamente por:

$$0! = 1$$

$$n! = n \cdot (n - 1)! \quad \text{para } n > 0;$$

ou na nossa notação empregando o nome de função FAT para fatorial:

ALGORÍTMO F. (calcula a função FAT (n) – fatorial de n, recursivamente)

F1. Se n = 0 FAT ← 1 e o algorítmo termina; senão

F2. FAT ← n. FAT (n - 1).

Vejamos como esta subrotina seria implementada no HIPO usando a técnica BSS. Suporemos que a subrotina seja armazenada em 0100, que a pilha seja armazenada a partir de 0500, e que a subrotina armazene o valor do fatorial por ela calculado no acumulador. Suporemos que a instrução BSS seja de código 91 e que o valor do número de parâmetros seja armazenado nos dígitos 0 e 1 da instrução, e que a instrução RET seja de código 92.

Enderêço	Conteúdo	Comentários
0000	+ 0000000500	Aponta inicialmente para o comêço da pilha
0009	+ 0000000000	IX9
		COMEÇO DA SUBROTINA
0100	+ 0011900002	carrega o valor do parâmetro (n) no acumulador
0101	+ 0053000104	se n ≠ 0 desvia para 0104.
0102	+ 0011000111	carrega 1 no acumulador (A ← 0!)
0103	+ 0092000000	retorna
0104	+ 0022000111	subtrai 1 do acumulador
0105	+ 0012000107	armazena para a chamada.
0106	+ 0191000100	chama FAT, com 1 parâmetro.
0107	+ 0000000000	o valor do parâmetro é armazenado aqui
0108	+ 0023900002	multiplica o parâmetro pelo acumulador
0109	+ 0063000010	guarda o resultado no acumulador
0110	+ 0092000000	retorna
0111	+ 0000000001	constante 1.

A tabela a seguir é a simulação da subrotina acima, desencadeada pela seguinte chamada:

```
0010 + 0191000100
0011 + 0000000003
0012      ...
  .       ...
```

que calcula 3 !

instruções	mnem	Variáveis Afetadas				Pilha											
		0000	IX9	A	0107	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
Situação Inicial		0500	0000														
0010	BSS	0503	0500			0012	0000	3									
0100-0101	LDA-BNZ			3													
0104-0105	SUB-STA			2	2												
0106	BSS	0506	0503						0108	0500	2						
0100-0101	LDA-BNZ			2													
0104-0105	SUB-STA			1	1												
0106	BSS	0509	0506									0108	0503	1			
0100-0101	LDA-BNZ			1													
0104-0105	SUB-STA			0	0												
0106	BSS	0512	0509												0108	0506	0
0100-0101	LDA-BNZ			0													
0102	LDA			1													
0103	RET	0509	0506														
0108-0109	MPY-SLQ			1													
0110	RET	0506	0503														
0108-0109	MPY-SLQ			2													
0110	RET	0503	0500														
0108-109	MPY-SLQ			6													
0110	RET	0500	0000														
0012	...																

Em subrotinas recursivas, especial cuidado deve ser tomado no uso de variáveis internas da subrotina, verificando se entre a definição da variável e qualquer uso da mesma não há uma chamada recursiva. Neste último caso a variável não deve ter um enderêço fixo na subrotina pois durante a chamada recursiva o seu valor prèviamente calculado pode ser destruido. Por êste motivo estas variáveis devem ser guardadas na pilha, o seu endereço mudando dinâmicamente como se fôssem parâmetros.

1.5. Tabelas e Buscas em Tabelas

Consideremos um conjunto S. Chamaremos um elemento de S de chave. Seja S_1 um subconjunto finito de S e T um outro conjunto de objetos. Uma tabela é uma função cujo campo de definição é S_1 e cujo campo de variação é T. Uma tabela, portanto, é um subconjunto finito de $S_1 X T$, ié um conjunto finito de pares ordenados (s_k, t_k) onde $s_k \in S_1$ e $t_k \in T$. Para cada chave, $s \in S_1$ existe e é único o par ordenado que pertença à tabela, com o primeiro elemento do par sendo s. Tabelas são de uso frequente em programas, e muitas vêzes a eficiência de um programa depende da adequada representação de suas tabelas e da organização conveniente dos algorítmos que processam estas tabelas.

Quando os elementos de S_1 e T podem ser representados em um número fixo de palavras, frequentemente uma representação conveniente da tabela na memória é guardar os n pares ordenados (s_k, t_k) da tabela em $n \times m$ palavras contíguas da memória, cada par ocupando m palavras consecutivas. As operações frequentemente encontradas relacionados com tabelas são: Dada uma chave s_i determinar o t_i correspondente; acrescentar mais um par (s_i, t_i) à tabela; alterar um par (s_i, t_i) da tabela, etc. Tôdas estas operações envolvem a localização de um par com uma determinada chave. Uma maneira de fazer isso é comparar a chave de sucessivas entradas da tabela com a chave procurada. Êste algorítmo simples é chamado de busca sequencial, ou de busca linear. Se a tabela tiver n elementos, e cada elemento é procurado com frequência uniforme então para uma busca, em média, são necessárias $(n+1)/2$ comparações. A busca linear é portanto um processo relativamente demorado. Várias técnicas foram por isso desenvolvidas para melhorar a eficiência da busca.

Para certas tabelas é possível ordenar os elementos de tal maneira que o endereço do par correspondente a cada chave é uma função de cálculo simples da chave. Por exemplo se $S_1 = \{1, 2, \ldots n\}$ e cada par ocupar 3 palavras, guardando os pares em ordem crescente das chaves, o endereço do par (m, t_m), relativo ao começo da tabela, será $3m-3$. Quando um arranjo dêste tipo é possível diremos que a tabela é de acesso direto. Neste caso a busca será extremamente eficiente, pois uns poucos cálculos permitem localizar qualquer elemento da tabela. Quando a tabela é de acesso direto, é desnecessário, em geral, guardar a chave na tabela, pois esta é implicitamente determinada pela localização do elemento correspondente na tabela. Assim, no caso do exemplo anterior, basta guardar na tabela t_m.

De uma maneira geral, porém, não é possível ordenar os elementos de modo a se obter uma relação tão simples entre a localização do elemento e a sua chave. Por exemplo tomemos para S o conjunto de todos símbolos possíveis formados com 3 letras, e S_1 o conjunto dos símbolos que são mnemônicos de instruções do HIPO. Os símbolos que são chaves válidas, ié que pertencem a S_1, são combinações de 3 letras bastante arbitrárias, (para que sejam mnemônicos), e assim não existe nenhuma ordem em que poderíamos guardá-los numa tabela e ter uma função simples para obter a partir das 3 letras de um determinado mnemônico o seu lugar na tabela. Se ordenarmos os 26^3 símbolos de S em ordem alfabética é fácil determinar o número de ordem de um símbolo particular qualquer dentre todos os símbolos de S. Mas para usarmos êste número como localização do símbolo dentro da tabela, precisaríamos uma tabela com 26^3 elementos apesar de que só um número muito menor de elementos vai ser efetivamente usado. Em casos como êste portanto não é possível termos uma tabela de acesso direto, e é conveniente o uso de técnicas especiais que permitam reduzir ao máximo o número de comparações necessárias para se achar um símbolo na tabela.

Uma das técnicas preferidas para a solução dêste problema é a aleatorização da chave. ("randomizing or hashing techniques"). Esta técnica consiste em subdividir a tabela em m subtabelas menores. Cada elemento da tabela é colocado em uma destas subtabelas conforme o valor de uma função pseudo-aleatória da chave que varia entre 0 e m-1. Assim, quando quisermos localizar um elemento da tabela com uma certa chave, aplicando a função pseudo-aleatória a esta chave determinamos a subtabela na qual o símbolo deve ter sido armazenado e assim precisamos efetuar uma busca numa tabela menor, reduzindo o número de comparações necessárias na localização do elemento. Em outras palavras a função aleatória nos dá a localização aproximada do elemento procurado facilitando a busca. Por exemplo se a nossa tabela tiver 100 elementos uma busca linear exigirá em média $101/2 \cong 50$ comparações. Se dividirmos a tabela em 10 subtabelas, cada um com 10 elementos, então gerando a partir da chave um número pseudo-aleatório entre 0 e 9, localizamos a subtabela que contém o elemento desejado sendo necessárias então em média apenas 5 comparações para localizar o elemento desejado. Um problema que pode ocorrer ao ser construída uma tabela com esta estrutura é ser selecionada uma subtabela para mais elementos do que a subtabela pode comportar. Neste caso seremos obrigados a invadir uma casa disponível de uma outra subtabela, que será selecionado por algum critério, por exemplo aplicando mais uma vez a função pseudo-aleatória. A busca entretanto só continuará eficiente se sobrecargas de subtabela dêste tipo forem relativamente raras, o que em geral pode ser conseguido reservando para a tabela um espaço maior do que o estritamente necessário.

Para ilustrar a técnica vejamos como ficaria a estrutura da tabela em que S_1 é o conjunto dos mnemônicos de instruções do HIPO. O HIPO possui 42 instruções descritas até agora. O número médio de comparações empregando busca linear seria portanto 21.5, para cada busca, se cada elemento da tabela fôsse procurado com a mesma frequência. Se porém a frequência fôr a mesma com que cada instrução é usada em programas, podemos esperar que certas instruções serão de uso mais frequente que outras. Por exemplo é razoável supor que LDA e STA são muito mais usadas do que STP ou BOF. Colocando na tabela estas instruções que são

usadas mais frequentemente antes das menos usadas, nas consultas mais frequentes o número de comparações será pequeno. Com esta providências, portanto, podemos reduzir o número médio de comparações de um acesso à tabela para, digamos, 15. Dividindo a tabela em 10 subtabelas, se a função aleatorizante conseguisse distribuir uniformemente as chaves pelas subtabelas teriamos 4.2 elementos por subtabela com o que o número médio de comparações por acesso cairia para 2.6. Colocando, porém, em cada subtabela as instruções mais usadas antes das menos usadas, êste número cairia mais ainda.

mnemônico (chave)	Representação interna no HIPO (X)	Número gerado a partir da chave (X^2)	Subtabela (8º dígito)
LDA	+ 0000736461	542374804521	0
STA	+ 0000828361	686181946321	4
LDQ	+ 0000736478	542399844484	4
STQ	+ 0000828378	686210110884	1
LZR	+ 0000738979	546089962441	6
LDG	+ 0000736467	542383642089	4
ADD	+ 0000616464	380027863296	6
SUB	+ 0000828462	686349285444	8
MPY	+ 0000747788	559186892944	9
DIV	+ 0000646985	418589590225	9
FAD	+ 0000666164	443774474896	7
FSU	+ 0000668284	446603504656	0
FMP	+ 0000667477	445525545529	4
FDV	+ 0000666485	444202255225	5
RVS	+ 0000798582	637733210724	1
RNW	+ 0000797586	636143427396	2
RNC	+ 0000797563	636106738969	3
RAW	+ 0000796186	633912146596	4
RAC	+ 0000796163	633875522569	2
DMP	+ 0000647477	419226465529	6
PNW	+ 0000777586	604639987396	8
PNL	+ 0000777573	604619770329	7
PAW	+ 0000776186	602464706596	0
PAL	+ 0000776173	602444525929	2
NOP	+ 0000757677	574074436329	3
BRN	+ 0000627975	394352600625	0
BNP	+ 0000627577	393852890929	9
BNZ	+ 0000627589	393867952921	5
BPS	+ 0000627782	394110239524	3
BZR	+ 0000628979	395614582441	8
BNG	+ 0000627567	393840339489	3
BNN	+ 0000627575	393850380625	8
BOF	+ 0000627666	393964607556	0
BST	+ 0000628283	394739528089	2
SLA	+ 0000827361	684526224321	2
SRA	+ 0000827961	685519417521	1
SLQ	+ 0000827378	684554354884	5
SRQ	+ 0000827978	685547568484	6
MDX	+ 0000746487	557242841169	4
STP	+ 0000828377	686208454129	5
BSS	+ 0000628282	394738271524	7
RET	+ 0000796583	634544475889	7

Distribuição pelas Subtabelas	
Subtabela	Nº de Elementos
0	5
1	3
2	5
3	4
4	6
5	4
6	4
7	4
8	4
9	3

Nº médio de comparações por acesso com frequência uniforme de buscas por elemento ≅ 2.7. Colocando as chaves mais usadas antes das menos usadas êste número seria ainda menor.

Um outro método de busca que reduz dràsticamente o número médio de comparações por acesso é o algorítmo de busca conhecido por busca binária, ou busca logarítmica. No algorítmo de busca binária a tabela deve estar ordenada em ordem lexicográfica, crescente por exemplo, das chaves. Esta restrição não chega a ser uma séria desvantagem em pelo menos dois casos:

1) Quando a tabela deve ser ordenada de qualquer jeito devido a outras razões, (como por exemplo numa tabela de símbolos de um programa quando se quer editar os símbolos usados em ordem alfabética);

2) Quando as chaves da tabela já são prèviamente conhecidos, e portanto, a tabela já pode ser construída em ordem lexicográfica. (Êste é o caso por exemplo das chaves que são mnemônicos de instruções do HIPO, que estudamos anteriormente).

Existem algorítmos eficientes de ordenação, (que exigem da ordem de $n\log_2 n$ comparações para ordenar n itens) e também métodos que permitem construir a tabela com uma estrutura de árvore binária tal que já leve em conta a ordem lexicográfica de seus elementos. [4]

O algorítmo de busca binária consiste em comparar a chave procurada com a chave do elemento que está no meio da tabela, ié com o $\lceil n/2 \rceil$-ésimo elemento. Se a chave procurada fôr menor do que a chave dêste elemento, então como a tabela está ordenada o elemento procurado certamente estará na primeira metade da tabela, já que as chaves de todos os elementos da segunda metade são maiores do que a chave do $\lceil n/2 \rceil$ ésimo elemento. Se a chave procurada fôr maior do que a chave dêste elemento, então por ra-

ciocínio análogo o elemento procurado estará na segunda metade da tabela. Evidentemente se a chave procurada fôr igual à chave do $\lceil n/2 \rceil$ ésimo elemento então localizamos o elemento procurado e o algorítmo termina; senão isolamos uma subtabela com $\lfloor n/2 \rfloor$ elementos que contém o elemento procurado e então repetimos os mesmos passos com esta subtabela. Assim por bisecções sucessivas isolamos subtabelas de tamanho cada vez menor até localizarmos o elemento procurado. Damos abaixo a definição recursiva dêste algorítimo. (é fácil a programação iterativa do algorítmo de busca binária. A descrição recursiva que damos a seguir não é conveniente portanto para efeitos práticos e foi escolhida apenas por sua concisão e elegância.)

Algorítmo B

(Função recursiva BB (BASE, n, C) cujo valor é o índice do elemento de chave C numa tabela que começa no elemento de índice Base + 1 e tem n elementos.)

$\lfloor X \rfloor$ indica o maior inteiro contido em: ié $\lfloor X \rfloor = \max_{m \leq x} m$

$\lceil X \rceil$ indica o menor inteiro contido em: ié $\lceil X \rceil = \min_{m \geq x} m$

B1. [Pegue o elemento do meio] Se $n = 0$ o elemento procurado não está na tabela e o algorítmo termina; senão faça BB ← BASE + $\lceil n/2 \rceil$;

B2. [Compare as chaves] compare as chaves C e CHAVE [BB];

i) se fôrem iguais o algorítmo termina;

ii) se C > CHAVE [BB] BB ← BB (Base + $\lceil n/2 \rceil$, $\lfloor n/2 \rfloor$, C) e o algorítmo termina;

iii) se C < CHAVE [BB] BB ← BB (Base, $\lfloor n/2 \rfloor$, C) e o algorítmo termina.

O número médio de comparações por acesso na busca binária, supondo que cada chave seja procurada na tabela com a mesma frequência está entre $\log_2 n - 1$, e $\log_2 n$. Para $n \geq 2^8$ êste número é com muito boa aproximação $\log_2 n - 1$. O número máximo de comparações para localizar um elemento da tabela é $\lfloor \log_2 n \rfloor + 1$.

Vimos portanto nesta secção duas técnicas que reduzem consideràvelmente o número de comparações por acesso a uma tabela em relação ao algorítmo de busca sequencial. Como o uso de tabelas é muito frequente, tem merecido considerável estudo o desenvolvimento de estruturas de armazenamento convenientes e algorítmos eficientes de busca em tabelas. Nos computadores mais novos em geral há instruções especiais que realizam uma busca com a execução de uma única instrução. Quando isto acontece em geral é mais conveniente usar estas instruções para efetuar buscas do que programar um algorítmo dos tipos descritos, pois apesar de as instruções de busca em geral fazerem uma busca sequencial, um algorítmo por "software" será provàvelmente menos eficiente. Contudo se alguns computadores possuem instruções de busca sequencial por "hardware", talvez dentro de algum tempo seja econômico implementar por "hardware" algorítmos de

busca binária, ou técnicas de aleatorização. Técnicas cada vez mais eficientes continuam a ter portanto intenso interêsse mesmo neste caso.

1.6 Sistemas Operacionais: uma visão geral.

O computador digital nos moldes descritos na secção 1.1. surgiu há menos de 30 anos. A grosso modo a primeira destas três décadas foi de desenvolvimento e de intensa pesquisa em Universidades muitas vêzes associado a projetos subvencionados por agências governamentais tendo em vista finalidades específicas como por exemplo o desenvolvimento de uma máquina capaz de fazer cálculos balísticos. Como resultado da tecnologia e conhecimento adquiridos nesta década surgiram há cêrca de 20 anos os primeiros computadores fabricados por firmas especializadas e vendidos comercialmente. Nos 20 anos seguintes o progresso alcançado no projeto de computadores e nas suas aplicações foi verdadeiramente prodigioso. Do ponto de vista tecnológico, viu-se o aparecimento de três gerações de computadores. São considerados computadores de primeira geração os computadores à válvula. O aparecimento do transistor foi o responsável pelo primeiro grande salto tecnológico surgindo então os computadores da segunda geração em que as válvulas foram substituídas por transistores. Nos computadores de terceira geração o transistor deu lugar ao circuito integrado. Paralelamente à estas inovações tecnológicas houve um enorme progresso no projeto do "hardware". Êste progresso foi responsável por um aumento gigantesco da velocidade dos computadores que se tornaram dezenas de milhares de vêzes mais rápidos que os primeiros modelos.

O desenvolvimento ímpar do "hardware" dos computadores foi acompanhado, e na realidade ìntimamente ligado, a um progresso igualmente notável do "software". As aplicações dos computadores foram enormemente diversificadas e facilitadas por uma multidão de programas de utilidade, linguagens de programação e seus compiladores, interpretadores, e sistemas de programação especialmente projetados para melhorar tanto a eficiência do uso do equipamento, como o atendimento de usuários de uma determinada instalação. Êstes últimos são chamados de sistemas operacionais e ao seu estudo será dedicada uma boa parcela dêste trabalho.

Os primeiros modelos de computadores eletrônicos funcionavam sem sistemas operacionais. A sua programação era em geral trabalhosa e a depuração dos programas era feita junto ao computador, frequentemente interrompendo o teste para o programador tentar descobrir pelo painel do computador alguma falha do seu programa. Em geral não havia operador e o próprio programador operava a máquina. Quando um programador terminava sua tarefa junto do computador, entregava o equipamento a um outro usuário que repetia com o seu programa o mesmo processo. Com o aumento da velocidade dos compuadores porém, logo ficou aparente que êste tipo de operação era muito ineficiente, pois o computador estava na maior parte do tempo parado esperando alguma intervenção do operador. O desnível entre a velocidade da máquina e a velocidade do operador era simplesmente tão grande que se o operador tivesse de intervir frequentemente, a máquina ficaria parada por intervalos consideráveis de tempo cada vez que tal intervenção se fizesse necessária. Era necessário mudar o método de operação tornando o processo mais automático. Foi desta necessidade que surgiu o

sistema operacional. O sistema operacional era de início um programa projetado para realizar automàticamente certas funções que sem êle seriam feitas pelo operador. Na época em que surgiram êstes primeiros sistemas, por volta de 1965, programava-se muitas vêzes em linguagem de máquina e em linguagem de montagem e depois de 1957 em FORTRAN. Os primeiros sistemas operacionais consistiam de um programa, denominado de monitor, que controlava a execução de diversos programas de diversos usuários, um atrás do outro, chamando quando apropriado as várias componentes do sistema: o montador, um compilador FORTRAN, ou o carregador para carregar na memória programas em linguagem de máquina (linguagem objeto). Várias tarefas eram agrupadas para formar um conjunto de tarefas denominado de "batch" (lote). A idéia era que cada "batch" seria executado sem intervenção do operador. O programador era obrigado a seguir certas regras para permitir esta operação automática, como: usar para entrada/saída rotinas do sistema que eram programadas adequadamente para evitar que uma tarefa lesse como dados os cartões de outra tarefa; ao fim do programa o usuário tinha de desviar para o programa monitor que se encarregava de iniciar a tarefa seguinte; etc. Uma evolução natural dêstes sistemas iniciais levou ao emprêgo extensivo de subrotinas, providas pelo sistema ou programadas pelo próprio usuário. Um programa portanto consistia em geral de um programa principal que chamava diversas subrotinas, do sistema ou não. Quando da execução do programa, era necessário carregar na memória o programa principal e tôdas as subrotinas necessárias, disponíveis prèviamente numa fita de sistema, ou numa fita de subrotinas de biblioteca do usuário. Como o enderêço absoluto onde as rotinas seriam carregadas não eram conhecidas ao tempo da compilação da rotina, as rotinas ficavam guardadas em forma relocável (ver cap. 3). Com a programação de sistemas de programas cada vez mais complexos logo ficou aparente que a escassez de memória era um problema a sêr enfrentado. O reconhecimento deste fato levou à uma melhor técnica de utilização da memória: a técnica de superposição ("overlay"). Essa técnica consiste em carregar na mesma área de memória, à medida que se tornam necessárias, cada parte intercambiável do programa. Isto é possível de ser feito com tôdas as partes do programa que não necessitem estar simultâneamente na memória. Entrementes a velocidade do processador central foi aumentando com cada novo modelo de computador. Embora também os equipamentos periféricos de entrada e saída fossem cada vez mais rápidos e aperfeiçoados, o aumento de velocidade destes era limitado devido ao fato desta velocidade estar ìntimamente ligada à velocidade do movimento mecânico de suas peças. Assim a velocidade do processador central aumentou comparativamente muito mais que a das máquinas periféricas. Em consequência a execução de cada instrução de entrada/saída levava o mesmo tempo que a execução de centenas de instruções "normais" (ié que não envolvessem entrada/saída)

Seria portanto interessante que a entrada/saída se processasse paralelamente com o processamento das demais instruções do programa tornando o sistema mais eficiente. O aparecimento do canal e do conceito de interrupção tornou isto possível, ao mesmo tempo em que estimulava um novo desenvolvimento dos sistemas operacionais, com a inclusão de técnicas especiais para o aproveitamento da capacidade de processamento paralelo do "hardware".

A observação cuidadosa de cada geração de sistemas, até o grau

de sofisticação visto até aqui, revelou que mesmo com as técnicas especiais de aproveitamento da capacidade de processamento paralelo mencionado acima, muitas vêzes tal capacidade era subaproveitada, quando da execução de programas que exigem enorme quantidade de informações para serem lidas e escritas, e ao mesmo tempo fazem transformações relativamente simples destas informações. Êstes programas pràticamente durante tôda a sua execução estão esperando alguma unidade de entrada/saída completar uma operação de leitura ou escrita requisitada. Em contraste existem aqueles programas que exigem enorme quantidade de cálculos mas leem e escrevem comparativamente poucas informações. Os primeiros são programas cuja velocidade de execução está limitada pela velocidade de entrada/saída do sistema, enquanto que os do segundo tipo têm sua velocidade de execução limitada pela velocidade de transformação de informações do processador central. É evidente portanto que a eficiência de utilização do sistema seria muito melhorada se os dois programas coexistissem na memória e fossem alternadamente executados: enquanto o programa do primeiro tipo espera pelo término de uma operação de entrada/saída, (não podendo prosseguir por falta das informações a serem lidas, ou por falta de espaço de memória em que produzir novas informações de saída), o programa do segundo tipo poderia ser executado, aproveitando assim ao máximo os equipamentos de entrada/saída e o processador central e a capacidade de processamento paralelo do sistema. Ao mesmo tempo um sistema deste tipo reduziria o tempo de espera entre a entrega do programa para processamento e a devolução dos resultados em muitas situações, tornando desnecessário esperar o término de um programa longo para executar um programa curto por exemplo. A implementação desta idéia levou aos sistemas de multiprogramação.

Percebeu-se também, gradativamente, a necessidade de ter recursos para contornar diversas situações de anomalia automàticamente. Assim tornou-se aparente que um possível êrro de programação não deveria inutilizar o sistema, pois isto exigiria a intervenção do operador para reinicializar o sistema e reiniciar a operação normal. Para conseguir a operação automática em qualquer situação foram incluidas no projeto do "hardware" interrupções geradas pelo processador central em situações anômalas, como tentativa de execução de uma instrução inválida, ocorrência de um êrro de endereçamento, ocorrência de uma sobrecarga ("overflow")etc. Era preciso também impedir que um programa de um usuário pudesse destruir parte do sistema operacional (por exemplo a parte residente na memória), ou interferir indevidamente no programa em execução de um outro usuário (no caso de sistemas de multiprogramação). Mecanismos adequados de proteção foram implementados para conseguir isto. Para evitar que um programa inadvertidamente usasse tempo excessivo do processador central ou gerasse uma saída de um número excessivo de páginas impressas, foram incluídos no sistema operacional mecanismos para o contrôle destas grandezas. Para possibilitar o contrôle do tempo foi incluido no "hardware" um relógio interno acessível pelo sistema operacional. Êste relógio gerava uma interrupção quando passasse um certo tempo (estabelecido pelo sistema operacional inicializando um registrador especial do relógio).

O relógio interno era necessário, depois, também para implementar um sistema de multiprogramação bem como possibilitou que o sistema operacional fizesse uma adequada contabilidade do tempo de uso entre os diversos usuários do sistema. Para evitar que o usuário pudesse ter acesso ao registrador do relógio e que fôsse obrigado a usar as rotinas de entrada/

saída do sistema surgiram as instruções privilegiadas, que só poderiam ser executadas pelo sistema operacional. A tentativa de execução de uma instrução privilegiada no programa do usuário gerava uma interrupção desviando o contrôle para o sistema operacional.

Os sistemas operacionais que descrevemos acima incorporam importantes melhoramentos na utilização do "hardware" disponível. Os objetivos fundamentais em todos êles porém tem em vista a utilização mais eficiente do "hardware", embora signifiquem muitas vêzes também uma melhoria na qualidade de atendimento do usuário do sistema. A qualidade de atendimento pode ser vista como a facilidade de usar o sistema, o que significa tanto uma diversidade dos recursos do mesmo facilitando um grande número de funções comumente utilizados pelo usuário, como sobretudo o tempo decorrido entre a entrega do programa para processamento e a devolução do programa com os resultados. Êste tempo é chamado de tempo de devolução ("turn around time"). Ora, a experiência demonstra que a eficiência de utilização do sistema pode ser elevado, especialmente se o equipamento tem muitos usuários, mas frequentemente, e sobretudo neste último caso, o tempo de devolução se deteriora para várias horas, ou mesmo dias. O atendimento do usuário portanto pode ser bastante deficiente. Isto é particularmente sensível na fase de depuração de um programa, quando o programador terá de esperar frequentemente várias horas apenas para descobrir que esqueceu uma vírgula em algum comando do seu programa por exemplo.

Tendo em vista êste problema é que surgiu a idéia dos sistemas de tempo compartilhado. ("Time sharing systems"). A idéia fundamental destes sistemas é dar ao usuário a impressão de que o sistema está sob seu exclusivo comando, respondendo a cada solicitação dêste em tempo bastante curto, tendo por assim dizer tempo de devolução de questão de segundos. Na realidade o sistema, para que isto seja econômicamente viável, dá esta impressão a vários usuários ao mesmo tempo, atendendo um atrás o outro em rápida sucessão. Isto é possível porque a velocidade do equipamento é milhares de vêzes superior à velocidade de reação do usuário. Portanto o sistema pode atender a dezenas de usuários dando a cada um a impressão que êle é o único a utilizar o sistema. Fundamentalmente portanto um sistema de tempo compartilhado é bastante semelhante a um sistema de multiprogramação. As diferenças surgem porque enquanto o objetivo fundamental do primeiro é dar ao usuário um excelente serviço de atendimento, fazendo com que o usuário seja parte integrante do sistema, a ênfase do segundo é na eficiência de utilização do equipamento. Isto usualmente implica em uma série de diferenças nas técnicas utilizadas como por exemplo: o número de programas ativos num sistema de tempo compartilhado é geralmente bem maior do que num sistema de multiprogramação; Em consequência a memória é um recurso extremamente solicitado sendo necessárias técnicas eficientes de sua utilização como paginação, etc.; Num sistema de tempo compartilhado em geral é exigido que possam ser a êle ligados terminais que permitam que o usuário possa interagir com o sistema "em linha"; devido ao uso frequente de compiladores, editores de texto etc. ... nos sistemas de tempo compartilhado é conveniente que êstes programas sejam compartilhados por diversos usuários no sentido de que apenas uma cópia do programa esteja na memória agindo em diversas áreas de dados, (um para cada usuário): É o que se chama de programa reentrante; etc.

Procuramos nesta secção dar uma idéia do desenvolvimento de sis temas operacionais até a presente data.

Nos capítulos subsequentes estudaremos várias das componentes dêstes programas.

Capítulo 2

A linguagem de montagem e o programa montador

2.1 A linguagem de montagem do HIPO:

As linguagens de programação surgiram para facilitar a programação dos computadores, cuja programação em sua linguagem de máquina é muito tediosa. Uma das linguagens mais simples, em geral de sintaxe extremamente elementar, e cujos comandos executáveis tem uma correspondência 1 - 1 com as instruções do programa correspondente em linguagem de máquina é a linguagem de montagem. Nesta secção descreveremos a linguagem de montagem do HIPO. A linguagem de montagem do HIPO foi denominado de HAL (de "Hipo Assembly Language"). O programa que transforma um programa fonte em HAL para um programa em linguagem de máquina do HIPO, (programa objeto), é o montador.

Um programa em HAL consiste de um número arbitrário de comandos HAL. Cada comando é perfurado num cartão com campos pré-fixados. O formato dêste cartão é o seguinte:

Colunas	Nome do Campo
1 - 6	Rótulo
7	b (branco)
8 - 11	Código de operação do comando HAL
12	b
13	Campo X
14	Campo I
15	b
16-72	Campo de Operandos
73-80	Comentários

Rótulo - Este campo pode conter um símbolo ou deixado em branco. Um símbolo consiste de um a seis caractéres alfanuméricos sendo o primeiro uma letra, que deve ser perfurado na coluna 1, e os demais podem ser qualquer do conjunto de caracteres alfanuméricos. b-s intercalados não são permitidos. A cada símbolo o montador vai associar um enderêço.

Código de operação do comando HAL - Cada comando do HAL é identificado por um símbolo mnemônico de 1 a 4 letras. Os comandos em HAL são de dois tipos : a) Os comandos executáveis que correspondem às instruções do HIPO e cujo código de operação HAL é o mnemônico já visto na descrição das instruções do HIPO no capítulo 1.

b) Os pseudo-comandos também denominados de pseudo-instruções ou simplesmente pseudos. Estes comandos não são executáveis, e a êles não corresponde nenhuma instrução no programa objeto em linguagem de máquina. Êstes comandos portanto não são instruções ao HIPO durante a execução do programa; são antes instruções ao próprio programa montador. São usados para definir áreas para variáveis, constantes, controlar a listagem do programa e finalidades similares.

<u>os campos X e I</u> - só são usados nas instruções executáveis e seu conteúdo determina o conteúdo dos campos X e I da instrução HIPO correspondente. O campo X pode ser b ou O quando não se quer usar indexador ou 1 a 9 indicando o indexador a ser usado.
O campo I pode ser b ou O para endereçamento direto, ou 1 ou I para indicar endereçamento indireto.

<u>o campo de operandos</u> - o uso do campo de operandos nos pseudos será explicado mais tarde. Nas instruções executáveis êste campo vai determinar o conteúdo do campo EEEE da instrução HIPO correspondente ao comando. Este campo pode conter:

a) um número - neste caso êste número será o conteúdo de EEEE

b) um símbolo - neste caso o endereço associado a êste símbolo será o conteúdo de EEEE.

c) uma expressão do tipo - <símbolo> + < cte. numérica > ou <símbolo> - < cte. numérica > e neste caso o contéudo de EEEE será a soma (diferença) do enderêço associado ao <símbolo> com a <cte. numérica>.

d) uma literal - o uso de literais permite que cada instrução do HIPO que em linguagem de máquina utiliza endereçamento direto, funcione como se fôsse uma instrução que utiliza endereçamento imediato. Uma literal pode ser numérica, (inteira ou de ponto flutuante), ou alfanumérica. O montador reconhece que o operando da instrução é uma literal se na coluna 16 estiver um sinal = . Literais alfanuméricos podem ser no máximo de 5 caracteres (para a literal poder ser representada numa palavra) e devem estar entre aspas ("). Literais numéricas que contém um ponto (.), são transformados numa constante flutuante. Quando o montador encontra uma literal reserva uma palavra na memória que ao tempo de carga será inicializada com a representação interna do valor da literal, e coloca no campo EEEE o enderêço da palavra que contém a literal. Quando se quer que o caráter aspas (") seja um dos caracteres da literal alfanumérica devem ser colocadas duas aspas no lugar correspondente.

literal	Conteúdo da palavra reservada pelo montador
Ex. = 3	+ 0000000003
= - 322	- 0000000322
= - 2.5	- 2500000051
= 2.5E - 3	+ 2500000048
= "ALABA"	+ 6173616261
= "A"	+ 6100000000
= " " " A " " "	+ 3261320000

e) * ou expressões envolvendo * do tipo: *+ <cte. numérica> ou
*- <cte. numérica>

Nos dois casos * funciona como se fosse um símbolo ao qual o montador associa o endereço da palavra sendo montada.

A alocação das instruções, símbolos, constantes e literais do programa na memória é feita pelo montador. Cada comando é alocado seqüencialmente na memória, a partir de uma origem que pode ser especificado pelo programador através de um pseudo ORG (vide abaixo). Se o programador não especificar o endereço a partir de onde o programa vai ser alocado o montador assume o começo do programa em 0010. Após alocar o programa e suas constantes e símbolos o montador aloca as literais usadas.

Para o montador traduzir um comando HAL executável para a correspondente instrução HIPO deve determinar o conteúdo de cada campo da instrução. Uma instrução, como já vimos, tem o seguinte formato : - | +00ØP X I EEEE |. O campo ØP é determinado pelo código mnemônico do comando. Para determinar êste campo basta que o montador descubra qual o código de instrução do HIPO que corresponde ao mnemônico em questão e isto é feito procurando-se numa tabela cuja chave s é o mnemônico e cujo valor t correspondente é o código numérico da instrução.

Os campos X e I da instrução HIPO ficam diretamente determinados pelos campos X e I do comando HAL, conforme foi visto acima.

O campo EEEE é determinado pelo campo de operandos do comando HAL. Para determinar êste campo da instrução o montador pode precisar do endereço associado a um símbolo ou o endereço de uma literal. Para isto o montador precisa, durante o processo de montagem construir uma tabela de símbolos, em que a chave s é o símbolo usado e o valor t correspodente é o endereço associado ao símbolo, e também, de maneira análoga, uma tabela de literais. Diz-se que um símbolo usado em HAL fica definido quando este símbolo aparecer no campo "rótulo" de um comando HAL. Quando um símbolo é definido num comando executável, o montador associa a êle o endereço em que êste comando HAL está sendo alocado. Um símbolo pode ser usado, porém, num comando anterior ao comando em que é definido.

Portanto, quando o primeiro é encontrado o montador ainda não associou ao símbolo usado no campo de operandos o seu endereço, isto é, nesta altura o montador não pode ainda determinar o campo EEEE da instrução correspondente. O mesmo ocorre se uma literal fôr usada. A solução a êste problema na maioria dos montadores é usar um algoritmo de tradução em dois passos: os comandos do programa HAL são lidos duas vêzes; da primeira vez, isto é, no primeiro passo do montador, os comandos são lidos e quando um símbolo é definido é alocado colocando na tabela de símbolos, na entrada correspondente, o seu endereço; quando uma literal é usada, é colocada na tabela de literais e quando todos os comandos já foram lidos no primeiro passo, o montador aloca tôdas as literais. Quando portanto se inicia o segundo passo o endereço de cada símbolo definido já está na tabela de símbolos, e a cada literal já há um endereço associado. Assim, os comandos ao serem lidos no segundo passo já podem ter completados todos os campos da instrução HIPO correspondente, isto é, o programa objeto pode ser gerado.

Os símbolos usados num programa HAL podem ser: o rótulo de um comando executável, quando então serão associados ao endereço da instrução HIPO correspondente a êste comando no programa objeto; ou o símbolo usado para uma variável ou constante do programa e neste caso devem ser reservadas uma ou mais palavras na memória para estas variáveis ou constantes e o endereço correspondente deve ser associado ao símbolo em questão. A posição na memória, relativamente às instruções e outras variáveis ou constantes do programa, e o número de palavras a ser reservado para a variável, ou o valor da constante, são especificados através de pseudos apropriados tendo por rótulo o símbolo da variável ou constante. Em HAL, a posição relativa é determinada pela posição da pseudo dentro do programa, isto é, a alocação do símbolo é feita quando a pseudo que define o símbolo é encontrada no endereço da palavra que está sendo montada.

Vejamos agora quais são as pseudos em HAL:

	Código de operação HAL	Significado
a)	ØRG	Esta pseudoinstrução indica ao montador o endereço da próxima palavra a ser alocada pelo montador. Êste endereço é especificado no campo de operando.
b)	TRØN	A posição desta pseudo indica o endereço de início do TRACE (v. cap. 1). Êste endereço vai ser colocado nas colunas 22 - 25 do cartão // EXECUTE HIPO a ser gerado pelo montador.
c)	TRØF	A posição desta pseudo indica o endereço de fim do TRACE. Êste endereço vai ser colocado nas colunas 27 - 30 do cartão // EXECUTE HIPO a ser gerado pelo montador.

d) ØFNP

A ocorrência desta pseudo gera uma letra Ø na coluna 32 do cartão //EXECUTE HIPO, indicando que não se quer uma mensagem "OVERFLOW" quando ocorrer uma sobrecarga durante a execução do programa.

e) END

Esta peseudo indica que êste é o último comando HAL dêste programa.

f) DC

Quando esta peseudo é encontrada é reservada uma palavra na memória cujo enderêço é associado, se o comando tiver um símbolo no campo de rótulo, a êste símbolo. Nesta palavra, ao tempo de carga do programa objeto será carregada a constante especificada no campo de operandos do comando, que pode ser alfanumérica, inteira, ou de ponto flutuante, valendo as mesmas regras descritas para a especificação de literais, respectivamente, alfanuméricas, inteiras ou de ponto flutuante.

g) DS

Quando o montador encontra esta pseudo reserva o número de palavras especificado no campo de operandos, e associa, se o comando tiver um símbolo no campo de rótulo, o enderêço da primeira destas palavras a êste símbolo.

h) DA

Quando o montador encontra esta pseudo reserva uma palavra na memória, cujo enderêço será associado, se o comando tiver um símbolo no campo de rótulo, a êste símbolo. Nesta palavra, durante a carga do programa objeto vai ser carregado um enderêço (que pode ser usado como enderêço indireto pelo programa) especificado pelos campos X, I e operando do comando, que pode ser qualquer tipo de operando dos usados em comandos executáveis, (exceto literal).

Exemplos:

Nos exemplos abaixo da pseudo DA suporemos que a próxima palavra a ser colocada pelo montador quando a pseudo é encontrada seja a de enderêço 0321 e que o símbolo A é associado ao enderêço 0583.

Ex 1: SYMBØL DA 3 I	A +7	Esta pseudo associará o endereço 0321 ao símbolo "SYMBØL". Durante a carga do programa objeto será armazenada nesta palavra o endereço + 0000310590.
Ex 2: EX2S DA	* -1	0321 associado ao símbolo "EX2S". Endereço carregado durante a carga em 0321: + 0000000320.
Ex 3: SYMB 3 DA	0800	0321 associado ao símbolo "SYMB 3". Endereço carregado durante a carga em 0321: + 0000000800.

i) EQ

O uso de um símbolo no campo de rótulo desta pseudo é obrigatória. Esta pseudo associa a êste símbolo o endereço associado ao operando que pode ser qualquer dos tipos usados em comandos executáveis, (exceto literal).

Exemplos: (suponhamos que o endereço associado ao símbolo A seja 0583, e que a próxima palavra a ser alocada pelo montador ao encontrar a pseudo seja 0321).

Ex 1 SYMBØL EQ	A + 7	Associa ao símbolo "SYMBØL" o endereço 0590. A próxima palavra a ser alocada continua sendo 0321.
Ex 2 EX2S EQ	* -1	Associa ao símbolo "EX2S" o endereço 0320. A próxima palavra a ser alocada continua sendo 0321.

A pseudo EQ apresenta uma diferença essencial em relação às outras pseudos que definem símbolos ou constantes: (DC, DS e DA), a saber que nesta pseudo o endereço associado ao símbolo que é o rótulo do comando não depende da posição da pseudo no programa HAL, mas é determinado pelo campo de operandos. Em conseqüência, se nenhuma restrição é feita ao seu uso, é possível que quando a pseudo é encontrada, êste endereço não possa ser determinado por ser usado no campo de operandos um símbolo ainda não definido, ou também definido numa pseudo EQ. Uma restrição freqüente em linguagens de montagem ao uso de comandos do tipo EQ, e que elimina êste problema, é que todos os símbolos usados no campo de operandos de uma pseudo EQ já tenham sido anteriormente definidos. Esta restrição é adotada no HAL.

2.2 As tabelas do montador:

As idéias básicas do montador já foram esboçadas na secção anterior. Foi visto que os montadores são normalmente algoritmos de dois passos devido ao fato de que o campo de operandos pode conter símbolos ou literais ainda não alocados; que a alocação de comandos é feita seqüencialmente; e que a montagem de cada instrução é feita consultando certas tabelas do montador.

Existem muitas maneiras diferentes de organizar um montador de dois passos. Em particular muitas das funções do montador podem ser incluídas no primeiro ou no segundo passo indistintamente. Nas secções seguintes veremos uma possível organização. As tabelas que usaremos serão:

1. Tabela de comandos - É uma tabela que contém por chave s, o código de operação HAL e por valor t associado à chave s, o código de operação HIPO correspondente. Só contém comandos executáveis.

2. Tabela de símbolos - Esta tabela é construída durante o processo de montagem e tem por chave o símbolo usado, e por valor associado à chave o endereço do símbolo correspondente.

3. Tabela de literais - Literais são constantes especiais que são alocadas pelo montador após todos os comandos do programa HAL terem sido alocados. A tabela de literais terá por chave a representação simbólica do literal, isto é, a cadeia de caractéres que definem o literal no campo de operando e por valor associado à chave, o endereço da palavra na qual o literal foi alocado, e a constante que deverá ser carregada nesta palavra durante a carga (isto é, a representação interna da constante definida pelo literal). Com esta organização, literais usados em vários comandos com a mesma representação simbólica serão todos associados à mesma palavra. Porém representações simbólicas diferentes da mesma representação interna serão alocadas em palavras diferentes. Por exemplo: =.61E22 ou =+.61E22 ou =6.1E+21 ou ="A ƀƀƀ K" ou = 6100000072 são todos literais com a mesma representação interna +6100000072. Uma alternativa para eliminar êste possível inconveniente seria usar por chave a representação interna da constante definida pelo literal, que porém apresenta a desvantagem de ser necessário transformar o literal para a representação interna (para determinarmos a chave), tôda vez que o literal é usado, mesmo quando a mesma representação simbólica é usada em vários comandos.

A alocação do programa, como já foi dito, é seqüencial, isto é, a memória vai sendo ocupada em ordem crescente de endereços pelos comandos na ordem em que êstes aparecem no programa HAL. A ordem de ocupação da memória pode ser alterada porém pelo uso de comandos ORG.

O programador porém nêste caso deve tomar cetos cuidados para não utilizar as mesmas posições de memória para finalidades distintas. O contrôle da alocação de memória será feito através de um contador de locação que chamaremos de C-LØC cujo valor durante a montagem, ao iniciar o processamento de um nôvo comando HAL, será o enderêço da próxima palavra da memória a ser alocada. Outros contadores podem ser usados para determinar certas quantidades que podem ser de valia para o programador como número total, de células de memória utilizadas pelo programa, número de células de memória utilizadas para instruções, número de células de memória utilizadas para dados etc. A determinação destas quantidades porém pode ser dificultada pelo uso de vários comandos ØRG.

As variáveis do montador serão denotadas em letras minúsculas sublinhadas. Os 80 caractéres de um comando simbólico serão lidos numa variável denominada de "comando". Os campos de rótulo, código mnemônico de operação, X, I, e operando de um comando serão denotadas por rót [comando], op[comando], x[comando], i[comando] e oper [comando], respectivamente. O campo de enderêço associado a um símbolo s na tabela de símbolos será denotada por end-simb [s]; o enderêço associado a um literal de representação simbólica 1 denotaremos por end-lit [1], e a representação interna correspondente do valor do literal por val-lit [1]; o código de operação HIPO associado na tabela de comandos a um mnemônico op de um comando executável será denotado por codop-com [op]; chave-simb [s] indicará o campo de chave, isto é, do símbolo s, na tabela de símbolos e chave-lit [1] o campo de chave, isto é, do símbolo da literal 1, na tabela de literais. b̸ representa o caráter branco.

2.3 O primeiro passo

PI 1. Inicialize a limpe tabelas; c-loc ← 10; tron ← 0; trof ← 0; of ← b̸;

PI 2. Leia próximo comando (cartão) em comando;

PI 3. Compare op [comando] com os códigos mnemônicos de pseudos;
Se fôr um pseudo vá para PI 13. (Análise de pseudos); senão, (ou é um comando executável ou é um comando ilegal. Para não fazer uma busca na tabela de comandos assumiremos neste ponto do primeiro passo que é um comando executável.):

PI 4. Faça s ← rót [comando]; se s = b̸b̸b̸b̸b̸b̸ vá para PI7 (análise do campo de operandos no primeiro passo); senão

PI 5. Procure o símbolo s na tabela de símbolos; se êste símbolo já está presente, é um êrro, (símbolo superdefinido, isto é, definido mais de uma vez), e vá para o passo PI 7; senão

PI 6. Coloque o simbolo s e seu endereço na tabela de simbolos, ié faça chave-simb [s] ← s; end-simb [s] ← c-loc;

PI 7. Se oper [comando] não fôr um literal (ié se o primeiro caráter deste campo não fôr o caráter: "=") vá para PI11;

PI 8. (É literal) faça l ← oper [comando] ;

PI 9. Procure a representação simbólica l na tabela de literais; se já estiver na tabela vá ao passo PI11; senão

PI 10. Faça chave-lit [l] ← l (ié coloque a representação simbólica da nova literal l na tabela de literais)

PI 11. Faça c-loc ← c-loc + 1;

PI 12. Grave comando (para leitura posterior no segundo passo) e vá a PI 2.

PI 13. (Análise de pseudos.)

Se op [comando] = TRØN, faça tron ←c-loc e vá para PI 12;
Se op [comando] = TRØF, faça trof ← c-loc e vá para PI 12;
Se op [comando] = ØFNP, faça of ← Ø e vá para PI 12;
Se op [comando] = END, grave comando e vá para PI 23 (alocação de literais);
Se op [comando] = ØRG, vá para PI 14.
Se op [comando] = DC vá para PI 15.
Se op [comando] = DS vá para PI 15.
Se op [comando] = DA vá para PI 15.
Se op [comando] = EQ vá para PI 20.

PI 14. (ØRG) (oper [comando] deve ser um endereço absoluto) Transforme a constante numérica em oper [comando] para a representação numérica interna, dando por resultado endereço; (em oper [comando] temos a representação alfanumérica deste endereço). Faça c-loc ← endereco e vá para PI 12;

PI 15. (DS ou DC ou DA) Faça s ← rót [comando]; se s = бббббб vá para PI 18; senão

PI 16. Procure o símbolo s na tabela de símbolos; se este símbolo já está presente é um erro (símbolo superdefinido), e vá para PI 11; senão

PI 17. Faça chave-simb [s] ← s; end-simb [s] ←c-loc;

PI 18. Se op [comando] ≠ DS vá para PI 11; senão

PI 19. (É DS) Transforme oper [comando] (que deve ser um número absoluto, indicando o número de palavras a ser reservada pelo pseudo DS) para a sua representação numérica interna dando por resultado endereço; Faça c-loc ← c-loc + endereço e vá para PI 12;

PI 20. (EQ) Faça s ← rót [comando] ; se s = bbbbbb é erro, e vá para PI 12; senão

PI 21. Execute a subrotina E, de cálculo do endereço especificado no campo de operandos, que dá por resultado endereço;

Procure s na tabela de símbolos; se já está presente é erro (símbolo superdefinido); senão:

PI 22. Faça chave-simb [s] ← s; end-simb [s] ← endereço; e vá para PI 12;

PI 23. (Alocação de literais) percorra a tabela de literais e para cada literal, faça:

l ← chave-lit [l] ; Transforme a literal l para a representação interna de seu valor, dando por resultado valor; Faça val-lit [l] ← valor e end-lit [l] ← c-loc; Faça c-loc ← c-loc + 1 e pegue a próxima literal; se não há mais literais a alocar na tabela vá para PI 11 (segundo passo do montador).

2.4 A Subrotina E.

Esta subrotina é utilizada tanto pelo primeiro passo (vide PI 21.) como no segundo passo e tem por objetivo determinar o endereço especificado no campo de operandos do comando, oper [comando] , dando por resultado endereço;

E1. Se oper [comando] é um número (ié endereço absoluto) Transforme-o para a sua representação interna, dando por resultado endereço e retorne da subrotina; senão

E2. Pegue o primeiro símbolo de oper [comando] colocando-o em s1;

E3. Se s1 ≠ * vá para E10; senão faça endereço ← c-loc;

E4. Faça s1 ← b̸; pegue o próximo caráter não branco de oper [comando] , se houver, e coloque-o em s1;

E5. Se s1 = b retorne da subrotina; senão

E6. Transforme o número que deve seguir o sinal não branco s1, para a sua representação interna, dando por resultado n;

E7. Se sl = +Faça endereço ← endereço + n e retorne da subrotina; senão

E8. Se sl = - Faça endereço ← endereço - n e retorne da subrotina; senão

E9. É erro, (expressão inválida no campo de operandos) e retorne da subrotina;

E10. Procure sl na tabela de símbolos; se não estiver presente é erro (símbolo não definido) e retorne; senão

E11. Faça endereço ← end-simb [sl] e vá para E4.

2.5. O Segundo Passo

No segundo passo todos os símbolos definidos no programa já estão na tabela de símbolos com os correspondentes endereços e todas as literais estão devidamente alocadas. Assim para cada comando simbólico lido é possível gerar a palavra correspondente no programa objeto (se é que há esta correspondência). O programa objeto é montado numa variável obj que tem dois campos: end [obj] e cont [obj] que são respectivamente os dois campos de um cartão de carga; cont [obj] e o conteúdo da palavra que será carregada na palavra de endereço end [obj]. No segundo passo o programa fonte e objeto é listado. Cada linha da listagem é montado em list com os campos com [list] e obj [list], contendo respectivamente o comando simbólico e o comando objeto.

PII1. Faça c-loc ← 10 e p ← 0;

PII2. Leia o próximo comando em comando, limpe as áreas list e obj

PII3. Se op [comando] fôr pseudo vá para PII14; senão

PII4. Se p = 0 faça endex ← c-loc e p←1; senão

PII5. Procure op [comando] na tabela de comandos; se não está na tabela é erro (comando ilegal) e vá para PII2; senão

PII6. Faça código ← codop com [op [comando]]

Transforme x [comando] para a sua representação numérica interna dando x ;

PII7. Se i [comando] = I ou 1 faça i ← 1; senão faça i ← 0;

PII8. Se oper [comando] não fôr literal vá para PII3; senão

PII9. Faça l ← oper [comando] ; procure l na tabela de literais; faça e ← end-lit [l] ;

PII10. Monte cont [obj] com a instrução do HIPØ de código de operação código; campo X x; campo I i; e campo EEEE e; transmita para end [obj] c-loc;

PII11. Faça c-loc ← c-loc + 1;

PII12. Faça com [list] ← comando e obj [list] ← obj; Imprima a linha list e grave obj ; e vá para PII2.

PII13. Execute a subrotina E que dá por resultado o endereço especificado em oper [comando] em endereço; Faça e ← endereço e vá para PII10;

PII 14. Se op [comando] = ØRG transforme a constante numérica em oper [comando] para a representação numérica interna, dando por resultado endereço; faça c-loc ← endereço e vá para PII12;

PII 15. Se op [comando] = END faça com [list] ← comando, imprima a linha list; percorra a tabela de literais e para cada literal l:
Transmita end-lit [l] para end [obj] e val-lit [l] para cont [obj] ; limpe list; faça obj [list] ← obj; imprima a linha list ; e grave obj ;
Quando não houver mais literais gere o cartão de execução em obj (// EXECUTE HIPØ...) com endereço de início de execução endex, início de trace tron, fim de trace trof, e controle de mensagem de sobrecarga of ; encerre a montagem.

PII16. Se op [comando] = DC, determine a representação interna da constante especificada em oper [comando] dando val; Transmita c-loc para end [obj] e val para cont [obj] e vá para PII11.

PII17. Se op [comando] ≠ DA vá para PII21; senão, transforme x [comando] para a sua representação interna dando x;

PII18. Se i [comando] = I ou 1, faça i ← 1, senão faça i ← 0;

PII19. Execute a subrotina E achando o endereço especificado em oper [comando] e dando por resultado endereço;
Faça e ← endereço;

PII20. Monte uma palavra em cont [obj] com o campo X x; Campo I i; e campo EEEE e; transmita c-loc para end [obj] e vá para PII1;

PII21. Se op [comando] = DS Transforme oper [comando] para a sua representação numérica interna dando por resultado endereço; Faça c-loc ← c-loc +endereço e vá para PII12; senão

PII22. (TRØN, TRØF, ØFNP, EQ) vá para PII12.

2.6. Observações:

O montador deve ter recursos para descobrir os erros detectáveis durante a montagem, e se o erro fôr suficientemente grave não gerar o programa objeto, mas apenas a listagem do programa indicando os erros descobertos. Alguns dos erros estão indicados na descrição do montador na secção anterior, como símbolos superdefinidos, subdefinidos, EQ ilegal de símbolos, comando ilegal etc. Outros porém foram omitidos devido a descrição dada não ser de suficiente grau de minúcia para incluí-los. Este é o caso dos erros de formação das diversas entidades sintáticas que compõem um comando, como rótulo inválido, expressão no campo de operandos mal formada, constante inválida, literal inválida, etc. Estes erros são detectados por um reconhecedor sintático, cujos detalhes de implementação não foram indicados.

Um comando consiste de uma cadeia de 80 caracteres alfanuméricos. No HIPØ, onde cada palavra contém 5 caractéres alfanuméricos, um comando seria lido em 16 palavras da memória, que indicamos na descrição do montador por comando. No HAL como vimos os diversos campos do comando aparecem em certas posições préfixadas. É função do reconhecedor sintático separar de cada vez o campo em análise do comando e analisar a cadeia de caracteres que o formam verificando se se trata de um campo bem formado sintaticamente. Por exemplo o campo de rótulo no HAL consiste de 6 caractéres, que podem ser ou deixadas em branco, ou conter um símbolo que consiste de 1 a 6 caracteres, sem brancos intercalados, sendo o primeiro caráter obrigatoriamente uma letra devendo ser perfurada na coluna 1. Assim é o analisador sintático que deve descobrir que 03bbb, AbBbbb, bALFAb, não são rótulos válidos.

O montador descrito é um montador de dois passos. A necessidade de dois passos foi exposta na secção 2.1. Se porém ao invés de construir cada instrução do programa objeto, uma por vez, todo o programa objeto fôr armazenado na memória, o montador pode ser de um passo só (no sentido de haver necessidade de ler os comandos fonte uma vez só) pois comandos que envolvem símbolos indefinidos podem ser parcialmente traduzidos, sendo completados quando o símbolo em questão fôr definido.

Se guardarmos todo o programa objeto na memória, a montagem de um passo só pode ser feito do seguinte modo:

Ao encontrar um comando simbólico, o único campo que pode eventualmente não ser imediatamente montado, por conter um símbolo ou literal não definido, é o campo de operandos. Todos os outros campos podem ser montados sem transtorno ao ser encontrado o comando simbólico. Vejamos então o que o montador deve fazer com o campo de operan

dos. O campo de operandos como sabemos pode conter um número, (ié, endereço absoluto), caso em que pode ser imediatamente montado; um * ou uma expressão do tipo * + constante ou * - constante casos em que o campo de operandos também pode ser montado imediatamente pois o "valor" do símbolo * é o conteúdo do contador de passos; um símbolo já definido ou uma expressão do tipo símbolo + constante ou símbolo - constante, são casos em que ainda o campo de operandos pode ser imediatamente montado pois o "valor" de um símbolo já definido encontra-se na tabela de símbolos; e finalmente uma literal, ou um símbolo ainda não definido ou uma expressão do tipo símbolo ainda não definido + constante ou símbolo ainda não definido - constante, são os casos em que o campo de operando não pode ser imediatamente montado. Se o operando fôr uma literal, quando a literal é encontrada pela primeira vez é colocada na tabela de literais, e o endereço da instrução é colocado também na tabela de literais; quando o segundo uso da literal é encontrado o endereço da instrução é colocado no campo de endereços da instrução do primeiro uso, e assim por diante: para cada literal formamos uma lista ligada de todos os seus usos. Assim quando a literal fôr finalmente alocada e seu endereço conhecido, este pode ser colocado nos campos de endereço de todos os comandos em que a literal é usada. O mesmo procedimento pode ser feito com símbolos ainda não definidos. O caso de expressões envolvendo símbolos não definidos exige um tratamento um pouco mais complexo, pois neste caso é necessário guardar a constante que deve ser algebricamente somada ao endereço do símbolo quando este fôr conhecido. A idéia fundamental no entanto continua a mesma: da tabela de símbolos temos um apontador para o primeiro uso do símbolo, de onde temos um apontador para um lugar que ou é o segundo uso, ou é um campo com um indicador de que se trata de uma constante a ser somada ao endereço do símbolo, o valor algébrico desta constante, e um apontador para o segundo uso, e assim por diante. Expressões que envolvem vários símbolos, e que no HAL não são permitidos exigem estruturas de dados ainda mais complexas mas a idéia fundamental é sempre a mesma: encadear os campos que não podem ser imediatamente montados para posterior revisão no momento em que todos os endereços envolvidos forem conhecidos.

Do mesmo modo, se não houver necessidade de alocar os comandos sequencialmente, o montador também pode ser de um passo só. Os computadores modernos em sua esmagadora maioria porém executam suas instruções em seqüência, exceto quando encontram uma instrução de desvio, o que exige alocação sequencial das instruções. Mesmo assim, já foi construído pelo menos um computador em que esta exigência foi levantada. Trata-se do IBM-605 que tinha instruções de 2 endereços: o primeiro endereço especifica o parâmetro da instrução tal qual no HIPO, e o segundo especifica o endereço da próxima instrução a ser executada. Tudo se passa como se cada instrução fosse combinação de uma instrução normal e uma de desvio incondicional. Para o IBM-605 portanto é possível fazer um montador de um passo só, pois não há necessidade de alocação sequencial.

O montador que foi visto gera um programa objeto absoluto, ié, os endereços gerados serão efetivamente as posições da memória em que o programa será armazenado para execução. Como se verá no cap. 3 isto em certos casos é inconveniente de modo que o montador deve ter uma opção em que gere um programa objeto relativo. Em particular a instrução ORG especifica uma origem absoluta no caso visto. Para poder

ser especificado uma origem relativa é necessário que no campo de operandos possa ser usado um enderêço simbólico. Deve-se porém exigir que todos os símbolos usados num ORG já tenham sido definidos anteriormente, caso contrário pode acontecer que não seja possível determinar qual o enderêço que deve ser designado a c-loc pelo ORG, ou que a determinação dêste enderêço exija mais que dois passos. Com a restrição acima, basta que para a determinação do enderêço especificado pelo operando se execute a subrotina E.

Capítulo 3

O processo de "link-edição" e a carga do programa objeto na memória

3.1. Introdução

Programas em linguagem de máquina, absoluta, podem ser carregados na memória, no HIPO, automaticamente através do seu estado de carga. Em outros computadores em geral não existe um recurso de "hardware" deste tipo, e programas têm de ser carregados por um pequeno programa chamado de carregador. Se o programa estiver em forma absoluta o carregador será um algoritmo muito simples, análogo ao algoritmo por "hardware" do estado de carga do HIPO. Como logo veremos porém, nem sempre é conveniente que tenhamos programas em forma absoluta. Nestes casos o carregador será mais complexo, e é o que examinaremos neste capítulo.

Programas são escritos geralmente numa linguagem de programação como a linguagem de montagem, FORTRAN, ALGOL, etc.. Um programa escrito em uma destas linguagens, antes de poder ser executado, precisa ser traduzido para a linguagem de máquina do computador, o que é feito por um programa denominado de compilador; este último é carregado na memória e então executado. Para a programação de problemas complexos em geral é conveniente subdividir o problema em subproblemas menores, exprimindo a solução do problema complexo em função da solução dos subproblemas. Se há à disposição várias linguagens de programação, pode ocorrer que para diversos subproblemas, diferentes linguagens sejam as mais adequadas. Assim, a solução de um problema complexo pode envolver a programação de diversos subproblemas programados em várias linguagens diferentes. Cada um dos subproblemas deve ser compilado independentemente dos demais e na hora da carga juntados, formando o programa em linguagem de máquina que corresponde ao problema total original.

As diversas partes independentes de um programa serão chamadas de módulos.

Para que um esquema deste tipo funcione é preciso que dois problemas sejam resolvidos. O primeiro problema é que sendo o módulo uma parte apenas de um programa maior, e devendo ser compilado independentemente dos demais módulos cuja extensão e outras características são desconhecidas na hora da compilação, não é possível determinar a posição absoluta em que o módulo deverá ser carregado, ié o compilador não pode compilar o módulo para linguagem de máquina em forma absoluta. Mais ainda, o mesmo módulo pode ser utilizado em programas diferentes, ocupando nos dois programas ao tempo de execução posições diferentes na memória. A solução para este problema é deixar variável a posição inicial em que o módulo vai ser carregado. O módulo é compilado como se fosse carregado a partir da célula 0000 na memória. Os seus endereços que se referem a células do módulo não são assim endereços absolutos, mas relativos à origem do módulo. Por exemplo o endereço relativo 0320 não se refere à palavra 0320 na memória, mas à 0321-ima palavra do <u>módulo</u>, ié o seu verdadeiro endereço será

p + 0320 onde p é o endereço da palavra em que o módulo vai ser efetivamente carregado. O módulo nesta forma diz-se estar em forma relocável.

O segundo problema é que sendo os diversos módulos partes de um programa maior é preciso que se intercomuniquem na hora da execução. Assim, embora sejam compilados independentemente é preciso que hajam recursos de intercomunicação entre módulos independentes incorporados às linguagens de programação em que os módulos são programados. De uma maneria geral estes recursos se limitam a ter acesso em um módulo a um endereço de outro módulo. Este endereço pode ser um ponto do programa do segundo módulo, para o qual o primeiro deve desviar em algum ponto de sua execução, ou pode ser o endereço de uma área de dados do segundo módulo, que se deseja utilizar no primeiro. Em qualquer dos casos o endereço deve ser associado a um símbolo definido no segundo módulo mas usado no primeiro. Sendo assim o endereço associado a este simbolo é desconhecido na hora em que o primeiro módulo é compilado. Assim o compilador não pode traduzir para linguagem de máquina, inteiramente, os comandos que utilizam este símbolo. Os outros comandos, entretanto, que independem dos recursos de intercomunicação podem ser inteiramente compilados. (Normalmente estes comandos constituem a maior parte do programa.) Na hora da execução porém todos os comandos devem estar em linguagem de máquina. Os comandos portanto que foram só parcialmente compilados precisam ser completados antes da execução. Esta tarefa cabe ao carregador, que por ter acesso a todos os módulos do programa pode completar os comandos incompletos.

Devido aos dois problemas mencionados os compiladores não podem compilar os diversos módulos para linguagem de máquina absoluta, pois os endereços que se referem a uma célula do módulo sendo compilado serão endereços relativos, e comandos que utilizam células de outros módulos não podem ser inteiramente compilados. Por este motivo os compiladores compilam não para linguagem de máquina absoluta, mas para uma linguagem intermediária, na qual os endereços são relocáveis e os recursos de comunicação intermódulos são preservados. Uma vez porém que os módulos estejam em linguagem intermediária, ao carregador é virtualmente impossível saber qual a linguagem fonte da qual o programa foi compilado. Para ele todos os módulos estão em linguagem intermediária, sendo por ele transformados para linguagem de máquina absoluta. Não constitui portanto nenhuma complicação adicional que os diversos módulos originalmente fossem programados em linguagens de programação diferentes.

3.2. O problema da relocação:

Um módulo, em forma relocável, conforme as idéias da secção anterior, é um conjunto de células de memória contíguas, cuja origem (ié endereço da primeira célula do módulo) é variável. Todos os endereços que se referem a células do módulo são endereços relativos ao começo do módulo e precisam ser ajustadas, ié relocadas, na hora em que o módulo é carregado a partir da célula de endereço p da memória. Relocar um endereço significa somar a este endereço relativo a origem p do módulo.

Por exemplo, o módulo num certo ponto possui uma instrução de desvio incondicional para o endereço relativo e. O endereço absoluto correspondente será e + p, e assim a parte de endereço da instrução de desvio deve ser incrementada de p. Nem todas as palavras do módulo devem sofrer este acréscimo porém. Assim, uma constante do programa (que não seja um endereço relativo) como o valor de π por exemplo não deve ser alterada; uma instrução que endereçe uma palavra fixa na memória, (como um indexador por exemplo) não deve ser alterada; uma instrução imediata não deve ser alterada, etc..

O carregador deve ter elementos para poder decidir quais as palavras do módulo que devem ser incrementadas de p. A partir de um módulo em linguagem de máquina é impossível determinar quais as palavras que devem ser incrementadas. Essencialmente a dificuldade é o fato de que num programa em linguagem de máquina, dados, constantes, endereços relativos, endereços absolutos, instruções, etc., são indistinguíveis. Por exemplo a instrução +001200 0001 (STA) tanto pode ser uma instrução para armazenar o acumulador no indexador 1, caso em que o endereço 0001 será absoluto como uma para armazenar o acumulador na palavra 1 do módulo, caso em que o endereço 0001 será relativo e deve ser acrescido de p. Durante a compilação porém cada endereço, constante, etc. tem seu significado claramente especificado, e assim ao compilador é relativamente fácil descobrir quais os endereços que devem ser relocados e quais as palavras que devem ser inalteradas durante o processo de relocação. O compilador indica então quais as palavras relocáveis através de um indicador associado a cada palavra do módulo, ou através de uma tabela relacionando todas as palavras do módulo a serem incrementadas.

3.3. Facilidades de intercomunicação intermódulos

Já vimos que o problema da comunicação intermódulos se resume em ter acesso num módulo A a um endereço associado a um símbolo S_1 de um módulo B. O símbolo S_1 em relação ao módulo A se diz um símbolo externo. Como na hora da compilação de A o endereço do símbolo S_1 é desconhecido, em cada uso do símbolo S_1 devemos preservar a informação de que neste ponto temos uma referência a um símbolo externo de nome S_1. Cada símbolo externo deve ser portanto relacionado numa tabela, que contém ainda os endereços (relativos) de todas as palavras nas quais S_1 é usado no módulo. Assim esta tabela que chamaremos de tabela dos símbolos externos ou tabela de uso terá a seguinte estrutura:

Símbolo externo	Nº de usos	Endereços dos usos			
S_1	n_1	E^1_1	E^1_2	....	$E^1_{n_1}$
S_2	n_2	E^2_1	E^2_2	...	$E^2_{n_2}$
⋮					
S_m	n_m	E^m_1	E^m_2	...	$E^m_{n_m}$

TABELA DE USO ou TABELA DE SÍMBOLOS EXTERNOS

O carregador na hora da carga deve descobrir que S_1 é um símbolo do módulo B, conseguir seu endereço e substituir este endereço em todas as palavras em que S_1 é usado, procedendo de modo análogo com os outros símbolos externos. Deste modo, ao tempo de execução A pode ter acesso ao endereço correspondente a um símbolo externo.

Por outro lado S_1 é um símbolo definido no módulo B. Quando B fôr compilado, fica determinado o endereço relativo que corresponde a S_1. Depois da compilação porém um símbolo normalmente fica dissociado de seu endereço. Assim, se o mesmo acontecesse a S_1, na hora da carga não haveria meio de determinar que S_1 é um símbolo de B e qual o seu endereço relativo em B. Deste modo não basta que guardemos S_1 na tabela de símbolos externos de A; é preciso também não deixar dissociar o endereço de S_1 do seu símbolo S_1, a fim de que o carregador possa determinar que S_1 é um símbolo de B e qual o seu endereço. Assim no módulo B é preciso guardar numa tabela todos os símbolos deste módulo que podem eventualmente ser usados por um outro módulo, juntamente com seu endereço. Um símbolo S_1 definido num módulo B mas eventualmente usado num outro módulo diz-se um símbolo público em relação a B. Portanto todos os símbolos públicos de um módulo devem ser relacionados numa tabela contendo o símbolo e seu endereço relativo dentro do módulo. Esta tabela se denomina TABELA DE SÍMBOLOS PÚBLICOS, ou TABELA DE DEFINIÇÃO.

Fica claro que símbolos não são por si mesmos nem externos nem públicos: o símbolo S_1 de nosso exemplo é externo em relação ao módulo A, e público em relação ao módulo B. A figura a seguir ilustra a tabela de definição:

Símbolo (Público)	Endereço relativo no módulo
S_1	E_1
S_2	E_2
⋮	⋮

A estrutura da tabela de uso descrita deixa a desejar pois o número de usos de um símbolo externo é variável de símbolo para símbolo, e assim o espaço ocupado por cada símbolo é variável na tabela. A estrutura pode ser melhorada eliminando este inconveniente, se observarmos que os campos em que o módulo utiliza o símbolo externo não contém informação relevante até a hora da carga dos módulos, quando o carregador ali coloca o endereço do símbolo externo. Assim estes campos podem ser aproveitados para conter o endereço do próximo uso do símbolo externo, tornando a tabela de uso com uma estrutura ligada. ("linked"). A tabela conteria assim o símbolo externo, e o endereço relativo do primeiro uso; no campo de endereço da palavra correspondente ao primeiro uso teremos o endereço do segundo uso; e assim por diante.

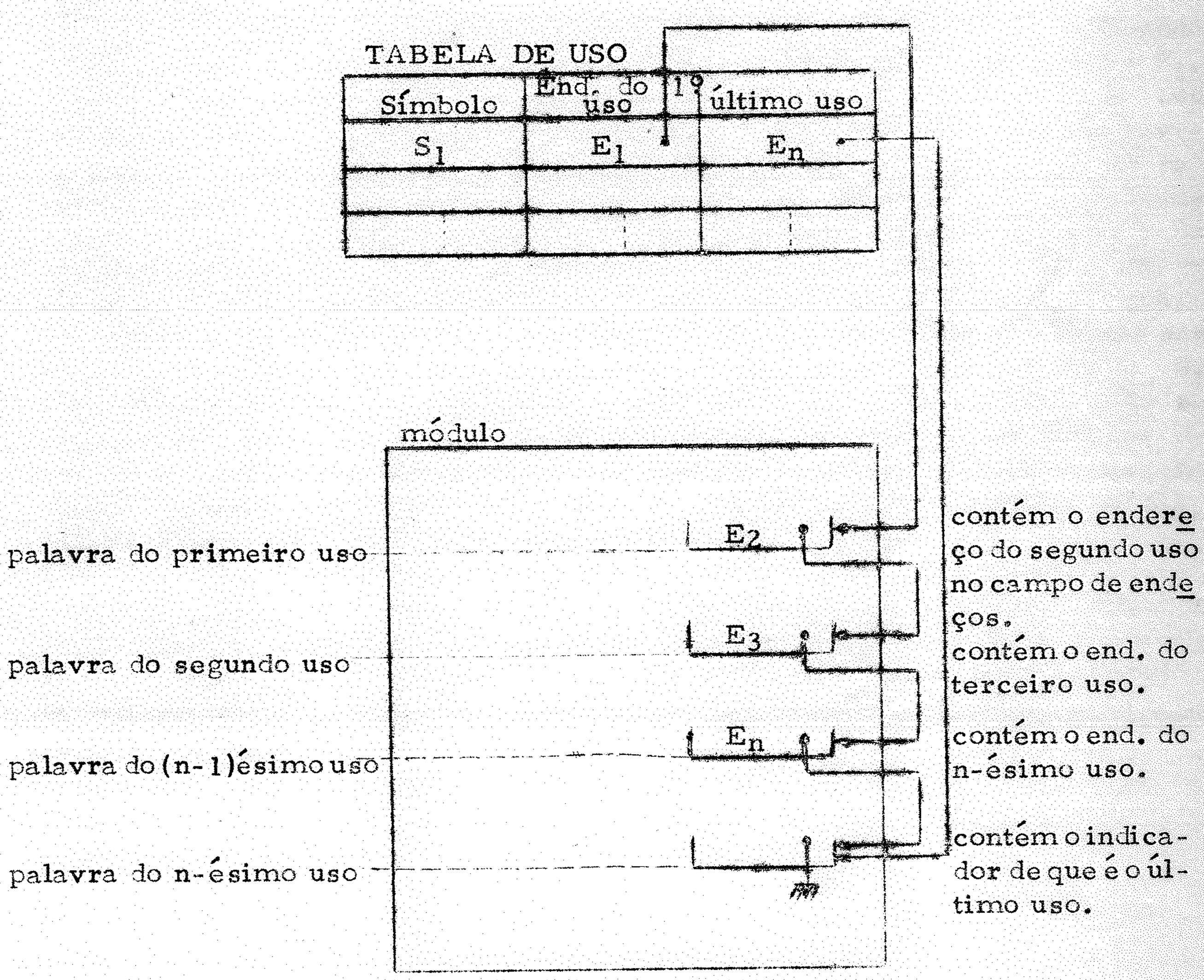
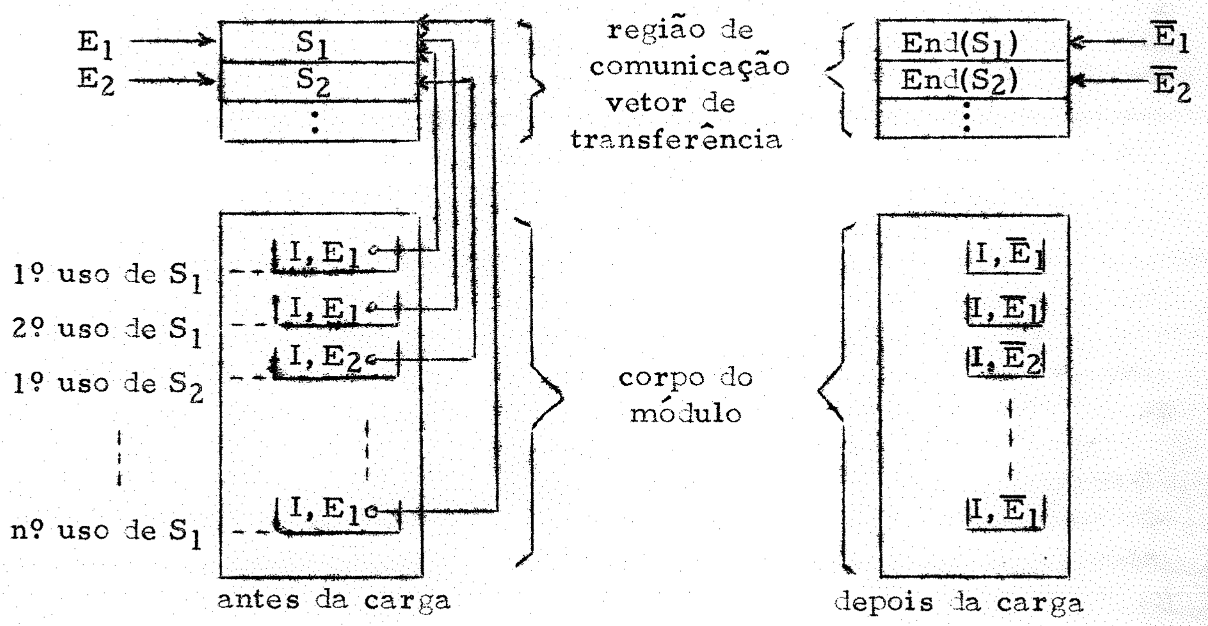


Tabela de uso com estrutura ligada

A figura anterior ilustra o esquema da tabela de uso com estrutura ligada. Para cada símbolo externo foi nela incluida um apontador para o último uso do símbolo externo. Assim quando a tabela é construída pelo compilador e este determina o endereço En + 1 do (n + 1)ésimo uso, o endereço do n-ésimo (ié o último até esta altura) uso está diretamente disponível. Basta portanto ali armazenar En + 1, colocar na palavra En +1 o apontador especial indicando que é o último uso, e atualizar o apontador de último uso armazenado nele En + 1.

O esquema da tabela de uso com estrutura ligada elimina o inconveniente anteriormente citado, e é uma solução adequada quando se quer plena generalidade de comunicação entre módulos. A solução tem o inconveniente de ser pouco eficiente, pois na hora da carga o endereço do símbolo externo S_1 tem de ser armazenado n_1 vêzes, a saber no campo de endereço de cada um dos n_1 usos do símbolo S_1. Se restringirmos um pouco a generalidade com que símbolos externos podem ser usados é possível evitar também este inconveniente, substituindo o símbolo externo pelo seu endereço apenas uma vez no módulo, independentemente do número de usos do símbolo. Isto pode ser conseguido com o auxílio de endereçamento indireto. Esta técnica consiste em relacionar numa região de comunicação do módulo, denominada de vetor de transferência todos os símbolos externos do módulo, e para cada uso de um símbolo S_1 no módulo fazemos uma referência indireta à palavra no vetor de transferência que contém o símbolo S_1. O carregador na hora da carga substitui o símbolo S_1 no vetor de transferência pelo seu endereço absoluto e assim na hora da execução, a instrução que utiliza S_1 tem acesso ao seu endereço através de endereçamento indireto. O vetor de transferência é portanto uma tabela de símbolos externos especial que faz parte do módulo durante a execução, pois há no módulo referências indiretas a seus elementos. A tabela de uso com estrutura ligada, em contraste, é utilizada apenas como informação auxiliar para o carregador, perdendo a função depois da carga. O mesmo acontece com a tabela de definição.

A figura a seguir ilustra o mecanismo do vetor de transferência.



A técnica do vetor de transferência.

Legenda: E_1 , E_2 - endereços (relativos) das palavras do vetor de transferência que contém S_1 e S_2.

$\overline{E}_1$, $\overline{E}_2$ - endereços (absolutos) das mesmas palavras.

End(S_1), End(S_2) - endereços (absolutos) de S_1 e S_2.

A técnica de usar um vetor de transferência restringe as facilidades de intercomunicação, pois, usando endereçamento indireto para ter acesso ao símbolo externo impede a utilização direta do endereço em instruções imediatas por exemplo. Em outras palavras a manipulação de endereços externos fica dificultada, embora não impossibilitada já que pode-se obter o endereço do símbolo externo em si fazendo-se acesso (não indireto) ao vetor de transferência.

3.4. Símbolos externos e públicos em FORTRAN E HAL

O esquema de comunicação intermódulos que vimos na secção 3.3. pode ser usada em diversas linguagens que permitem a subdivisão do programa em módulos. Como o compilador deve construir além das informações para a relocação do módulo uma tabela de definição e uma tabela de uso (ou então um vetor de transferência) para os símbolos externos, para cada módulo, as linguagens devem ter recursos projetados para o compilador poder reconhecer quais os símbolos externos e públicos do módulo. Nesta secção examinaremos como isto é feito em FORTRAN e no HAL.

No FORTRAN um símbolo externo pode ser o nome de uma subrotina, associado à primeira instrução a ser executada desta subrotina. Os módulos em FORTRAN são as subrotinas e o programa principal. Um módulo

usar um símbolo externo S em FORTRAN significa chamar a subrotina S neste módulo. Uma subrotina em FORTRAN pode ser de tipo função ou não. Se a subrotina não fôr do tipo função a chamada é feita através de um comando CALL que tem o formato:

CALL <nome da subrotina> (<par$_1$> , <par$_2$> , . . . ,<par$_n$>)

Este comando especifica além dos parâmetros, que se deve desviar para a subrotina <nome da subrotina> , e que este símbolo é um símbolo externo para este módulo. Em resposta a um comando CALL o compilador, supondo que a técnica usada para a comunicação intermódulos seja por meio do vetor de transferência, irá verificar se o símbolo externo especificado pelo comando já está no vetor de transferência; se não estiver reserva uma palavra no vetor de transferência onde coloca o símbolo externo em questão e compila uma instrução de desvio para subrotina, indireto, para esta palavra do vetor de transferência que contém agora o símbolo externo;se o símbolo externo já estiver presente no vetor de transferência apenas compila uma instrução de desvio para a subrotina, indireto para a palavra do vetor de transferência que contém o símbolo externo. Os parâmetros são transmitidos na hora da execução conforme foi discutido no capítulo 1.

Se a subrotina fôr do tipo função a chamada é feita dentro de um comando de definição de variável numa expressão que contém uma referência à função através da especificação do nome da função seguida de seus parâmetros separados por vírgula e entre parentesis. Ex.: suponhamos que XM seja uma função:

A = B * C + D(I, J) - XM (A,B)

Quando o compilador encontra um símbolo seguido de abre parêntesis, (, verifica se o símbolo não é uma matriz, ié se não foi previamente definido num comando DIMENSION; se não fôr verifica se não é uma função definida dentro do próprio módulo por um comando de definição de função, ("statement function"); se ainda não fôr então o compilador assume que o símbolo é um símbolo externo nome de uma função, e compila uma instrução de desvio para subrotina, indireto para a palavra de vetor de transferência que contém este símbolo externo, analogamente ao caso discutido de chamada de subrotina por meio de CALL.

Vimos assim como o compilador FORTRAN conclui que um determinado símbolo é externo. Examinemos agora como determina que um símbolo é público. Um nome de subrotina é definida em FORTRAN através de um comando FUNCTION se fôr do tipo função, ou SUBROUTINE em caso contrário. Em ambos os casos estes comandos indicam que o símbolo seguinte é um símbolo público, que pode ser usado externamente e deve portanto ser incluido na tabela de definição da subrotina, juntamente com o seu endereço que será o endereço do início da subrotina.

Assim: SUBROUTINE <nome da subrotina> (<par$_1$> , . . . , <par$_n$>) indica que <nome da subrotina> é um símbolo público. Análogo é o caso em: FUNCTION <nome da função> (<par$_1$> ,<par$_2$>, . . . ,<par$_n$>).

Alguns compiladores FORTRAN permitem que uma subrotina tenha vários pontos de entrada. Os pontos de entrada são especificados neste caso por um comando ENTRY que também define um símbolo público.

Em HAL símbolos públicos e externos são especificados através de pseudos reservados para este fim. Símbolos públicos são declarados através de um pseudo ENTR que tenha no campo de operandos o símbolo que se deseja declarar público. Como um símbolo público é um símbolo definido dentro do módulo ele é definido como qualquer outro símbolo interno do módulo, comparecendo no campo de rótulo de algum comando executável ou pseudo. Símbolos externos são declarados através de um pseudo EXT , que tenha por rótulo o símbolo externo em questão.

3.5. O algoritmo do carregador e link editor

A função do link-editor é juntar os módulos de um programa, estabelecer as necessárias ligações de intercomunicação, e transformar o programa em linguagem de máquina absoluta. O carregador tem por função carregar o programa na memória. O algoritmo que exporemos incorpora estas duas funções, embora as duas funções possam ser encaradas como independentes e em certos casos pode ser conveniente fazer a link-edição sem necessariamente carregar o programa na memória.

Para que o carregador link-editor possa fazer a sua tarefa é necessário que tenha acesso a todos os módulos do programa, em linguagem intermediária, em um meio de armazenamento como por exemplo o disco magnético . Suporemos que no disco o sistema operacional mantenha um diretório de todos os módulos nele armazenados, e dos respectivos símbolos públicos.

Para controlar a link-edição o algoritmo que veremos mantém uma tabela de todos os símbolos públicos já carregados na memória, com os respectivos endereços já relocados. A técnica de comunicação intermódulos será o vetor de transferência. O algoritmo que veremos mantém também uma tabela da localização do próximo símbolo externo a ser processado em cada vetor de transferência e o número de seus símbolos externos ainda a processar. Deste modo quando um símbolo externo é achado pelo algoritmo, é possível verificar se o módulo correspondente já está na memória, verificando se este símbolo já está presente na tabela de símbolos públicos carregados. Em caso afirmativo o módulo correspondente já está na memória e o algoritmo pode obter o endereço (absoluto) do símbolo em questão; em caso negativo o módulo ainda não foi carregado. Com este esquema o algoritmo evita que o

mesmo módulo seja carregado mais de uma vez na memória.

As variáveis do algoritmo serão denotadas com letras minúsculas. As variáveis principais com os respectivos significados são as seguintes:

S - símbolo externo sendo processado.

c-rel - constante de relocação.

i - índice do vetor de transferência sendo processado.

n - número de vetores de transferência na memória.

m-req - memória requerida pelo módulo.

m-disp - memória total disponível.

TABELAS:

vetores de transferência

endereço do próximo símbolo externo a ser processado no vetor de transferência i. variável: end-vt[i]	número de símbolos externos a serem processados ainda no vetor de transferência i. variável: n-se [i]

vetor de transferência i

S_1 S_2 S_3 ... S_n

tabela de símbolos públicos carregados

Símbolo público	endereço: end - sp [i]

LE1. Leia nome do módulo principal em s ; Faça c-rel ← 0; i ← 1; n ← 0;

LE2. Procure no diretório módulo com símbolo público s ;

LE3. Se s está ausente no diretório o algoritmo termina ; (módulo com símbolo público s ausente do disco); senão localiza o módulo correspondente no disco ;

LE4. Lê do disco a memória requerida pelo novo módulo em m-req ; Se c-rel + m-req > m - disp o algoritmo termina; (programa exige mais memória do que a memória disponível); senão

LE5. Incorpora na tabela de símbolos públicos carregados a tabela de símbolos públicos do novo módulo somando ao endereço de cada símbolo público lido a constante de relocação c-rel ; Incorpora à tabela dos vetores de transferência a localização do primeiro símbolo externo no vetor de transferência deste módulo incrementado de c-rel e o número de símbolos neste vetor de transferência: Faça n←n + 1 ;

LE 6. Carrega e reloca o módulo, a partir do endereço c-rel em palavras contíguas, somando c-rel aos endereços que devem ser relocados conforme o indicador de relocação;

LE 7. Se n-se [i] = 0 vá ao passo LE 10; senão
Faça s ← CONT [end-vt(i)] (Pegue próximo símbolo externo)

LE 8. Procura símbolo s na tabela de símbolos públicos carregados;
Se ausente vá ao passo LE 2.;
Se presente há entrada j da tabela, Faça ·

CONT [end-vt(i)] ← end-sp [j] ; (substitui símbolo externo no vetor de transferência pelo seu endereço); n-se [i] ← n-se [i] - 1;

LE 9. Faça end-vt [i] ← end-vt [i] + 1 (endereço do próximo eventual símbolo externo no vetor de transferência i); vá ao passo LE 7.

LE 10. (vetor de transferência i já processado) Faça i ← i + 1;
Se i ≤ n vá ao passo LE 7 .; senão o algoritmo termina. (todos os módulos do programa estão carregados na memória, e todos os símbolos externos foram substituidos pelos seus endereços).

O algoritmo acima ou termina com sucesso no passo LE 10. com todos os módulos devidamente carregados, e todos os símbolos externos em todos os vetores de transferência substituidos pelos respectivos endereços, ou termina por falta de memória no passo LE 4., ou por falta de algum dos módulos necessários no disco no passo LE 3.

3.6. A técnica da superposição (overlay)

O carregador que vimos na secção anterior carrega todos os módulos na memória. Durante a execução do programa porém, num ponto arbitrário da execução, em geral não é necessária a presença de todos os módulos do programa. Como já vimos na secção 1.6., com a programação de sistemas de programação cada vez mais complexos a memória requerida pelo programa era cada vez maior, tornando a escassez de memória disponível um problema a ser enfrentado. Se portanto, durante a execução de um determinado módulo a presença de um outro fôr desnecessária na memória, deixar não obstante todos os módulos na memória constitui um desperdício de memória que eventualmente poderia ser evitado. O reconhecimento destes fatos levou à técnica de superposição de módulos na memória ("overlay"), que é um método de utilização da memória mais eficiente do que a carga pura e simples de todos os módulos antes de iniciar a execução do programa.

A técnica de superposição consiste essencialmente em carregar um módulo na memória apenas quando a sua presença se torna necessária. Naturalmente se o módulo fosse nesta altura carregado na mesma posição em que seria carregado se todos os módulos fossem carregados simultaneamente, como o fizemos na secção anterior, então nenhuma economia de memória resultaria carregá-lo apenas quando fosse necessário; a economia resulta do fato de este módulo poder ser carregado na mesma posição em que está um outro e cuja presença é desnecessária na memória enquanto o primeiro permanecer na memória. Vários módulos podem assim ocupar a mesma área da

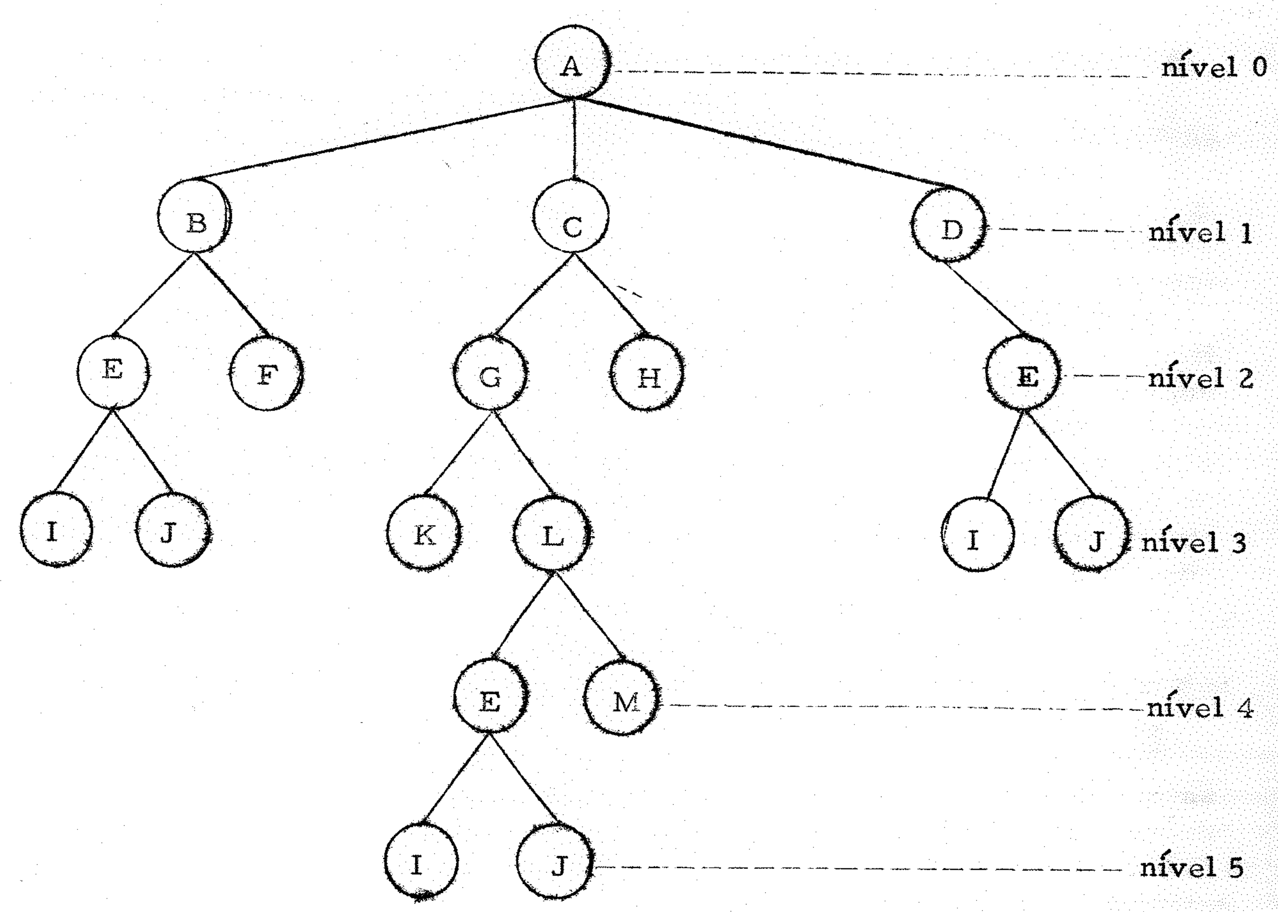
memória, em instantes distintos durante a execução.

Na técnica de superposição os módulos são guardados em forma absoluta, portanto já link-editados, a fim de tornar a carga de cada módulo durante a execução o mais rápido possível. Para que o programa possa ser link-editado é necessário portanto, ao tempo da link-edição, determinar quais os módulos que podem ocupar as mesmas áreas de memória, ié que podem ser superpostos. Muitos sistemas deixam a cargo do programador especificar as superposições possíveis. Uma vez especificados os diversos conjuntos (disjuntos) de módulos, tais que todos os módulos de cada conjunto podem ser superpostos o processo de link-edição em si muda muito pouco em relação ao algoritmo visto; é necessário apenas que todos os módulos que compartilham uma área da memória sejam relocados nesta área. Durante a execução antes de um símbolo externo ser referenciado, o programa deve chamar uma subrotina que verifica se o símbolo em questão está presente na memória, carregando o módulo correspondente se não estiver, e em seguida retornar ao módulo que o chamou. As instruções de ligação com esta subrotina devem ser portanto inseridas pelos compiladores nos pontos apropriados.

Existem certas situações no entanto em que é possível construir um algoritmo sistemático, que determina quais os conjuntos de módulos que podem ser superpostos. Examinaremos agora um destes casos. O algoritmo não determina a maneira mais eficiente de superposição; determina apenas uma divisão de módulos superponíveis que funciona.

Como vimos em FORTRAN os símbolos externos são nomes de subrotinas. No caso de um programa em que os módulos são subrotinas é possível determinar uma partição dos módulos do programa em conjuntos de módulos superponíveis, analisando apenas os vetores de transferência dos módulos do programa.

Isto é possível porque existe entre as subrotinas de um programa uma hierarquia natural que pode ser aproveitada para construir a partição mencionada. A figura a seguir representa a árvore de subrotinas do módulo principal (A) de um programa.

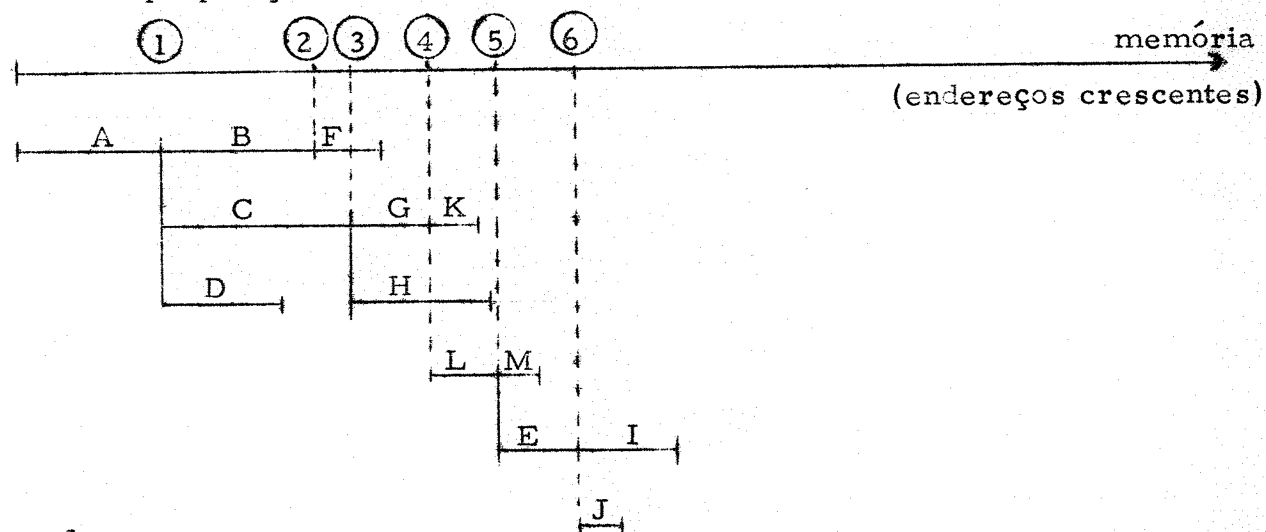


A árvore acima pode ser construída a partir dos vetores de transferência de um programa FORTRAN. A regra para a construção da árvore é simplesmente cada nó, (que é o nome de um dos módulos do programa), ter por filhos todos os seus símbolos externos. Como cada símbolo externo é o nome de uma subrotina, isto significa que cada nó tem por filhos as subrotinas que ele chama. Durante a execução, em cada instante, todas as subrotinas encadeadas precisam ficar simultaneamente na memória. Esta é a única restrição a ser obedecida pela partição de módulos. Quais as subrotinas que serão efetivamente encadeadas não é possível determinar apenas das informações disponíveis nesta altura, pois pode depender de condições que ocorrerem durante a execução. Potencialmente porém as subrotinas que estão ao longo de um caminho que vai da raiz até uma folha da árvore são as que eventualmente podem ser encadeadas. Assim, por exemplo, na árvore acima A, B, E, e J são módulos que não podem ser sobrepostos. Com esta estratégia a mínima memória necessária ao programa é a que se obtém quando cada módulo é carregado no endereço de menor valor maior que o endereço da última palavra de todos os módulos que podem chamá-lo. Assim no exemplo da árvore dada o endereço em que o módulo E pode ser carregado deve ser maior que o endereço da última palavra de B, L e D. A relação definida acima estabelece uma relação de ordem parcial sobre o conjunto dos módulos do programa. Alocar os módulos do programa de modo a requerer a menor memória possível significa ordenar os módulos segundo esta relação de ordem parcial. Um algoritmo de ordenar uma lista em ordem parcial pode ser encontrado em [10]. Indicando por $\angle$ a relação $S_1 \angle S_2$ (S_1 precede S_2) se e só se S_1 chama S_2, temos as seguintes relações de precedência no nosso exemplo:

$A \angle B$, $A \angle C$, $A \angle D$, $B \angle E$, $B \angle F$, $C \angle G$, $C \angle H$, $D \angle E$, $E \angle I$, $E \angle J$, $G \angle K$, $G \angle L$, $L \angle E$, e $L \angle M$

(Cada relação de precedência do tipo $S_1 \angle S_2$ significa que o endereço da primeira palavra de S_2 tem de ser maior que o endereço da última palavra de

S_1.) Representando cada módulo por um segmento proporcional à memória necessária ao módulo, e representando a memória por um eixo de abscissas as relações acima ordenadas pelo algoritmo mencionado dão o seguinte esquema de superposição.



Damos abaixo uma descrição resumida do algoritmo que mencionamos acima. A partir das relações de precedência que podem ser facilmente obtidas da árvore de subrotinas constrói-se uma tabela sequencial, de acesso direto com um nó X[K] para cada módulo. Cada nó X[K] desta tabela é constituído por 5 campos:

X[K]

NOME [K]	M-REQ[K]	ORIGEM [K]	CONT [K]	TOPO [K]

onde NOME[K] - é o nome (símbolo público) do módulo k ;

M-REQ[K] - é a memória requerida pelo módulo k ;

ORIGEM[K] - é o endereço da primeira palavra do módulo (inicialmente O; o objetivo do algoritmo é determinar ORIGEM para todos os módulos)

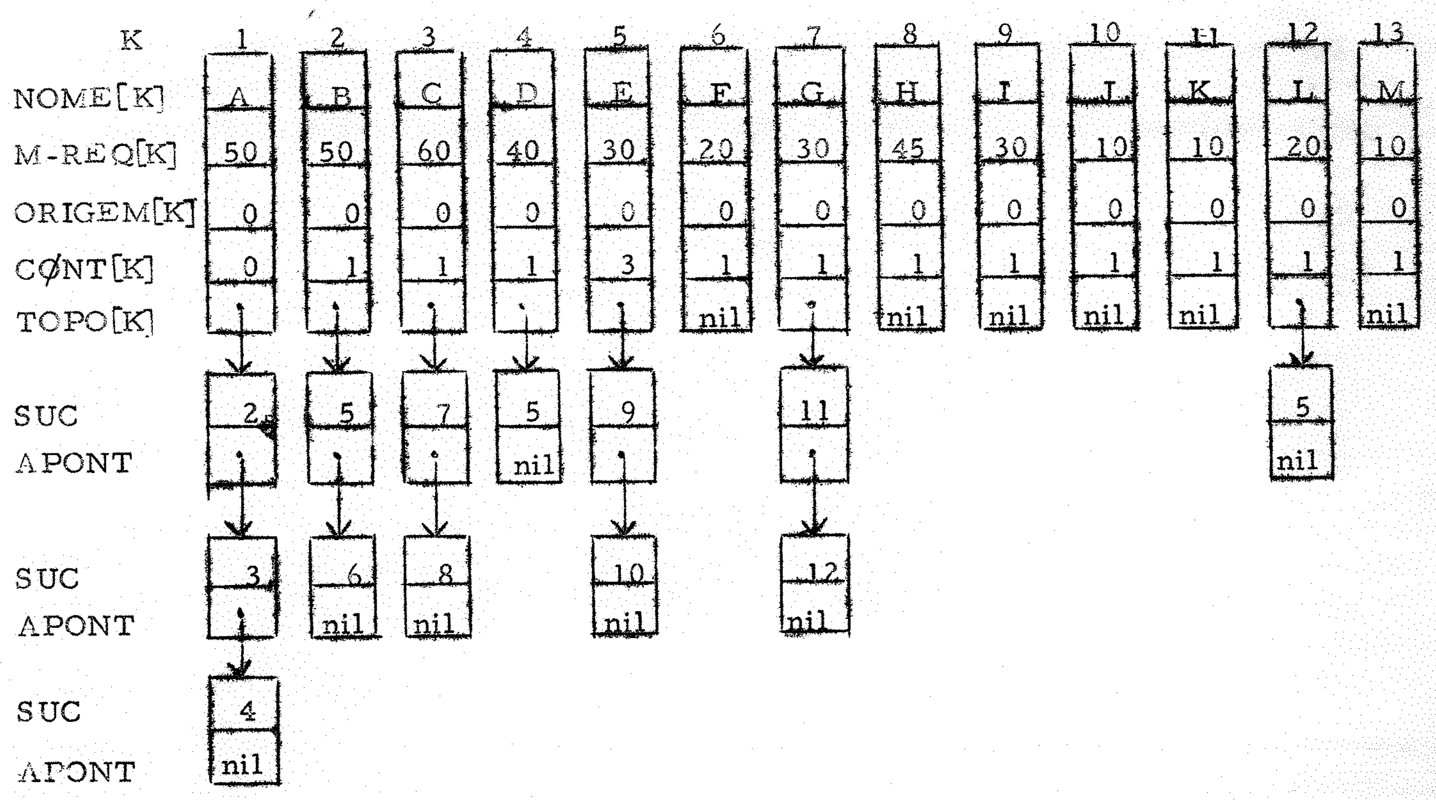
CONT[K] - é um contador que contém o número de predecessores diretos do módulo k ;

TOPO[K] - é um apontador para uma lista linear ligada ("linked") que contém os sucessores diretos do módulo k ;

Esta lista é composta por elementos com os campos:

SUC	APONT

onde SUC é o índice k de um sucessor direto do módulo e APONT é um apontador para o próximo elemento da lista. O último elemento da lista de sucessores diretos tem no campo APONT um apontador especial (pode ser 0000 por exemplo) que denotaremos por <u>nil</u>, e que indica que este é o último elemento da lista. A figura a seguir indica o conteúdo inicial da estrutura descrita para o nosso exemplo:



Uma vez construída esta estrutura,(o que é fácil a partir da árvore dada) o algoritmo consiste em:

1. sair com o módulo com campo CONT = 0;

2. obter o endereço da primeira palavra seguinte a este módulo somando ORIGEM deste módulo com M-REQ ;

3. Decrementar o CONT de cada sucessor dele; se o campo CONT ficar nulo para um determinado sucessor k, colocar em ORIGEM(k) o endereço obtido em 2.; e encadear este sucessor numa lista de elementos com o campo CONT = 0 (o próprio campo CONT, que não vai mais ser usado pode ser usado para este encadeamento);

4. Após fazer a operação 3. para todos os sucessores do módulo em questão pegar o próximo nó da lista de elementos com o campo CONT = 0 repetindo as operações a partir de 1..

Para maiores detalhes sobre o algoritmo esboçado acima e informações sobre listas ligadas ver [10] .

3.7. Observações:

1. As técnicas de intercomunicação vistas neste capítulo foram inicialmente desenvolvidas para a comunicação entre subrotinas FORTRAN ou de tipo FORTRAN escritas em outras linguagens. Uma discussão de como estas técnicas poderiam ser extendidas ao ALGOL pode ser encontrada em [3] .

2. Os métodos de ligação vistos pressupõem que cada módulo, ao tempo de execução ocupará um conjunto de células contíguas, que inclui as

eventuais instruções e áreas de dados do módulo. A memória requerida pelo módulo é determinada ao tempo de tradução do mesmo. Estes métodos portanto tal qual foram expostos não são apropriados para módulos cuja memória requerida não é conhecida ao tempo da carga, ié para estruturas de informação que "crescem ou decrescem" dinamicamente durante a execução do programa.

3. A estrutura da tabela de uso descrita neste capítulo permite que se usem símbolos externos num programa. Entretanto a referência a símbolos externos no programa tem de ser, com esta estrutura, tal que cada referência corresponda à substituição do endereço do símbolo externo em uma das palavras do módulo. Em HAL por exemplo esta restrição significa que expressões envolvendo símbolos externos não podem ser usadas. Assim o operando: TABELA seria admissível, mas TABELA + 5 seria um operando inválido, supondo que TABELA seja um símbolo externo. Evidentemente adaptações menores da tabela de uso permitiriam o uso de expressões envolvendo símbolos externos.

4. Programas FORTRAN e subrotinas FORTRAN podem ter um outro tipo de acesso à áreas de outros módulos, através de blocos de símbolos declarados "comuns" por um comando do tipo "COMMON". O uso de um símbolo declarado num comando COMMON é uma referência externa de tipo diferente do descrito, pois o valor simbólico neste caso é irrelevante. Neste caso o endereço relativo de um item no bloco "COMMON" é determinado pela posição do respectivo símbolo na lista de declarações no comando "COMMON" e portanto conhecido ao tempo da compilação. A origem do bloco de "COMMON" é no entanto desconhecida ao tempo de compilação. O problema portanto se resume em relacionar os lugares do módulo em que se refere a um símbolo de um bloco de COMMON na tabela de relocação, sendo a constante apropriada de relocação conhecida ao tempo de carga.

Capítulo 4

Entrada/Saída e seu contrôle

4.1. Introdução:

Entrada e saída é necessária num computador porque a sua memória central é bastante restrita sendo assim imperativo armazenar pelo menos parte das informações utilizadas na computação em outros meios de armazenamento de informação, e também para o sistema poder se comunicar com o usuário, programador ou operador de maneira adequada.

As instruções de entrada/saída do HIPO não são representativas do processo de entrada/saída nos computadores reais modernos de alta velocidade por várias razões. Num computador real é necessária uma flexibilidade maior do que a permitida pelo HIPO, tanto em termos da variedade de equipamentos de entrada/saída que são ligados ao sistema, (por exemplo leitora de cartões; perfuradora de cartões; unidades de fita magnética;tambor magnético; disco magnético, removível ou não; fita de papel, máquina de escrever; tubo de raios catódicos, ("display"); traçador de gráficos; impressora em linha; etc.), como no grau de contrôle sobre estes equipamentos.

Neste capítulo examinaremos como a entrada/saída é organizada no "hardware" e como é organizado o contrôle da entrada e saída no sistema operacional, para facilitar as operações de entrada/saída para o programador, manter o sistema a salvo de erros do programador referente à entrada/saída, e aproveitar as potencialidades do "hardware" tornando o processo possivelmente mais eficiente do que se estas funções fossem deixadas sob responsabilidade do programador.

4.2. Processamento e entrada/saída simultânea

Equipamentos de entrada/saída (E/S) geralmente envolvem algum movimento mecânico para completar uma operação de leitura (transmissão de informação do meio de E/S para a memória) ou de escrita (transmissão de informação da memória para o meio de E/S). Principalmente por esta razão as velocidades envolvidas nas operações de entrada/saída são bem menores do que as velocidades de execução de instruções que não são de E/S. Além disso dentro da ampla variedade de equipamentos de entrada/saída mencionados na secção anterior as velocidade de transmissão variam enormemente desde alguns caracteres por segundo para uma máquina de escrever até dezenas ou mesmo centenas de milhares de caracteres por segundo para uma unidades rápida como discos e tambores magnéticos. Estes fatos motivaram a introdução de recursos no "hardware" que permitissem que várias operações de entrada/saída se processassem paralelamente a instruções do programa executadas pelo processador central.

Na maioria dos computadores E/S paralela a processamento de instruções do programa pelo processador central é conseguida através da in-

trodução de processadores especiais denominados de canais. Outras denominações comuns do mesmo dispositivo são processadores de entrada/saída controladores de entrada/saída, etc.. Embora a organização dos processadores de entrada/saída e sua interligação com as demais componentes do sistema, como unidades de E/S, memória, processador central, etc., possam variar de sistema a sistema, um grande número de sistemas tem essencialmente a mesma organização que veremos em seguida.

No esquema que veremos as operações de entrada/saída são iniciadas pelo processador central mas são executadas pelo canal que está ligado com a unidade de E/S envolvida, e serve de intermediário entre a unidade e a memória durante a transmissão de informação entre os dois. Uma vez iniciada uma operação de E/S num canal pelo processador central este continua a execução do programa independentemente do canal que paralelamente executa a operação de E/S. Uma operação de E/S pode assim se processar simultaneamente com uma instrução do processador central e outras operações de entrada/saída iniciadas previamente em outros canais de entrada/saída. O canal é ligado a várias unidades de E/S do mesmo tipo. Na organização que estamos descrevendo num canal apenas uma unidade, de cada vez, poderá estar envolvida numa operação de E/S. A figura 4-1 ilustra os conceitos vistos.

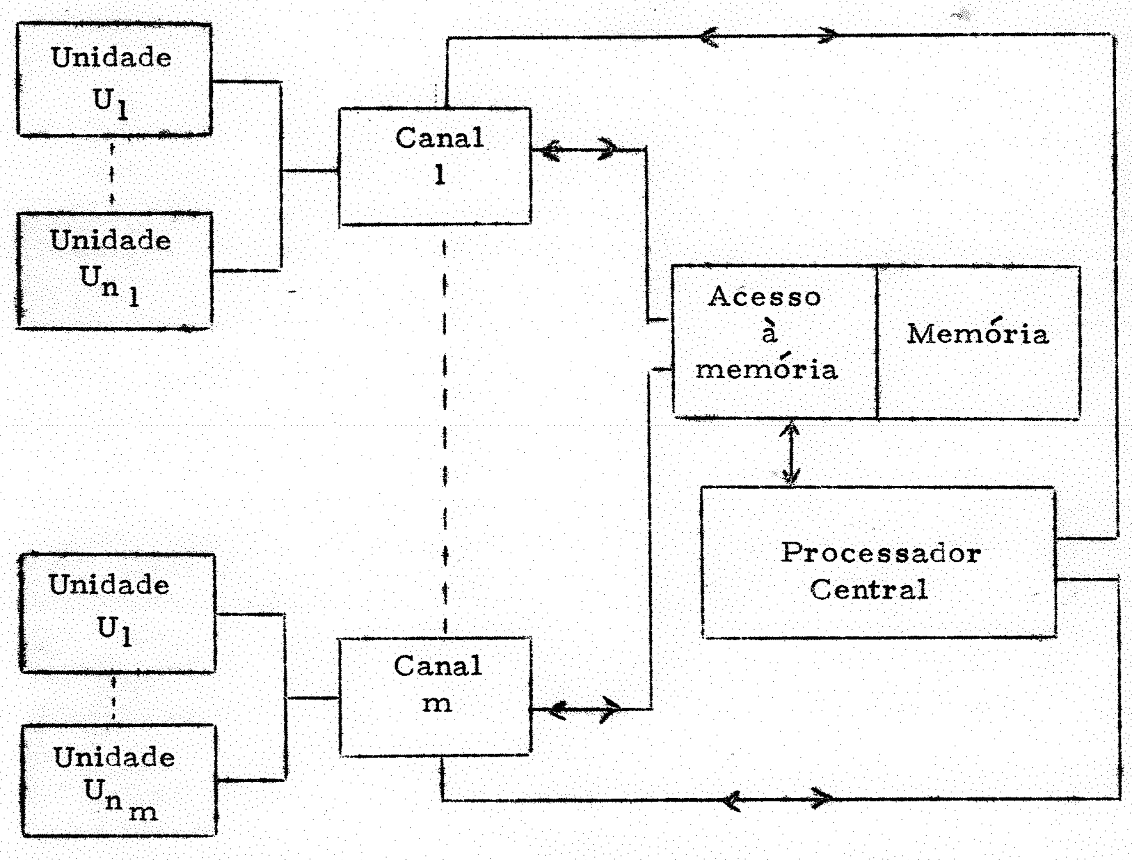


Fig. 4-1.

Nesta estrutura o processador central para iniciar uma operação de E/S transmite ao canal especificado pela instrução uma descrição da operação que deve ser executada pelo canal contendo as informações:

a) código da operação a ser executada (ex. para um canal que

está ligado a unidades de fita magnética: leia da fita magnética para frente, leia para trás, reenrole, grave na fita, pule um registro para frente, pule um registro para trás, etc.; ex. para um canal que está ligado a unidades de leitura de cartões: leia um cartão binariamente, leia um cartão com códigos EBCDIC, teste o estado da unidade, etc.;);

b) A unidade que deve fazer a operação especificada;

c) Se a operação envolver transmissão de informação de ou para a memória, o endereço da primeira palavra envolvida na operação; e

d) o número de palavras da memória envolvidas.

É comum que durante a tramissão de dados se processe também automaticamente a conversão de códigos. O canal interpreta estas informações recebidas, inicia a operação adequada na unidade especificada, requisita sempre que necessário durante a operação acesso a uma palavra da memória, transmitindo seu conteúdo no momento oportuno para a unidade numa operação de escrita, ou recebendo da unidade no momento oportuno a informação e transmitindo-o à palavra da memória em questão, numa operação de leitura, verifica se a operação se desenrola normalmente e informa ao fim da operação o processador central das eventuais condições de exceção ocorridas durante a operação, (tais como: unidade não disponível, perda de informação durante a transmissão, erro de paridade, começo de fita ou fim de fita detetado, etc.), através de uma interrupção. (v. secção 4.3.)

A execução de uma operação de E/S num canal como já dissemos se processa independentemente dos demais processadores (processador central e outros canais) exceto quanto aos acessos à memória. Tanto os canais como o processador central fazem acessos à memória. O processador central entretanto faz mais acessos à memória durante a execução de suas instruções do que o canal devido à maior lentidão do último. Para evitar conflitos o mecanismo de acesso possui a lógica necessária para atender um acesso por vez numa escala de prioridades. Um canal possui prioridade sobre o processador central. Assim quando o canal precisa de um acesso à memória o processador central é bloqueado por um ciclo de memória enquanto o acesso é atendido, recomeçando em seguida normalmente o processamento. Esta técnica denomina-se "roubo de um ciclo" pelo canal ("cycle stealing"), e evita que dois processadores tenham acesso simultaneamente à memória.

O canal possui prioridade sobre o processador central quanto ao atendimento de um acesso à memória pois suspendendo a operação do processador central apenas se está atrasando o processamento durante o atendimento ao canal, mas se o acesso ao canal não fôr atendido dentro de um certo tempo, haverá perda de informação. Por exemplo na leitura de uma unidade de fita magnética de velocidade de transmissão 100 kc (100 000 caracteres por segundo) quando um acesso é requisitado pelo canal o pedido deve ser atendido em menos de 10μ s (tempo de leitura de um novo caráter), para que o primeiro caráter da palavra a ser transmitida à memória não se perca. O sistema deve ser projetado de tal maneira que com operação normal nunca ocorra perda de informação, mas se por alguma razão excepcional houver perda de informação o canal informe o processador central desta ocorrência.

4.3. Interrupções

Uma interrupção é um sinal para o processador central, gerado internamente ou por algum outro processador. Se o processador central recebe um sinal de interrupção, após terminar a instrução que estava executando quando o sinal foi recebido, desvia para uma posição, fixa para cada tipo de interrupção, salvando automaticamente o conteúdo de certos registradores como o IAR. A posição fixa em geral contém um desvio para uma rotina de tratamento da interrupção. Após a execução desta rotina, se outras interrupções não ocorrerem, o processador central reinicia o processamento do ponto em que este foi interrompido.

Muitos sistemas tem associado às diversas interrupções possíveis um sistema de prioridades mediante o qual uma rotina de tratamento de interrupções de um certo nível só é interrompido por alguma interrupção de prioridade mais alta. Se a prioridade de interrupção é mais baixa ou igual, a interrupção só transfere controle para a respectiva rotina de tratamento após a rotina de tratamento da interrupção de mais prioridade terminar a sua execução ou "desligar" sua prioridade.

O mecanismo de interrupção é um meio de intercomunicação entre processadores e pode ser aproveitado para controlar o tratamento de situações excepcionais. A interrupção pode ser gerada por algum canal informando o processador que uma operação de E/S foi completada, transmitindo um bloco de dados ou terminando uma operação como de posicionamento de um mecanismo de braço de uma unidade de disco, ou término de uma operação de pulo de página numa impressora etc.. Neste caso a interrupção é característica do canal e identifica a unidade que causou a interrupção, e eventuais condições de exceção que ocorreram durante a execução da instrução de E/S pelo canal. Outros tipos de interrupção são gerados no próprio processador central e podem ser aproveitados para controlar no sistema operacional situações de exceção. Exemplos típicos são: interrupção do relógio interno, que ocorre depois de decorrido um certo tempo especificado num registrador associado ao relógio interno, ou se o relógio atinge valor nulo e que pode ser usado para fins de contabilidade do tempo de uso do sistema e outras funções; interrupções do programa do usuário, gerados por uma ocorrência de sobrecarga ("overflow") numa operação aritmética; tentativa de execução de uma operação inválida; tentativa de acesso a um endereço inválido; etc..

4.4. O sistema de controle de E/S (SCES)

Nas secções anteriores foi vista uma organização da entrada/saída no "hardware". A programação da E/S de um programa é complexa porque o número de situações possíveis de ocorrer, e para as quais o programa deve ter previsão é muito grande. Além disso, para que se possa aproveitar a capacidade de processamento, paralelo à atividade de E/S do "hardware", deve-se utilizar de mecanismos mais ou menos complexos, assegurando o máximo de paralelismo e ao mesmo tempo cuidando por exemplo de que um certo dado não seja utilizado pelo programa antes de ser produzido por um canal.

Por outro lado, como vimos, cada operação de entrada/saída é

precisamente especificada no "hardware" através da designação do canal e unidade envolvida, da operação particular que se quer executar na unidade e as palavras da memória envolvidas. Seria evidentemente de todo conveniente que o programa pudesse ter, não obstante, suficiente flexibilidade para poder ser executado colocando por exemplo uma determinada fita numa das execuções em uma unidade e numa outra execução numa unidade diferente; ou que o programa não tenha de ser refeito ou recompilado se quisermos executá-lo numa instalação que tenha um número diferente de canais da instalação usualmente utilizada. Por todas estas razões é conveniente o sistema operacional controlar toda a E/S do usuário, facilitando enormemente a programação, e utilizando o "harware" com eficiência. Uma outra razão ainda para a inclusão no sistema operacional deste controle, é impedir que um usuário possa interferir indevidamente num processo de um outro usuário, lendo por exemplo inadvertidamente os cartões de controle deste último como dados de seu programa, etc.. Por este motivo em geral a instrução de iniciar a operação num canal é privilegiada, no sentido de que somente poder ser executada num estado do processador, que é usado sob controle do sistema operacional e é reservado somente para este. Assim uma instrução de E/S quando o sistema está sob controle do sistema operacional só pode ser executado por ele, sendo vedado ao usuário. (A tentativa de execução de uma instrução privilegiada pelo programa do usuário gera uma interrupção).

Os objetivos principais do sistema de controle de E/S são portanto conseguir uma certa independência das operações de E/S dos dispositivos de E/S ao nível do programa fonte; aproveitar a capacidade de paralelismo do "hardware"; e proteger as informações de um usuário de eventual uso indevido por outro.

4.5. Arquivos

As informações de entrada de um programa do ponto de vista deste são obtidas de uma entidade lógica (isto é. eventualmente diferente de uma unidade física) chamada de arquivo. Analogamente as informações de saída são transmitidas a um arquivo. Um arquivo é um conjunto finito ordenado de registros lógicos, associado a um símbolo - o nome do arquivo. Cada registro lógico se compõe de certas informações. A uma parte das informações do registro lógico se dá o nome de campo. A organização do registro lógico é responsabilidade do programador: para o sistema operacional a unidade de informação num arquivo será o registro lógico. Em geral os registros lógicos de um arquivo contém informações correlatas. Por exemplo: um arquivo X típico que contém o cadastramento de certas informações sobre os n alunos de uma universidade, se compõe de n registros lógicos; cada registro lógico se compõe das informações sobre um dos alunos da universidade com 10 campos com as informações respectivamente: nome do aluno, número do aluno, disciplinas cursadas com as respectivas notas, disciplinas em que está matriculado, idade do aluno, sexo, data de nascimento, local de nascimento, filiação e endereço. As informações de um arquivo são armazenadas como uma cadeia de "bits", caracteres, ou palavras em um meio físico apropriado (fita magnética, cartão, disco magnético, etc.). Como cada tipo de unidade tem características próprias de operação, em cada operação de entrada/saída física é transmitido um conjunto de "bits", caracteres, ou palavras do

do arquivo denominado de registro físico. As limitações e restrições do registro físico dependem do tipo de unidade em geral e não serão discutidas aqui. Diremos, apenas, que em geral é conveniente (e possível) fazer com que um registro físico de um arquivo se componha de um ou mais registros lógicos. O número de registros lógicos por registro físico chama-se fator de bloqueamento.

Uma outra característica importante de um arquivo é a ordem em que os seus registros lógicos serão processados pelo programa. Se o programa faz acesso aos registros lógicos na mesma ordem em que comparecem no arquivo isto é, se o programa fizer acesso ao registro lógico número n após ter acesso aos registros lógicos 1, 2, ..., n-1, dizemos que o acesso ao arquivo é sequencial; se o faz em alguma outra ordem dizemos que o acesso é aleatório. Esta distinção é importante por duas razões:

1) Certos meios físicos só podem ser usados eficientemente se o acesso aos registros físicos do arquivo fôr sequencial. (Ex. uma leitora de cartões só pode ler o n-ésimo cartão depois de ler os n-1 anteriores;numa fita magnética só é possível ter acesso ao n-ésimo registro físico depois de a fita ser enrolada até o ponto em que começa este registro; etc.). Isto quer dizer que acesso aleatório ou é inteiramente impossível em arquivos armazenados nestes meios físicos (leitora de cartões, por exemplo), ou então torna-se muito ineficiente devido ao tempo necessário para localizar um determinado registro físico, (fita magnética). Em contraste, outros meios, como disco magnético, tambor magnético, etc. são adequados para processar um arquivo tanto sequencialmente como aleatoriamente, pois o tempo necessário para localizar um registro nestes dispositivos é virtualmente independente da posição relativa do registro físico dentro do arquivo.

2) Arquivos que são processados sequencialmente tem a importante característica de que em qualquer instante do processamento é fácil o sistema operacional prever qual será o próximo registro físico a ser processado.

4.6. "Buffers"

Nas secções anteriores foi visto que para o funcionamento mais eficiente do sistema de computação foram incluídos recursos no "hardware" (como canal, interrupção, etc.) que permitem que dados sejam transmitidos de ou para a memória, para ou de um dispositivo de E/S, simultaneamente com a execução de instruções do programa pelo processador central. O programa porém, que o processador central executa, corresponde a um algoritmo, e como foi visto os passos de um algoritmo são pela sua própria definição executados de maneira seqüencial. Desta maneira quando um certo dado é lido pelo algoritmo é preciso assegurar que este dado não seja utilizado pelo programa antes que a unidade de entrada produza o referido dado na memória. Se a instrução seguinte à instrução de leitura só fosse executada depois de terminar a transmissão de dados esta sincronização seria automática; como porém, conforme vimos tal não acontece, é preciso tomar cuidados especiais para que a capacidade de processamento paralelo com atividade de E/S seja adequadamente aproveitada. A técnica usual para assegurar isto é o emprego de "buffers" múltiplos que examinaremos nesta secção. (O caso de uma opera-

ção de escrita pelo programa é inteiramente análogo ao discutido acima: ié quando um certo dado é escrito pelo algoritmo é preciso que a área correspondente da memória não seja destruida pelo programa antes que a unidade de saída tenha recebido o referido dado. No parágrafo abaixo discutiremos sempre o caso de entrada, mas o caso de saída é todas as vêzes inteiramente análogo).

Um "buffer" é uma área da memória em que armazenamos um registro físico, para ser processado subsequentemente pelo programa no caso de leitura, ou para ser transmitido em seguida a uma unidade de E/S no caso de uma operação de saída. Quando um novo registro físico é necessário ao programa, a unidade de E/S é como vimos, relativamente lenta em relação ao processador central, e assim antes que o programa possa utilizar o respectivo "buffer" terá de esperar que a unidade encha o "buffer" considerado; logo em seguida porém o processador central terá de processar os registros lógicos deste registro físico e eventualmente o tempo necessário para processá-los é maior do que o tempo necessário à unidade para encher um "buffer" com um registro físico. Assim é natural tentar aproveitar este tempo de processamento para a unidade encher um outro "buffer" com o registro físico seguinte. Para que o sistema operacional, que como já dissemos controlará todo este processo, possa antecipar qual será o próximo registro físico é evidente que o arquivo correspondente deve ser de acesso seqüencial, (ou o programa deve fornecer esta informação de alguma maneira, se isto fôr possível, no caso de arquivos de acesso aleatório). Esta é a idéia básica da técnica de "buffers" múltiplos.

Vamos considerar primeiro o caso de 2 "buffers", representados esquematicamente na figura 4.2..

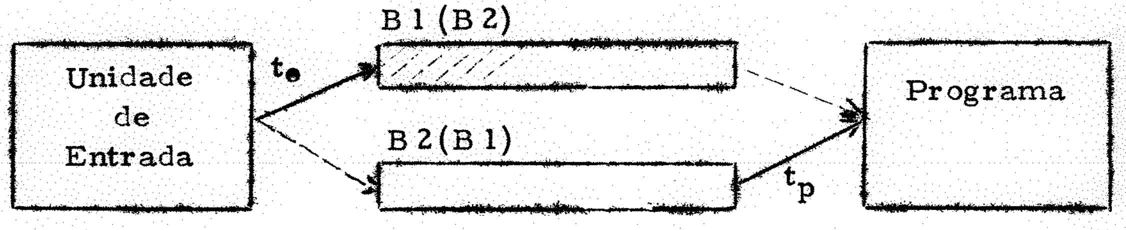


Fig. 4.2.

Inicialmente temos os dois "buffers" vazios. Na primeira oportunidade o sistema de controle de E/S enche o "buffer" B 1. Após B 1 ficar cheio, as identidades de B1 e B2 são trocadas (ação indicada na figura entre parentesis); o programa começa o processamento de B2 (o buffer que acabou de ser enchido pela unidade), ao mesmo tempo em que a unidade de entrada, através do canal respectivo, inicia a leitura do "buffer" B1 de novo registro físico; enquanto o programa processa B2 a unidade enche B1; consideremos inicialmente o caso mais favorável em que o tempo necessário para a unidade-canal encher B1, t_e, é igual ao tempo de processamento de B2 pelo programa, t_p; neste caso B1 fica cheio ao mesmo tempo em que B2 acaba de ser processado, e assim, as identidades de B1 e B2 podem ser novamente troca-

das recomeçando-se o ciclo, o programa iniciando o processamento de B2 enquanto B1 começa de novo a ser enchido.

Se $t_p \geq t_e$ o programa não terá de esperar nenhuma vez (exceto talvez pelo primeiro registro físico), pela unidade de entrada, sempre tendo à disposição um "buffer" para processar. Neste caso tudo se passa como se as operações de entrada da unidade consideradas fossem instantâneas graças à técnica de usar 2 "buffers" e a capacidade do "hardware" de processar E/S e programa simultâneamente. (Se $t_p > t_e$ a unidade de entrada é que vai ter de esperar para o processador desocupar um buffer). Se $t_p < t_e$ o processador terá de esperar $t_e - t_p$ para cada registro físico. É fácil ver que no caso de 2 "buffers", e admitindo que a unidade e canal estejam disponíveis em todos os instantes em que o sistema quiser iniciar nova leitura, leitura e processamento de novos registros começam sempre no mesmo instante, independentemente se t_p é maior, igual ou menor que t_e, pois o início de ambas as atividades depende da conclusão da outra. (Na realidade a leitura do registro n + 1 começará ligeiramente antes do processamento do registro n, pois ambas as atividades são iniciadas pelo processador central). O mesmo porém não acontece se utilizarmos mais de 2 "buffers".

Empregando 2 "buffers" o tempo de espera do programa pela leitura do registro n + 1 é, conforme se depreende da discussão acima, $\max \{0, t_e(n+1) - t_p(n)\}$, onde indicamos $t_e(n+1)$ e $t_p(n)$ para realçar que se trata do tempo de enchimento do buffer para o registro físico n+1 e do tempo de processamento do registro físico n. O uso alternado de "buffers" pode ser facilmente extendido para um número arbitrário de k "buffers", que são usados rotativamente tanto pela unidades de E/S como pelo processador central. A análise neste caso porém fica consideravelmente mais complexa devido ao fato de que, como notamos acima, com mais de dois "buffers" os instantes de início de atividade no canal-unidade e no processador para processar um novo "buffer" podem agora ser distintos. É fácil ver porém que se $t_p(n)$ e $t_e(n)$ forem constantes com n, ié se o processamento de um "buffer" levar sempre o mesmo tempo, bem como o seu enchimento então não há nenhuma vantagem no emprego de mais de dois buffers, (a não ser, talvez, durante um certo tempo transitório no início do processo). De fato sendo $t_p = t_p(n)$ e $t_e = t_e(n)$ constantes, se $t_p \geq t_e$ o processador não espera nada pela entrada, mesmo com dois "buffers" e portanto neste caso nada há a ganhar com o emprego de mais de 2 "buffers". Se $t_p < t_e$, se o canal-unidade teve tempo no início do processo para encher vários "buffers", durante algum tempo o processador não espera nada pela entrada, "consumindo" o que a unidade produziu anteriormente; porém como $t_p < t_e$, cada "buffer" é processado antes que um novo "buffer" possa ser enchido e portanto após alguns buffers processados o processador não terá mais nenhum "buffer" para processar e terá de esperar o canal-unidade encher um novo "buffer"; após este momento canal e processador iniciarão sempre no mesmo instante o processamento de um novo "buffer" e como $t_p < t_e$ o processador terminará sua tarefa antes do canal-unidade e assim terá de esperar $t_e - t_p$ para recomeçar o seu trabalho, exatamente como no caso de 2 "buffers". Assim o emprego de mais de 2 "buffers" só se justifica quando t_p e/ou t_e variam consideravelmente e assim o canal pode aproveitar quando $t_p > t_e$ para se adiantar ao processador e assim quando $t_p < t_e$ o processador tem vários "buffers" a processar antes de ter de esperar pela entrada. Um exemplo simples de um caso destes é um

programa que lê n cartões, e depois os processa, aí lê de novo mais n cartões e os processa, etc.. Neste caso durante a leitura dos n cartões $t_p(1)$, $t_p(2)$, ..., $t_p(n-1)$, são muito pequenos mas $t_p(n)$ é possivelmente maior do que o tempo de enchimento de n "buffers".

4.7. Ciclo de entrada e ciclo de saída de um "buffer"

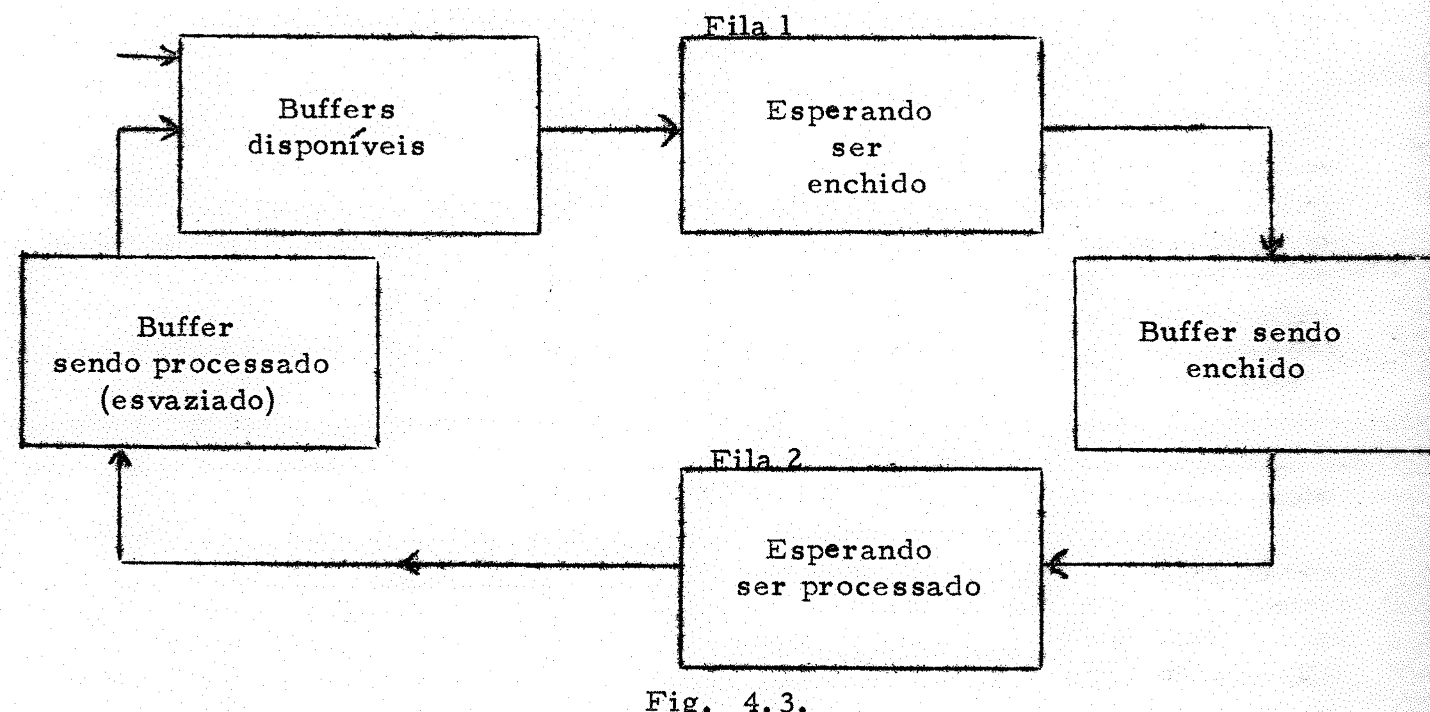


Fig. 4.3.

Consideremos o caso de k "buffers" usados rotativamente. A figura 4.3. representa os diversos estados em que o "buffer" fica durante um ciclo de leitura. Inicialmente temos os k "buffers" vazios disponíveis. Estes "buffers" entram numa fila dos "buffers" esperando pela unidade para serem enchidos. Quando o canal e a unidade puderem atender o pedido de leitura o primeiro "buffer" da fila é retirado desta e começa a receber o registro físico da unidade de entrada; ao terminar com sucesso a transmissão de dados, este "buffer" está agora cheio e entra numa fila dos "buffers" esperando para serem processados; quando o processador necessitar de novo "buffer" o primeiro da fila esperando por processamento é retirado desta e se inicia o seu processamento; quando o processador terminar o processamento o buffer volta a ser disponível e recomeça o mesmo ciclo. A fig. 4.4. mostra o ciclo de saída de um "buffer", que é inteiramente análogo ao ciclo de entrada.

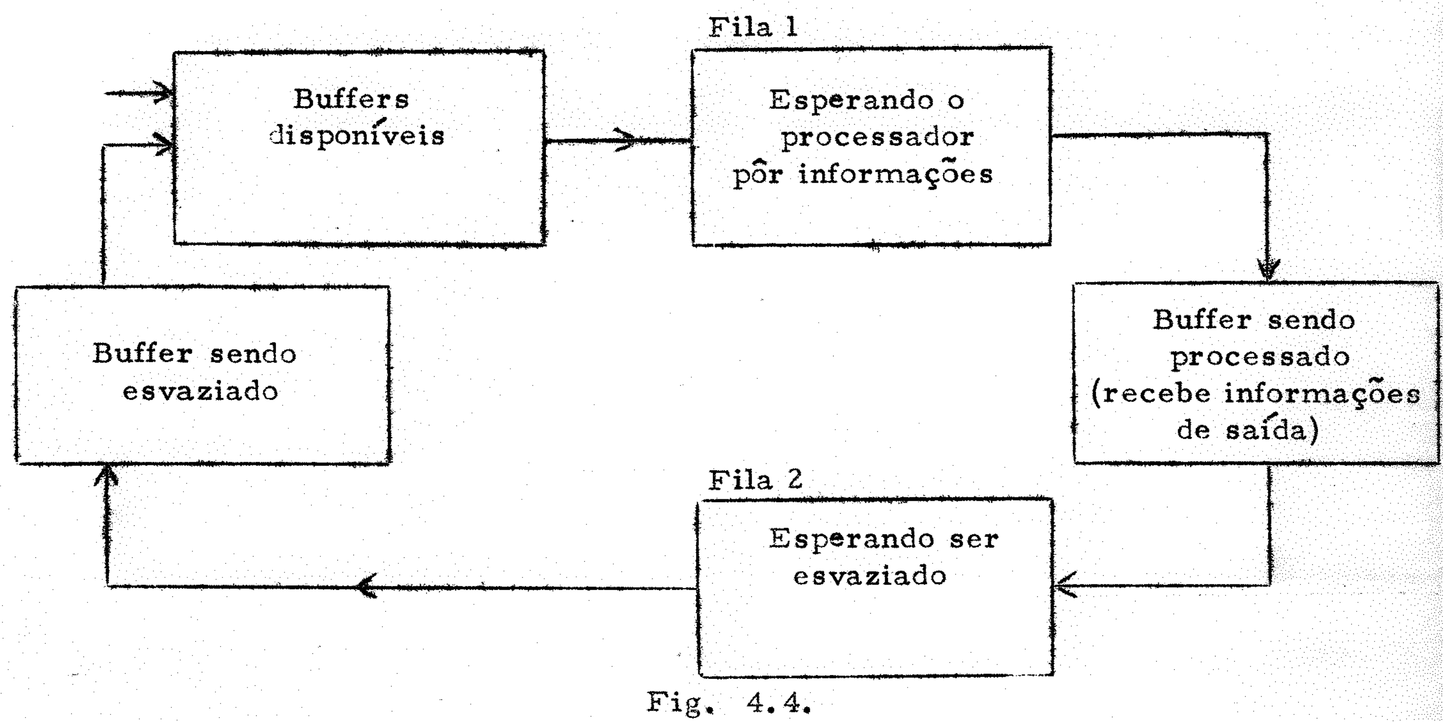


Fig. 4.4.

4.8. As tabelas de atividade e sua manipulação

Nas secções anteriores esboçamos as funções principais do sistema de controle da E/S. Para desempenhar estas funções o sistema de controle da entrada/saída dispõe de uma série de subrotinas cada uma com tarefas específicas, e manipula uma série de tabelas em que as informações necessárias ao controle dos diversos arquivos, "buffers", filas, canais e unidades são armazenadas e no momento apropriado atualizadas. Nesta secção discutiremos estas tabelas e suas interrelações. O acesso aos arquivos é nesta secção suposto sequencial.

Como já foi dito anteriormente o programa do usuário ao solicitar uma "leitura" de um novo registro lógico refere-se apenas a um determinado arquivo. O sistema em resposta a esta solicitação deve devolver ao programa do usuário o registro lógico em questão. Supondo que o registro físico que contém o referido registro lógico já tenha sido transmitido a um "buffer", duas maneiras são comuns para o sistema dar acesso ao programa do usuário a este registro lógico: uma é fornecer ao programa do usuário o endereço do registro lógico em um indexador; o outro é transmitir o registro lógico todo a uma área de trabalho especificada no programa. Evidentemente em algum instante anterior o registro físico correspondente foi fisicamente lido da unidade que contém o arquivo. Como vimos, para que o sistema possa ler um registro físico precisa especificar precisamente o canal e a unidade envolvida, o endereço da área memória ("buffer") envolvida e o número de palavras a transmitir. Como nada disso é especificado na solicitação do usuário o sistema precisa manter uma tabela que descreve o arquivo. Esta tabela é o FAB, (de "File Activity Block"), e contém as seguintes informações:

a) informações sobre a estrutura do arquivo:

a1. identificação do arquivo: nome, número do arquivo, data de validez, etc.;

a2. informações de controle: Tipo de unidade (fita, cartão, etc.)
Entrada ou Saída
Tamanho do registro lógico
Fator de bloco
Aberto ou fechado

b) informações sobre os "buffers" associados:

b1. número de "buffers" alternados e endereço do primeiro "buffer";

b2. endereço do "buffer" em uso pelo programa;

b3. número do registro lógico do "buffer" em uso pelo programa;

b4. endereço do primeiro "buffer" esperando pelo programa;

b5. número de "buffers" esperando pelo programa;

c) informações sobre a unidade associada:

c1. identificação do canal e unidade;

d) histórico;

d1. registro dos erros ocorridos;

Por sua vez cada unidade ligada ao sistema precisa ter uma tabela denominada de UAB (de "Unit Activity Block") que descreve a situação da unidade, e cada canal uma denominada de CAB (de "Channel Activity Block") com a situação do canal:

UAB :

a) identificação da unidade:

a1. tipo de unidade;

a2. número do canal que controla esta unidade; endereço do CAB do canal;

a3. número da unidade;

b) arquivo na unidade:

b1. rótulo (nome) do arquivo;

b2. endereço do FAB que descreve este arquivo;

c) endereço do "buffer" em uso pela unidade;

d) informações sobre a fila de "buffers" esperando pela unidade:

d1. número de "buffers" esperando pela unidade;

d2. endereço do primeiro "buffer" da fila;

e o CAB contém:

a) identificação do canal;

b) endereços dos UAB-s das unidades controladas por este canal:

c) canal ocupado ou livre e
unidade em uso no caso de canal ocupado.

Os CAB-s e UAB-s fazem parte do sistema de controle da E/S. Os "buffers" e FAB-s fazem parte do programa do usuário.

Vejamos agora como se processa a entrada de um arquivo (sequencial) de um certo programa.

Os "buffers" associados ao arquivo e o espaço para cada arquivo são reservados no programa do usuário pelo compilador. A estrutura do arquivo não é importante nesta altura e pode ser declarada quando o programa vai ser carregado, através de cartões de controle apropriados. Cada leitura do arquivo é compilada como uma chamada a uma rotina apropriada do SCES (Sistema de controle de Entrada e Saída), tendo por parâmetro o endereço do FAB que vai descrever o arquivo. Antes de poder fazer a primeira "leitura" do primeiro registro lógico, o programa deve abrir o arquivo através da chamada de outra subrotina cujas funções logo veremos.

Antes de iniciar a execução do programa os arquivos devem ser declarados através de cartões de controle específicos para este fim e que inicializarão parte dos FAB-s respectivos. Estas informações são a identificação do arquivo, o tipo de unidade, tamanho do registro lógico, fator de bloqueamento, e a unidade que se quer associar a este arquivo (opcional). O número de "buffers" alternados e o endereço do primeiro destes "buffers" é preenchido pelo compilador quando reserva espaço ao FAB e aos "buffers".

Vários arquivos podem ser associados à mesma unidade, em tempo de execução porém apenas um pode estar ativo, quando a unidade é do tipo exclusivo para um só arquivo como uma unidade de fita magnética. Qual a unidade associada a um arquivo é na realidade irrelevante até o momento em que o arquivo fica ativo. Por este motivo a designação de unidade é opcional na especificação do arquivo. Para evitar que uma unidade seja usada acidentalmente para a atividade errada (como leitura de um arquivo A em vez de B; ou escrita em uma fita que se tenciona ler, etc.) e designar uma unidade se isto ainda não foi feito é preciso executar uma rotina que abre o arquivo. A rotina de abertura de um arquivo torna-o disponível ao programa que em seguida pode ter acesso aos seus registros lógicos. A rotina de abertura tem por parâmetro o endereço do FAB correspondente e um indicador especificando se o arquivo é de entrada ou saída. (O arquivo pode ser ainda de entrada/saída para unidade de acesso aleatório que não discutiremos aqui).

ROTINA DE ABERTURA

AB 1. Se o arquivo estiver aberto (conforme o FAB) retorna. (Obs.: Alguns SCES consideram isto como um erro, ié uma tentativa de abertura inválida); senão:

AB 2. Se a abertura é de um arquivo de saída vá ao passo AB 6.

AB 3. (arquivo de entrada) Se o arquivo ainda não tem unidade associada vá ao passo AB 5; senão verifique se o rótulo do arquivo presente na unidade coincide com o nome do arquivo em FAB. Se não coincidir dê instruções apropriadas ao operador; senão reserve esta unidade ligando o UAB com o FAB. (Se já estiver reservado é um erro).

AB 4. Chame a rotina de Leitura n vêzes, onde n é o número de "buffers" associados ao arquivo tendo por parâmetro o endereço do FAB, (Esta rotina vai colocar o "buffer" do arquivo na fila esperando pela unidade e se o canal estiver desocupado iniciar a leitura do primeiro registro físico); retorne.

AB 5. (arquivo de entrada ainda não tem unidade associada) Procure todas as unidades do tipo indicado por FAB que contenham um arquivo disponível com o rótulo especificado em FAB; se não achar ou achar mais de uma unidade dê instruções apropriadas ao operador; senão desgine a unidade encontrada a FAB, e reserve a unidade ligando UAB ao FAB, e vá ao passo AB 4.

AB 6. (Arquivo de saída) (O tratamento de um arquivo de saída é inteiramente análogo ao de entrada: Se a unidade já estiver designada verifique se ela está em condições de escrever; dê instruções apropriadas ao operador em caso contrário; reserve a unidade ligando UAB a FAB e se o arquivo fôr rotulado escreva um registro físico padrão com as informações do rótulo, e retorne; se o arquivo não tiver unidade designada procure a primeira unidade do tipo indicado em FAB disponível e em condições de escrever. Se não achar dê instruções apropriadas ao operador; senão reserve esta unidade e escreva o rótulo se o arquivo fôr rotulado e retorne).

Uma vez aberto o arquivo o programa pode ter acesso aos seus registros lógicos. Cada "leitura" de um novo registro lógico é um desvio para a subrotina apropriada do SCES que vai devolver ao programa o registro lógico solicitado. A rotina de leitura de registro lógico verifica se o arquivo está aberto para leitura e indica erro se não estiver; verifica no FAB se há um "buffer" em uso; se houver avança um registro lógico neste "buffer"; se o registro lógico fôr anterior ou o último registro lógico deste "buffer" devolve o registro lógico ao programa e retorna; se o ponteiro avançou para um registro lógico posterior ao último "buffer" então este "buffer" já foi processado pelo programa e é portanto liberado para nova leitura. Assim chama a rotina de leitura do arquivo (que vai colocar este buffer na fila de espera pela unidade; depois de a rotina de leitura devolver o comando, ou no caso de não haver "buffer" em uso no programa verifica se há um "buffer" esperando pelo pro-

grama; se não houver o programa é colocado num estado de espera pois não pode continuar até que a unidade encha novo "buffer"; senão retira o primeiro "buffer" da fila de espera pelo programa, coloca-o como "buffer em uso", inicializa o apontador de registros lógicos para o primeiro registro lógico deste "buffer" e devolve este registro lógico ao programa e retorna.

A rotina de leitura recebe o endereço de um "buffer" do arquivo e o coloca na fila de espera da unidade. Em seguida se o canal (e portanto também a unidade) estiver desocupado, retira um "buffer" da fila de espera da unidade, coloca-o como "buffer" em uso pela unidade, e chama a rotina de ativação do canal tendo por parâmetro a unidade; e retorna após a rotina de ativação devolver o comando. A rotina de ativação marca o canal como ocupado em CAB, coloca a unidade como a unidade usada pelo canal em CAB, monta a partir do UAB (que contém o buffer em uso) e FAB que contém o tamanho do registro físico um comando de leitura, e inicia uma operação de E/S no canal-unidade em questão, retornando a seguir.

Se o canal estiver ocupado durante a execução de rotina de leitura nenhuma operação de E/S é iniciada, e as rotinas vistas acima simplesmente manipulam certas filas; mas se o canal estiver ocupado, ao fim da operação que está executando informará o processador central através de uma interrupção. A rotina de tratamento de interrupções, depois de tomar as providências necessárias relativamente à interrupção propriamente dita, é a que verifica se há trabalho para o canal e se houver escolhe uma das operações esperando pelo canal e inicia nova operação. Desta maneira enquanto houver trabalho para um canal, este é mantido ocupado constantemente. O tratamento de interrupções de E/S é complexo porque cada tipo de unidade tem a sua característica própria que requer uma ação apropriada para cada caso. Devido ao grande número de dispositivos cada um com uma variedade de tipos de interrupção a rotina de tratamento correspondente não será aqui abordada detalhadamente.

De uma maneira geral uma rotina de interrupção de E/S começa suspendendo outras interrupções de mesma prioridade ou de prioridade mais baixa, salva todos os registradores importantes como o Acumulador, Extensão, IAR, etc., que permitirão que o processador central recomece seu trabalho após o tratamento da interrupção do mesmo ponto em que foi interrompido, e recebe do canal as informações que descrevem a interrupção. Em seguida estas informações são analisadas, verificando se a operação completada pelo canal terminou normalmente, e em caso negativo, toma uma ação apropriada para cada tipo de situação anormal; por exemplo a providência comumente tomada no caso de um erro de paridade na transmissão de um registro físico de uma unidade de fita, disco ou tambor magnético é tentar repetir a operação um certo número de vêzes (baseado na experiência de que erros deste tipo ocorrem muitas vêzes por causas transitórias). Se a operação terminou normalmente, tratando-se de uma leitura, retira o endereço do "buffer" em uso da UAB, e coloca na fila de "buffers" esperando pelo programa; verifica se este programa não estava em estado de espera por esta unidade e se estava retira-o deste estado pois o programa pode continuar agora, já que tem um "buffer" à disposição; chama então a rotina de alocação de tempo do canal, e após executar esta rotina restaura os registradores salvos e as interrupções e retorna.

A rotina de alocação de tempo tem por função determinar qual a unidade seguinte que vai ser atendida pelo canal: se não houver nenhuma unidade esperando pelo canal marca o canal como desocupado e retorna; se houver alguma unidade esperando pelo canal escolhe mediante um algoritmo de alocação de horário a próxima unidade a ser servida pelo canal, retira um "buffer" da fila de "buffers" esperando por esta unidade, coloca-o como "buffer" em uso, e chama a rotina de ativação do canal tendo por parâmetro a unidade. A rotina de ativação já foi examinada.

A rotina de alocação de tempo pode seguir diversas estratégias para a escolha da próxima unidade a ser servida pelo canal. Uma estratégia simples é servir as unidades na ordem em que as requisições de operação de E/S foram feitas. Neste caso é comum encadear os diversos pedidos numa fila à medida que vão chegando, e a rotina simplesmente retira o primeiro desta fila e escolhe a unidade correspondente como a próxima a ser servida pelo canal. Uma outra estratégia simples é servir as unidades ligadas ao canal ciclicamente. Uma estratégia mais complexa é examinar a situação de cada arquivo ativo ligado ao canal e escolher o que tem maiores probabilidades de paralisar o programa por falta de "buffers", eventualmente baseando essa decisão em informação estatística colhida do processamento do arquivo até o momento. É preciso no entanto não esquecer que o que se ganha em cada operação de E/S simultânea com o processamento do programa são em geral apenas algumas dezenas de milissegundos e portanto a rotina de alocação deve ser uma rotina rápida. Uma estratégia aparentemente melhor portanto pode não causar significativa melhora de performance do sistema, devido ao tempo adicional necessário à execução da própria rotina de alocação.

Um arquivo depois de aberto fica ativo até que seja fechado.

A rotina de fechamento de arquivo é análoga à rotina de abertura. Se o arquivo fôr de entrada a rotina retira os "buffers" esperando pela unidade; se o arquivo fôr de saída chama diversas vêzes a rotina de escrita para escrever fisicamente todos os "buffers" esperando pela unidade e o último "buffer" eventualmente incompleto que estava em uso pelo programa, escreve o rótulo de fim de arquivo se o arquivo fôr rotulado, marca o arquivo fechado em FAB; em geral executa ainda uma operação de controle conforme requisitado na chamada como por exemplo reenrolar uma fita magnética e retorna.

Nas rotinas descritas nesta secção tratamos em geral o caso de arquivos de entrada. O caso de arquivos de saída é inteiramente análogo e não será visto aqui.

As rotinas descritas mostram como o processo de E/S é controlado. Evidentemente o SCES controla também a execução de operações de E/S que não envolvem transmissão de dados, solicitadas pelo programa tais como voltar um registro numa fita magnética, posicionar um braço de um disco, pular de página numa impressora, etc..

Estas atividades em geral são controladas por rotinas bastante simples no entanto, e por isso não são descritas com maiores pormenores.

Uma situação que precisa ser prevista pelas rotinas do SCES é o caso de fim de arquivo e de fim de rôlo. Normalmente um programa prevê um desvio para uma certa instrução, no caso de durante a leitura de um novo registro ser encontrado o fim de um arquivo. Como porém no caso de emprêgo de "buffers" múltiplos o programa após a leitura física do fim de arquivo precisa ainda processar vários registros lógicos o SCES deve providenciar o desvio desejado apenas depois de processados todos estes registros. A rotina de leitura de registro lógico deve ter previsão para este caso. Evidentemente depois de ocorrer o fim de arquivo nenhuma leitura física deve ser feita. O caso de fim de rôlo também deve ser previsto, tanto na entrada como na saída, prevendo inclusive mudança eventual na unidade associada ao arquivo.

4.9. Observações:

1. Neste capítulo examinaremos diversos problemas associados à E/S num programa. Em particular foram examinadas as técnicas de uso de "buffers" para controlar E/S simultânea com o processamento de instruções pelo processador central; para arquivos sequenciais. Foi visto que o uso de mais de dois "buffers" por um arquivo é raramente vantajoso. Se vários arquivos possuem registros físicos de mesmo tamanho uma técnica de economizar na memória necessária para os "buffers", utilizada por alguns SCES é a Associação de "Buffers" ("Pool of Buffers"). Nesta técnica n "buffers" de mesmo tamanho são utilizados por m arquivos. O SCES vai retirando da "Associação" os "buffers" à medida que são necessários a um arquivo e liga-os ao arquivo. Quando o programa processou um "buffer" de um arquivo de entrada ou uma unidade esvaziou um "buffer" de saída, o "buffer" é devolvido à associação. Como os "buffers" são retirados da associação e devolvidos por diversos arquivos é necessário encadear os "buffers" uns aos outros numa lista ligada ("linked list"), cada "buffer" apontando para o "buffer" seguinte da lista. O processo funciona numa maneira bastante similar ao visto nas secções anteriores, exceto que o sistema de controle da E/S precisa controlar também a "Associação de "buffers"", o que é feito com o auxílio de uma outra tabela de atividade, o PAB, (de "Pool Activity Block"), e precisa decidir dinamicamente como repartir os n "buffers" entre os m arquivos.

2. Como ressaltamos, também anteriormente, o emprego de "buffers" para arquivos de acesso aleatório em geral é de pouca utilidade, e assim geralmente no caso de arquivos aleatórios é conveniente usar apenas um "buffer" e o programa terá de esperar normalmente pelo término da operação de E/S. O controle de unidades de acesso aleatório como o disco magnético é mais complexo pois estas unidades em geral podem estar associados a diversos arquivos ao mesmo tempo. Para tomar conta do espaço de discos magnéticos (ou dispositivos similares de acesso aleatório) o sistema precisa manter um mapa do espaço disponível e um diretório de arquivos que descrevem os arquivos gravados no disco. O diretório em geral possui a identificação e localização no disco de cada arquivo e informações quanto ao tipo de acesso permitido ao arquivo e outras informações de eventual interêsse (por ex.: data de criação, frequencia de uso, etc.). Como em geral unidades de acesso aleatório contém arquivos de diversos usuários especialmente se o meio de armazenamento não fôr removível, é necessário o sistema prover mecanismos de

proteção dos arquivos de um usuário de outro. Uma maneira simples de estruturar o diretório para se ter um mecanismo adequado a acesso a arquivos por usuário é a de uma árvore de dois níveis esquematizada na figura 4.5..

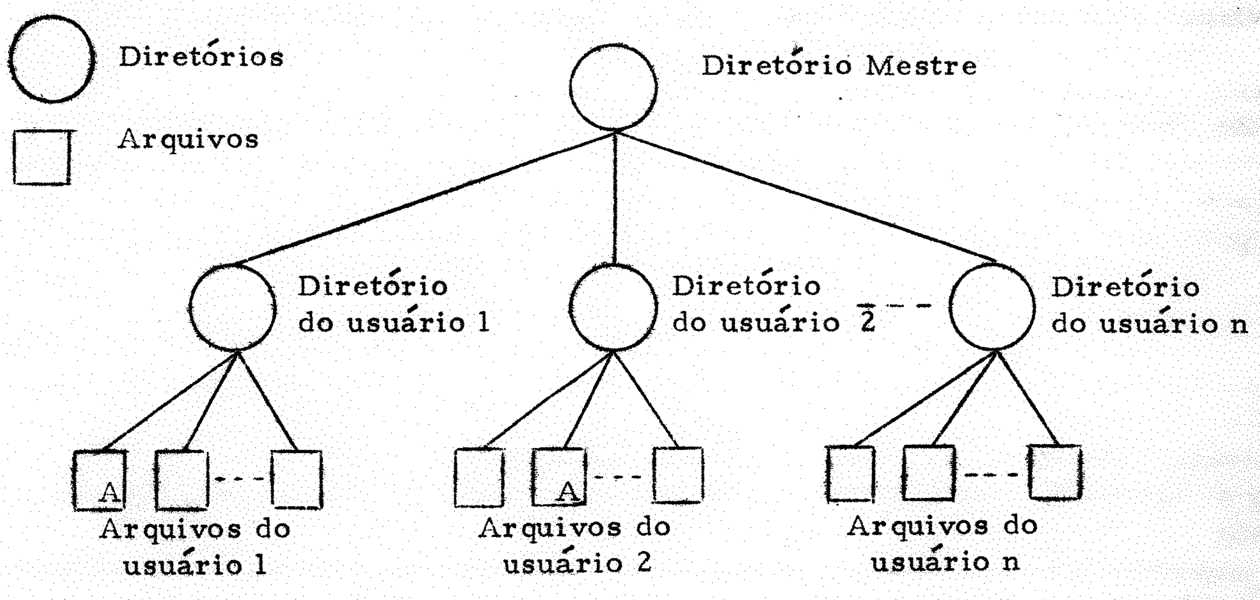


Fig. 4.5.

Esta estrutura permite que dois usuários tenham arquivos com o mesmo nome. Para se ter acesso a um arquivo é necessário especificar o usuário (que se não especificado pode ser assumido como o usuário solicitante do acesso) e o nome do arquivo. Cada usuário ganha acesso aos seus arquivos e ao seu diretório podendo mudar nomes de seus arquivos, acrescentar ou eliminar arquivos, mudar o tipo de acesso a seus arquivos, etc., e ter acesso aos arquivos de outros usuários, desde que autorizado a tanto. (Normalmente a proteção do acesso de um arquivo consiste de uma lista dos usuários que podem ter acesso ao arquivo e o tipo de acesso que cada um pode ter, como: só pode ler, só pode escrever, pode escrever e ler, só pode executar).

Outras estruturas de diretório mais genéricas podem ser encontradas em 1 .

3. O SCES é programado cuidadosamente para sincronizar processos paralelos evitando que um dado seja utilizado antes de ser produzido, que um processo espere quando seu progresso não é bloqueado, que menos ou mais "buffers" sejam utilizados do que os disponíveis. Parte desta sincronização é evidente pela estrutura dos algoritmos expostos. Como porém interrupções ocorrem em instantes aleatórios, certos problemas podem surgir no sistema exposto, se cuidados especiais não forem tomados para evitá-los, no caso de uma interrupção ocorrer numa situação "infeliz". Por exemplo se durante a execução da rotina de leitura de registro lógico, sendo processado paralelamente à entrada de um registro físico num "buffer" do mesmo arquivo, houver necessidade de novo "buffer" mas a fila de espera pelo programa estiver vazia, e ocorrer uma interrupção logo depois desta fila ser consultada, a rotina de interrupção colocará o "buffer" que acabou de ser lido na fila de espera pelo programa e retornará à rotina de leitura, que no en-

tanto, baseado numa informação anterior à interrupção colocará o programa em estado de espera por um "buffer" que já foi lido ! Este problema, e outros similares, são agravados pelo caráter aleatório do instante em que uma interrupção ocorre, ficando virtualmente impossível reproduzir a mesma situação repetidas vêzes, o que torna o teste exaustivo do SCES extremamente difícil.

Discutiremos o problema da sincronização de processos paralelos no capítulo seguinte em maior detalhe.

Capítulo 5

Conceitos fundamentais de sistemas multiprogramados e de tempo compartilhado

5.1. Introdução

Na secção 1.6. foi esboçada rapidamente a evolução dos sistemas operacionais. A complexidade de um sistema operacional pode ser muito variada dependendo de que provisões o sistema tem. Se o sistema fôr por lotes (tipo "batch") e de uniprogramação, ié apenas um programa está na memória por vez, e é executado do início ao fim sem sair da memória então o sistema poderá consistir de um monitor simples que interpreta os cartões de controle do sistema. Ele delega tarefas para as diversas componentes do sistema à medida que são necessárias: um compilador, o link-editor, o sistema de controle de entrada/saída, etc.. É provável que para que o sistema possa efetuar com razoável eficiência as suas funções pelo menos certas partes do sistema devem permanecer permanentemente na memória, podendo sempre que necessários desempenhar rapidamente suas funções.

Já vimos porém que estes sistemas têm certas limitações, como:

1. O "hardware" não será utilizado com eficiência quando um programa usar E/S intensamente.
2. Uma vez iniciado um programa ele prossegue até o seu término. Assim programas curtos que são submetidos para processamento depois que um programa longo começa a ser executado precisarão esperar o término deste para poderem ser executados.

Estas limitações motivaram o desenvolvimento de sistemas operacionais mais sofisticados. As duas idéias que surgiram para tentar resolver estes problemas foram os sistemas multiprogramados e de tempo compartilhado, como já o mencionamos na secção 1.6.. Sistemas de multiprogramação são vistos em geral como uma extensão de sistemas do tipo "batch", no sentido de que para o usuário os dois sistemas parecem idênticos: nos dois sistemas ele entrega a sua tarefa ("JOB"), ao operador, e recebe algum tempo depois os resultados. O tempo de devolução, é de se esperar, provavelmente será melhor num sistema de multiprogramação, supondo que ambos os sistemas recebam a mesma carga de serviço. Em sistemas de tempo compartilhado o usuário terá acesso ao sistema através de terminais podendo interagir com este. Vários usuários tem acesso simultaneamente ao sistema e assim estes sistemas são chamados de acesso múltiplo. Sistemas de tempo compartilhado podem ser uniprogramados movendo-se a intervalos de tempo curtos o processo que está na memória para um meio auxiliar de armazenamento, denominado em geral de memória auxiliar, como um tambor ou disco magnético, para dar lugar a um outro processo. Este foi o caso de sistemas pioneiros como o CTSS desenvolvido no MIT na primeira metade da década de 1960. Porém sistemas modernos de tempo compartilhado, para terem maior eficiência, são multiprogramados, e na realidade a distinção entre sistemas de mul-

tiprogramação e de tempo compartilhado vem se tornando mais difícil na medida em que se aproveitam técnicas inicialmente desenvolvidas para sistemas de tempo compartilhado em sistemas de multiprogramação e viceversa.

Neste capítulo examinaremos as técnicas e os conceitos básicos que aparecem em sistemas de multiprogramação e tempo compartilhado. No restante deste capítulo usaremos o termo sistema multiprogramado para designar um sistema em que vários programas coexistem na memória, podendo ser um sistema de acesso múltiplo (tempo compartilhado) ou um sistema por lotes ou misto.

5.2. Programas reentrantes

Em sistemas multiprogramados é comum vários usuários usarem num determinado momento um compilador, ou algum outro programa muito utilizado. Uma maneira de se fazer isto é prover uma cópia do compilador para cada usuário. Desta maneira em determinados momentos teremos diversas cópias do mesmo programa na memória. Em virtude porém de a memória ser um recurso escasso, convém utilizá-lo da melhor maneira e assim é óbvio que é de interesse estudar uma maneira de conseguir que uma só cópia do compilador pudesse ser compartilhada por diversos usuários. Um programa que pode ser usado simultaneamente por diversos processos é denominado de programa reentrante, ou procedimento puro. O conceito de processo será definido mais tarde. Definiremos informalmente um processo como um programa em execução.

A execução de um programa consiste na aplicação de uma seqüência de instruções ao conjunto de dados sobre a qual o programa trabalha. O programa é a especificação destas instruções. O estado do processo fica após cada instrução perfeitamente caracterizado pelo conteúdo das células de memória que contêm o programa e os dados que são processados pela execução do programa, e por um indicador que determina qual a próxima instrução a ser executada. Um processo portanto pode ser interrompido num certo ponto e recomeçado do mesmo ponto mais tarde, sem nenhuma conseqüência quanto à tarefa que está sendo executada, exceto quanto à velocidade de execução, desde que no intervalo de tempo entre a interrupção da execução e o reinício da mesma não se alterem as instruções que descrevem o programa, e os dados que são processados, enfim o estado do processo. Em particular, desde que as condições acima sejam satisfeitas, as próprias instruções do programa podem ser executadas neste intervalo de tempo, atuando sobre uma outra área de dados, ié o mesmo programa pode ser executado como um outro processo.

Para que um programa possa ser reentrante é necessário portanto que:

1) o programa possa endereçar várias áreas de dados distintas por algum mecanismo, uma área para cada processo;

2) as instruções não devem se modificar a si mesmas.

O primeiro requisito é conseguido separando-se as instruções

da área de dados, e com certas técnicas de endereçamento que veremos na secção 5.3.. O segundo requisito pode ser conseguido disciplinando a programação de compiladores, e outros programas que se desejam reentrantes; evitando utilizar técnicas de programação que modificam as instruções do programa.

É interessante observar a analogia entre programas reentrantes e subrotinas recursivas que examinamos na secção 1.4.. Também em subrotinas recursivas as mesmas instruções são executadas sobre áreas de dados diferentes, pois o valor dos parâmetros e do ponto de chamada varia de chamada para chamada e variáveis temporárias utilizadas pela subrotina, devem também em geral ter um endereço para cada chamada da mesma, conforme vimos na secção 1.4..

5.3. Métodos de relocação dinâmica e endereçamento

Em sistemas multiprogramados a memória é utilizada simultâneamente por diversos processos. Este fato pode ocasionar uma série de problemas que procuraremos discutir nesta secção.

Na secção anterior vimos o conceito de compartilhamento de um programa por vários processos com uma só cópia do progrma na memória como um meio de utilizar melhor a memória. Como o tamanho da memória é bastante limitado, devido ao preço elevado e tamanho físico, é em geral necessário também que durante a execução de um processo seja necessário remover partes do mesmo ou de outros processos sendo executados para a memória auxiliar, para dar espaço à (parte) de um outro processo. Esta técnica se chama de "swapping". A memória auxiliar é de acesso mais lento que a memória principal, mas comparativamente tem uma capacidade de armazenamento enorme, em alguns casos podendo-se considerar praticamente de capacidade ilimitada. Na disucssão dos problemas de alocação de memória nesta secção teremos sempre em vista estes dois conceitos.

Neste ponto é conveniente introduzir a noção de memória virtual. Um processo utiliza elementos de informação de um espaço de objetos que chamaremos de seu espaço de endereçamento. Um item de informação é identificado pelo seu endereço neste espaço. Por motivos que se tornarão logo aparentes, é conveniente distinguir 2 espaços de endereçamento:

i) o espaço físico — que consiste das células de memória físicas individualmente endereçáveis, ou memória física e

ii) o espaço lógico — também denominado de espaço abstrato, espaço de nomes, ou memória virtual que consiste de células de memória de uma memória imaginária. O conteúdo da memória virtual frequentemente é armazenado na memória auxiliar.

Um processo se refere a posições de sua memória lógica: cada endereço do processo é um endereço virtual que se refere a um objeto da memória virtual. Quando um processo é executado o programa correspondente é interpretado pelo "hardware" e pelo menos a parte do programa que está sendo executada precisa estar na memória física, bem como a parte dos da-

dos que estão sendo processados. Em suma uma parte da memória virtual do processo deve estar armazenada na memória física. Cada endereço do processo, que é uma referência a uma posição da memória virtual precisa ser transformado pelo "hardware" por meio de uma função apropriada no endereço físico da célula física que neste instante contém a informação endereçada. Esta função, ou mapeamento pode ser de diversos tipos conforme veremos a seguir. Até este ponto não fazíamos distinção entre a memória virtual e a memória física, porém as memórias convencionais em que isto era feito também poderiam ser vistas como uma memória virtual e uma memória física, como o exposto acima, em que a função que transforma um endereço lógico em um endereço físico é a função identidade. Neste caso o tamanho da memória virtual é igual ao da memória física, mas é importante observar que a memória virtual pode ser menor ou maior do que a memória física, conforme o método de mapeamento.

Suponhamos que "swapping" seja utilizado pelo sistema para o emprego mais eficiente da memória física. Se tivermos numa memória convencional um processo que temporariamente vai para a memória auxiliar para dar lugar a um outro processo, quando o primeiro for movido para a memória de novo para poder continuar sua execução, mais tarde, ele deverá ser movido ou exatamente para as mesmas posições físicas que ocupava antes de ser removido, ou seus endereços relocáveis devem ser relocados conforme o método visto no capítulo 3. Mover para a mesma posição é um procedimento excessivamente rígido para um sistema multiprogramado em que partes da memória são utilizadas por processos diferentes, e por outro lado o método de relocação visto no capítulo 3 é excessivamente demorado para poder ser utilizado com eficiência freqüentemente. Este método de relocação chama-se também de relocação estática devido a esta razão. A estrutura de memória de um sistema convencional não é, portanto, prática para a implementação de "swapping".

Para termos a flexibilidade de poder carregar um processo durante a sua execução, em consequência de "swapping", para posições distintas das originalmente ocupadas, são necessários métodos práticos de relocação que sejam eficientes mesmo que o processo tenha de ser relocado frequentemente. Estes métodos chamam-se métodos de relocação dinâmica.

Um método de relocação dinâmica usado freqüentemente consiste no emprego de registradores especiais no "hardware", denominados de <u>registradores-base</u>. Suponhamos inicialmente que o processador central tenha um registrador-base. Neste caso o endereço físico é determinado somando-se ao endereço lógico o conteúdo do registrador-base, que deve apontar para o endereço físico do início da área de memória física que contém o processo. Usando registrador-base um processo pode ser facilmente relocado dinamicamente, em consequência de "swapping", pois se o processo for carregado numa outra área da memória física basta mudar o conteúdo do registrador-base que deve apontar para o novo início da área que contém o processo agora. Normalmente o emprego de um outro registrador permite que se tenha proteção física por "hardware" de qualquer outra área não ocupada pelo processo de acessos invertidos deste. Trata-se de um registrador que aponta para o fim da área que contém o processo. Cada endereço físico gerado é comparado automaticamente pelo "hardware" com o endereço do começo e do fim da área que con-

têm o processo. Uma interrupção é gerada se o processo tentar endereçar uma célula fora dos limites permitidos.

O compartilhamento de um programa reentrante por dois ou mais processos pode ser conseguido com o emprego de dois registradores-base em vez de um. O registrador-base 1 aponta para o início da área que contém o programa reentrante e o registrador-base 2 aponta para o início da área de dados do programa. Em cada acesso um dos dois registradores-base é utilizado pelo "hardware". A especificação de qual o registrador-base que deve ser utilizada pode ser tanto explícita, ié fazendo parte do endereço lógico, como implícita conforme o tipo de acesso que se está fazendo (ex.: uma instrução de desvio utiliza o registrador-base 1; uma instrução de carga ou descarga do acumulador utiliza o registrador-base 2; etc.). Quando o programa reentrante está sendo utilizado pelo processo A o registrador-base 2 aponta para a área de dados de A; quando o processo B está sendo executado o registrador-base 2 aponta para a área de dados de B. A fig. 5-1 ilustra este processo.

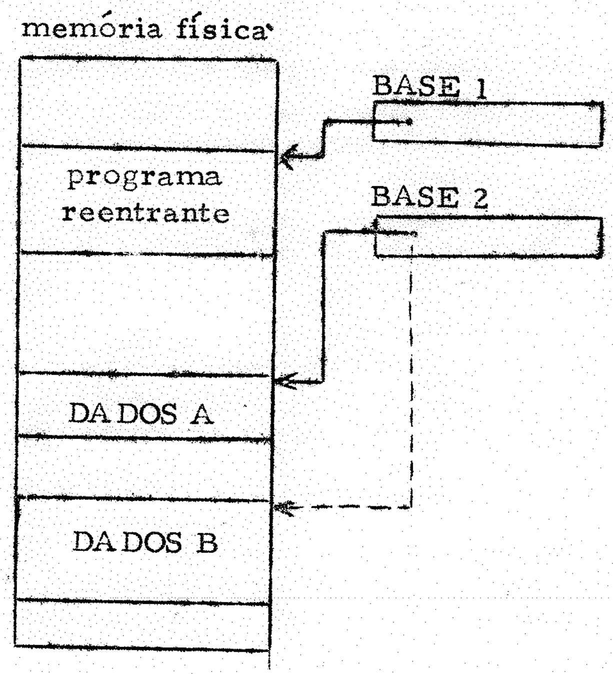


Fig. 5-1. Compartilhamento de programas por meio de dois registradores-base.

O emprego de um ou dois registradores-base resolve o problema da relocação dinâmica mas apresenta duas características inconvenientes:

a) a memória virtual conseguida será menor ou igual à memória física.

b) cada processo precisa ocupar uma área contigua de memória física no caso de um registrador-base, ou duas áreas contiguas, uma para programa e outra para a área de dados no caso de dois registradores-base. Como num sistema multiprogramado a memória é utilizada dinâmicamente por diversos processos, áreas livres são criadas quando

um processo termina e ocupadas quando um processo se inicia. Eventualmente, como consequencia deste mecanismo, teremos a situação em que temos n áreas ocupadas intercaladas por m áreas livres. Se um novo processo precisa ser iniciado numa situação da memória deste tipo, é possível que aconteça que não exista nenhuma área contígua de tamanho suficiente para abrigar o processo embora a soma das áreas livres seja maior do que a área de memória requisitada, sendo necessário ou compactar as áreas ocupadas ou remover parte de um processo da memória física.

Estes dois problemas motivaram o aparecimento de uma nova técnica de relocação dinâmica denominada de paginação.

Se quisermos que a memória virtual de um processo seja maior que a memória física é necessário evidentemente que de cada vez apenas uma fração da memória virtual esteja na memória física. Basicamente é o que fizemos com a técnica de superposição vista no capítulo 3. Esta técnica é entretanto uma solução apenas relativamente modesta, pois ou é de responsabilidade do programador, ou exige uma disciplina rígida de programação para se poder controlar quais as áreas superponíveis. Podemos pensar também na técnica de superposição como num método que permite não o emprego de uma memória virtual maior que a memória física mas como um método engenhoso de alocar as mesmas áreas da memória virtual para finalidades diferentes durante a execução do processo em instantes diferentes, e que acaba dando a impressão de uma memória virtual maior do que a memória física. Se porém a memória virtual for de "nascença" maior que a memória física o sistema precisa ter um mecanismo automático que permita que se tenha apenas uma parte da memória virtual na memória física. O problema com o registrador-base que impede isto é que a memória ocupada pelo processo é ocupada de uma vez, contiguamente na memória física. Foi uma evolução natural portanto dos problemas que um registrador-base causa, tentar solucioná-los permitindo que um processo seja fracionado na memória física, embora seja contíguo na memória lógica. É precisamente isto o que se faz na técnica de paginação.

Num sistema paginado a memória virtual é subdividida em n áreas de mesmo tamanho, chamadas de páginas. Cada página tem um número de 0 até n-1 e tem m células numeradas de 0 a m-1. Uma célula é endereçada pelo número da página que o contém e pelo número da célula dentro da página denominado de número da linha. Em geral $m = b^{k_1}$ e $n = b^{k_2}$ onde b é a base em que é expresso o endereço pois assim reservando-se k_1 casas para a linha na página e k_2 casas para o número de página a célula da página p linha l é a célula de número $(pl)_b$, ié pl na base b, da memória virtual, (começando a numeração de 0). (Por exemplo se b = 10, $k_1 = 3$ e $k_2 = 3$ temos 013980 representando a linha 980 da página 013, e é a célula 13980 da memória virtual pois há 13 páginas de 0 a 12 antes da página considerada, cada uma com 1000 células).

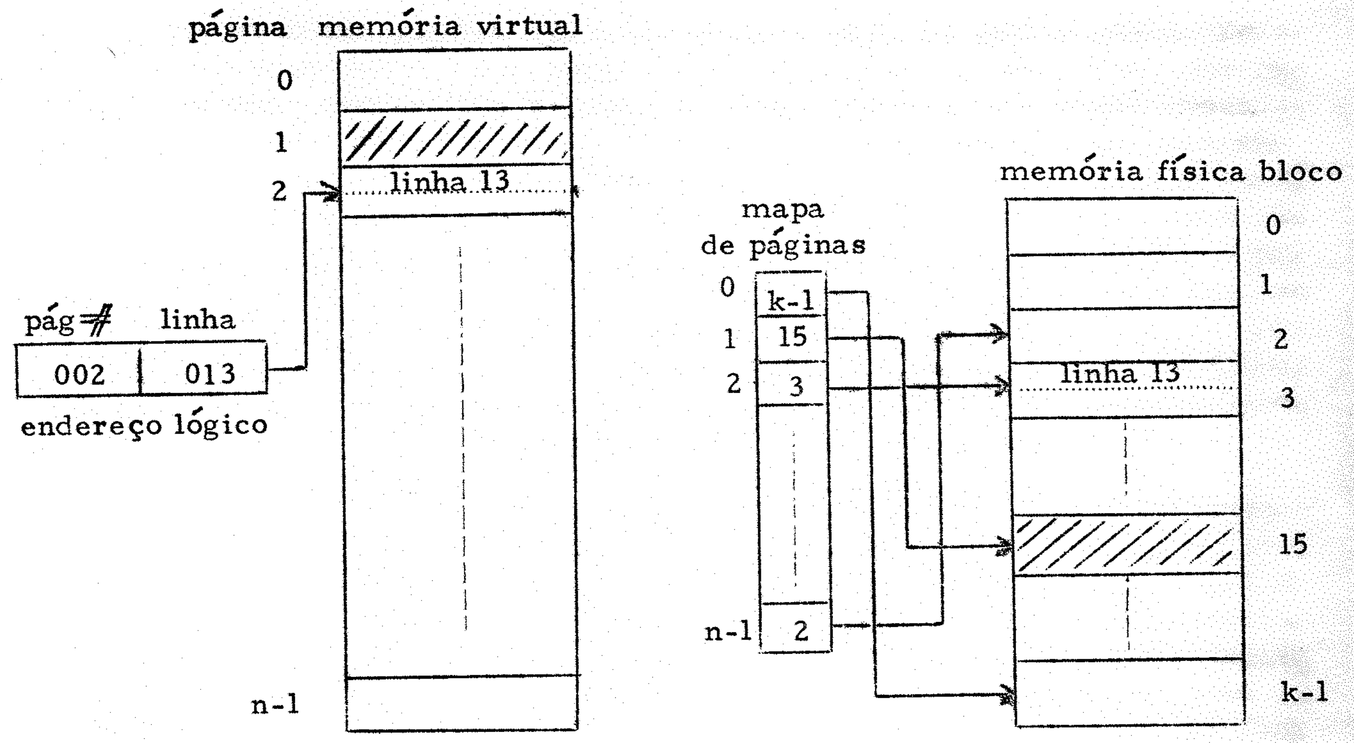


Fig. 5-2 - Paginação

A memória física é subdividida em k blocos, de mesmo tamanho que as páginas de modo a podermos ter uma página da memória virtual armazenada num dos blocos da memória; n pode ser diferente de k, ié a memória virtual pode ser maior (ou menor) que a memória física. A correspondência entre uma página e o bloco que a contém é feita através de um mapa de páginas que é uma tabela de acesso direto cuja entrada i contém o número do bloco que contém a página i se esta página estiver presente na memória, e um indicador se a página não está presente. Um endereço físico é formado a partir de um endereço lógico pelo "hardware" com o auxílio da tabela de páginas: o número da página é a entrada da tabela que contém o número do bloco se a página estiver presente; o número do bloco é então concatenado com o número da linha determinando-se o endereço físico. Se a página não estiver presente é gerada uma interrupção, para o sistema operacional, em consequência, localizar a página faltante na memória auxiliar, achar um bloco da memória disponível e carregar a página necessária ao processo atualizando a tabela de páginas. Se uma página precisa ser removida do bloco de memória que ocupa durante o "swapping" a correspondente entrada na tabela de páginas do processo que utiliza a página deve ser atualizada refletindo esta situação. Cada processo em execução tem uma tabela de páginas refletindo a correspondência entre os blocos das páginas presentes na memória física e a memória virtual. O mecanismo está ilustrado na Fig. 5-2.

Este método de endereçamento resolve os dois inconvenientes citados em conexão com registradores base, pois permite que um processo fique fragmentado na memória física embora suas páginas sejam contíguas na memória lógica; permite que a memória virtual seja maior que a memória fí-

sica, pois não há necessidade que todas as páginas de um processo estejam presentes na memória física; utiliza com eficiência a memória física pois para cada processo precisamos de áreas contíguas do tamanho de uma página apenas. É verdade que ocasiona algum desperdício na última página de cada processo, que em geral não estará inteiramente ocupada. Este desperdício é de 1/2 página em média, pois alguns processos utilizarão mais que metade da página outros menos que a metade, sendo razoável supor que em média o desperdício será de 1/2 página. Qualquer página que é removida de um bloco pode ser depois colocada em qualquer outro bloco, bastando atualizar de acordo a tabela de páginas do processo, portanto a paginação é um método eficiente de relocação dinâmica também. Como veremos depois, compartilhamento de programas pode ser conseguido também com relativa facilidade.

Um problema que precisa ainda ser respondido é onde é armazenada a tabela de páginas.

Uma maneira é evidentemente guardar a tabela na memória e ter um apontador para a tabela de páginas do processo que está sendo executado pelo processador central. Esta maneira no entanto não é adequada pois para cada célula que tivermos de endereçar será necessário fazermos 2 acessos à memória em vez de 1; um acesso para determinar o bloco que contém a página e outro para fazer o acesso à célula propriamente dita. Esta maneira portanto tem de ser rejeitada pois dividiria a velocidade de acesso à memória por 2. Se o número de páginas da memória não for muito grande poderíamos ter uma tabela de páginas por registradores do processador central. Neste caso esta tabela seria carregada com a tabela de páginas do processo quando este começasse ou recomeçasse a sua execução. Mas um número de páginas pequeno significa memória virtual pequena e portanto no caso em que a memória virtual é significativamente maior não é economicamente viável ter uma tabela de páginas com muitos registradores. A solução neste caso é o emprego de um dispositivo chamado de memória associativa. A tabela com todas as páginas fica na memória mas os números dos blocos das páginas mais utilizadas são armazenados na memória associativa. A memória associativa é uma tabela de registradores especial que contém a chave, (neste caso a página), e o valor associado à chave (que neste caso é o número de bloco); uma busca na memória associativa por uma página é feita comparando-se simultâneamente todas as entradas da tabela com a chave procurada. Desta maneira na maior parte dos acessos não haverá necessidade de fazermos um acesso extra à memória, que agora só será necessário se a página procurada não estiver na memória associativa, e devido ao fato de todas as entradas da tabela serem simultaneamente varridas na busca o número do bloco correspondente a uma página é determinado com extrema rapidez.

Num sistema paginado podemos compartilhar um programa sem problema entre dois processos, bastando para isso que as páginas da área de dados sejam distintas das páginas do programa reentrante e que as páginas do programa sejam colocadas na tabela de páginas dos processos que compartilham o programa. Como porém temos endereços no programa reentrante, as páginas deste devem estar nas mesmas páginas da memória virtual dos processos que compartilham o programa. Pelo mesmo motivo a área de dados do programa tem de começar na mesma página, embora o número de páginas ocupadas pela área de dados possa ser variável nos diversos processos.

Uma área de dados também pode ser compartilhada por dois ou mais processos. Se a área não tiver referências (endereços indiretos) ao programa, então as páginas do programa não necessitam ocupar as mesmas páginas das respectivas memórias virtuais; analogamente se não houver referências na área de dados a si mesma, esta poderá ocupar páginas distintas nas memórias virtuais dos processos que a compartilham.

É digno de notar que os métodos de relocação dinâmica vistos até este ponto não eliminaram a necessidade de termos relocação estática e um link-editor do tipo visto no capítulo 3. Aquelas técnicas foram desenvolvidas para se poder juntar um programa a partir de módulos independentemente programados e compilados. O compilador que compila um determinado módulo não tem a informação da posição em que o módulo ficará na memória virtual e portanto tem de compilar o módulo como se este fosse para a posição zero da memória virtual sendo necessário na hora da link-edição relocar o módulo estaticamente na memória virtual.

Para finalizarmos esta secção vamos estudar a técnica de segmentação. A motivação para o desenvolvimento de mais uma técnica de relocação dinâmica reside em certos inconvenientes que um sistema paginado ainda apresenta. Estes inconvenientes são os seguintes:

a) Se uma página é compartilhada por vários processos ela comparece em cada uma das tabelas de páginas dos processos que a compartilham. Se a página portanto for removida da memória, devido a "swapping" a tabela de páginas de todos os processos devem ser alteradas. O mesmo acontece se uma página compartilhada é carregada na memória. Isto não só, pode ser demorado, como ainda exige um certo trabalho de contabilidade para sabermos quais os processos que compartilham uma determinada página.

b) Se quisermos compartilhar não só programas, mas também módulos arbitrários como subrotinas por exemplo, então como vimos uma subrotina compartilhada tem de ocupar as mesmas páginas em todos os processos que o compartilham. Isto significa que na hora da link-edição as subrotinas já link-editadas em algum processo precisam ir para as mesmas posições na memória virtual, o que não só significa um desperdício de memória virtual, exigindo portanto uma memória virtual enorme, como um trabalho de contabilidade bastante considerável.

c) Uma memória virtual paginada apresenta dificuldades quando temos estruturas de dados num processo, que podem crescer ou decrescer dinamicamente. Neste caso o programador deverá alocar explicitamente estas estruturas na memória virtual, deixando espaço para as estruturas poderem crescer. Isto também causa desperdício de memória virtual e exige também memórias virtuais enormes que serão usadas no entanto de maneira bastante esparsa.

As dificuldades acima motivaram o aparecimento da técnica de segmentação. Há diversas maneiras para se implementar o conceito de segmentação. Examinaremos o método empregado no MULTICS ("multiplexed information and computing service") que é um sistema de tempo compartilhado extremamente versátil e complexo desenvolvido no MIT para o GE-645.

Num sistema segmentado a memória virtual se compõe de um certo número de áreas de memória (virtual) contígua, denominados de segmentos. Um item de informação é endereçado especificando-se qual o segmento da memória virtual que o contém e qual o seu endereço relativo ao início deste segmento. Um segmento pode conter um módulo de um processo seja de dados ou de instruções. A posição de um segmento de um processo na memória virtual, ié a especificação de qual o segmento que o contém, deve ser determinado apenas ao tempo de execução permitindo que um segmento compartilhado por vários processos possa ocupar posições diferentes nas respectivas memórias virtuais, apesar de termos apenas uma cópia do segmento na memória auxiliar e/ou na memória principal. O que examinaremos nos parágrafos subsequentes desta secção é qual o mecanismo pelo qual o conceito de segmentação é implementado no MULTICS para o GE-645 e como este mecanismo permite resolver os problemas mencionados que surgem num sistema apenas paginado.

No MULTICS - GE-645 a memória virtual se compõe de 2^{18} (ié 256k onde $1k = 2^{10} = 1024$) segmentos cada um com 2^{18} palavras individualmente endereçáveis de 36 bits cada. Cada segmento tem um número associado que pode ser visto como o endereço do segmento na memória virtual, denominado de número do segmento. Na memória auxiliar a parte ocupada de cada segmento é armazenada como um arquivo sendo portanto identificado por um nome, que indicaremos por < s>. O número do segmento <s> na memória virtual de um processo será indicado por <s #>, e pode ser, como logo veremos, diferente para vários processos que compartilhem um mesmo segmento <s>. Por outro lado cada célula de um segmento < s> tem um número associado que é o seu endereço dentro do segmento. Como um segmento é potencialmente grande, cada segmento é paginado de modo que num certo instante podemos ter apenas a parte do segmento necessária ao processo presente na memória principal em blocos eventualmente não contíguos.

Um item de informação num segmento <s> é identificado <u>no programa fonte</u> por um par de símbolos <s> | [α] onde <s> é o nome simbólico do segmento e [α] o nome simbólico do item de informação dentro do segmento. [α] é transformado pelo compilador no endereço relativo ao começo do segmento <s>, que indicaremos por [α #], eventualmente modificado ao tempo de execução por endereçamento indexado e indireto mas <s #>, o número do segmento que conterá <s> durante a execução do processo será obtido apenas na hora da execução, pelo "hardware" de um registro apropriado. < s #> será fixado para um processo durante a execução do mesmo quando <s> for endereçado pela primeira vez.

O GE-645 além dos registradores usuais como acumulador, indexadores, IAR (denominado de "program counter" no GE-645), etc., possui mais alguns registradores usados especialmente no processo de endereçamento. Estes registradores são o PBR ("Procedure base register") que contém o <u>número do segmento</u> que está sendo executado pelo processador central; o DBR ("descriptor base register") que contém um apontador para uma tabela denominada de SDT ("segment descriptor table"), e 4 registradores chamados BPR ("base-pair-registers") cada um contendo um número de segmento e um endereço base neste segmento. O SDT é por sua vez um segmento que contém a localização de todos os segmentos já endereçados pelo processo, ié dos seg-

mentos já conhecidos do processo. Um segmento $<s>$ quando é endereçado pela primeira vez recebe, por um mecanismo que veremos depois, um número $<s\#>$ que é o índice do primeiro elemento livre nesta altura do SDT. Na entrada correspondente do SDT é colocado um apontador para o bloco de memória que contém a tabela de páginas deste segmento. Para endereçar um item de informação $<s>|[\alpha]$ de um segmento já conhecido o "hardware" procede do seguinte modo:

1) obtém $<s\#>$, de um dos registradores PBR, DBR ou de uma palavra da memória, conforme o tipo de acesso;

2) consultando DBR obtém a localização do SDT; com $<s\#>$ obtém do SDT o bloco de memória que contém a tabela de páginas de $<s>$;

3) a partir de $[\alpha\#]$ que é composto de 2 partes, número da página e número de linha, determina o bloco de memória que contém a página em questão, da tabela de páginas de $<s>$;

4) obtido o número do bloco que contém o item de informação $<s>|[\alpha]$ a célula correspondente é atingida endereçada pelo número da linha dentro do bloco que contém a página em questão.

Estes conceitos são ilustrados na fig. 5-3. (a) e (b).

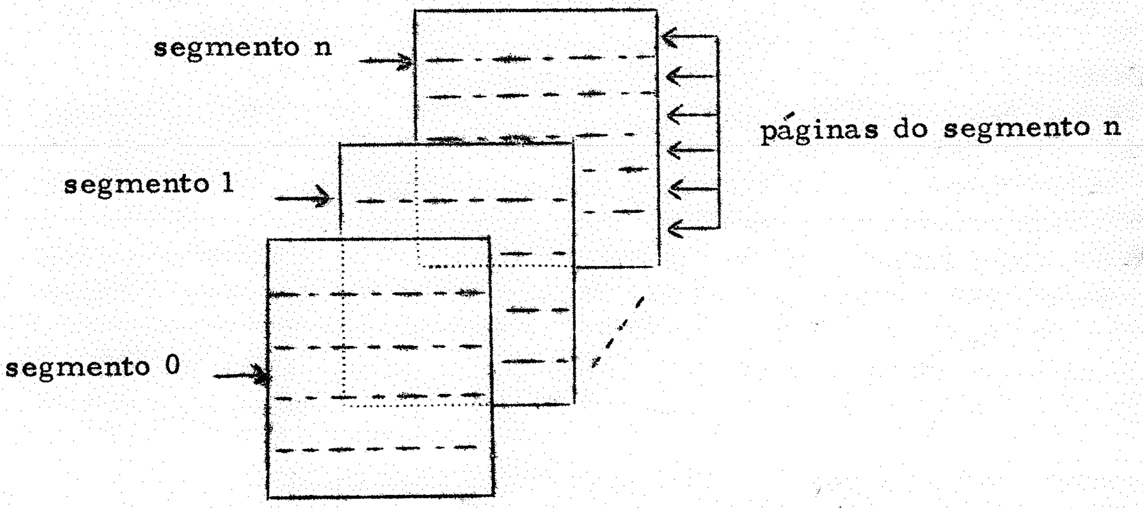
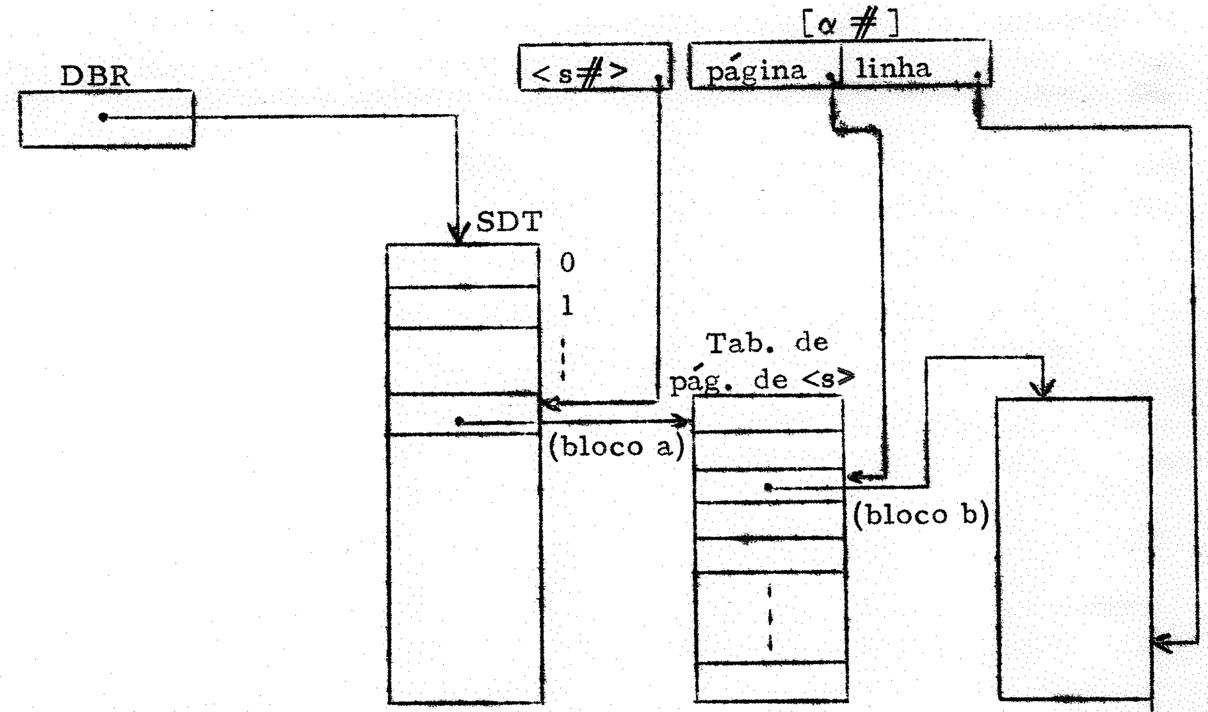


Fig. 5-3(a) - A memória virtual segmentada



bloco a contém a tabela de páginas de <s> .
bloco b contém a página de < s> endereçada.

Fig. 5-3(b) - Endereçamento num segmento < s #> .

Deve-se observar que o SDT é um segmento, e portanto paginado também. Assim na realidade o DBR aponta não para o SDT diretamente mas para a tabela de páginas do SDT, que está num bloco da memória principal, e o < s #> se compõe também de uma parte que determina qual a página do SDT endereçada por < s #> , e do número da linha desta página que contém o apontador para o segmento < s > (ié para a tabela de páginas do segmento < s >).

Para compreender este mecanismo de endereçamento falta apenas examinarmos como é obtido o número do segmento < s #> , de algum dos registradores: PBR, DBR, ou de alguma palavra da memória, e o endereço dentro do segmento [α #] .

A instrução do GE-645 tem o formato representado na fig. 5-4.

sbpr	campo de endereço	código op.	bit s-e	modo

Fig. 5-4. Formato da instrução do GE-645

Quando o acesso à memória é para o processador central obter nova instrução, < s #> é o número do segmento sendo executado e portanto é

obtido do PBR; o endereço da instrução dentro do segmento está no IAR; obtido assim o par $<s\#>|[\alpha\#]$ o hardware procede conforme o descrito acima.

Se uma instrução faz um acesso à memória, então se o bit s-e (bit de segmento externo) estiver desligado a referência é interna ao segmento sendo executado, e $<s\#>$ é obtido do PBR; se o bit s-e estiver ligado o campo <u>sbpr</u> (seleciona "base-pair-register") indica qual dos 4 BPR deve ser usado, e $<s\#>$ é obtido do BPR indicado; o campo <u>modo</u> especifica se o endereço dentro do segmento especificado deve ser indexado por algum indexador e/ou indireto; o campo de endereço portanto depois de eventualmente indexado dá um endereço que já é $[\alpha\#]$ no caso de referência interna ao segmento sendo executado ou em caso contrário é ainda somado ao endereço base contido no BPR usado para dar $[\alpha\#]$, obtendo assim o par $<s\#>|[\alpha\#]$. Se este endereço for indireto $<s\#>|[\alpha\#]$ aponta para um <u>par de palavras</u> da memória principal cujo conteúdo vai determinar um novo endereço $<s'\#>|[\alpha'\#]$ que por sua vez pode ser ou não indireto e assim por diante.

Um par de palavras localizado por uma referência indireta a $<s\#>|[\alpha\#]$ (que aponta para a primeira palavra do par), pode ser interpretado de 3 maneiras conforme a configuração de certos bits na primeira palavra do par:

a) $<s'\#> = <s\#>$ e $[\alpha'\#]$ é obtido da primeira palavra do par;

b) $<s'\#>$ está na primeira palavra do par e $[\alpha'\#]$ na segunda palavra;

c) $<s'\#>$ está no BPR indicado na primeira palavra e $[\alpha'\#]$ na segunda palavra.

O mecanismo de endereçamento descrito até aqui funciona perfeitamente do ponto de vista lógico, mas seria muito lento pois para cada acesso a uma palavra de endereço $<s\#>|[\alpha\#]$ são necessários vários acessos à memória: um acesso para a tabela de páginas do SDT, um para obter do SDT a localização da tabela de páginas do segmento $<s>$, um para obter o bloco de $<s>$ endereçado, e finalmente o acesso da palavra endereçada. Para tornar na maior parte dos acessos à memória a consulta a estas tabelas desnecessária o GE-645 utiliza ainda uma memória associativa em que são guardados apontadores para os blocos mais frequentemente endereçados, tal qual era feito no caso de paginação. Apenas, agora, a chave nesta tabela é uma concatenação do número do segmento e da página deste segmento que o bloco apontado contém. Portanto o "hardware" obtendo um endereço $<s\#>|[\alpha\#]$ procura primeiro na memória associativa o bloco que contém o item de informação endereçado, não sendo necessário recorrer à tabela de página do SDT, ao SDT e à tabela de página de $<s>$ para localizar a palavra endereçada, se o apontador do bloco em questão estiver na memória associativa.

Vejamos agora como o mecanismo descrito pode ser aproveitado para resolver os problemas que motivaram o desenvolvimento da técnica de segmentação, mencionados antes.

Se vários processos compartilham um segmento <s>, <s#> pode ser diferente em cada processo, ié a posição do segmento não é necessariamente igual nas memória virtuais dos diversos processos que o compartilham. Como um segmento pode ser compartilhado apesar disso examinaremos logo adiante. O apontador para a tabela de páginas de <s> está em cada um dos SDT-s dos processos que o compartilham e para os quais <s> já é conhecido. A tabela de páginas porém é única e assim se uma página de <s> é removida da memória principal, apenas uma tabela de páginas deve ser alterada. Um segmento por outro lado pode crescer ou decrescer dinamicamente, sem nenhum problema até o tamanho máximo de um segmento que é entretanto bastante grande. (2^{18} palavras de 36 bits). Normalmente apenas uma parte do segmento está ocupada, mas isto não causa transtorno pois o desperdício é apenas na memória virtual, sendo armazenado, na memória auxiliar apenas a "informação útil" do segmento, sob forma de um arquivo, e na memória principal, graças à paginação, apenas a parte desta informação necessária aos processos que compartilham o segmento.

Falta apenas ver como segmentos podem ser compartilhados e como um segmento se torna conhecido para um processo e o método de ligação entre os diversos segmentos-módulos independentes de um processo.

No MULTICS todos os procedimentos são reentrantes, ié não se modificam e trabalham em áreas de dados diferentes, uma para cada processo que compartilha o segmento do procedimento. Referências internas ao procedimento são feitas pelo mecanismo visto acima, relembrando apenas que quando o procedimento está sendo executado, o PBR contém o número do segmento do procedimento no processo que está sendo executado pelo processador central. Referências a símbolos externos são feitas por um mecanismo essencialmente igual ao visto no Capítulo 3, com a diferença que agora a ligação intermódulos é feita na hora da execução. Assim se o procedimento <A> faz uma referência a um símbolo externo <s> | [α] o compilador não conhecendo o endereço do símbolo externo constrói uma referência indireta à secção de ligação de <A> que contém um apontador para a cadeia de caractéres simbólico <s> | [α] e uma configuração de bits que indica que o endereço está sob forma simbólica. Na hora da execução se a configuração de bits que indica que um endereço está sob forma simbólica é encontrada durante o processo de endereçamento pelo "hardware", é feito automaticamente um desvio ao supervisor. Cada processo tem associado uma tabela de segmentos já conhecidos que contém o nome do segmento conhecido e o número do segmento no processo (esta tabela é análoga à tabela de símbolos públicos já carregados discutida no capítulo 3). O supervisor procura o nome do segmento em questão nesta tabela. Se o segmento já é conhecido deste processo o número do segmento, <s#> é colocado na primeira palavra do par indireto na secção de ligação de <A>. Se o nome <s> não está na tabela, <s> passará agora a ser conhecido do processo. Para isto o supervisor procura a primeira entrada livre do SDT do processo: o índice correspondente será o <s#> para este processo, e <s> e <s#> são colocados na tabela de segmentos conhecidos do processo, e <s#> na primeira palavra do par indireto na secção de ligação de <A>.

Para poder colocar a localização na memória de <s> na entrada correspondente do SDT o MULTICS chama o sistema de arquivos, que verificará se este segmento é conhecido de algum outro processo, obtendo neste caso a localização de <s> na memória, (ié o bloco que contém sua tabela de páginas). Se <s> não é conhecido de nenhum processo ele não está na memória principal e então uma tabela de páginas é criado para ele. Até este ponto portanto, em qualquer caso determinamos <s #> que substitui a primeira palavra do par indireto referenciado na secção de ligação de A. É preciso agora determinar [α #] que corresponde ao símbolo [α] de <s>. [α #] é obtido da tabela de símbolos públicos de <s> que está numa posição prefixada no segmento <s>. Obtido [α #] este substitui o conteúdo da segunda palavra do par indireto. Finalmente os bits da primeira palavra são modificados indicando que o endereço agora não é mais simbólico. Assim todas as referências futuras de <A> ao ítem <s>| [α] serão feitas através do mecanismo normal do endereçamento indireto. Como a secção de ligação de <A> é modificada e <A> é reentrante esta não pode fazer parte do segmento <A>. Assim a secção de ligação de <A> está num outro segmento, sendo necessária uma secção de ligação para cada processo que usa <A>. Podemos pensar na secção de ligação como num segmento separado, embora às vêzes podemos agrupar diversas secções de ligação num só segmento. Se porém a secção de ligação de <A> é um segmento separado de <A> a referência a esta secção precisa obter o número deste segmento no processo. Um dos registradores BPR é reservado para este fim.

Para o controle de desvio de subrotina é empregado uma pilha do modo como foi discutido no capítulo 1. A pilha precisa ser porém externa tanto ao procedimento que chama uma subrotina como à subrotina chamada pois ambas são reentrantes. Assim cada processo tem um segmento reservado para esta pilha e o número deste segmento fica num dos registradores BPR reservado para este fim. O apontador para o início atual da pilha fica no registrador base associado ao BPR em questão. É preciso notar ainda que quando é feita uma transferência de um segmento <A> para um segmento <B> de um processo é necessário mudar o BPR que aponta para a secção de ligação do processo em execução. No MULTICS isto é feito desviando-se para a secção de ligação de <B> através da secção de ligação de <A>. (A instrução de desvio já muda o PBR). Na secção de ligação de <B> o BPR que aponta para a secção de ligação de <A> é salvo e é carregado com o número de segmento da

secção de ligação de <B>, só então desviando para o ponto de entrada apropriada de <B>.

Compartilhamento de qualquer segmento, seja um procedimento ou de dados é assegurado pelo mecanismo de referência a símbolos externos visto acima. Cada processo que compartilha um segmento tem uma cópia separada do seu segmento de ligação. O segmento correto de ligação é utilizado sempre que o segmento compartilhado é executado como parte da execução do processo X, pois o BPR reservado para a secção de ligação apontará para o segmento de ligação associado ao processo X. O sistema operacional se encarrega de carregar os BPR com a informação correta sempre que um processo é iniciado ou reiniciado. Se o segmento compartilhado for de dados e se este contém referências a um segmento externo,uma secção de ligação é criada pelo sistema. Referências internas não criam nenhum problema pois basta armazenar o endereço relativo ao começo do segmento. O início do segmento por sua vez é obtido do SDT e portanto em cada processo que compartilha o segmento de dados este pode ocupar posições diferentes na memória virtual correspondente.

5.4. Processos, processadores virtuais, e sincronização

Nas secções anteriores, definimos informalmente um processo como um programa em execução. Nesta secção examinaremos esta noção mais de perto, definindo-a mais precisamente, indicando como a sua execução é controlada pelo sistema operacional, e quais as relações entre vários processos paralelos.

Denning define um processo em [17] como um par (s,p) onde p é um programa e s é um estado. Um programa pode ser associado a uma seqüência de ações $a_1, a_2, \ldots, a_k \ldots$ e uma seqüência de estados $s_o, s_1, \ldots, s_k \ldots$ tal que para $k \geq 1$ a_k é uma função que depende de s_{k-1}, e $s_k = a_k(s_{k-1})$. (Observe-se a similaridade entre esta definição e a dada na secção 1.3). Um processo (s,p) significa que a seqüência de ações de p $a_1, a_2, \ldots, a_k \ldots$ deve gerar a seqüência $s_o, s_1, \ldots s_k \ldots$ onde $s_o = s$. Nós interpretaremos as ações a_i como as instruções do programa p e os estados s_i como o conteúdo de todas os registradores do processador (inclusive o IAR) e da área de dados do programa p. Em cada instante i a próxima ação a_i a ser executada depende não só de p mas de s_{i-1}. Interromper um processo (s,p) após k passos é equivalente a definir um novo processo (s_k,p).

Imaginemos agora m processos (s_i, p_i) $1 \leq i \leq m$ se processando ao mesmo tempo. Dizemos que estes processos são paralelos. Entre estes processos podemos ter relações de precedência no sentido de que certas ações do processo k devem ser executadas antes de certas ações do processo j. Se não houver entre dois processos nenhuma relação de precedência, dizemos que os dois processos são independentes. A propriedade importante sobre um conjunto de processos paralelos, é que respeitadas as relações de precedência entre eles, o resultado de cada um independe da velocidade de execução dos demais e dele mesmo, ié suspendendo o progresso de um processo temporariamente, não tem nenhum efeito sobre o estado final deste processo

ou de qualquer outro.

Num sistema de computação moderno temos vários processadores capazes de funcionar simultaneamente. Podemos ter vários processadores centrais ligados à mesma memória principal e periféricos, e temos vários processadores de entrada/saída, ié canais. Assim existe a possibilidade de termos num sistema deste tipo efetivamente vários processos progredindo simultaneamente. Por outro lado o número de processos que coexistem num certo instante pode ser maior que o número de processadores de que dispomos e assim teremos de temporariamente suspender alguns processos, para poder dar sequência a outros, e por isso a propriedade mencionada no parágrafo anterior é extremamente importante, visto que mesmo respeitando as relações de precedência entre os processos em execução, há várias sequências de ações que um processador pode executar. Por exemplo se tivermos um processador executando três processos, P_1 com as ações a_1b_1 , P_2 com as ações $a_2b_2c_2$ e P_3 com as ações a_3b_3, sendo que todas as ações de P_1 e de P_2 devem preceder as de P_3, as seguintes sequências de ações são todas possíveis: $a_1b_1a_2b_2c_2a_3b_3$; $a_1a_2b_1b_2c_2a_3b_3$; $a_1a_2b_2b_1c_2a_3b_3$; $a_1a_2b_2c_2b_1a_3b_3$; $a_2a_1b_1b_2c_2a_3b_3$; $a_2a_1b_2b_1c_2a_3b_3$; $a_2a_1b_2c_2b_1a_3b_3$; etc..

Tendo em vista as considerações acima é natural pensar no próprio sistema operacional como um conjunto de processos paralelos, que progridem com velocidades indefinidas, e que são sujeitas a certas relações de precedência. Esta maneira de encarar o sistema foi introduzida em [15]. A cada programa do usuário iniciado corresponde um processo; a cada arquivo de entrada corresponde um processo cuja função é ler registros físicos do arquivo enchendo os "buffers" reservados ao arquivo; a cada arquivo de saída, um processo cuja função é escrever registros físicos do arquivo esvaziando os "buffers" reservados ao arquivo, etc..

Como podemos ter mais processos do que processadores físicos para processá-los é necessário salvar o estado de um processo quando este é interrompido, para que mais tarde a execução possa ser reiniciada do mesmo ponto em que foi interrompido. O sistema operacional mantém para este fim uma tabela para cada processo, denominado de vetor de estado, que pode ser imaginado como os registradores de um processador virtual que está executando o processo. O vetor de estado consiste de todos os registradores do processador, como o IAR, acumulador, registradores usados para o mapeamento de endereços, etc,, indicadores do processador como o indicador de sobrecarga, eventuais indicadores do estado de dispositivos de entrada, e informação sobre o espaço de endereçamento do processo, e outras informações, como porque um processo não pode continuar por exemplo, quando este é o caso.

Vejamos agora como podemos sincronizar, ié assegurar que as relações de precedência sejam respeitadas, num conjunto de processos paralelos. Um mecanismo geral de sincronização foi definido por Dijkstra, [15]. Este mecanismo se baseia no conceito denominado de semáforo.

Um semáforo s é uma variável inteira, com um valor inicial não negativo, designado a s quando ele é criado. Após s ser inicializado os processos paralelos que tem acesso a s podem fazê-lo apenas através de

duas operações denominadas de primitivas de sincronização. Estas operações são a operação espere (s) e sinalize (s), (originalmente denominadas por Dijkstra de operação P e operação V; a nomenclatura espere e sinalize são usadas por Habermann em [19]). Um processo P executa uma operação espere num semáforo s seguindo o seguinte algoritmo:

Algoritmo E. : espere(s)

E1. $s \leftarrow s-1$

E2. se $s \geq 0$ E. termina (e P pode continuar a sua execução);
senão (ié se $s < 0$) P é colocado numa fila de espera associado ao semáforo s e seu progresso é blocado, (ié P entra em estado de espera), e E. termina.

Um processo Q executa uma operação sinalize (s) num semáforo s seguindo o algoritmo:

Algoritmo S. : sinalize(s)

S1. $s \leftarrow s+1$

S2. Se $s > 0$ E termina; senão (ié se $s \leq 0$) um dos processos na fila de espera associado a s é retirado da fila, seu progresso é novamente permitido, tirando-o do estado de espera, e E termina.

As operações espere(s) e sinalize(s) são indivisíveis. Isto significa que se dois processos tentarem executar estas duas operações simultaneamente, a execução de uma das operações será retardada até o término da outra.

As seguintes observações se aplicam ao mecanismo acima descrito:

A) se $s < 0$ há $-s$ processos na fila de espera associado a s
se $s \geq 0$ não há nenhum processo na fila de espera associado a s ;
ié o número de processos na fila de espera de s é dado por
$\max\{-s ; 0\}$

B) se após uma operação sinalize(s) $s < 0$ pela propriedade (A) havia mais de um processo na sua fila de espera. Como resultado de sinalize(s) um destes processos pode agora continuar a sua execução; qual o processo que é removido da fila não é relevante do ponto de vista da lógica dos processos paralelos.

Examinemos agora como estas primitivas podem ser utilizadas para realizar certos tipos de sincronização.

Um tipo de sincronização comum é a exclusão mútua. Imaginemos dois processos paralelos P e Q que contém uma secção "crítica" cada um, e suponhamos que é necessário que enquanto uma secção "crítica" é executada a outra não pode ser executada, ié que as duas secções "críticas" são mutuamente exclusivas. Este problema aparece sempre que vários processos

utilizam e (modificam) uma mesma tabela de variáveis de estado. As secções que fazem acesso às variáveis de estado são "críticas" no sentido acima, pois se ambas pudessem ter acesso simultâneo o processo P poderia por exemplo consultar uma variável de estado e tomar uma decisão enquanto Q modifica a mesma variável de estado. Um exemplo deste tipo foi visto na secção 4.9.. Em outras palavras, as variáveis de estado são compartilhadas pelos dois processos, e se ambos puderem ter acesso simultâneo à tabela, o estado final desta pode ser bastante diferente do que o desejado. Exclusão mútua pode ser conseguida seguindo o esquema:

É definido um semáforo mutex com valor inicial 1, antes de iniciar P ou Q.

	Processo P		Processo Q
P1.	parte A do processo P	Q1.	parte A do processo Q
P2.	espere (mutex) secção crítica de P sinalize (mutex)	Q2.	espere (mutex) secção crítica de Q sinalize (mutex)
P3.	parte B do processo P	Q3.	parte B do processo Q

O mesmo esquema pode ser generalizado para n processos.

Um outro exemplo de sincronização é a requerida no problema denominado de problema do produtor/consumidor, que é uma abstração do processo de uso de "buffers" na entrada ou saída de um processo. Trata-se da sincronização de dois processos cíclicos paralelos P e Q que compartilham n "buffers". O processo P, ou o produtor, deposita um item de informação num "buffer" vazio para uso posterior por Q; Q "consome" um "buffer" cheio removendo o item de informação. Os "buffers" são usados ciclicamente pelos dois processos. A sincronização neste caso é necessária, pois P deve ser impedido de depositar um novo item num "buffer" cheio; e Q deve ser impedido de retirar um item de um "buffer" vazio, ou sendo enchido por P. A sincronização destes dois processos pode ser conseguida pelo esquema:

São usados dois semáforos cheio e vazio com os valores iniciais 0 e n respectivamente, antes de iniciar P ou Q.

	Processo P (valor inicial de k=1)		Processo Q (valor inicial de i = 1)
P 1.	espere (vazio) deposite item no buffer [k] sinalize (cheio)	Q1.	espere (cheio) retire item do buffer [i] sinalize (vazio)
P 2.	k ← (k + 1) mod n ; resto do processo P, vá a P1.	Q2.	i ← (i + 1) mod n resto do processo Q vá a Q1.

É fácil provar que o esquema sincroniza P e Q conforme é requerido.

Um outro esquema geral de sincronismo pode ser obtido com o emprego de semáforos denominados de semáforos privativos. Um semáforo privativo de um processo P é um semáforo s tal que apenas P pode executar uma operação *espere*(s) neste semáforo. Um semáforo privativo pode ser utilizado para controlar o progresso de P sempre que este progresso dependa dos valores atuais de certas variáveis de estado. Se as variáveis de estado indicarem que o progresso de P não pode ser permitido, uma operação *espere*(s) após a inspeção das variáveis de estado paralizará o processo. Outros processos que alterem as variáveis de estado, devem executar uma operação *sinalize*(s), no semáforo s quando a barreira para o progresso de P for removida. Um exemplo em que este tipo de sincronismo é necessário é o caso em que um processo P necessita de um certo recurso do sistema como por exemplo uma unidade periférica. Se não houver nenhuma unidade disponível o processo é paralizado. Quando eventualmente uma unidade ficar disponível, o processo é retirado do estado de espera executando uma operação *sinalize*(s) no semáforo privativo correspondente. O esquema geral desenvolvido por Dijkstra para este tipo de sincronismo é:

São usados dois semáforos mutex e privativo inicializados com o valor 1 e 0 respectivamente. Como variáveis de estado são consultados e modificados, mutex é usado para conseguirmos exclusão mútua das secções críticas dos processos que compartilham as variáveis de estado, conforme vimos acima; privativo é um semáforo privativo de P. Q é o processo que eventualmente remove a barreira para o progresso de P.

Processo P

P1. *espere* (mutex)

P2. (secção crítica de P) que inspeciona e modifica as variáveis de estado. Se o processo pode continuar executa *sinalize* (privativo); senão

P3. *sinalize* (mutex)

P4. *espere* (privativo)

P5. restante do processo P

Processo Q

Q1. *espere* (mutex)

Q2. (secção crítica de Q) inspeciona e modifica as variáveis de estado se conforme esta inspeção há necessidade de sinalizar algum semáforo privativo de algum processo, (eventualmente mais de um) executa *sinalize* (privativo) correspondente.

Q3. *sinalize* (mutex)

Q4. restante do processo Q.

São várias as vantagens de um mecanismo de sincronização uniforme, como o visto acima. Em primeiro lugar o comportamento de processos paralelos pode ser estudado, de maneira precisa demonstrando-se que esquemas de sincronização gerais como os esquemas vistos nesta secção funcionam de acordo com as especificações. Uma outra consequência favorável é o fato de que como as velocidades de execução não interferem no funcionamento lógico de um conjunto de processos paralelos devidamente sincronizados, o número de processadores físicos de que se dispõe para executá-los é irrelevante do ponto de vista lógico. Se tivermos um número menor de processado-

res o conjunto de processos paralelos será executado corretamente do mesmo modo como se tivessemos mais processadores físicos, tudo se passando como neste último caso, com a única diferença de que parecerá que temos processadores mais lentos. Finalmente um mecanismo geral de sincronização incorporado ao sistema operacional permite que um processo do usuário possa durante a sua execução criar, por meio de chamadas apropriadas ao sistema operacional, outros processos que serão executados paralelamente com o primeiro, devidamente sincronizados entre si.

5.5. Alocação de recursos

Um dos problemas que um sistema operacional precisa resolver é o de reservar as diversas componentes do sistema computacional disponível às tarefas sob controle do sistema, nos momentos apropriados, ié distribuir o serviço que precisa ser feito de uma maneira "racional", de modo a melhor satisfazer certos objetivos.

No capítulo 4, a rotina de alocação de tempo do canal, discutida brevemente, é um exemplo de um problema deste tipo. Naquele problema o SCES colocava em filas de espera pela unidade de E/S pedidos de operações de E/S; quando um canal ficava desocupado era necessário decidir qual dos pedidos devia ser atendido em seguida pelo canal. Problemas semelhantes ocorrem na distribuição de trabalho para a memória, processador central, etc.. Algoritmos que resolvem estes problemas são referidos em geral por algoritmos de alocação de recursos ou de alocação de tempo ("scheduling"). Estes algoritmos são necessários no sistema operacional não só para aliviar o programador da tarefa de alocação, como de conseguir atingir as metas da instalação proporcionando utilização mais eficiente da mesma e serviço de qualidade mais aceitável ao usuário.

Uma maneira de pesnar, útil do ponto de vista de alocação de recursos, é encarar um processo como uma entidade que necessita dinamicamente durante a sua execução de "recursos". Um recurso é uma entidade do sistema, seja de "hardware" ou "software" que pode ser alocada ao processo. Assim um bloco de memória, uma unidade central de processamento, uma trilha de disco, um canal, uma unidade de fita magnética, etc... são recursos de "hardware"; um processador virtual, uma página da memória virtual, uma entrada no diretório do disco, uma entrada numa tabela do sistema são recursos de "software". Recursos do "hardware" e "software" são limitados e é por isso que são necessários algoritmos de alocação de recursos. Recursos do "hardware" são limitados porque só temos um certo número de processadores centrais, um número limitado de blocos na memória principal, de canais, de unidades de E/S, etc.. Recursos do "software" também podem ser limitados, porque o número de elementos de uma certa tabela pode ser restrita a um certo máximo, por exemplo.

A alocação de um recurso de um certo tipo pode não ser independente da alocação de um outro tipo de recurso. Por exemplo a alocação do processador central depende da alocação da memória: suponhamos por exemplo que um certo processo P está sendo executado pelo processador central

num sistema paginado; evidentemente parte do programa do processo P e sua área de dados devem estar na memória principal; se uma página que não está na memória for referenciada, P não pode continuar enquanto a referida página não for para a memória; a rotina de alocação da memória iniciará a carga da página na memória eventualmente desalojando uma outra página; como a transferência de uma página da memória auxiliar para a memória principal leva um tempo relativamente longo, podemos aproveitar este tempo para processar algum outro processo Q ; o processador central é alocado então ao processo Q; se porém Q também não tiver uma página na memória, que será logo mais referenciada por ele pouco depois que Q for iniciado ele também será interrompido, sendo necessário recomeçar um outro processo R com o qual poderá acontecer o mesmo fenômeno. O que esta descrição sugere é que se um processo não tiver alocada uma certa quantidade crítica de memória não poderá usar eficientemente o processador central. A política de alocação de um recurso pode portanto afetar a alocação de outro, motivo pelo qual o problema da alocação de um certo recurso não deve ser encarada isoladamente mas como um aspecto de um só problema de grande dimensão que é a alocação dos recursos do sistema. Costuma-se referir a este problema como o de garantir o uso balanceado dos recursos do sistema. Na prática é comum o balanceamento de apenas a memória e o processador central. Discutiremos no restante desta secção algumas das técnicas de alocação de processadores e memória e como eventualmente se pode balancear o sistema em relação a estes dois recursos.

Examinemos inicialmente os aspectos da alocação de processadores físicos supondo que a memória disponível no sistema é ilimitada. Esta maneira de analisar o problema é a de Saltzer, [20].

Se não houvesse limitações nem de memória nem do número de processadores físicos, então todos os processo cuja execução é solicitada ao sistema poderiam ser carregados na memória e cada um poderia ser executado num processador físico. Como os processos precisam ser eventualmente sincronizados, certos processos em determinados instantes necessitarão esperar por outros. Podemos imaginar diversos métodos pelos quais a execução de um certo processo seja suspensa pelo tempo necessário, quando seu progresso não é logicamente permitido. Por exemplo o processo poderia consultar continuamente uma certa variável, que quando devidamente alterada por um outro processo, reiniciaria a execução do primeiro. O problema com este método é que consultar uma variável exige consultas à memória o que causa atrasos no atendimento de acessos à memória de outros processos. Uma solução mais conveniente seria "parar" fisicamente o processador, e permitir que um outro processador mande um sinal que reiniciaria a execução. Como porém o número de processadores na prática não é ilimitado, este método também não é conveniente pois o processo durante o tempo de espera reserva para si o processador, impedindo-o de produzir algo útil. Assim, com número limitado de processadores, a melhor solução é cessar a execução do processo temporariamente, (bloquear o processo), guardar na memória em tabelas apropriadas o estado do processo, ié do "processador virtual", e liberar o processador para a execução de algum outro processo. Quando o progresso do processo for novamente permitido, este fato será devidamente sinalizado, e o alocador de processadores assim que tiver um processador disponível, escolherá, por um mecanismo adequado, um dos processos que podem continuar

a sua execução. O mecanismo de sinalização entre processos poderá ser a utilização de semáforos conforme foi visto na secção anterior. No primeiro método todos os processos estão sempre correndo ("running"); no segundo método um processo pode estar correndo, ou bloqueado (quando o processador pára), e no terceiro caso, que é a solução adequada no nosso caso, o processo pode estar correndo (quando seu progresso está permitido e um processador físico o está executando), bloqueado (quando seu progresso não é permitido), ou pronto-para-a-execução, ou simplesmente pronto ("ready"), (quando seu progresso é permitido mas nenhum processador o está executando). Associamos portanto a um processo 3 estados: correndo, bloqueado e pronto. Vejamos agora quais as mudanças entre estes estados que são permitidas.

Um processo quando é iniciado, antes de entrar no estado "correndo", precisa ser carregado na memória, (que como já dissemos é suposta ilimitada nesta discussão). Depois de carregado o processo está "pronto", mas é possível que não haja nenhum processador livre neste momento, e não seja conveniente paralizar nenhum outro processo para iniciar a execução deste. Assim um processo P, sempre que é iniciado, entra em primeiro lugar no estado "pronto". Eventualmente, mais tarde, um processador fica livre e este processo é escolhido pelo algoritmo alocador de processadores, e sua execução é iniciada entrando no estado "correndo". Como consequência de uma operação <u>espere</u>(<u>s</u>) em algum semáforo <u>s</u> o processo pode ser impedido de prosseguir entrando no estado "bloqueado". (Entrando neste estado, o processo libera o processador e o algoritmo alocador irá escolher algum outro processo "pronto", reiniciando sua execução). Em consequência de uma operação <u>sinalize</u>(<u>s</u>) executada por algum outro processador o progresso de P pode ser novamente permitido e neste caso o processo entrará novamente no estado "pronto". Um processo nunca pode passar diretamente do estado "pronto" para "bloqueado", pois para ser "bloqueado" precisa antes prosseguir na sua execução, ié passar pelo estado "correndo". Do mesmo modo não pode passar diretamente do estado "bloqueado" para o estado "correndo" sem passar pelo estado "pronto" porque eventualmente não tenhamos nenhum processador para executá-lo quando o processo deixa o primeiro estado. (Se não for este o caso, ié tivermos um processador livre, o alocador de processos sempre pode escolher P para reiniciar a sua execução). Resta discutir uma mudança: do estado "correndo" para o de "pronto".

Pode acontecer que um processo num certo ponto da sua execução não se "bloqueie" mais, por exemplo ou entrando numa malha infinita por erro de programação, e assim nunca termine, ou então não ser mais bloqueado até o seu término, que pode ocorrer um longo tempo depois. Em geral, deixar um processo correr um "longo" tempo não é conveniente, não só porque o processo pode estar no primeiro caso e assim poderá passar do tempo permitido ao processo pelo usuário, como poderá impedir que outros processos sejam atendidos satisfatoriamente. Evidentemente podemos conseguir que tal não aconteça através da utilização do relógio interno que já foi mencionado antes. Assim quando um processo entra no estado "correndo" não recebe o comando do processador por tempo indefinido mas recebe uma "fatia de tempo" ("time slice") do processador através da colocação de uma certa quantia, que pode ser eventualmente calculada dinamicamente para cada processo pelo alocador do processador, no registrador do relógio interno. Se o processo não se bloquear antes, decorrido este tempo será interrompido pela "interrupção"

gerada pelo relógio interno. Uma outra eventualidade que precisa ser considerada é um processo ser incorporado aos processos "prontos" e cuja execução é mais importante, conforme a política de alocação de processadores, que algum dos processos que estão correndo. Em ambos os casos, fatia de tempo esgotada e processo "pronto" mais importante do que algum dos processos sendo executado, é conveniente que um processo seja retirado do estado "correndo" antes de ser bloqueado. Este processo poderia continuar o seu progresso mas vai ter sua execução suspensa, ié temos uma transição do estado "correndo" para o estado "pronto". Caso o processo tenha sido suspenso por ter tido sua fatia de tempo esgotada, o alocador de processadores poderá chegar à conclusão que este mesmo processo deve receber nova fatia de tempo. Neste caso o tempo gasto com o alocador de processadores terá sido inútil, mas este é o único meio de evitar os inconvenientes anteriormente citados.

Neste ponto vamos discutir a maneira pela qual podemos passar um processo escolhido para ser executado num certo processador, para o estado "correndo". Evidentemente precisamos carregar os registradores do processador com os conteúdos dos respectivos "registradores" no processador virtual deste processo, que estão guardados no vetor de estado deste processo. (Antes disso evidentemente já salvamos estes registradores, caso estes representem o estado de um outro processo que acabou de deixar o controle do processador). Se o procedimento do sistema operacional que deve fazer isto está sendo executado num outro processador isto não causa maiores problemas, sendo necessário porém que o "hardware" permita que nesta circunstância um processador modifique os registradores de outro. Se porém este procedimento for executado neste mesmo processador, (que é certamente o caso, muito comum, em que só temos um processador central na instalação, por exemplo), cuidados especiais precisam ser tomados. O problema é que o espaço de endereçamento do processador precisa ser mudado, enquanto o procedimento do sistema operacional é executado, que por sua vez tem o seu próprio espaço de endereçamento. Saltzer sugere 3 soluções para este problema, [20]:

1. Ter no processador uma única instrução, que salva todos os registradores do processador, e carrega todos os registradores a partir de um vetor de estado;

2. Simular uma tal instrução complexa por programa, tendo uma opção no "hardware" que permita que durante a execução deste programa os endereços sejam considerados endereços físicos absolutos, ao invés de referências à memória virtual como é normalmente;

3. Tomar os cuidados necessários para que o procedimento, que deverá ser um programa reentrante, seja compartilhado por todos os processos, e esteja na mesma posição nas memórias virtuais de todos os processos.

Vimos portanto que algumas das tarefas do alocador de processadores são: escolher um dos processos que estão no estado "pronto" e eventualmente um dos processadores, que pode ser o mesmo em que o alocador está sendo executado; se for este o caso passar o processo respectivo para o estado "pronto"; passar o processo "pronto" escolhido para o estado "correndo" colocando-o no comando do processador em questão. Para o alocador po

der escolher por algum esquema o processo que recomeçará seu processamento, é necessário termos uma lista dos processos no estado "pronto". Esta lista tem normalmente estrutura ligada.

Existem diversas estratégias comuns para o alocador escolher o próximo processo a ser executado da lista dos processos no estado "pronto". O objetivo de qualquer estratégia é otimizar certas metas desejáveis do sistema. Às vêzes estas metas podem ser enunciadas numa forma matematicamente precisa caso em que modelos matemáticos podem ser formulados cuja análise eventualmente indicará uma política ótima. É importante observar porém que qualquer que seja a estratégia escolhida ela terá de ser suficientemente simples para que o algoritmo alocador não necessite de tempos consideráveis para escolher o próximo processo a ser executado.

A técnica mais simples de alocação é a política "o primeiro que chega é o primeiro a ser servido". Neste caso basta que a lista de processos "prontos" seja uma fila. Cada processo "pronto" que chega é colocado no fim da fila e o alocador simplesmente retira o primeiro da fila. Um esquema pouco mais elaborado é o denominado "de prioridade fixa". Neste caso cada processo recebe um número de prioridade fixa. A lista de processos "prontos" neste caso pode ser separada em várias filas, cada um com processos de mesma prioridade. O alocador escolhe simplesmente o primeiro da fila de processos de prioridade mais alta. Na terceira estratégia denominada de "prioridade calculada", cada processo recebe um número de prioridade quando chega à lista de processos "prontos", calculado através de algum algoritmo, que pode levar diversos fatores em conta, como o tempo de processamento deste processo, a memória necessária, a quantidade de outros recursos utilizada, o tempo de espera de outros processos na lista de processos "prontos" etc.. Esta técnica pode permitir que a prioridade de cada processo seja calculada por um algoritmo diferente. O valor de outra variável importante que o alocador precisa obter para um processo é a sua fatia de tempo. Obviamente o esquema mais simples é termos uma fatia de tempo constante. A fatia de tempo porém pode ser calculada também por algum algoritmo, permitindo uma variação dentro de certos limites, e pode, de fato, ser uma das variáveis que determinam a prioridade do processo.

Os três esquemas vistos calculam a posição de um processo na lista de processos "prontos" na hora em que este é colocado nesta lista. Uma outra possibilidade seria escolher a prioridade do processo na hora de retirar um processo da lista. Em geral isto não é feito pois provavelmente envolveria um tempo para o alocador proporcional ao número de processos na lista.

O alocador deve ser chamado toda vez que um processo passa para o estado "pronto", pois neste caso um processo precisa ser colocado na lista de processos "prontos", no lugar apropriado; que um processo passa de "correndo" para "bloqueado", pois neste caso um processador ficou "livre". Estas mudanças em geral acontecem como resultado do tratamento de alguma interrupção, razão pela qual as interrupções são dados de entrada importantes para o alocador.

Antes de passarmos a examinar a questão da alocação de processadores com memória limitada, vamos considerar o problema do ponto

morto ("deadlock") que pode ocorrer quando temos um conjunto de processos paralelos. Diz-se que dois (ou mais) processos entram num ponto morto, ou que ocorre um ponto morto, se ambos os processos são paralizados, cada um esperando um evento a ser produzido pelo outro. Esta espera circular paraliza os dois processos, A e B, definitivamente pois o processo A não pode progredir antes de um sinal a ser produzido por B e B não produzirá este sinal pois está paralizado esperando por um sinal de A. Um ponto morto é portanto uma situação indesejável, que pode ocorrer quando se tem um sistema de processos paralelos. Um ponto morto pode surgir devido a duas causas:

1 - Como os processos se intercomunicam, o ponto morto pode ser consequência de um erro lógico na concepção dos processos; um mecanismo de sincronização geral como o visto na secção anterior pode facilitar a prova de que um tal ponto morto não ocorrerá num determinado sistema de processos.

2 - Como o mesmo conjunto de recursos é compartilhado pelos processos podemos ter o caso em que temos dois (ou mais) processos, sendo executados paralelamente, enenhum processo exija individualmente mais recursos que o disponível no sistema mas em conjunto exijam. Quando o tipo de recurso com que isto acontece for intercambiável entre os processos, ié pode ser alocado e dealocado ao processo antes que este termine de utilizá-lo, tais como processadores, memória, etc., não haverá problema de ponto morto. Se porém isto ocorre com um recurso que uma vez alocado a um processo só pode ser "dealocado" depois de liberado pelo processo, então pode ocorrer um ponto morto. Por exemplo se tivermos na instalação duas leitoras de cartões e um processo A utilizar dois arquivos em cartão, simultaneamente durante o decorrer do processo, o primeiro arquivo no intervalo de tempo $I_{1A} = [t_{1A}, t'_{1A}]$, o segundo no intervalo de tempo $I_{2A} = [t_{2A}, t'_{2A}]$, e $I_{1A} \cap I_{2A} \neq \emptyset$ (entenda-se que A abre o arquivo 1 após decorrido o tempo <u>t_{1A} de processamento de A</u>, e fecha após decorrido t'_{1A} etc.), e um processo B com $I_{1B} = [t_{1B}, t'_{1B}]$ e $I_{2B} = [t_{2B}, t'_{2B}]$, $I_{1B} \cap I_{2B} \neq \emptyset$, sendo $t_{1A} \leq t_{2A}$ e $t_{1B} \leq t_{2B}$ então <u>conforme</u> o <u>progresso conjunto</u> de A e B <u>pode</u> ocorrer um ponto morto. Até que A ou B abram o primeiro arquivo, os dois processos podem progredir sem problema; se porém um abrir um arquivo o outro não <u>deve</u> abrir o seu primeiro arquivo até que o outro abra também o segundo pois se o fizer os dois poderão progredir por mais algum tempo, quando certamente ocorrerá um ponto morto; depois que um abrir os dois arquivos não poderá mais ocorrer ponto morto embora um dos processos possa ser bloqueado temporariamente. A questão suscitada pelo problema do ponto morto é, em síntese, se é possível construir um alocador de recursos prático que evite pontos-morto do segundo tipo. O exame desta questão é assunto de pesquisa muito recente. Já foram desenvolvidos algoritmos que detectam se o estado de um certo conjunto de processos contém um ponto-morto, e se um estado é "seguro", no sentido de que não leva necessariamente a um ponto morto, [17], [23], [24], [25].

Até agora foi suposto nesta secção que a memória é ilimitada no nosso sistema. Este não é evidentemente o caso real e portanto examinaremos agora o que acontece se esta hipótese é abandonada. Se a memória é limitada vários problemas precisam ser resolvidos. Alguns destes problemas foram discutidos em conexão com métodos de relocação dinâmica na sec

ção 5.3. Em particular tratamos naquela secção dos problemas de compartilhamento de informações na memória por vários processos, e ficou evidenciada a necessidade de termos tabelas para controlar o espaço livre na memória principal e auxiliar. Como consequência dos métodos de relocação dinâmica vistos na secção 5.3., tornou-se possível manter apenas parte de um processo na memória e movê-lo, ou mover partes dele, quando conveniente entre a memória auxiliar e a memória física ("swapping"). "Swapping" por sua vez suscita problemas de alocação de recursos, que discutiremos agora. A questão fundamental, que precisa ser resolvida, é evidentemente o sistema alocador decidir que partes de que processos devem ficar na memória principal. Já foi apontado que este problema está relacionado com o problema de alocação de processadores, e portanto é preciso também examinar como a alocação destes dois recursos ficará interligada no sistema.

Duas estratégias extremas podem ser consideradas na alocação de qualquer recurso a um processo. Uma é alocar antes de iniciar a execução do processo todos os recursos de que irá necessitar, durante a sua execução. A outra é alocar os recursos apenas quando estes se tornam necessários. Em particular, em conexão com a alocação de memória estas duas estratégias significam, carregar todo o processo na memória cada vez que ele vai ser iniciado ou reiniciado, ou carregar na memória apenas as partes necessárias de cada vez, que num sistema paginado implica em carregar uma página apenas quando é feito um acesso a ela. Esta última estratégia é denominada de paginação sob demanda. Foram feitos alguns estudos estatísticos concernentes à maneira pela qual um processo faz acesso às suas páginas. Estes estudos parecem indicar que um processo quando inicia sua execução recebendo uma fatia de tempo faz acesso muito rapidamente a um certo número de suas páginas, e depois o conjunto de páginas endereçadas varia lentamente. Se portanto usarmos paginação sob demanda, é bem possível que o sistema de processamento obtido seja muito ineficiente executando um grande número de transferências de informação entre a memória auxiliar e a memória principal, e fazendo pouco processamento realmente útil. Algumas estratégias de alocação de memória tem sido sugeridas tendo em vista resolver estes problemas. Todas estas estratégias tem por objetivo remover da memória as páginas que serão endereçadas com menor probabilidade no futuro imediato, sendo usadas em conjunção com uma política de paginação sob demanda.

A estratégia mais simples, e de implementação mais fácil é remover as páginas da memória, quando necessário, na mesma ordem em que as páginas foram trazidas para a memória, ié a primeira página removida será a página que foi carregada em primeiro lugar. Esta estratégia é de fácil implementação, pois basta manter uma tabela das páginas carregadas numa fila.

Uma estratégia mais elaborada é remover a página menos recentemente usada, que parece uma boa estratégia admitindo que a página me-

nos recentemente usada é também a que tem menor probabilidade de ser endereçada em seguida. Para implementar esta estratégia associa-se a cada bloco da memória um contador denominado de registrador de idade. Quando uma página é endereçada, o registrador de idade correspondente é carregado com um certo valor. Todos os registradores de idade são periodicamente decrementados. Num certo instante portanto o conteúdo de um registrador de idade indica quanto tempo faz que a página correspondente não foi referenciada: quanto menor este conteúdo maior é o tempo decorrido entre o instante em que a página foi endereçada e o instante considerado. Assim a página menos recentemente usada é a que tem o menor valor no registrador de idade. Na prática, o tamanho de cada registrador de idade sendo finito, este pode pode indicar o tempo decorrido entre a última referência e o instante considerado apenas até um certo máximo. Depois de decorrido este máximo o registrador de idade conterá zero , e portanto todas as páginas com o contador zero são consideradas de "mesma idade" e são as primeiras candidatas a serem removidas. Há diversas idéias para implementar registradores-idade. Por exemplo cada vez que uma página é referenciada um capacitor pode ser carregado, que depois descarrega lentamente ligando um bit quando ficar descarregado. Outra idéia é ter um registrador-idade associado a cada bloco com o seguinte funcionamento: quando a página correspondente é endereçada o bit mais à esquerda é carregado com 1; periodicamente todos os registradores-idade são deslocados para a direita de 1 bit, entrando 0 no bit mais à esquerda. Assim a posição do primeiro bit 1 no registrador indica o tempo decorrido desde a última referência à página. É necessário tomar precauções entretanto para que o tempo gasto no manejamento dos registradores-idade não seja uma carga significativa para o sistema. [1]

A terceira estratégia foi desenvolvida por Denning, [18]. Analisando as estratégias anteriores, pode-se chegar à conclusão de que um de seus inconvenientes é que as páginas de todos os processos são candidatas a remoção. Quando um certo processo é executado as páginas dos processos que não estão correndo evidentemente não são endereçadas (exceto talvez as páginas compartilhadas) e portanto vão sendo removidas da memória; quando a fatia de tempo do processo que está correndo **esgotar-se pode acontecer** que qualquer dos processos que receber a nova fatia de tempo bloqueie-se quase imediatamente devido a falta de algumas de suas páginas que serão logo endereçadas. Denning baseia a sua estratégia no conceito de conjunto de trabalho de um processo ("working set"). O conjunto de trabalho de um processo pode ser visto como a menor parcela do processo que precisa estar na memória para processamento eficiente do processo. Formalmente define-se o conjunto de trabalho (num sistema paginado), $W(t, T)$, ao tempo t de um processo, onde T é denominado de parâmetro do conjunto de trabalho, como o conjunto

de páginas distintas referenciadas pelo processo no intervalo de tempo [t - T, t] ; o número de páginas em W(t, T) é por definição o tamanho de W(t, T) e é indicado por w(t, T). As primeiras candidatas à remoção na estratégia do conjunto de trabalho no instante t são as páginas do processo que deixaram de pertencer a W(t, T). W(t, T) pode ser determinado com o auxílio de registradores-idade carregando a constante T no registrador-idade quando uma página é endereçada. Assim todas as páginas que tiverem o registrador-idade contendo zero foram referenciadas há um tempo maior que T e portanto não pertencem a W(t, T). Do raciocínio acima conclui-se que a estratégia do conjunto do trabalho coincide com a estratégia da página menos recentemente usada, se considerarmos candidatas à remoção apenas as páginas do (ou dos) processo(s) sendo executado(s). Várias propriedades interessantes do modelo são expostas e demonstradas sob certas hipóteses em [18] e [21] .

Uma das vantagens do modelo proposto por Denning é a possibilidade de se relacionar a alocação de processadores e de memória. Ele sugere que um processo não deve ser carregado na memória a não ser que se tenha espaço para o seu conjunto de trabalho. Poder-se-ia também imaginar que o alocador de processadores não comece a execução de nenhum processo antes que o respectivo conjunto de trabalho esteja na memória física. Denning porém argumenta que isto pode ser fetivamente contraproducente, pois o conjunto de trabalho de um processo pode ser radicalmente mudado depois que ele é bloqueado.

5.6. Proteção

Num sistema multiprogramado vários níveis de proteção são em geral necessários. É necessário prover a proteção do sistema do usuário, ié é necessário evitar que o usuário possa paralizar o sistema ou destruir informação essencial ao sistema; É necessário proteger um usuário de outro, ié é preciso que um usuário não possa interferir no processo de um outro usuário de uma maneira não desejada por êste; É necessário proteger um processo do usuário de um outro processo do usuário, ié o usuário pode querer que certos processos seus tenham direitos de acesso restritos, a um outro processo seu.

Finalmente é necessário proteger o sistema de si mesmo, restringindo o dano que um subsistema deficiente possa causar ao resto do sistema.

A proteção acima exige uma variedade de mecanismos tanto no "harware" como no "software". No "hardware" costuma-se prover dois tipos de proteção:

a) proteção de memória
b) proteção contra a execução de certas instruções

Vários mecanismos de proteção de memória foram implementados. Normalmente isto consiste em classificar áreas de memória em algumas classes tais como:
1 - inacessível
2 - lê sòmente ("Read-only")
3 - execute-sòmente ("Execute-only")
4 - lê e escreve

Certas áreas precisam ser inacessíveis à outras áreas. Por exemplo um usuário não deve ter eventualmente nenhum acesso à área de um outro usuário, que contenha informação confidencial. Às vêzes é necessário que uma área possa ser lida mas não modificada. Outras vêzes é necessário que uma área só possa ser executada. Por exemplo um programa de um usuário que alugue a execução do mesmo a um outro usuário, não deve poder ser lido ou modificado pelo último afim de garantir o direito "autoral" do primeiro usuário. Finalmente uma área pode ser lida e escrita, ié ter garantido pleno acesso.

Proteção pode ser garantida à memória física ou à memória lógica. Uma maneira comum de proteção da memória física é ter uma "chave" de proteção associada a cada bloco de memória física, especificando o tipo de acesso ao bloco. Êste método porém não é conveniente pois todos os processos tem o mesmo tipo de acesso a um bloco, portanto para que o mecanismo funcione adequadamente será necessário mudar as chaves cada vez que o processador muda de processo.

Proteção à memória lógica é mais interessante, pois neste caso não teremos o problema mencionado acima. O método mais simples de proteção à memória lógica é o já mencionado na secção 5.3, que se consegue com o uso de registradores base e limite. Conforme porém Graham [22] observa, êste mecanismo tem o defeito de ser do tipo tudo ou nada, ié um processo ou tem acesso irrestrito a uma área ou não tem nenhum acesso. A pro

teção mais adequada pode ser conseguida num sistema paginado ou segmentado. Associando no mapa da memória lógica uma chave descrevendo o tipo de acesso, vários processos podem ter acessos de vários tipos a várias áreas da sua memória lógica. Assim por exemplo, uma página compartilhada num sistema paginado pode ser "lê-sòmente" num processo mas "lê e escreve" em outro. Num sistema segmentado a proteção é garantida ao nível de segmento. No MULTICS por exemplo no SDT ("segment descriptor table") é guardado para cada segmento uma chave que pode ser escreve-apenas, lê-apenas, lê-escreve, executa-sòmente, inacessível ou acesso sòmente para processos do sistema. Um problema com êste esquema é que todos os segmentos de um processo tem o mesmo tipo de acesso a um determinado segmento do processo. Êste inconveniente poderia ser resolvido tendo um vetor de acesso para cada segmento, com o tipo de acesso a todos os outros segmentos do processo.

Outro tipo de proteção necessária por "hardware", também já mencionado antes é a inclusão de instruções privilegiadas. Em geral instruções de E/S, instruções de manejo do relógio interno, instruções de mudança do espaço de endereçamento, etc. são privilegiadas, ié só podem ser executadas em um estado especial do processador central.

A tentativa de endereçamento de um tipo não permitido a um processo, ou a tentativa de execução de uma instrução privilegiada no estado do processador em que êste executa um processo do usuário geram automàticamente uma interrupção. A mudança do estado reservado para processos do usuário e o estado reservado para um processo do sistema pode ser feito de várias maneiras. Por exemplo pode-se ter uma instrução especial de "chamada para o sistema", ou pode-se compartilhar os segmentos do sistema necessário em cada processo como é feito no MULTICS, tratando um desvio para o sistema do mesmo modo como uma transferência entre segmentos do processo. Os bits de contrôle do segmento do sistema no SDT indicam que o segmento deve ser executado no estado reservado ao sistema operacional. Como o SDT é construído pelo sistema operacional e o usuário não tem acesso direto a êle, tem-se um mecanismo adequado de proteção.

Os mecanismos de proteção por "hardware" devem ser complementados por outros de "software". Em geral vários níveis de proteção são necessários. Só dois níveis de proteção, um reservado para o sistema e outro para o usuário, por exemplo, não é adequado normalmente, pois de um lado às vêzes deseja-se extender parte dos privilégios do sistema ao usuário, e outras vêzes é inconveniente que um certo processo do sistema tenha todos os privilégios, pois isto torna qualquer procedimento do sistema potencialmente muito perigoso, sendo ilimitado o dano que pode ser causado ao sistema por um seu procedimento defeituoso. Vários níveis de proteção portanto é uma meta desejável.

Proteção por "software" é garantido em vários pontos. Por exemplo o acesso ao sistema de arquivos é feito através de consultas a diretórios conforme vimos no capítulo 4, onde parte das informações se destina à verificação se o tipo de acesso solicitado é permitido ao processo solicitante. Em [22] encontra-se a base da proteção no MULTICS. A proteção por "hardware" no MULTICS, conforme foi visto acima protege um segmento e

não o "caminho de acesso" ao segmento. Isto é feito por "software".Segmentos no MULTICS são agrupados em classes. O acesso a segmentos de classes distintas é controlado pela sistema. O modelo na qual o mecanismo é baseado chama-se anéis de proteção. Imagina-se anéis concentricos de proteção. Acesso a êstes anéis é feito através de uma "porta" para cada anel. Quanto mais interno é o anel mais restrito é o acesso a êle. O anel 0 é o mais interno, vindo depois o anel 1, e assim por diante. Segmentos são associados a êstes anéis. Os segmentos mais críticos do sistema são associados ao anel 0, segmentos menos críticos ao anel 1 e assim por diante, podemos ter vários níveis de proteção no sistema operacional. Segmentos do usuário são associados a níveis ainda mais externos. Um segmento que está no anel k tem acesso direto a todos os segmentos deste anel. Se o segmento quiser acesso a um anel diferente, o acesso é garantido no sistema operacional. Se o acesso solicitado é a um segmento de um anel externo ao solicitante, o acesso é permitido; se porém é para um segmento de um anel mais interno apenas acessos através da "porta" são permitidos. A porta é um conjunto de informações que precisam ser verificadas antes que o acesso seja garantido. Primeiro é verificado se o segmento pode ter acesso ao segmento chamado; em seguida é verificado se o ponto de entrada ao segmento chamado é um ponto válido. Em seguida são verificados os parâmetros da chamada. Isto é necessário pois o segmento interno tem maiores direitos de acesso e assim é necessário verificar caso êstes contenham endereços, se o segmento que solicita a chamada tem acesso aos segmentos endereçados pelos parâmetros. Os parâmetros devem ser copiados antes desta verificação numa área de dados do mesmo anel que o segmento chamado, pois caso contrário se o segmento solicitante fôr compartilhado os parâmetros poderiam ser modificados após as verificações por um outro processo. A área de dados interna ao anel do segmento chamado só pode ser modificada por um segmento deste anel ou de aneis com direitos de acesso ainda maiores. Graham sugere alguns mecanismos adicionais de "hardware" que poderiam facilitar a implementação do sistema de aneis proposto, embora mecanismos adicionais não sejam estritamente necessários. Modelos matemáticos de proteção gerais, em função dos quais vários sistemas particulares podem ser expressos foram desenvolvidos. Um modêlo,está descrito em [17] que contém também as referências básicas nas quais o modelo está baseado.

5.7 Observações e conclusões

1. Algumas observações adicionais convém serem feitas nesta altura sôbre o conteúdo dêste capítulo:

Na secção 5.3 foi introduzida a noção de "swapping". Essencial à idéia de "swapping" é um algorítmo que escolhe uma área da memória que precisa ser removida para criar espaço para uma outra área que deve ser carregada. Algumas das estratégias mais comumente encontradas nesta escolha foram vistas na secção 5.5. Uma área de memória que deve ser removida não pode ser uma das áreas <u>sendo</u> referenciadas pelos processos em execução. Em particular se uma operação de E/S está sendo feita por algum canal nesta área, então enquanto a operação se desenvolve não se deve remover a área, pois o canal irá depositar ou retirar informações endereçando a área através de <u>endereços absolutos</u>, via roubo de ciclos, automáticamente. Outra observação importante em conexão com esta remoção é

que se a área em questão não foi modificada não há necessidade de ser copiada na memória auxiliar, pois uma cópia idêntica já se acha lá. É o que ocorre por exemplo com páginas "lê-sòmente".

Na secção 5.5. foram vistos também os estados de um programa necessários num sistema de multiprogramação, e as mudanças de estado possíveis. Foi dito que para mudar do estado "pronto" para um estado "bloqueado" o processo precisa necessàriamente passar pelo estado "correndo". Às vêzes é conveniente que um processo A possa bloquear um processo B. Se isto fôr permitido pelo sistema operacional então B na hora que fôr bloqueado por A pode estar no estado "pronto" e portanto a mudança direta de "pronto" para "bloqueado" é necessário neste caso.

2. Neste capítulo foram vistos alguns dos conceitos básicos para a construção de sistemas multiprogramados. Os conceitos expostos são mais ou menos aceitos geralmente como fundamentais e baseiam-se em um certo número de artigos básicos citados nas referências.

Uma bibliografia excelente está relacionada em [17] que contém também comentários valiosos sôbre alguns dos trabalhos referenciados. Em [1] também há uma lista de referências bastante extensa. Nas referências listadas neste trabalho só foram incluídos os que foram efetivamente consultados na confecção dêste trabalho.

REFERÊNCIAS

[1] Watson, R.W. "Timesharing system design concepts" Mc Graw-Hill, 1970.

[2] Flores, I. "Computer Sftware". Prentice-Hall, 1965.

[3] Wegner, P. "Introduction to Systems Programming" Academic Press, 1964.

[4] Wegner, P. "Programming Languages, information Structures, and Machine organization" Mc Graw-Hill, 1968.

[5] Barron, D.W. "Assemblers and Loaders", Macdonald & Co, 1969.

[6] Hopgood, F.R.A. "Compiling Techniques", Macdonald & Co, 1969.

[7] Rosen, S. "Programming Systems and Languages" Mc Graw-Hill, 1967.

[8] ACM. "Papers from the 3rd ACM Symposium on Operating System Principles" CACM 15, 3, Março/1972.

[9] ACM. "Papers from the 1st ACM Symposium on Operating System Principles" CACM 11, 5, Maio/1968.

[10] Knuth, D.E. "The Art of Computer Programing" Vol. I Addison -Wesley, 1968.

[11] "Time-Sharing memory management" University of Waterloo, Waterloo, Out. Canada.

[12] Simon, I. "especificações do computador hipotético HIPO" notas técnicas do CCE-USP.

[13] Markov, A.A. "The Theory of Algorithms", Academy of Sciences of the USSR, distributed by Clearinghouse for Federal Scientific and Technical Information, 1954.

[14] Rosin, R.F. "Supervisury and Monitor Systems" ACM. Computing Surveys 1, 1, Março/1969.

[15] Dijkstra, E.W. "The Struchue of the "THE" multiprogramming system". CACM. 11, 5, Maio/1968.

[16] Baskett, F. "Mathematical models of multiprogrammed Computer Systems". Ph. D. Thesis, The University of Texas at Austin; 1970.

[17] Denning, P.J. "Third Generation Computer Systems" ACM Computing Sinveys, 3, 4, Dezembro/1971.

[18] Denning, P.J. "The Working set model for program behavior" CACM. 11, 5, Maio/1968.

[19] Habermann, A. N. "Synchronization of Communicating Processes" CACM, 15, 3, Março/1972.

[20] Saltzer, J. H. "Traffic Control in a multiplexed Computer System" Ph. D. Thesis Rep. MAC - TR - 30, Proj. MAC. M. I. T. 1966.

[21] Denning, P. J. and Schwartz, S. C. "Properties of the Working set model" CACM. 15, 3, Março/1972.

[22] Graham, R. M. "Protection in an information processing utility" CACM. 11, 5, Maio/1968.

[23] Habermann, A. N. "Prevention of system deadlock" CACM. 12, 7, Julho/1969.

[24] Coffman, E. G.; Elphick, M.; and Shoshani, A. "System deadlocks" ACM. Computing Surveys 3, 2, Junho/1971.

[30] Holt, R. C. "Deadlock in computer systems" Ph. D. Thesis, Rep TR - 7191, Cornell University, 1971.