MEMÓRIA ESTENDIDA

PARTE IV

O Modo Protegido de trabalho do processador Intel de 32 bits foi idealizado, concebido e desenvolvido pelos engenheiros da Intel com um importante objectivo em vista: dar um bom suporte a nível do «hardware» do microprocessador às actividades de **Multitarefa** (ou «Multitasking» no seu nome de origem), isto é, a execução simultânea de diferentes programas ou processos.

Pretendia-se com isso acompanhar a linha de evolução previsível e desejável para os novos sistemas operativos e, aqui, o pensamento estava certamente dirigido (nessa altura, claro) para o OS/2 da IBM que vinha desde há muito sendo anunciado como o futuro grande sucessor do DOS.

Como os leitores estarão eventualmente lembrados, o OS/2 tivera um ínicio de carreira pouco fulgurante, conduzindo a um desastroso insucesso comercial na opinião de alguns. A generalidade dos utilizadores não emigrou para o OS/2 como esperado e continuou calmamente apegado ao velho DOS.

Houve que fazer então um apuramento de responsabilidades para essa estranha situação e, talvez sem grande surpresa, concluiu-se que o maior culpado era o microprocessador 80286, em particular pelo seu fraco apoio às actividades de **Multitarefa.** Isto porque nos microprocessadores 80286 cada segmento de memória estando limitado a um máximo de 64 KB, não é possível endereçar mais de 16 MB de memória e a compatibilização do **Modo Protegido** com o **Modo Real** é verdadeiramente problemática.

A resposta da Intel a essas questões veio com os microprocessadores de 32 bits e foi nada mais nada menos que:

a) Levantamento das restrições ao tamanho dos segmentos de memória. A partir de então um segmento de memória poderá ter até 4 Gigabytes de tamanho.

b) A memória endereçável poderá existir ou não fisicamente em RAM. O microprocessador dispõe de um mecanismo de *Paging* que, se for utilizado, permite mapear endereços lineares para endereços físicos completamente distintos. Deste modo é possível, entre outras coisas e só a título de exemplo, manter parte da informação de um programa (código ou dados) no disco rígido, carregá-la para uma zona tampão da memória física, quando for necessário, mapear quaisquer endereços lineares para esses endereços de memória física e fazer prosseguir a execução do programa normalmente. E quando essa informação deixar de ser necessária, ela pode ser descarregada de novo para o disco rígido.

Naturalmente que consecutivos acessos ao disco rígido tornam o sistema significativamente mais lento, mas também não se pode ter tudo e 1 MB de RAM é muito mais caro que 1 MB de espaço em disco rígido.

c) Um modo de emulação do **Modo Real** de funcionamento dos processadores 8086, mas compatível com o **Modo Protegido** e a gestão de memória em **Modo Protegido**.

Este modo de trabalho é designado por **Modo Virtual 86**.

Programas como o EMM386.EXE o QEMM386.SYS e o 386MAX.SYS colocam o processador em **Modo Virtual 86** e utilizam o mecanismo de *Paging* do **Modo Protegido** para mapear RAM nas zonas de memória conhecidas por **UMBs** (*Upper Memory Blocks*). As janelas de DOS no Windows e no OS/2 também são obtidas colocando o processador a trabalhar em **Modo Virtual 86**.

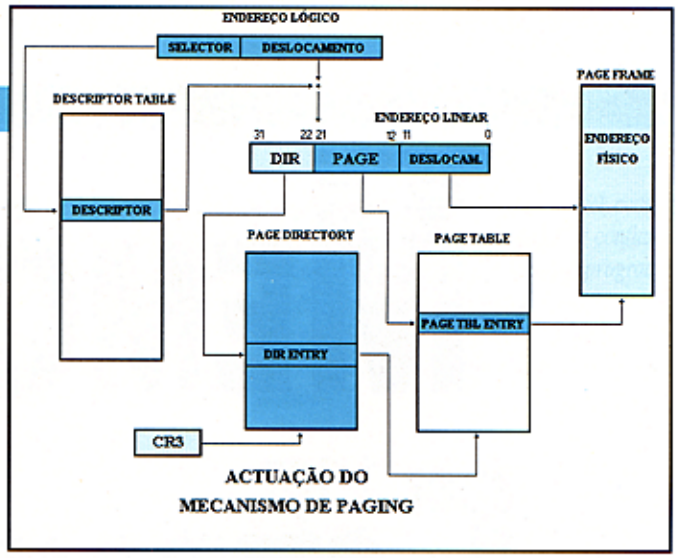
Neste nosso artigo vamos então abordar (infelizmente de modo sumário) dois tópicos extremanente importantes para complementar o estudo que temos vindo a realizar sobre o **Modo Protegido**:

- O mecanismo de *Paging*.
- O suporte dado pelos processadores de 32 bits às actividades de **Multitarefa**.

Vamos ainda ficar a saber o que são as *Call Gates*, e porque existe necessidade de mais este «portão» para que todo o modelo do **Modo Protegido** funcione sem atritos.

Para o próximo número da revista apresentaremos o programa VIRT86.EXE, muito interessante e didático em nossa opinião, e que ilustrará amplamente todos os temas da matéria coberta ao longo dos vários artigos sobre as bases fundamentais do **Modo**

Vamos ainda ficar a saber o que são as *Call Gates*, e porque existe necessidade de mais este «portão» para que todo o modelo do *Modo Protegido* **funcione sem atritos.**



ACTUAÇÃO DO MECANISMO DE PAGING

Protegido, e ainda a faceta mais enigmática e incompreendida do funcionamento do processador Intel de 32 bits, o **Modo Virtual 86**, irá (nunca tal vimos até hoje) ser integralmente divulgada em público através do código fonte de um programa.

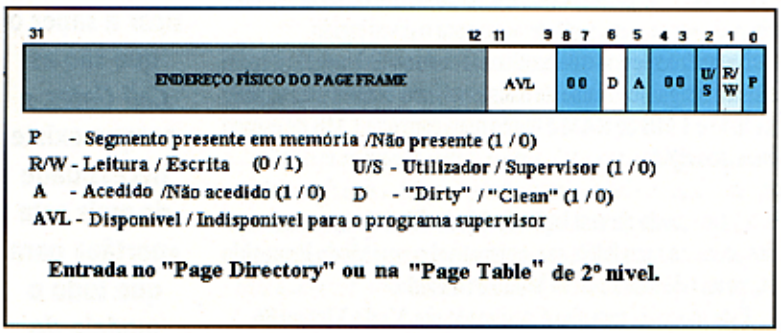
Mas até lá recomendamos um bom estudo de linguagem *Assembly*, pois creiam-me, não obstante o muito que se diz por aí, vale mesmo a pena.

O MECANISMO DE PAGING

Quando o mecanismo de *Paging* é utilizado, a correspondência endereço linear– endereço físico é feita através de tabelas colocadas em memória pelo programa supervisor do Modo Protegido. Essas tabelas designam-se por *Page Tables* e gerem «talhadinhas» de 4 KB de memória designadas por *Pages*.

Existem 2 níveis de *Page Tables*, o *Page Directory* e as *Page Tables* de 2º nível (muitas vezes designadas simplesmente por *Page Tables* quando não houver possibilidade de permanecer qualquer confusão).

Tanto um como outro dos níveis de *Page Tables* é constituído internamente por entradas de 4 bytes designadas por *Page Table Entries* e que apresentam o formato da figura seguinte.



P - Segmento presente em memória /Não presente (1 / 0)
R/W - Leitura / Escrita (0 / 1) U/S - Utilizador / Supervisor (1 / 0)
A - Acedido /Não acedido (1 / 0) D - "Dirty" / "Clean" (1 / 0)
AVL - Disponível / Indisponível para o programa supervisor

Entrada no "Page Directory" ou na "Page Table" de 2º nível.

O mecanismo de *Paging* considera um endereço linear de 32 bits como sendo constituído por 3 partes:

– Os bits 22 até 31 (últimos 10 bits) do endereço linear constituem um índice relativamente ao endereço físico base de um *Page Directory*.

– Os bits 12 a 21 do endereço linear constituem um índice relativamente ao endereço físico base de uma *Page Table de 2º nível*.

– Finalmente, os bits 0 a 11 constituem o valor de deslocamento (ou *offset*) na página de memória definida na entrada da *Page Table de 2º nível* obtida através dos restantes bits do endereço linear.

Por sua vez o valor do endereço físico base (valor de 32 bits) do *Page Directory* consta de um registo do microprocessador designado por CR3, ou PDBR (*Page Directory Base Register*).

Os endereços físicos base quer do *Page Directory*, quer das *Page Tables de 2º nível* quer das *Pages* de memória são sempre localizados em endereços físicos de memória múltiplos de 4 KB. Repare agora na figura seguinte pois talvez ajude bastante na consolidação de ideias:

Quando o microprocessador descodifica um endereço linear, procede do seguinte modo:

– O valor correspondente aos bits 22 a 31 do endereço linear é multiplicado por 4 (número de bytes numa *Page Table Entry*) e adicionado ao valor existente no registo CR3. Encontra assim o endereço físico de uma entrada no *Page Directory*. Nessa entrada do *Page Directory* os bits 12 a 31 (zona designada por Endereço Físico do *Page Frame* ou mais vulgarmente e no original por *Page Frame Adress*) são os 20 bits superiores do endereço físico base de uma *Page Table de 2º nível*.

Como se disse atrás, os endereços base destas tabelas situam-se em endereços múltiplos de 4KB ou 2^{12} bytes, por conseguinte os bits 0 a 11 do endereço físico têm sempre o valor 0, não havendo portanto qualquer ambiguidade na determinação do endereço físico.

– Seguidamente o valor correspondente aos bits 12 a 21 do endereço linear é multiplicado por 4 e adicionado ao endereço físico base da *Page Table de 2º nível*.

O microprocessador obtém deste modo o endereço físico de uma entrada na *Page Table de 2º nível*.

– Nessa entrada da *Page Table de 2º nível* o *Page Frame Adress* representa os 20 bits superiores do endereço físico base de um *Page Frame* de memória.

Os 12 bits inferiores são sempre 0, conforme se referiu acima.

– Finalmente os bits 0 a 11 do endereço linear são adicionados ao endereço físico base do *Page Frame* de memória, ficando assim completada a descodificação de endereço linear para endereço físico.

Repare agora de novo na figura representativa de uma *Page Table Entry* e nos vários campos que aí aparecem para além do endereço físico do *Page Frame*:

– Campo **P** (**Present** bit): Informa se o *Page Frame Adress* mapeia para uma *Page* existente em memória física. Se o campo **P** for 0 numa *Page Table Entry* de qualquer das *Page Tables*, a tentativa do processador para utilizar essa *Page Table Entry* para descodificar um endereço dará origem a uma *Page-Fault Exception*.

Essa *Exception* possibilitará a um sistema que utilize memória virtual em disco, copiar a *Page* em falta do disco para a memória física, actualizar a informação na *Page Table Entry* e fazer prosseguir o programa como se nada se tivesse passado.

– Campo **R/W** (**Read/Write** bit) informa se o acesso à *Page* definida pela *Page Table Entry* é só de leitura ou pode ser tanto de leitura como de escrita.

– Campo **U/S** (**User/Supervisor** bit). Algumas *Pages* são para

utilização exclusiva do Sistema Operativo ou programas supervisores do **Modo Protegido**. Se o CPL (*Current Privilege Level*) for 0, 1 ou 2 o processador encontra-se a executar em modo Supervisor (do ponto de vista do mecanismo de *Paging*) e pode utilizar *Pages* definidas por *Page Table Entries* com o valor 0 no campo U/S. Se o CPL for 3 então só são utilizáveis *Pages* definidas por *Page Table Entries* com o valor 1 no campo U/S.

– Campo **A** (**Acessed** bit). Como o nome aparenta dizer, este campo informa se uma *Page* ou *Page Table de 2º nível* foi acedida em leitura e/ou escrita.

– Campo **D** (**Dirty** bit) informa se uma *Page* foi acedida para escrita.

– Campo **AVL** (**Available** bits) está livre para a utilização que for julgada mais conveniente pelos programadores do sistema.

A activação do mecanismo de *Paging* requer como mínimo a seguinte sequência de acontecimentos:

1) O sistema gestor do **Modo Protegido** criou pelo menos duas *Page Tables*, ou seja um *Page Directory* e uma *Page Table de 2º nível*.
2) O registo do processador CR3 é carregado com o endereço físico base do *Page Directory*.
3) O processador encontrando-se já em Modo Protegido, o bit 31 do registo CR0 do processador é carregado com o valor 1, o que pode ser conseguido com as seguintes instruções *Assembly*:

```
MOV EAX, CR0
OR EAX, 8000000H
MOV CR0, EAX
```

E é tudo o que de mais fundamental interessa conhecer sobre o mecanismo de *Paging*. Para uma aplicação e consequente aprofundamento destes conhecimentos, tente analizar muito cuidadamente o código fonte do programa VIRT86.EXE que apresentaremos no próximo número da nossa revista.

A MULTITAREFA

Existem dois tipos possíveis de *descriptors* nas *Descriptor Tables* GDT (*Global Descriptor Table*) ou LDT (*Local Descriptor Table*) que dão apoio ao mecanismo de Multitarefa.

O primeiro designa-se por *Task State Segment Descriptor* ou abreviadamente por *TSS Descriptor*, e o segundo designa-se por *Task Gate Descriptor*.

Quando um *interrupt*, *exception*, instrução *assembly* JMP ou CALL passa a execução de um programa para um *TSS Descriptor* ou *Task Gate Descriptor* dá-se aquilo que se designa por «Mudança de Tarefa» (ou talvez mais vulgarmente por *Task Switch*).

Durante um *Task Switch* o conteúdo de quase todos os registos do processador da Tarefa anterior são automáticamente guardados pelo processador numa estrutura em memória designada por *Task State Segment* (ou «TSS») antes da execução poder ser iniciada no ambiente da nova Tarefa. E quando a execução dessa nova Tarefa tiver terminado, o retorno à Tarefa anteriormente interrompida pode ser invocada com a simples execução de uma instrução *assembly* IRET. A figura seguinte mostra-nos o formato dos primeiros 104 bytes de um TSS

31 ... 15 ... 0		
'I/O MAP BASE'	0000000000000000T	64
0000000000000000	LDT	60
0000000000000000	GS	5C
0000000000000000	FS	58
0000000000000000	DS	54
0000000000000000	SS	50
0000000000000000	CS	4C
0000000000000000	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
EIP		20
CR3		1C
0000000000000000	SS2	18
ESP2		14
0000000000000000	SS1	10
ESP1		0C
0000000000000000	SS0	08
ESP0		04
0000000000000000	'BACK LINK' PARA TSS ANTERIOR	00

TASK STATE SEGMENT

Nem todos os campos do TSS são dinâmicamente alterados durante o *Task Switch*, alguns são estáticos e apenas lidos pelo processador nessa ocasião:

– Campo **LDT**, o qual contém o «selector» da *Local Descriptor Table* da nova Tarefa.

– Os endereços lógicos dos *stacks* para os níveis de privilégio 0, 1 e 2. O modelo de **Modo Protegido** impõe que todos os anéis de protecção de nível superior (número cardinal inferior) a 3, devem dispor de *stacks* próprios para receber chamadas de níveis inferiores. Deste modo evita-se a possibilidade de um nível superior poder ser danificado porque o *stack* que lhe é fornecido por um nível inferior teve a infelicidade de ser pequeno demais.

– O bit **T** (*Trap bit*). Se este bit tiver o valor 1, o processador accionará uma *Debug Exception* sempre que se der o *Task Switch*.

– O endereço base do Mapa de Permissão de I/O (*I/O Permission Bitmap*). Este Mapa, no caso de existir, faz parte do TSS e é localizado algures acima do byte 67H.

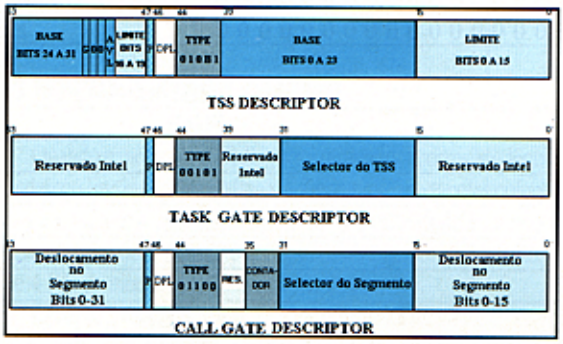
O seu endereço base é o deslocamento desde o ínicio do TSS, sendo o valor mínimo por conseguinte 68H. O valor máximo está estabelecido em 0DFFFH. O último byte do Mapa deverá ter o valor 0FFH. Com o auxílio deste Mapa o processador pode gerar *exceptions* sempre que houver acesso a determinados portos de Entrada/Saída, permitindo por conseguinte o sistema Supervisor emular ou aceder por si próprio a esses portos. Os portos que geram *exceptions* são especificados no *I/O Permission Bitmap* com o valor 1 no bit respectivo.

No programa VIRT86.EXE ilustraremos também este assunto numa perspectiva em que todo e qualquer programa irá ficar «iludido» sobre a quantidade de memória *extended* existente no

sistema enquanto o VIRT86.EXE estiver em Supervisor.

– No caso de se estar a utilizar o mecanismo de *Paging* o campo CR3 do TSS pode ser utilizado para dar a cada Tarefa as suas próprias *Page Tables*.

Vejamos agora a figura seguinte e fixemos a nossa atenção no formato de um *TSS Descriptor*.



Verifique que o campo Type apresenta um bit **B**, designado por *Busy bit*. Se o *Busy bit* tiver o valor 1, a Tarefa encontra-se em execução. As Tarefas não são **reentrantes**, qualquer tentativa de executar uma tarefa com o *Busy bit* ligado dará origem a uma *exception*.

Em Multitarefa um dos registos do processador, o *Task Register*, contém sempre o «selector» do *TSS Descriptor* corrente na GDT. Por este motivo os *TSS Descriptors* só podem existir na GDT e nunca numa LDT ou numa IDT. Para obviar a esse inconveniente utilizam-se algumas vezes *Task Gates*. Aqui o campo designado por Selector do TSS indexa para um *TSS Descriptor* na GDT.

O campo designado por DPL controla o acesso ao *TSS Descriptor*, de modo que uma rotina não terá acesso a mudança de Tarefa via *Task Gate* se o RPL do «selector» constante do segmento CS e o CPL da rotina não forem tanto ou mais privilegiados que o DPL da *Task Gate*.

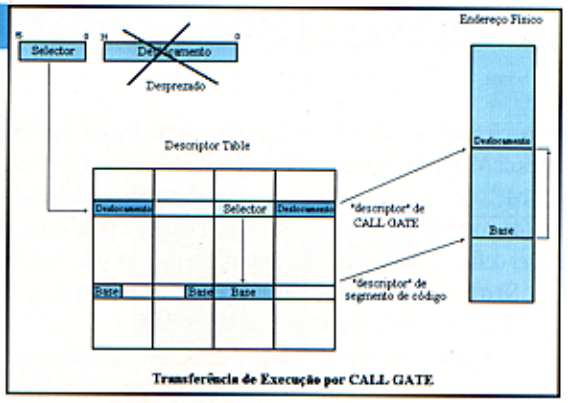
De notar que quando se utiliza uma *Task Gate* para mudar de Tarefa o DPL do *TSS Descriptor* de destino não é utilizado pelo processador.

E, como haviamos anunciado, vamos tratar agora das

CALL GATES

É possível também efectuar transferências para segmentos de código com níveis mais elevados de privilégio recorrendo a *Call Gates*. A figura anterior mostra-nos, além de outras, também a estrutura de um *Call Gate Descriptor* o qual pode residir quer na GDT, quer numa LDT, mas nunca na IDT.

O *Call Gate Descriptor* contém um «selector» para um *descriptor* de segmento de código numa das *Descriptor Tables*. Contudo, aqui será o valor do deslocamento contido no próprio *Call Gate Descriptor* que irá ser utilizado e não o valor de deslocamento determinado dentro do processo em execução, o qual será pura e simplesmente desprezado. Isto assegura que uma *Call Gate* fará a transferência de execução para um ponto perfeitamente bem determinado. Vejamos no esquema da figura seguinte como isto tudo se processa.



Durante a transferência de execução por *Call Gate* são verificados pelo processador os seguintes níveis de privilégio:

- CPL
- RPL
- DPL do *Call Gate Descriptor*.
- DPL do *descriptor* do segmento de código de destino.

O DPL do *Call Gate Descriptor* determina de que níveis de privilégio esta pode ser chamada. Quando as *Call Gates* são acedidas por instruções *Assembly* do tipo CALL vigoram as seguintes regras:

1) O DPL do *Call Gate Descriptor* será sempre maior ou igual que o CPL ou o RPL.
2) O DPL do *descriptor* do segmento de código de destino será sempre menor ou igual (maior ou idêntico nível de privilégio) que o CPL.

Quando se efectua uma transferência para uma rotina em nível mais elevado de privilégio através de uma *Call Gate*, o processador faz automáticamente o seguinte:

1) Actualiza o CPL
2) Muda de *stack*

O novo CPL corresponderá ao DPL do *descriptor* do segmento de código de destino.

Para mudar de *stacks* o processador encontra a informação necessária recorrendo ao *Task State Segment* tal como se escreveu atrás.

Os *Call Gate Descriptors* contêm ainda de espécial um campo designado por «Contador» cujo valor informa o processador do número de *doublewords* que deverá copiar do velho para o novo *stack*.

E enfim, ficamos por aqui, a temática é por natureza muito difícil e alguns leitores terão já certamente pensado em desistir de nos continuar a ler, isto se outros entretanto não acabaram por adormecer com a luz do candeeiro acesa.

Demos uma panorâmica dos conhecimentos que consideramos mais cruciais. Um conhecimento efectivo obterá o leitor lendo algumas obras e estudando listagens de programas que versem estas matérias. As melhores obras são, sem dúvida, embora de leitura pouco fácil, os manuais de referência para programadores que são publicados pela Intel para os vários microprocessadores. Listagens de programas são efectivamente muito difíceis de encontrar, porém o programa VIRT86.EXE a ser publicado no próximo número da **Spooler** talvez contribua um pouco para preencher essa lacuna.

José Páscoa