|  |
| --- |
|  |
| Sistemas Operativos |
|  |
| LI41d |

André Martins

Leonardo Freire

Sistemas Operativos

LI41d

## Exercício 1

1

1. Tendo em conta que cada página possui no máximo uma capacidade de 212 bytes e que cada entrada da página contém 8 bytes, o numero de entradas de uma tabela de paginas corresponde à divisão da capacidade pelo tamanho de uma entrada logo,   
   212 /23 , 29.

Sabendo o número de entradas, é preciso ter em conta que esta arquitetura está dividida em 3 níveis, portanto, para navegar entre os níveis são necessários 3\*9 bits, logo 27.

Como as páginas são de 212 bytes para especificar um endereço são necessários 12 bits.

Conclui-se que o número de bits uteis é 12+ 27 , portanto 39.

1. A dimensão máxima da memória física corresponde à multiplicação do número de entradas pelo tamanho de cada entrada.

Assumindo que cada PTE armazena 38 bits de endereço físico, estes 38 bits são utilizados apenas como base para um endereço na memória física. À base existe a necessidade de somar o offset, ou seja, mais 12 bits.

Pode dizer-se que o número de entradas corresponde a 238+12, 250.

Sabendo que o número de bytes ocupados por cada entrada é 8, a capacidade máxima da memória física é 250 + 3. Após pesquisa verificou-se que 250 corresponde a *PetaBytes*, portanto a memória possui no máximo 8 *PetaBytes.*

1. Se o endereço virtual for constituído por 36 bits, para manter os 3 níveis de tabelas de páginas, vai ser necessário diminuir algures o tamanho do endereço virtual. Isto porque já se tinha verificado que seriam necessários 39 bits.

A solução escolhida foi:

6 BITS - > tabela 1º nível

9 BITS - > tabela 2º nível

9 BITS - > tabela 3º nível

12 BITS - > offset

Se se trocasse os bits da tabela de 1º nível com os de 2º, o número de tabelas que não ocupam todas as entradas possíveis ia aumentar, tornando a memória mais fragmentada. Portanto decidiu-se que a tabela de primeiro nível iria ser a que possuía menos bits.

Não se optou por uma divisão de 8-8-8 porque faria com que existisse em todos os níveis informação redundante.

d)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Endereço de memória virtual | 1ºnivel | 2ºnivel | 3ºnivel | offset | Endereço de memória fisica |
| 0x4304f0 | 0x0 | 0x2 | 0x30 | 0x4f0 | 0x6a000004f0 |
| 0x432fE8 | 0x0 | 0x2 | 0x32 | 0x5e8 | 0x6c000005e8 |
| 0x430700 | 0x0 | 0x2 | 0x30 | 0x700 | 0x6a00000700 |
| .L2 = 0x26000 | 0x0 | 0x0 | 0x26 | 0x0 |  |
| .L3 = 0x29000 | 0x0 | 0x0 | 0x29 | 0x0 |  |
| .L4 = 0x2C000 | 0x0 | 0x0 | 0x2c | 0x0 |  |

C:\Users\Druh\AppData\Local\Microsoft\Windows\INetCache\Content.Word\transferir.png

ii

Se se retirar a permissão de escrita por parte do índices 0x32 ocorrerá *segmentation* *fault* na instrução   
- *mov* [ 430700 ], *eax* ,

## Exercício 2

1. 1.A pagina referida não está submetida, ou seja, o programa tentou aceder a um endereço que não tem mapeamento valido (por exemplo, se o endereço já foi libertado). Geralmente o *handler* termina o processo.

2.Acesso a uma pagina sem as devidas permissões, ou seja, um processo tenta realizar uma operação invalida, como o exemplo de escrever numa pagina que apenas permite leitura. Novamente o programa é terminado.

3.O problema é parecido com o 2., ou seja, o processo tenta realizar uma operação invalida sobre uma pagina, mas desta vez o *handler* resolve o problema de forma diferente, pois agora a pagina estava marcada como *copy*-*on*-*read*, e assim o *memory* manager não assume que houve uma violação de acesso, e faz ao invés disso uma copia privada da pagina ao processo atual. Quando o processo volta ao controlo da pagina e tenta fazer novamente a operação, esta já executada de forma normal.

4.Quando uma *thread* insere um valor no seu *stack* e cruza com uma pagina que ainda não foi inicializada ou alocada, porem o *memory* manager esta preparado para detetar esta situação e reconhece como um caso especial, e o que faz é disponibilizar uma nova pagina, colocar todos os seus valores a 0, e mapeá-la ao processo, isto desde que ainda exista espaço nas paginas virtuais para o *stack*.

5.Falha normal na pagina. Existem vários tipos de falhas normais nas paginas e diferentes resoluções consoante o tipo de erro.

1. O esquema de carregamento de páginas usado no Windows no arranque dos processos em alternativa ao *demand paging*, é designado de *Super-Fetch*. O *Super-Fetch* consiste no pré-carregamento de páginas que, provavelmente, irão ser utilizadas no processo. Estas páginas são carregadas numa lista *standby*, onde são facilmente alcançadas caso haja necessidade de colocá-las no processo. Tal mecanismo promove o aumento da velocidade de execução porque permite melhorar os tempos de leitura. É também utilizado no processo de *boot* do sistema, quando uma aplicação passa para primeiro plano e quando se reinicia o sistema, após hibernação.
2. Uma *soft* *fault* é quando o *memory* manager consegue resolver o problema acedendo à página em memória, sem ser necessário voltar a ler a página a partir do disco, no caso de ser necessário aceder ao disco para recuperar a página estamos perante uma *hard* *fault*.

As *soft* *fault* segundo o autor podem ocorrer por causa de uma página partilhada já estar mapeada a um processo, ou quando é necessária uma página a zero. Também pode ocorrer quando uma página é comprimida para aumentar o tamanho da memória física.

1. A introdução do *swap* *file* nos sistemas operativos modernos foi introduzido para que as aplicações do WUP (*Windows* *Universal* *Program*) quando não estiverem a ser utilizadas em *foreground* são suspensas, e não executam até o utilizador voltar a selecionar essa aplicação. Quando o sistema necessita de memória, é selecionada uma aplicação que não esteja a ser usada há algum tempo e é feito o *swap* de todas as páginas em várias operações IO ao invés de ser feito página a página. Depois quando a aplicação voltar a ser executada é feito uma leitura sequencial o que é mais eficiente que *random* *reads*.

A motivação para o aparecimento do *swap* *file* foi o facto de otimizar o uso de recursos do sistema, aumentando a duração da bateria, visto que as aplicações cada vez usam mais recursos e mais aplicações são usadas em simultâneo.

1. A partir do Windows 8 é utilizada uma funcionalidade chamada *Physical* *Address* *Extension*(PAE) que define uma hierarquia de tabelas de página de 3 níveis, com entradas de tabela de 64 bits em vez dos tradicionais 32 bits, permitindo ao processador aceder a endereços físicos num espaço físico superior aos 4 GB. Assim o autor do texto não está a cometer um erro ao dizer que as PTE são de 8 bytes.

### Exercício 3

Neste exercício era-nos pedido que verificássemos qual a maior região do espaço de endereçamento para um determinado processo.

A resolução deste exercício passou por obter um *Handle* para o processo através do seu id, para tal usamos a função da biblioteca do *Windows*, *OpenProcess*.

A partir daí, usamos uma estrutura que representa a região de memoria. Mais uma vez, esta também faz parte da biblioteca do Windows. Fazemos uma *query* para obter as informações da região de memoria atual, onde o resultado dessa *query* fica guardado na estrutura. Basta então comparar o tamanho da região com o tamanho da região de maior dimensão encontrada até esse ponto, e assim sucessivamente até obter o maior valor.

### Exercício 4

1. A vantagem de as bibliotecas do sistema ocuparem os mesmos endereços independentemente dos processos em observação é a de permitir uma maior eficiência no tempo de carregamento e na gestão da memória.
2. O *loader* resolve esta questão procedendo a realocação em tempo de carregamento das *dll’s* para outros endereços de memória. Para evitar esta situação de realocar a memória no tempo de carregamento é possível utilizar uma ferramenta *rebase* para mudar os endereços antecipadamente e evitar as interceções.

### Exercício 5

Para representarmos os dados da imagem é necessário considerar alguns aspetos; primeiro recebemos o nome do ficheiro, e mapeamos esse ficheiro em memória, mas agora temos de garantir que o ficheiro tem o formato desejado; segundo, temos de saber como estão representados os dados da imagem em memória, porque o mapeamento varia consoante o driver de processamento que é usado, existe o da Motorola e o da Intel, que representam os dados em *big endian* e *little endian*, respetivamente; terceiro, temos de saber como obter as informações consoante o seu formato e a sua representação em memoria, para isso usamos funções auxiliares de obtenção dos dados.

Tendo em conta esses aspetos principais foi do nosso interesse enfatizá-los e fizemo-lo da seguinte maneira, criámos estruturas para representar o mapeamento em memória, e para guardar as informações da imagem. Para obter essas informações da memória, foi necessário compreender o formato do ficheiro e em que zonas estão contidas as informações relevantes. Após isso foi necessário iterar sobre essa informação, e para isso guardamos os valores importantes do “cabeçalho” de cada zona, como os *offsets* para cada elemento, a *tag*, o tamanho, e os dados. Consoante a *tag* sabemos quais os dados estão representados nessa zona e assim preenchemos a nossa estrutura. No final apresentamos essas informações na consola.