Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Sistemas Operativos, Inverno de 2016/17
Primeira Série de Exercícios

1) Considere uma MMU com espaço de endereçamento virtual de 256 GiBytes e 3 níveis de tabelas de páginas, com a dimensão máxima de uma página e com entradas de 4 bytes. Responda às questões justificando devidamente as suas respostas.

   a) Qual o tamanho mínimo da página para suportar esta arquitectura? Considerando o espaço de endereçamento virtual igual ao espaço de endereçamento físico (RAM), quantos bits existem em cada PTE para especificar o endereço físico?

   b) Considerando que são utilizados 6 *bits* para *flags* em cada PTE, indique quais as dimensões máximas de espaço de endereçamento virtual e físico mantendo a dimensão de página indicada na alínea anterior e o número de níveis indicado no enunciado. Indique as consequências de se introduzir um bit de controlo adicional em cada PTE.

   c) Comente a seguinte afirmação: "na arquitectura apresentada, cada acesso à memória, requerido por uma instrução de um programa em execução, implica sempre 4 acessos físicos, uma vez que existem 3 níveis de tabelas de páginas".

   d) Considerando uma política de carregamento em que inicialmente apenas é carregada a página que contém o *entry point* de código e é alocada uma *page frame* para *stack*, faça um diagrama com o estado inicial da estrutura de páginas. Admita que o *entry point* corresponde ao endereço virtual `0x486140` e o *stack pointer* é iniciado com o valor `0x1fffffffff`. Admita também que as *page frames* alocadas correspondem, respectivamente, aos endereços físicos `0x40000` e `0xC0000`.

2) Na gestão de memória dos sistemas operativos da família Windows, indique:

   a) A motivação para a existência das listas *standby* e *modified*.

   b) Porque razão, de uma forma geral, as listas *Free* e *Zero* indicam um valor de memória disponível consideravelmente inferior ao valor total de memória disponível no sistema?

   c) Indique, pelo menos, duas funções da Windows API que possam levar páginas libertadas de um *Working Set* directamente para a lista de páginas *Free*. Justifique a sua resposta.

   d) Faça um diagrama de estados que apresente os estados por que passa uma *page frame* no Windows. Considere como estados iniciais e finais do ciclo de vida aqueles em que a *page frame* se encontra completamente dissociada de qualquer espaço de endereçamento (*free* ou *zeroed*). Dê nomes às transições entre estados, descrevendo cenários ilustrativos da sua ocorrência.

3) Utilizando as funções de sistema `GetPerformanceInfo`, `GetProcessMemoryInfo`, `GlobalMemoryStatusEx` e `QueryWorkingSet`, escreva a função `PrintMemInfo` que apresenta na consola a seguinte informação acerca do sistema e da execução do processo com o *id* passado como argumento:

   Informações globais:

   - Total da memória física existente
   - Total da memória física ocupada
   - Total de memória virtual existente
   - Total de memória virtual ocupada
   - Dimensão das páginas de memória

   Informações locais ao processo:

   - Total de espaço de endereçamento existente
   - Total de espaço de endereçamento ocupado
   - Número de (*hard*) *page faults*
   - Dimensão total do *working set*
   - Dimensão do *working set* privado

   Nota: a utilização das funções `GetPerformanceInfo, GetProcessMemoryInfo e QueryWorkingSet` implica a ligação com a biblioteca `psapi.lib` ou `kernel32.lib` (dependendo da versão do sistema operativo) e a utilização do ficheiro de *include* `psapi.h`.

4) Para cada versão do Windows as bibliotecas dinâmicas de sistema (`kernel32.dll`, `user32.dll`, `psapi.dll`, etc) foram construídas para serem mapeadas em endereços bem definidos, sem conflitos (interseções) entre as várias bibliotecas.

   a) Qual a vantagem deste procedimento?

   b) No caso de ocorrerem intersecções entre duas *dll's*, que não de sistema, necessárias a determinado processo, como resolve o *loader* essa questão?

5) A fig. 1 apresenta de forma sumária um excerto da organização de um módulo executável no formato PE. O ficheiro começa com um *header* de compatibilidade com antigas aplicações MS-DOS, com o *layout* especificado na estrutura `IMAGE_DOS_HEADER`. O campo `e_lfanew` contém o *offset* dentro do ficheiro onde se encontra a componente obrigatória do *header* PE (estrutura `IMAGE_NT_HEADERS`). O campo `OptionalHeader,` do tipo `IMAGE_OPTIONAL_HEADER,` contém um *array* de `16` estruturas do tipo `IMAGE_DATA_DIRECTORY`, que localiza tabelas importantes, nomeadamente, na entrada 1 (`IMAGE_DIRECTORY_ENTRY_IMPORT`), a directoria de tabelas de importação (secção `.idata`), que permite aceder às tabelas de funções importadas de cada DLL usada pelo módulo. Em anexo é entregue um documento (`pecoff_v83.pdf`) com a especificação do formato PE.

   Notas:

   1. O conteúdo do ficheiro executável está mapeado em memória a partir de um endereço base. Para obter o endereço base onde está mapeado o executável utilize a função `GetModuleHandle` passando `NULL` como argumento. É importante referir que a localização presente nas entradas `IMAGE_DATA_DIRECTORY`, e em geral os campos que referem endereços virtuais, e que já se encontravam presentes no ficheiro, antes de ocorrer a ligação dinâmica, na verdade indicam a posição relativa ao endereço virtual base do carregamento do módulo em memória. Designam-se por isso como *Relative Virtual Addresses-RVAs*.

   2. As estruturas apresentadas existem em variantes de 32 e 64 bits, consoante o módulo executável seja de 32 ou 64 bits. Na questão seguinte considere apenas executáveis de 32 bits.

   Implemente uma DLL que forneça um serviço para apresentar no *standard output* a lista de funções importadas pelo processo invocante, organizadas por DLL a que pertencem. Por cada função apresente o seu nome e o endereço onde está presente. Abaixo apresenta-se um exemplo do *output* pretendido.

```
DllName=KERNEL32.dll
        Function GetModuleHandleW(76C73480)
        Function GetModuleFileNameW(76C74920)
        Function VirtualProtect(76C7432F)
        Function VirtualAlloc(76C71856)
        Function GetLastError(76C711C0)
        Function GetProcessId(76C9CEF4)
        Function GetCurrentProcess(76C71809)
DllName=MSVCR100D.dll
        Function exit(5F8F8080)
        Function wcsstr(5F9259A0)
        Function getchar(5F90F660)
        Function wprintf(5F9211A0)
        Function printf(5F917EE0)
```

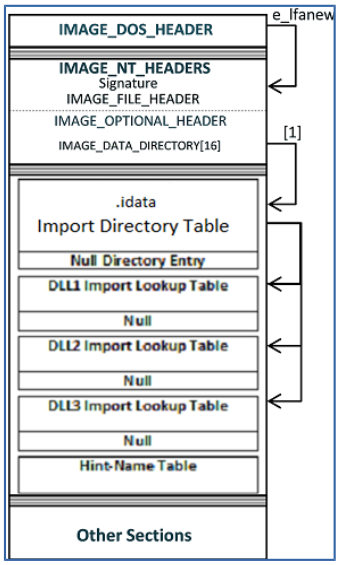


Fig.1 Excerto da estrutura de um módulo executável no formato PE

Data limite de entrega: 11 de Novembro de 2016

Jorge Martins e João Patriarca
ISEL, 14 de Outubro de 2016