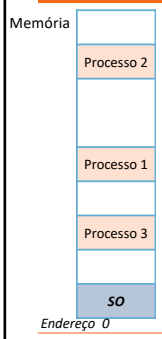
Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 22

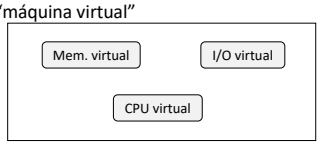
Execução de vários programas



- O SO cria **processos** para executar os programas:
  - Define a memória onde o programa vai executar
  - Lê o programa executável de disco para essa zona de memória (código e variáveis inicializadas)
  - Inicia tudo para começar a executar na instrução certa
    - Incluindo Pilha (SP) e IP

Máquina virtual de um processo

- Processo = instância de um programa em execução
  - O SO mantém informação sobre cada processo para garantir a sua "máquina virtual"



- O SO gere e mapeia na máquina real
  - Mem. Virtual → mapa de memória privado
  - CPU virtual → partilhado (time-sharing)
  - I/O virtual → exemplo: canais de I/O (streams) ficheiros

O SO assegura uma máquina virtual para cada processo

- Para um processo é como se ocupasse a máquina sozinho
- O estado da computação é preservado nas trocas de contexto: CPU, memória e IO
- As interações com periféricos e ficheiros são realizadas pelo SO
- Um processo só consegue escrever na memória que lhe está reservada.
  - Mais nenhum processo consegue aceder a essa memória.

Mecanismos Indispensáveis ao SO

- O hardware deve permitir, com o SO:
  - Proteção/partilha de memória
  - Proteção/partilha do CPU
  - Proteção/partilha do I/O
  - → Instruções de modo supervisor e interrupções ...
- Garantir que os processos não acedem diretamente aos recursos e que só o SO pode controlar esses recursos

Partilha do CPU

- Como partilhar o CPU entre os processos e SO?
- *Temporizador* – interrompe o processamento ao fim de um tempo pré-determinado.
  - Decrementado a cada "tick" do relógio.
  - Quando chega a 0 lança uma interrupção de hardware.
  - Assegura que o SO nunca perde o controlo do CPU
- *Load-timer* é uma instrução privilegiada.

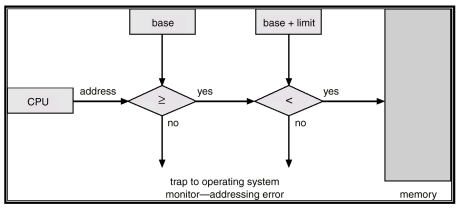
Memória - Problemas

- O programa tem de ser carregado sempre na mesma posição de memória?
  - Como podemos ter mais de um programa na memória?
  - Se o ficheiro executável for carregado em diferentes endereços, os endereços usados no programa têm de ser alterados?
- Não podemos ter programas (código+dados+pilha) maiores que a memória instalada?

Proteção de Memória

- É necessário proteger a memória entre programas e a do Sistema Operativo
- Uma solução simples é o SO fixar uma faixa legal de endereços (no CPU) para cada processo:
  - Registo Base
  - Registo Limite
- A memória fora dessa zona é proibida ao programa.
  - Os endereços no programa são alterados no carregamento
  - O CPU não usa endereços fora do intervalo

Protecção de memória

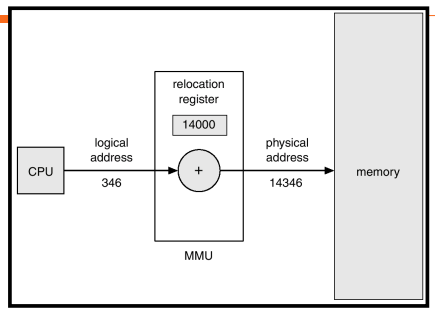


- As instruções para alterar os registos *base* e *limite* são privilegiadas.
  - Só o SO as pode usar

Proteção de Memória – Exemplo 2

- Definição da zona de memória de um processo:
  - **Registo Base** e **Registo Limite** para cada processo
  - Estes fazem parte duma unidade de transformação de endereços (**MMU**-*Memory Management Unit*)
  - Os endereços efetivos do programa são **virtuais** e relativos ao registo base para obter o **endereço real ou físico**
  - A memória fora dessa zona é proibida ao programa.

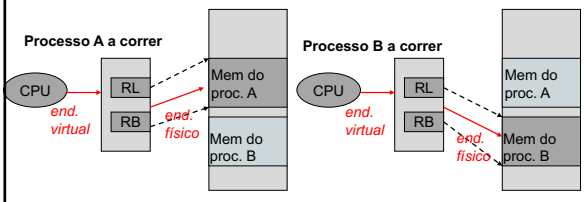
Registo de recolocação



Esta técnica permite carregar o programa em qualquer posição da memória física

Troca de segmento de memória física

- Suporte *hardware*: registos base e limite
  - O valor corrente dos registos base (RB) e limite (RL) são específicos de cada processo
  - O S.O. gere estes valores, usando instruções privilegiadas



Espaço físico gerido por afetação de segmentos contíguos

- Cada executável indica a necessidade máxima de memória
- A imagem de um processo é criada num segmento de memória de dimensão igual ou superior às necessidades do programa
- Suporte *hardware*: registos base e limite
  - Asseguram a recolocação do programa e a protecção entre processos;
  - Definidos pelo SO para cada processo criado
- Transformação dos endereços:
  - ***End.físico = End.Virt + End.Base***
  - *Se dentro do limite válido*

Espaços de endereços lógicos (ou virtuais) e físicos (ou reais)

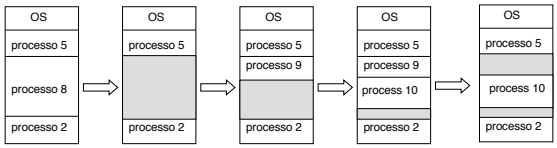
- Cada processo tem um espaço de endereçamento lógico que é separado do dos outros processos
- O conceito de um espaço de endereçamento lógico é independente do espaço de endereços físico:
  - Endereços lógicos – usados no programa (definidos pelo programador, compilador e *linker*); na execução são os vistos pelo CPU; também conhecidos por endereços virtuais.
  - Endereços físicos – obtidos a partir dos endereços lógicos e recebidos pela cache e pela memória física

Espaços de endereços lógicos (ou virtuais) e físicos (ou reais)

- Existe uma Unidade de Transformação de Endereços ou MMU
- Os endereços virtuais e físicos diferem, claro!
  - E os físicos dependem de onde o processo é colocado em cada execução
- Em cada momento, estão carregados em memória imagens (código+dados+pilha) de vários processos
  - Torna independentes as organizações dos dois espaços
  - o SO gere a memória física disponível em cada instante

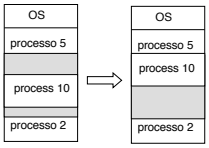
Inconvenientes da atribuição de memória por segmentos contíguos

- Zonas livres de diferentes dimensões estão espalhadas pela memória física.
- Quando um processo é criado, é necessário atribuir-lhe uma zona livre contígua suficientemente grande
- As zonas de memória livre podem-se encontrar dispersas pela memória → **Fragmentação**



Combatendo a fragmentação

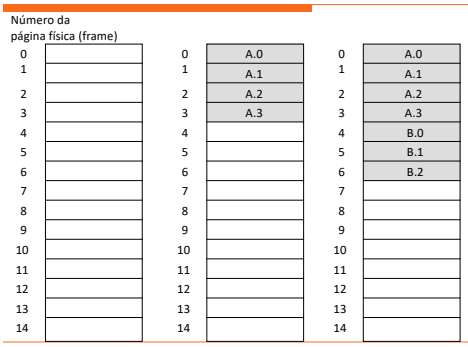
- A fragmentação pode ser eliminada juntando a memória ocupada
  - demorado: exige movimentar os vários segmentos em memória
- Alternativa?
- Todos os processos usam segmentos do mesmo tamanho
  - um processo pode necessitar de mais espaço (ou vários segmentos)
  - ou não necessitar de todo o espaço → **fragmentação interna**



Solução: Paginação

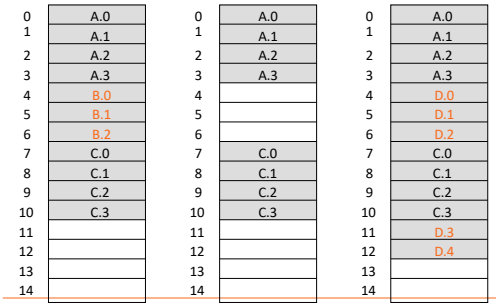
- Divide-se a memória física em pequenos segmentos de tamanho fixo chamados de **páginas físicas** (ou **frames**)
  - o tamanho é uma potência de 2 (p.e. 4 ou 8Kbytes).
- Divide-se o espaço de endereçamento lógico em blocos do mesmo tamanho, chamados de **páginas lógicas** (ou páginas virtuais).
- Cada processo pode ocupar várias páginas, para acomodar imagens maiores que uma só página
- Existe uma **tabela** que relaciona páginas virtuais com os frames físicos
  - Esta é usada pela unidade de transformação de endereços

Carregando os processos A e B em memória



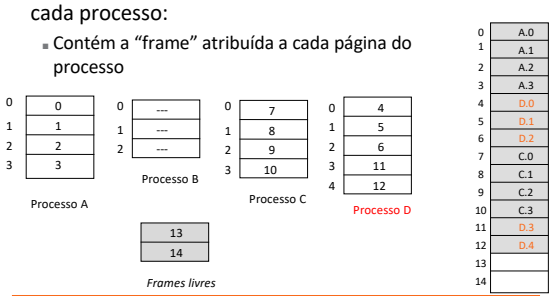
Paginação: atribuição não contígua

- o processo B termina e inicia-se o processo D:

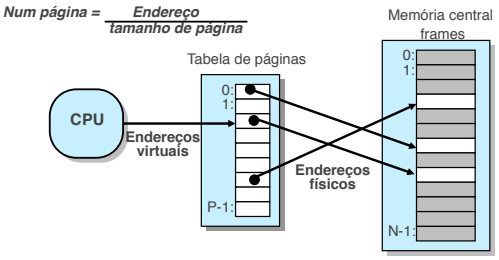


Tabelas de páginas para o exemplo

- O OS mantém uma tabela de páginas para cada processo:
  - Contém a "frame" atribuída a cada página do processo



Transformação de endereços (1)

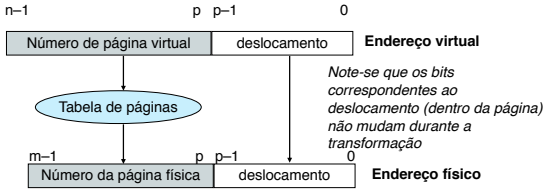


Transformação de endereços: Unid. Transf. (MMU – *Memory Management Unit*) converte páginas em frames físicos através da tabela de páginas (gerida pelo SO)

Transformação de endereços (2)

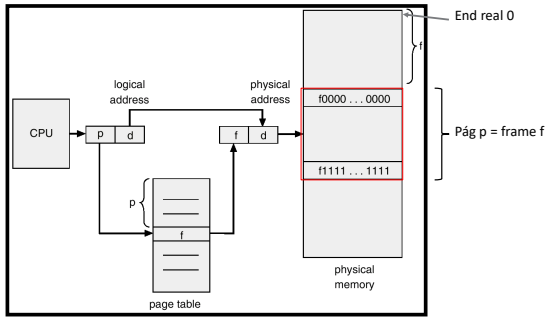
- Parâmetros:
  - p = 2^p = tamanho da página (usa p bits).
  - n = 2^n = espaço virtual (endereço tem n bits)
  - m = 2^m = espaço físico (endereço tem m bits)

Num página = Endereço / tamanho de página



Note-se que os bits correspondentes ao deslocamento (dentro da página) não mudam durante a transformação

Transformação de endereços (3)



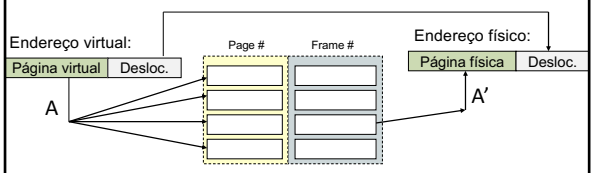
Implementar a tabela de páginas

- Onde se guarda a tabela de páginas de um processo?
- No caso do Pentium, end. 32bits e páginas de 4Kbytes:
  - tabela tem 2^{20} entradas (2^{32} /2^{12}); se cada uma tiver ~3 bytes, são ~3 Mbytes!
  - E há uma tabela de páginas por processo …
  - Não é tecnicamente viável guardar a tabela de páginas na MMU
- Solução:
  - Guardar a tabela de páginas em memória central

Implementação da tabela de páginas (2)

- Neste esquema cada acesso a dados ou código obriga a **dois acessos à memória**:
  - um para ler a tabela de páginas obtendo o endereço físico
  - e outro para ler/escrever o dado ou ler a instrução.
- O problema dos dois acessos à memória pode ser mitigado com uma **cache** da tabela de páginas:
  - ***Translation Look-aside Buffer*** (TLB)
  - *hardware* especial para consulta rápida, baseia em memória associativa

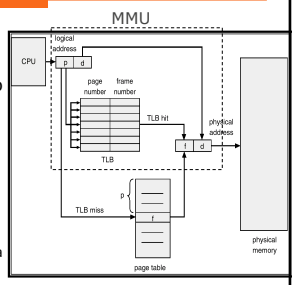
TLB – memória associativa para frames



- Transformação de endereços:
  - Relaciona números de páginas com números de frames
  - Se A está na memória associativa, TLB responde com o número da "frame" A'
  - Senão, temos um TLB miss…

MMU com TLB

- Transformação de endereços: A→ A'
  - Se A está na TLB responde logo com o número da "frame" A'
  - Se não está, obtém o número da "frame" da tabela de páginas na memória e atualiza a TLB
    - Podendo ter de eleger uma vítima na TLB para dar lugar à nova página



Tempo de acesso efectivo

- Se tempo de acesso à RAM é T unid. de tempo
- Se *TLB Hit rate* = α (percentagem das vezes em que o número da *frame* está num registo do TLB)
  - E se acesso à TLB desprezável
- Tempo de acesso efetivo (TAE)

  TAE = α T + (1 – α) 2T

  = (2 – α)T

Exemplo: para α = 90% e T=10ns

TAE = 11ns

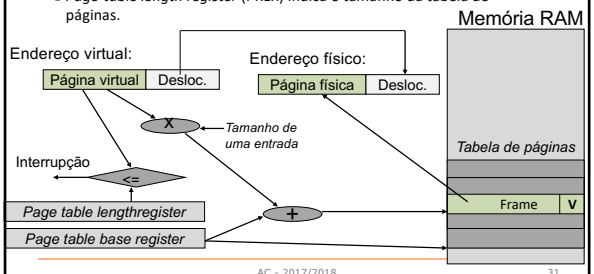
Neste exemplo os endereços virtuais e sua transformação está a custar 1ns por acesso

Proteção de memória por páginas

- A proteção de memória está associada ao preenchimento que o SO faz da tabela de páginas de cada processo
- A MMU tem de referenciar a tabela do processo em execução (controlado pelo SO)
- Mais informação pode estar associada à tabela ou a cada página. Exemplos:
  - Pode limitar o espaço de endereçamento limitando o tamanho da tabela de páginas
  - Cada página pode ter associada informação descrevendo os acessos permitidos (por exemplo: só leitura, pode executar)
  - Podem haver entradas inválidas → não mapeadas na memória real

Tabela de páginas com de bit validade

- Para cada processo:
  - *Page-table base register* (PTBR) aponta para a base da tabela.
  - *Page-table length register* (PRLR) indica o tamanho da tabela de páginas.



Páginas da imagem de um processo