

Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Escalonador do CPU

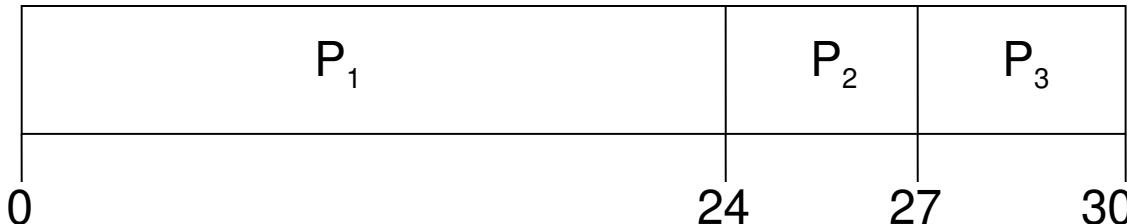
- Seleciona de entre os processos *Ready* qual o que irá ser executado no(s) CPU(s)
- Escalonador é activado quando o processo:
 - Muda do estado de *running* para *waiting*
 - Muda do estado *running* para *ready*
 - Muda do estado *waiting* para *ready*
 - Termina
- Os escalonadores que usam apenas 1 e 4 são designados *non preemptive*
- Escalonadores que usam 2 e 3 são *preemptive*

Escalonamento FCFS

- *First-Come, First-Served*

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Se os processos chegarem pela ordem 1, 2, 3, então:



- Tempo de espera: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Tempo médio de espera: $(0 + 24 + 27)/3 = 17$

Escalonamento SJF

- *Shortest Job First*
- Ordena os processos considerando a duração do próximo CPU burst. Executa primeiro os processos com CPU burst mais curtos
- Duas opções
 - *Nonpreemptive* – uma vez atribuído o CPU o processo fica em *Running* até terminar o CPU burst
 - *Preemptive* – se um processo entra na fila de *Ready* com um CPU Burst menor do que o tempo restante do CPU burst do processo em execução, atribuir o CPU ao processo que entrou em *Ready*. Também conhecido como *Shortest-Remaining-Time-First* (SRTF)
- SJF é óptimo do ponto de vista do tempo médio de espera de um conjunto de processos

Escalonamento por prioridades

- *Priority scheduling*
- É associado um nível de prioridade (inteiro) com cada processo
 - Não existe acordo sobre se a prioridade mais alta corresponde a valores baixos ou altos do nível de prioridade
 - Iremos assumir que números baixos representam maior prioridade
- O CPU é atribuído ao processo com maior prioridade
 - Preemptive
 - Nonpreemptive
- SJF é um caso particular de escalonamento por prioridades
- Problema: Adiamento indefinido
 - Processos com prioridade baixa podem nunca executar
- Solução: Contar com o tempo de espera (*aging*)
 - Aumentar a prioridade dos processos em espera à medida que o tempo passa

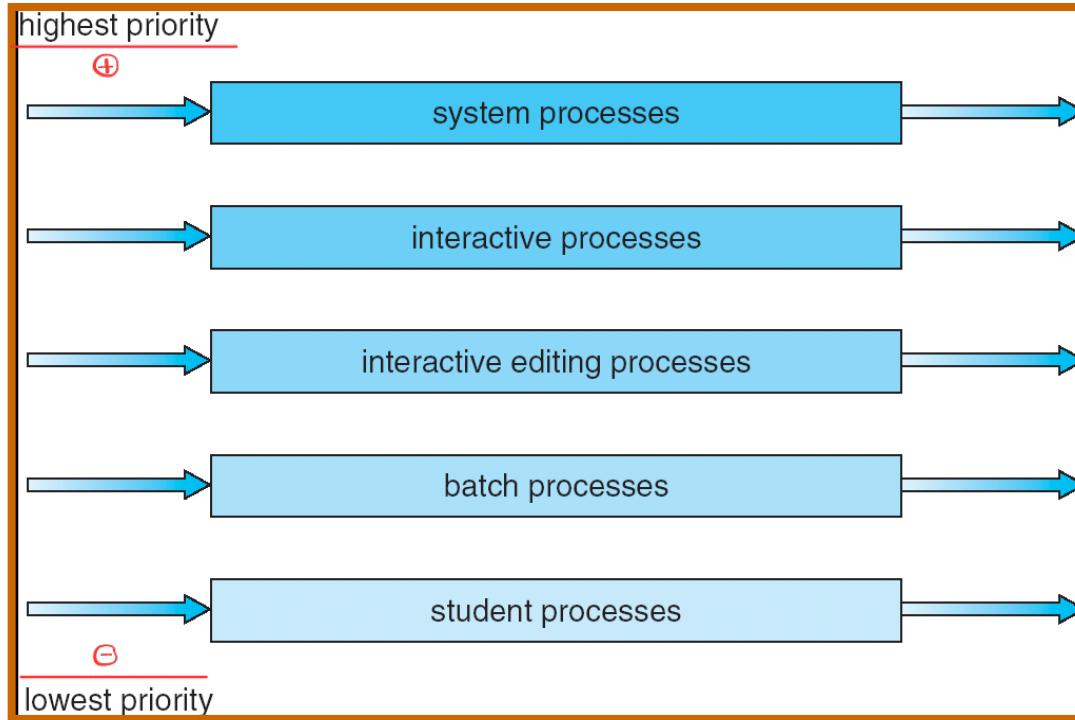
Round Robin

- Versão *Time sharing* e *preemptive* de FCFS
- Cada processo pode usar o CPU, no máximo, por determinado tempo (*time quantum*). Se o processo não bloquear antes do tempo definido é retirado da execução e passa para o fim da lista de Ready
 - *Time quantum* varia, em geral, entre 10 e 100ms
- Se existem n processos na fila de *Ready* (nenhum em execução) e o *time quantum* é q então:
 - cada processo usa cerca de $1/n$ do processador
 - Um processo nunca espera mais do que $(n-1).q$ unidades de tempo
- Desempenho
 - Q grande \uparrow FCFS
 - Q pequeno \uparrow o overhead da mudança de contexto pode ser significativo

FIFO multi-nível

- **Multilevel queue** → vários queues por níveis
- Fila de *Ready* é dividida em 2:
 - *Foreground* (interactiva) → modo interativo
 - *Background* (batch) → modo não interativo (e.g.: resultado guardado em ficheiro)
- Cada Fila tem pode ter a sua política de escalonamento
 - *Foreground – RR* → Tempos de resposta certos! //
 - *Background – FCFS* → minimizam o overhead da mudança entre processos!
- Escalonamento entre as 2 filas
 - **Baseado em prioridades fixas** → Processos em "foreground" têm @ prioridade! //
 - Só executar processos em background se fila Ready de *foreground* estiver vazia
 - **Divisão do tempo (Time slice)**
 - Cada fila tem um certo tempo de CPU disponível (Ex: 80% RR 20% FCFS)

FIFO multi-nível



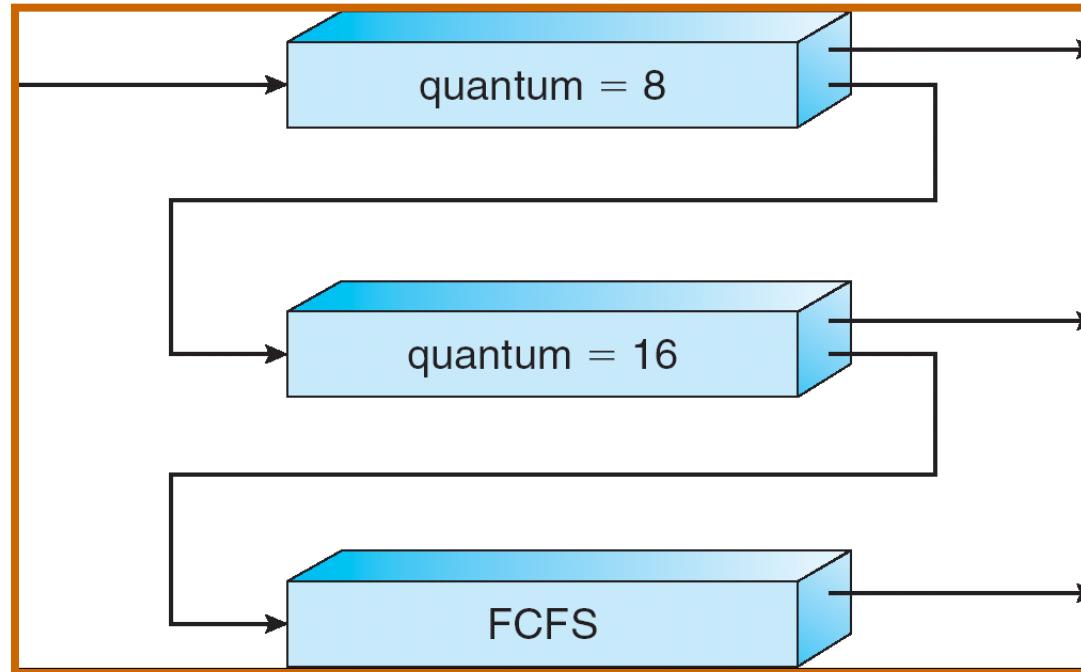
- *Multilevel Feedback Queue* ↗️ eles movem-se entre os vários filhos!,,
- Um processo pode mover-se entre as várias filas
- Parâmetros
 - Número de filas
 - Algoritmo de escalonamento de cada fila
 - Algoritmos de elevar e descer a prioridade de um processo => Quando é que mudamos de uma fila para a outra ...
 - Algoritmo de atribuição da prioridade inicial de um processo

Exemplo:

- Três Filas:
 - Q_0 – RR com *time quantum* 8 ms
 - Q_1 – RR com *time quantum* 16 ms
 - Q_2 – $FCFS$
- Escalonamento
 - Um novo processo começa em Q_0 . Se esgota os 8ms antes de bloquear, passa para Q_1
 - Em Q_1 , se o processo ao executar esgota 16ms antes de bloquear passa para Q_2



FIFO multi-nível com realimentação



Linux scheduler

→ O SO define o escalonamento!

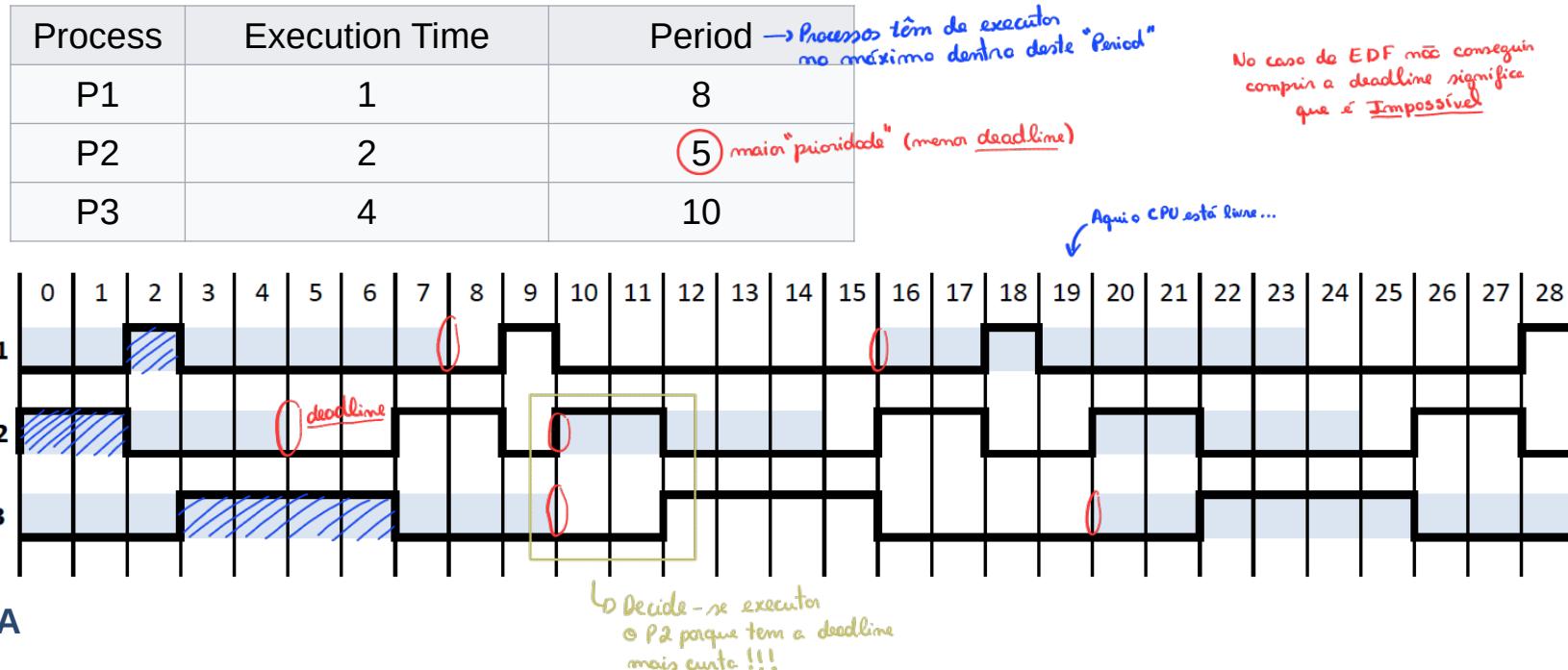
- o Linux considera três classes “clássicas” de *scheduling*, cada uma incorporando prioridades múltiplas, que, quando ordenadas por ordem decrescente de prioridade, são:
 - SCHED_FIFO** → processos de tempo real (“nonpreemptive” executam até terminarem ou irão para waiting)
 - só utilizados em super user
 - As vezes podemos ver utilização para garantir condições temporais...
 - SCHED_RR** → processos de tempo real (tem um time quantum associado)
 - Processos “normais” de utilizador
 - SCHED_OTHER** – classe formada pelos processos restantes, o processador só é atribuído a processos desta classe se não houver outro tipo de processos prontos a serem executados;
- as classes **SCHED_FIFO** e **SCHED_RR** estão associadas a processamento de tempo real e a processos de sistema e o valor das suas prioridades é fixo;
- a classe **SCHED_OTHER** está associada aos processos utilizador;

Linux scheduler

- Foram recentemente incorporadas no Linux novas classes de scheduling:
 - **SCHED_DEADLINE** – classe formada por *threads* de tempo real; para cada thread são indicados: período, deadline relativa e tempo de computação; escalonador usa algoritmo Global Earliest Deadline First; disponível desde kerner 3.14 (Março 2014)
 - **SCHED_BATCH** – escalonador assume que processos nesta classe são cpu-bound; usa *timeslices* maiores; aplica penalty quando processo acorda.
 - **SCHED_IDLE** – classe formada por processos de muito baixa prioridade; o valor nice não tem efeito neste processos
 - prioridade
 - utilizado também no SCHED_OTHER

Earliest Deadline First

- Escolhe para execução sempre o processo que tem a *deadline* mais próxima *→ Funciona com base em deadlines...*
- É óptimo do ponto de vista de que se é possível correr um conjunto de processos (com tempo de chegada, tempo de processamento e deadline) de forma a todos cumprirem as deadlines, o EDF cumpre as deadlines



Linux scheduler

- a partir da versão 2.6.23 do *kernel*, o Linux passou a usar um algoritmo de scheduling para a classe **SCHED_OTHER** conhecido pelo nome de *justiça total (total fairness)*;
- assume-se um processador ideal que tem tantos elementos de processamento quantos os processos que correntemente coexistem; assim, se existirem N processos, o processador executa-los-á em paralelo distribuindo a potência de cálculo por todos eles de um modo uniforme (a velocidade de processamento será 1/N da velocidade se existisse um só processo em execução);
- o algoritmo vai procurar modelar este comportamento num processador real;
- são definidas duas variáveis associadas a cada processo:
 - *tempo de execução virtual* – tempo de execução associado ao processo se ele estivesse a ser executado no processador ideal;
 - *tempo de espera* – tempo real que o processo aguarda na fila de espera dos processos prontos a serem executados pela atribuição do processador;

Linux scheduler

- o *scheduler* organiza os processos numa fila de espera dos *processos prontos a serem executados* única e calendariza para execução o processo cuja diferença entre o *tempo de espera* e o *tempo de execução virtual* é maior, procurando desta forma minimizar o grau de injustiça existente;
- sempre que um processo é calendarizado para execução, o seu *tempo de espera* é decrementado do valor correspondente à janela de execução, o dos restantes presentes na fila de espera é incrementado do mesmo valor, e todos eles tem o *tempo de execução virtual* incrementado do valor de execução virtual;
- os processos bloqueados mantêm os valores destas variáveis inalterados e não entram naturalmente no esquema de seleção enquanto não forem acordados;
- o algoritmo de *scheduling* usa, pois, como elemento central de decisão, a história passada de execução do processo, não havendo propriamente uma distinção entre os processos I/O-intensivos e os processos CPU-intensivos.

- **Runqueue**

- 1 por CPU
- Representa as *runnable tasks* desse CPU
- Tem 2 *priority queues*
 - Active e Expired

- **Priority Array**

- Desempenho $O(1)$
constante
- Permite acesso a tarefa com prioridade mais alta
- Tarefas com mesma prioridade são servidas por ordem

Não depende do n.º de processos que está a escalonar?

Tópico prático: comando **nice**

- Através do comando **nice** um utilizador pode baixar a prioridade a um processo
 - Executar **program** através de:
nice -20 program
- O comando **nice** não tem grande efeito enquanto o número de processadores for suficiente para a execução de todas as tarefas/processos

Só tem efeito quando
→ esgotarmos o número
de CPU's

Memória Virtual

(Saber Teoria (objetivos + Segurança) para o teste)

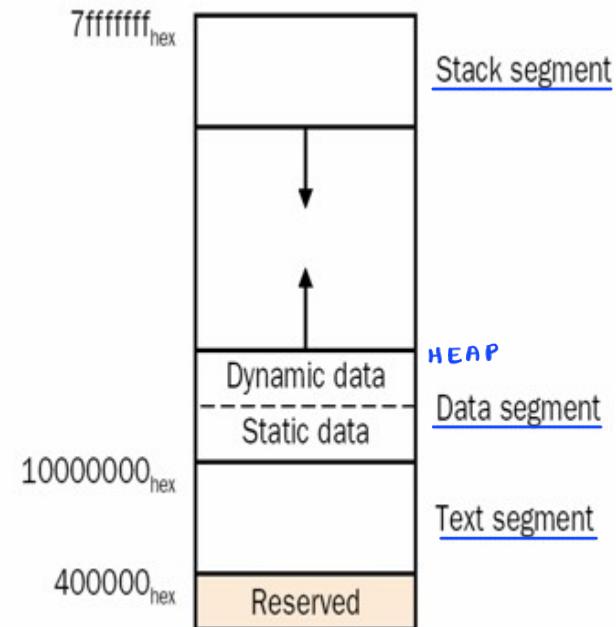


Sabem Isto !!!

- Eficiência da utilização da memória
 - Memória deve ser partilhada pelos processos
 - Manter em memória apenas o necessário
 - Endereços usados pelos processos não são endereços da memória física
→ Endereço 0x100 na memória virtual pode não ser o 0x100 na memória física (normalmente não é!)
(apenas o só pode utilizar memória física)
- Segurança
 - Mecanismos de segurança que impeçam que um processo altere as zonas de memória dos outros processos.
- Transparência
 - Processo tem acesso a muita memória (eventualmente mais do que a memória física)
virtual!
 - Processo corre como se toda a memória lhe pertencesse
- Partilha de memória
 - Vários processos acedem à mesma zona de memória (de forma controlada)
No caso de pretendermos...

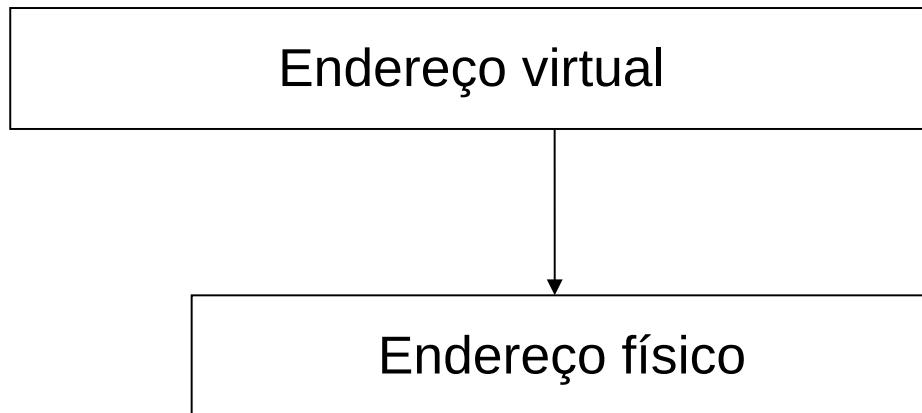
Memória virtual

- Cada processo corre num espaço de endereçamento virtual (igual para todos)
 - Os endereços usados pelos processos e os endereços físicos que lhes correspondem podem ser distintos
 - Os endereços que o processo usa são virtuais, o mesmo endereço virtual de 2 processos pode corresponder a endereços físicos distintos
 - Os endereços da memória virtual têm de ser convertidos em endereços de memória física
 - Alguns endereços de memória virtual podem estar armazenados em disco
- ↳ Coisas que não são muito utilizadas ...*



Mapeamento Virtual-Físico

- É necessária a conversão (rápida) do endereço virtual para o endereço físico



Memória física como cache

- O conteúdo dos endereços virtuais pode ser armazenado em disco

Endereço virtual

0:	0x10
4:	0x3
8:	0x33
12:	0x21

O processo nem sabem!

Endereço físico

0:	0x21
4:	0x3

Disco

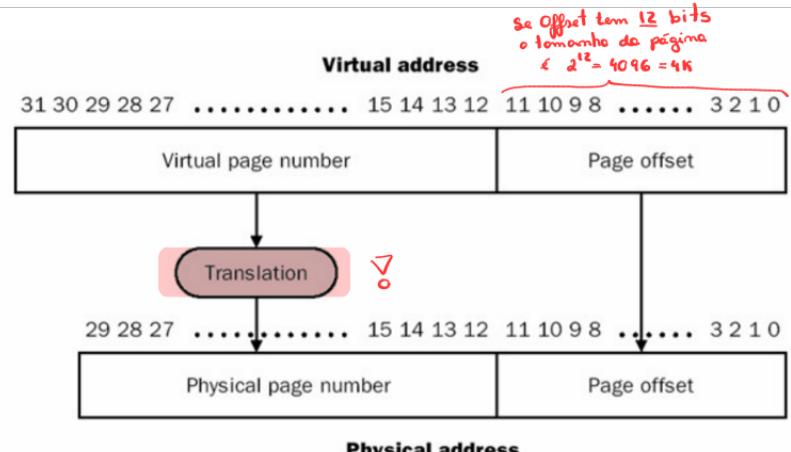
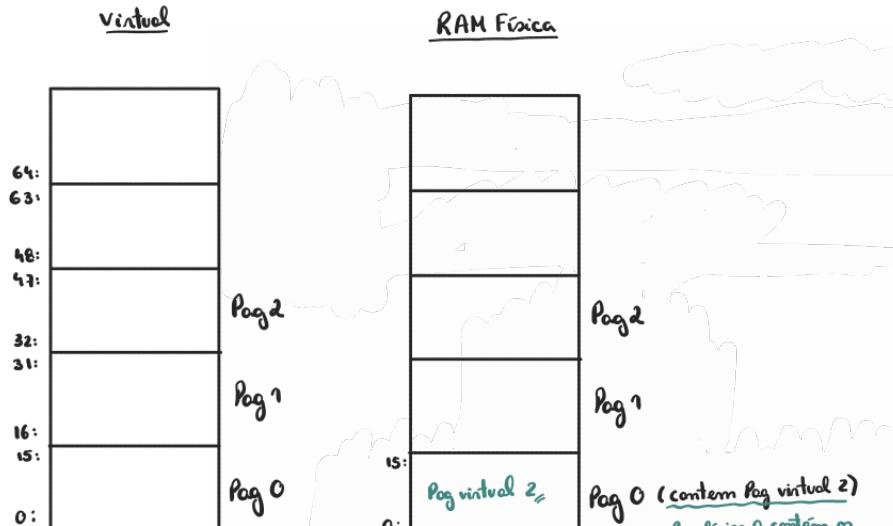
0x33
0x10

- Páginas podem estar em disco
- Páginas podem estar em memória física

Endereço virtual paginado

- Nº página virtual
- Offset no interior da página

Tamanho da página?



Paginação e Segmentação

• Paginação

- Memória dividida em páginas
- Páginas têm tamanho fixo
- Fragmentação interna \Rightarrow Cada página só pode pertencer a um processo... (e por vezes fica memória desperdiçada)
- Um processo necessita de um certo número de páginas livres

Atenção !

→ Um processo pode ter associado um conjunto grande de páginas

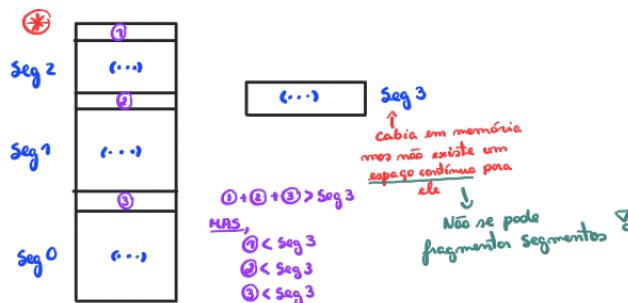
} = guardados em "page tables"



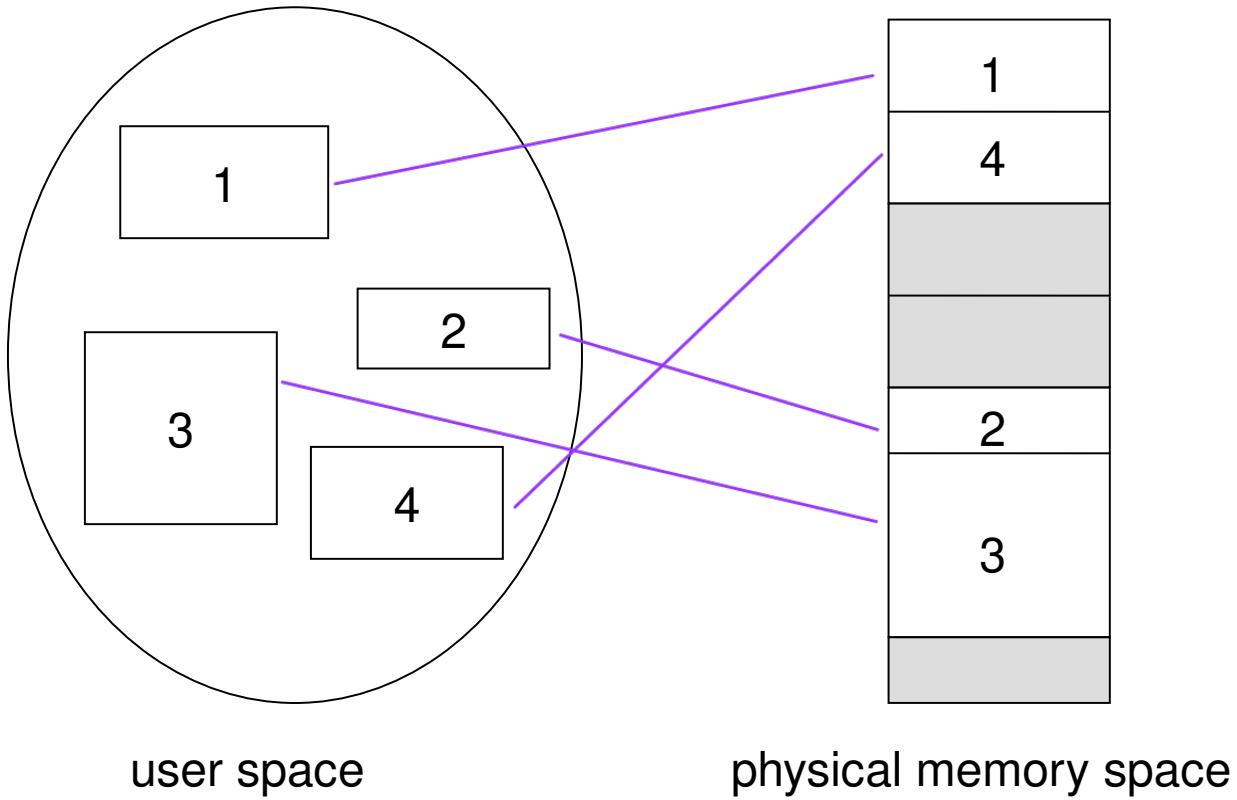
• Segmentação

- Memória dividida em segmentos
- Segmentos podem ter tamanho variável
- Fragmentação externa *
- Um processo necessita de uma zona contígua de memória livre para cada segmento

Página
→ tamanho fixo Segmento
→ tamanho variável



Segmentação



Paginação e Segmentação

	Página	Segmento
Palavras por endereço	1 (<i>Nº da página : offset</i>)	2 (seg. e offset)
Visível ao programador	Não <i>modelo uniforme ...</i>	Depende <i>por vezes o segmento tem de ser escolhido de forma explícita</i>
Substituir um bloco	Trivial	Difícil <i>?</i>
Utilização memória	Fragmentação interna <small><i>Memória perdida → pois os páginas têm tamanho fixo</i></small>	Fragmentação externa <small><i>A excesso das espacamentos livres pode ser grande risco o espaço livre pode ser muito pequeno para 1 processo</i></small>
Acesso ao disco	Eficiente	Menos eficiente <small><i>Segmentos de tamanho variável ...</i></small>

Características de memória paginada

- O custo de um **page fault** é muito elevado
 - Quando é preciso, pela primeira vez, associar o endereço da memória virtual para a memória física.
- Páginas são relativamente grandes para amortizar o tempo de acesso
 - 4KB a 64KB
- Usar uma organização completamente associativa → não existem critérios pré-definidos
 - Qualquer Pág. Virtual pode ser associada a qualquer Pág. Física
- *Page faults* são tratados por software pois o atraso principal é o acesso ao disco.
 - Algoritmos mais sofisticados
- **Write-back**
 - Sempre que escrevemos numa página virtual esse valor também é escrito na página física correspondente (levando atraso no DISCO)
 - só quando a página tiver de sair da mem. física ...

Encontrar uma página

- **Organização completamente associativa**
 - Página virtual pode ser mapeada em qualquer página física
 - Reduzir *page faults*
- Não é possível usar procura associativa
- Tabela relaciona página virtual com a sua posição
 - *Page table*
 - Indexada por nº página virtual
 - Entrada indica posição real da página virtual
 - **Cada processo tem a sua tabela de página**

Pág. Virtual	Pág. Física
...	...
...	...

→ Um processo pode usar vários páginas...

e.g.:
Cache associativa
 - podemos aceder ao mesmo tempo por hardware se ela lá está

e.g.:
 P_1 : Pág. VO \rightarrow Pág. F1
 P_2 : Pág. VO \rightarrow Pág. F4
 Transparência !

Tabela de página (Data Structure)

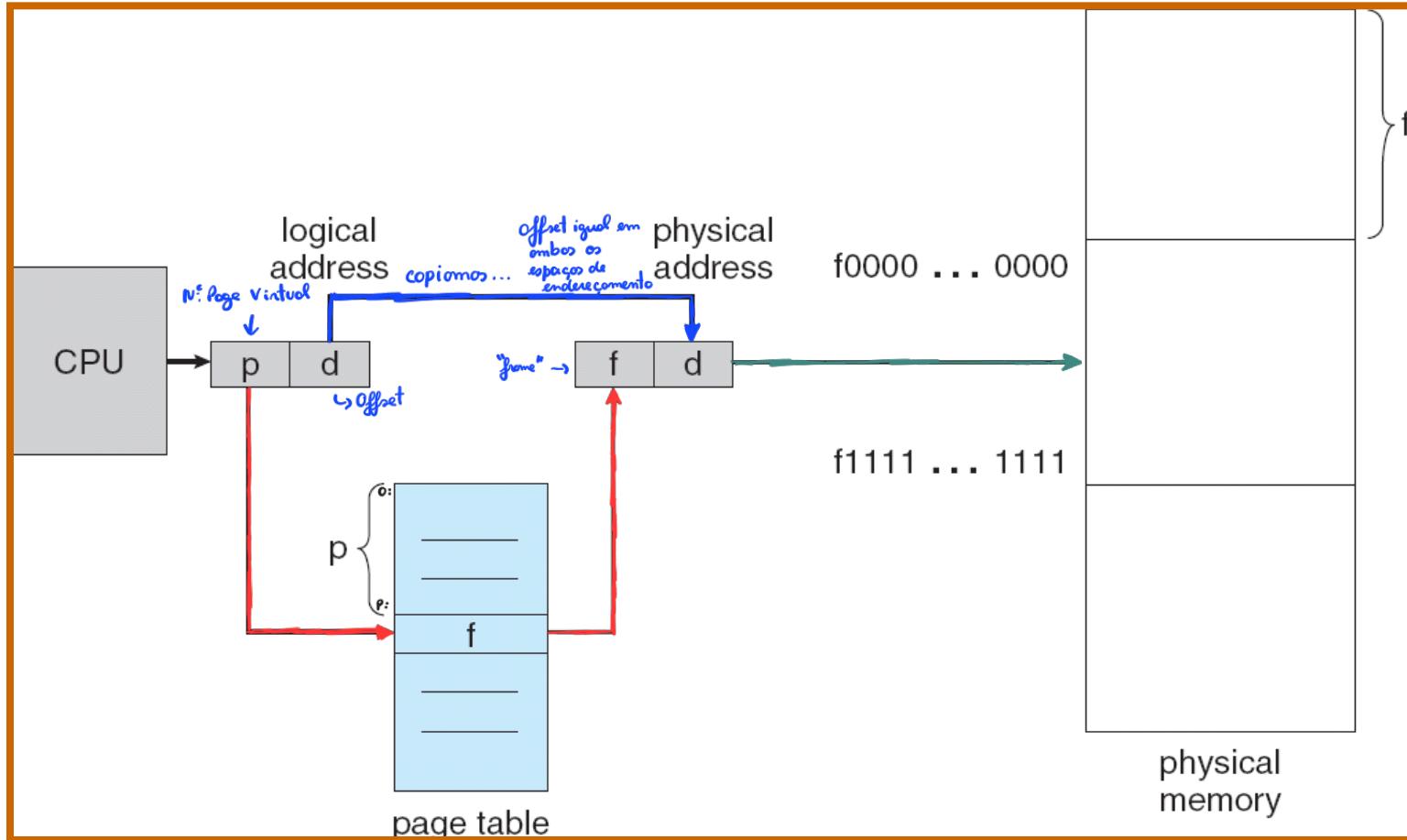
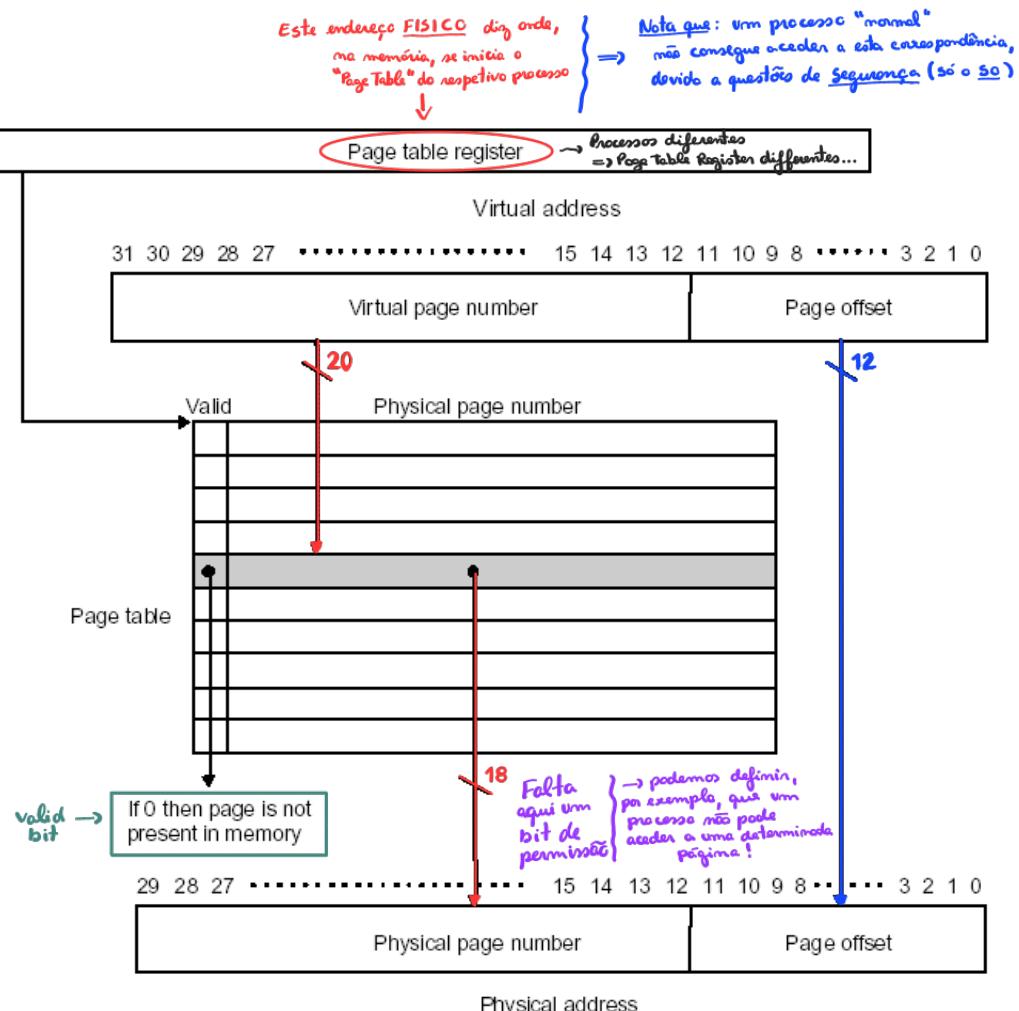


Tabela de página

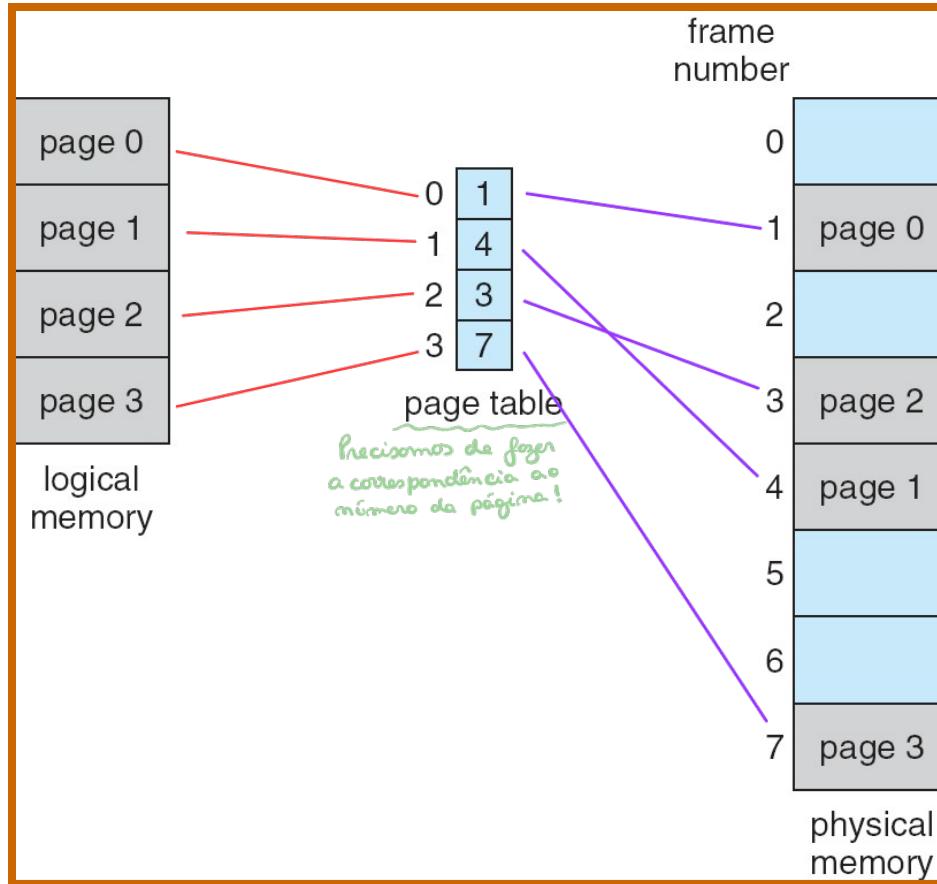


- Se o valid bit for 0 temos um "page fault" → temos de ir buscar ao disco
- Se o valid bit for 1 podemos proceder

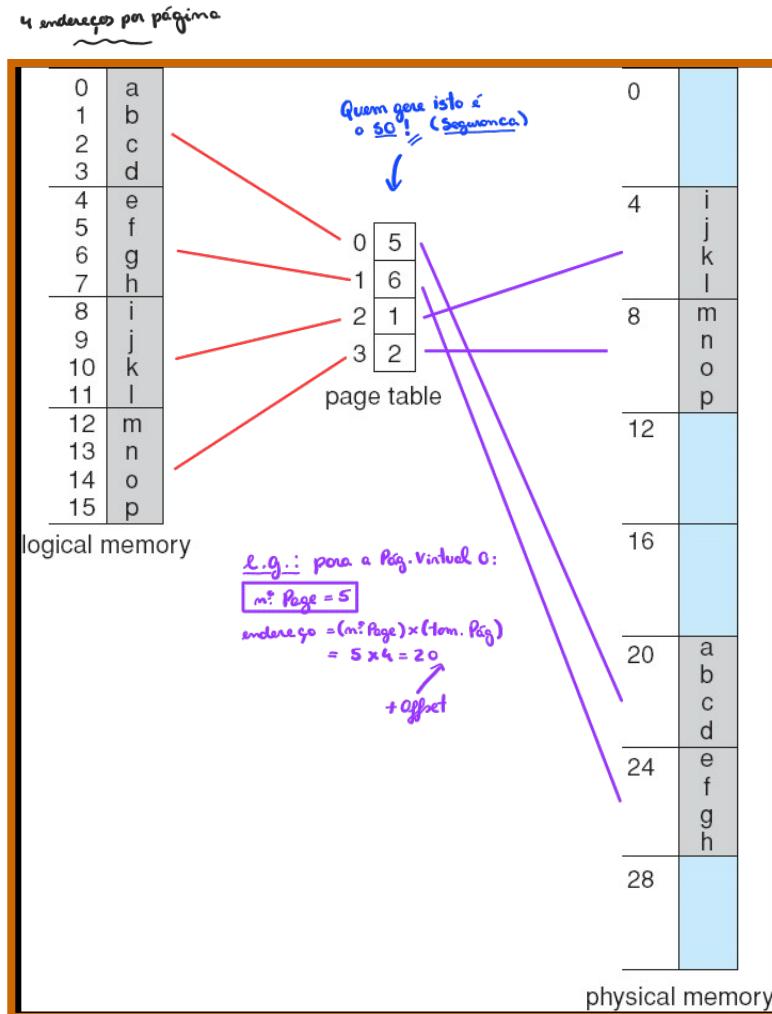
- Registo de tabela de página
- *Valid bit*
- Permissões

• Ainda podem conter "reference bit"
 → utilizado para ver a periodicidade com que aquela página é utilizada...

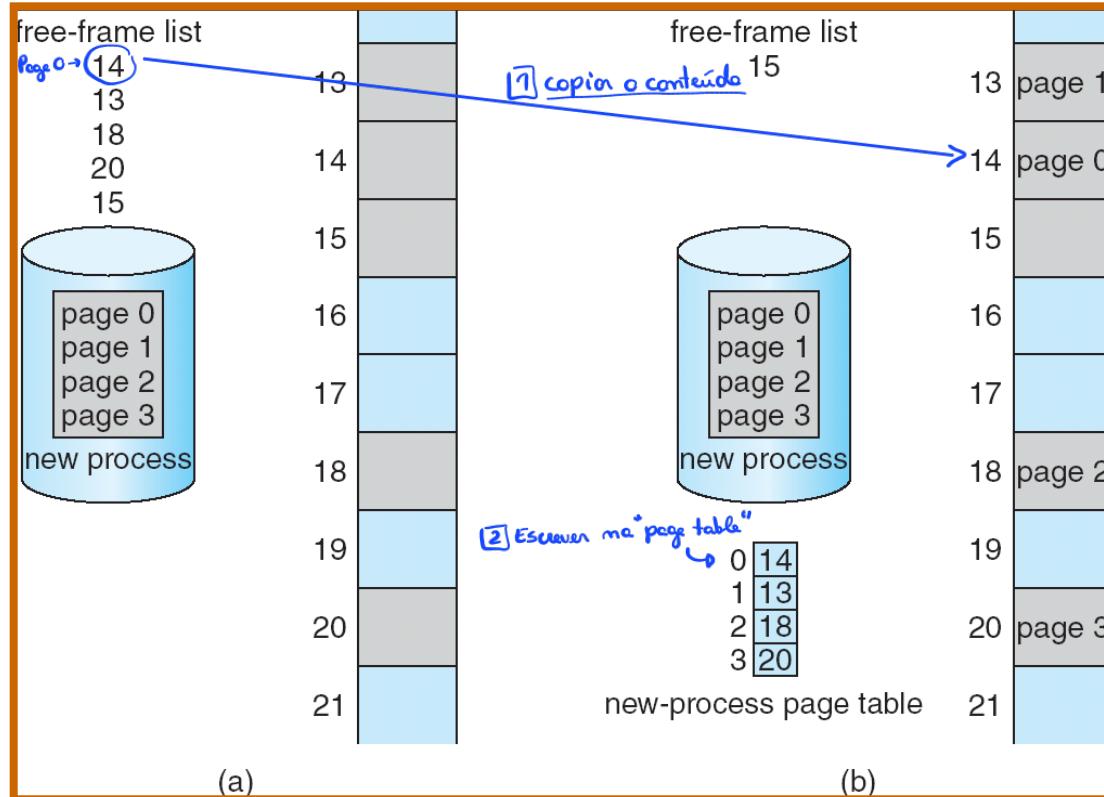
Endereços lógicos e físicos



Endereços lógicos e físicos



Endereços lógicos e físicos



Before allocation

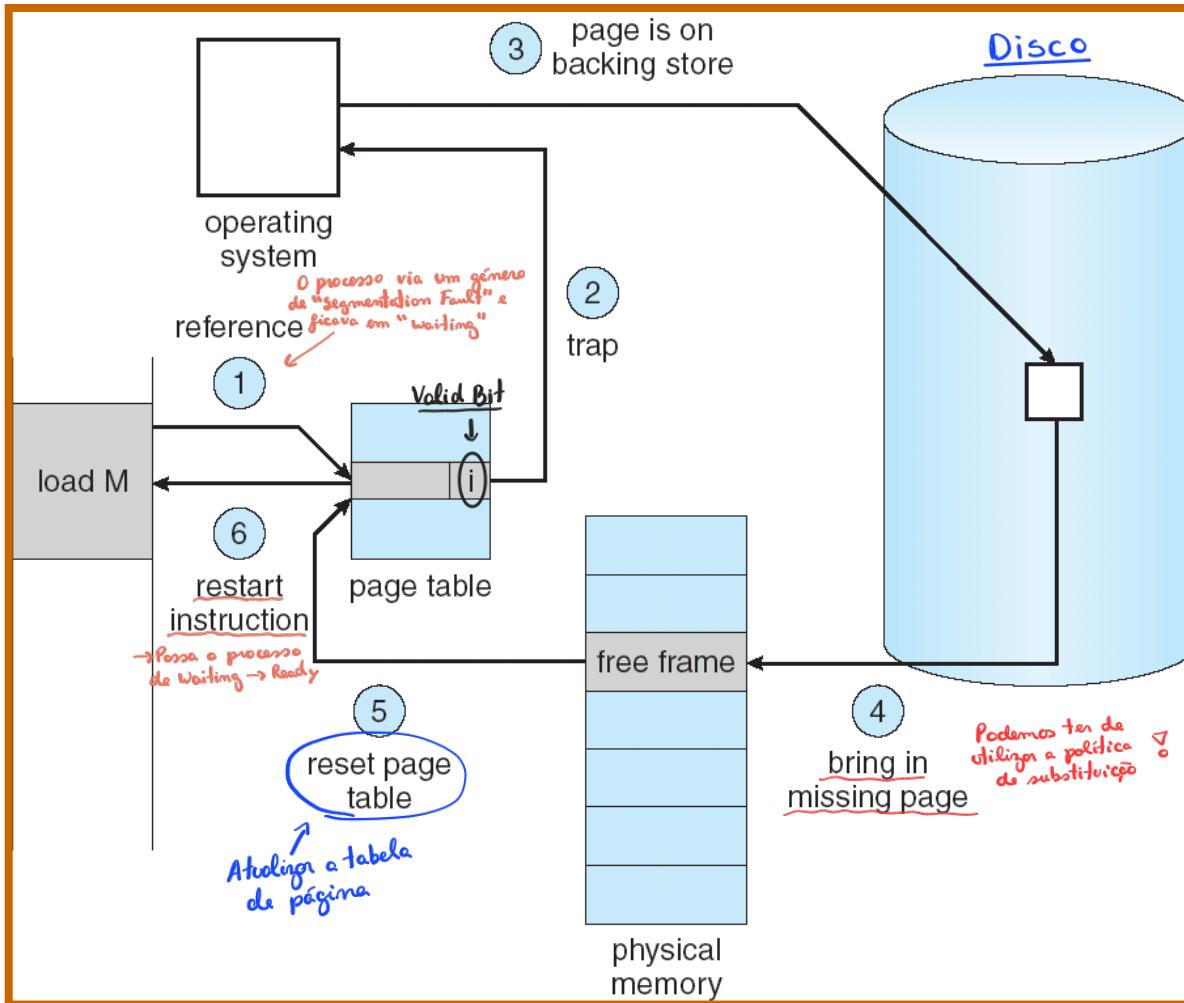
After allocation

Page fault

→ Gerado um endereço virtual que
não está mapeado nesse instante
na memória física ...

- **Valid bit da tabela de página com 0** ← 0 so deve garantir isto!
- Tratado pelo sistema operativo
- Estrutura mantém posição das páginas virtuais em disco → Agora iria copiar do disco para a memória física! → E quando não houverem mais páginas livres? → termos de tirar o mapeamento de uma das páginas físicas ... → termos de associar agora a 2 páginas virtuais ...
- **Política de substituição tipo LRU aproximado** → Least-recently used (nunca de memória a página que não é utilizada a mais tempo!) → Mas neste caso é uma aproximação!
 - Tabela de páginas contém reference bit
 - Periodicamente reference bits colocados a zero
 - Se página é acedida (*touched*) reference bit a 1
 - Substituir páginas com reference bit a 0

Page fault



Tamanho da tabela de página

- Considerem um sistema com endereços de 32 bits, páginas de 4KB e 4 bytes por entrada na tabela de página. Qual o espaço ocupado pela tabela de página?

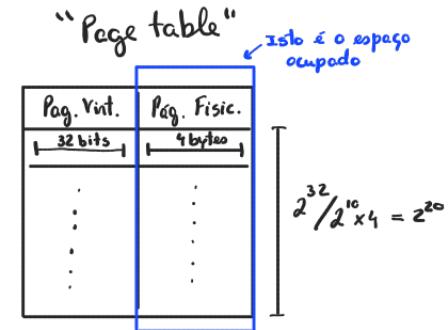
“São endereços onde o Offset = 0!.”

Endereço: 0xXXXXX XXXXX → Número de páginas: $\frac{2^{32}}{4\text{KB}} = \frac{2^{22}}{4} = 2^{20}$

4KB = $4 \times 2^{10} = 4096\text{bytes}$

4 bytes por entrada de página → Tamanho da Tab. de Páginas: $2^{20} \times 4 = 4\text{MB}$

Quatro bytes por entrada.



Nº de entradas = $2^{32} / 2^{12} = 2^{20}$

Tamanho da tabela = $2^{20} * 4 = 4\text{MB}$

100 processos implicaria 400MB de memória usados em tabelas de página!

→ Isto são estruturas auxiliares ...

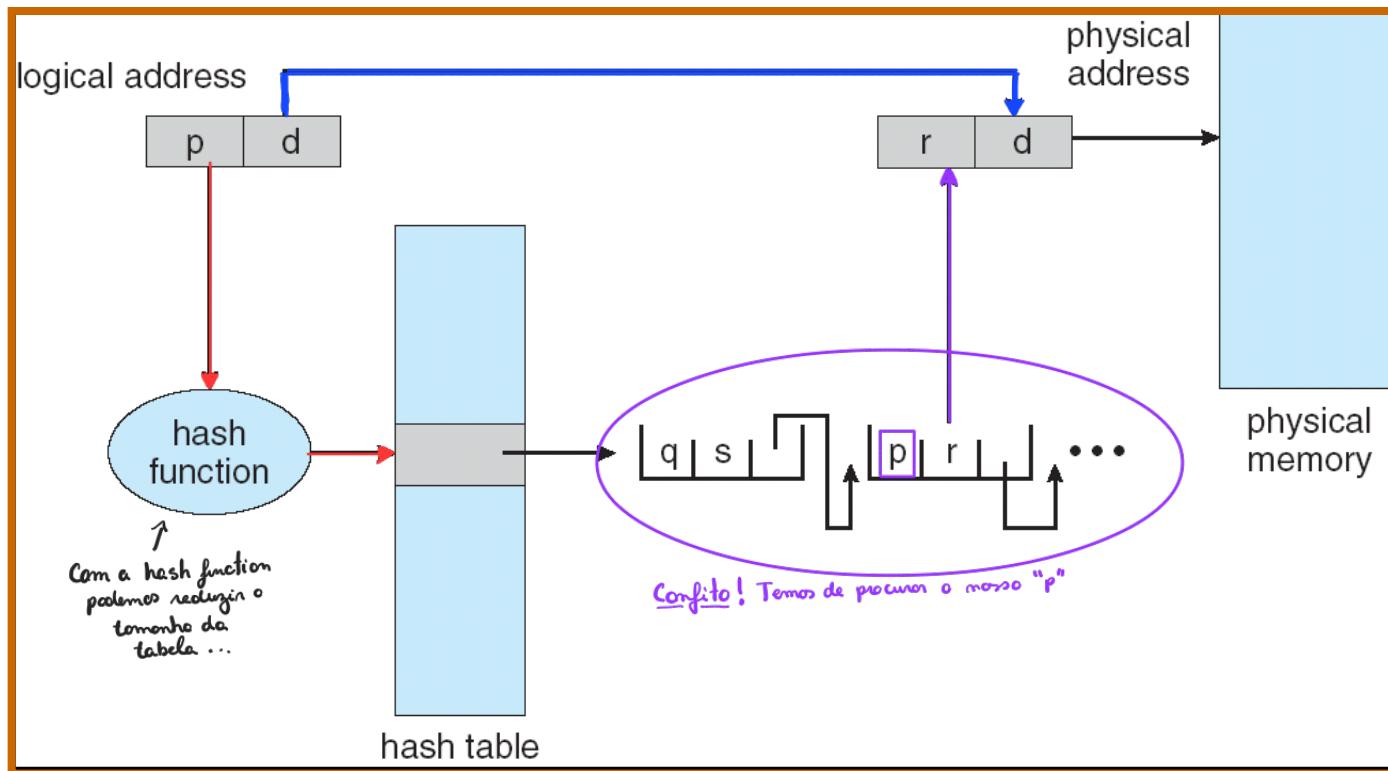
Tamanho da tabela de página

- Técnicas para reduzir tabela de página
 - Registo limita o tamanho da tabela
 - Memória associada ao processo cresce num só sentido
→ Digo que não preciso de tantos bits para endereçar ...
→ Mas a stack costuma crescer de cima para baixo ...
 - 2 tabelas de página por processo com 2 registos limite
 - 2 segmentos (*stack* e *heap*) crescem em direcções opostas
 - Hashing
 - Tabela de página depende do tamanho da memória física
 - Tabelas de página com vários níveis
 - Tabelas de página em memória virtual!



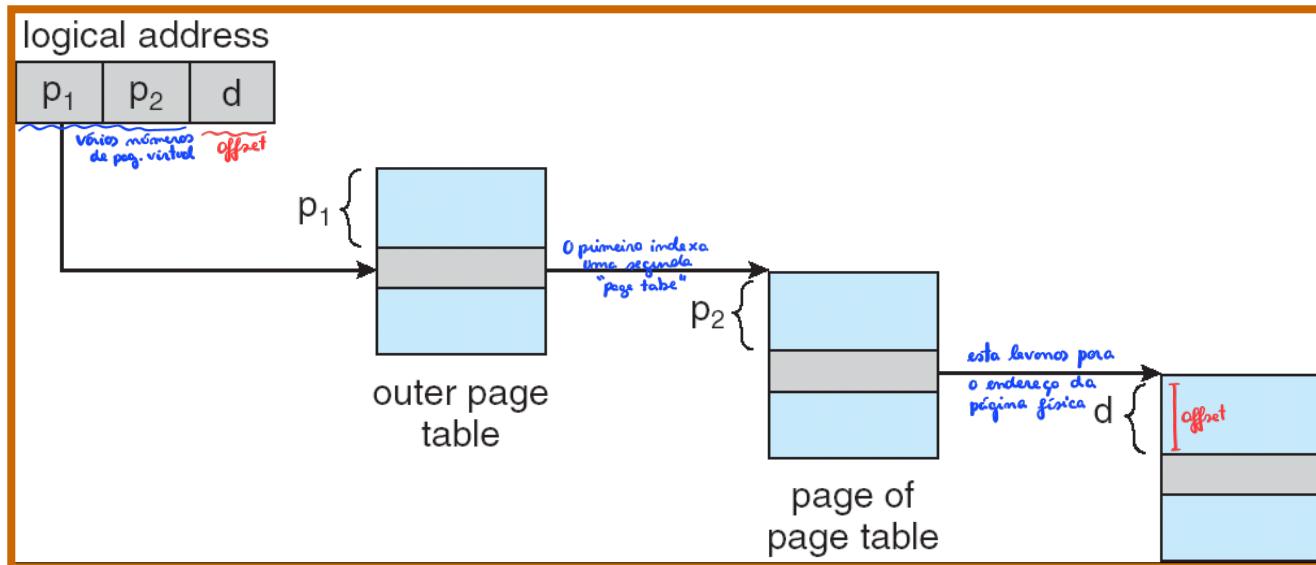
Tabelas de página com *hashing*

- Função de *hash* determina entrada a ser usada
- Na mesma entrada podem co-existir várias traduções



Tabelas de página com vários níveis

- 2 níveis



Permite redimensionar!

→ alguns valores de p_1 não têm uma segunda tabela associada.

⇒ Tabela P_2 pode ser mais pequena
(...)

Política de escrita

- Write-through não é eficiente

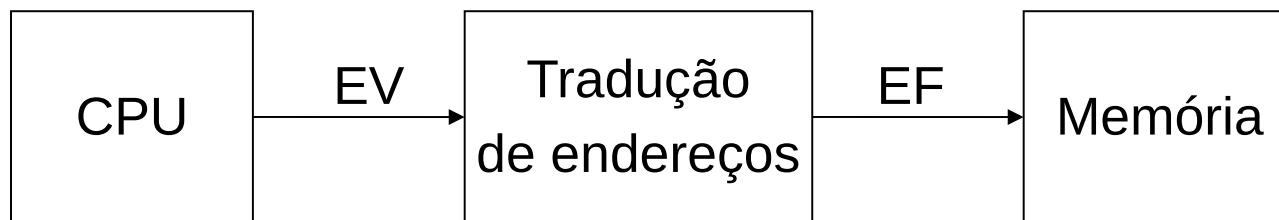
escrever na memória física ⇒ escrever no disco
- Write-back →
 - Escrita em disco página a página
 - Página só é escrita em disco quando necessita de ser retirada de memória física...
- Dirty bit

→ É ou não necessária fazer a cópia?
 • se o que está na Pag. Físic. é igual ao que está no Disco ele mete o Dirty Bit = 0 // Quando o processo escrever nequela página → Dirty Bit = 1

Isto serve para não estarmos a copiar uma coisa que está igual
- Write allocate
 - Escrita numa página que não reside em memória física, carrega essa página para a memória física e escreve
 - 1 → Mem. Física
 - 2 → Escrever em Disco

Problema

- Memória virtual obriga a 2 referências à memória para aceder a um endereço virtual!
 - Tabela de página reside em memória
 - Acesso à tabela de página
 - Acesso à memória física
- Para cada acesso à memória teremos 2 acessos à memória física !!!*
- Com base no resultado inimos ao outro ...*



Solução

- Os acessos à memória virtual contêm localidade espacial e temporal
 - *e.g.: telemóvel: lista de chamados mais recentes*
 - *O acesso à lista de endereços não é uniforme*
 - *está tudo seguido*
 - *l-o-g: um acesso a um endereço é um pedaço de acesso com localidade espacial (normalmente está seguido)*
- Cache de endereços traduzidos recentemente
- **TLB – Translation-lookaside buffer**
 - ↳ *Memória associativa com procura paralela!,,*
 - *Já é definida em Hardware, designada por: TLB*
- Entrada do TLB pode incluir *tag*, página física e *reference*, *dirty*, *access bits*
- Em cada acesso o TLB é verificado
 - *Hit* - continua
 - *Miss* - verificar a tabela de página
 - Podem originar (ou não) *page faults*
 - Podem ser tratados por hardware ou software

TLB – Memória Associativa

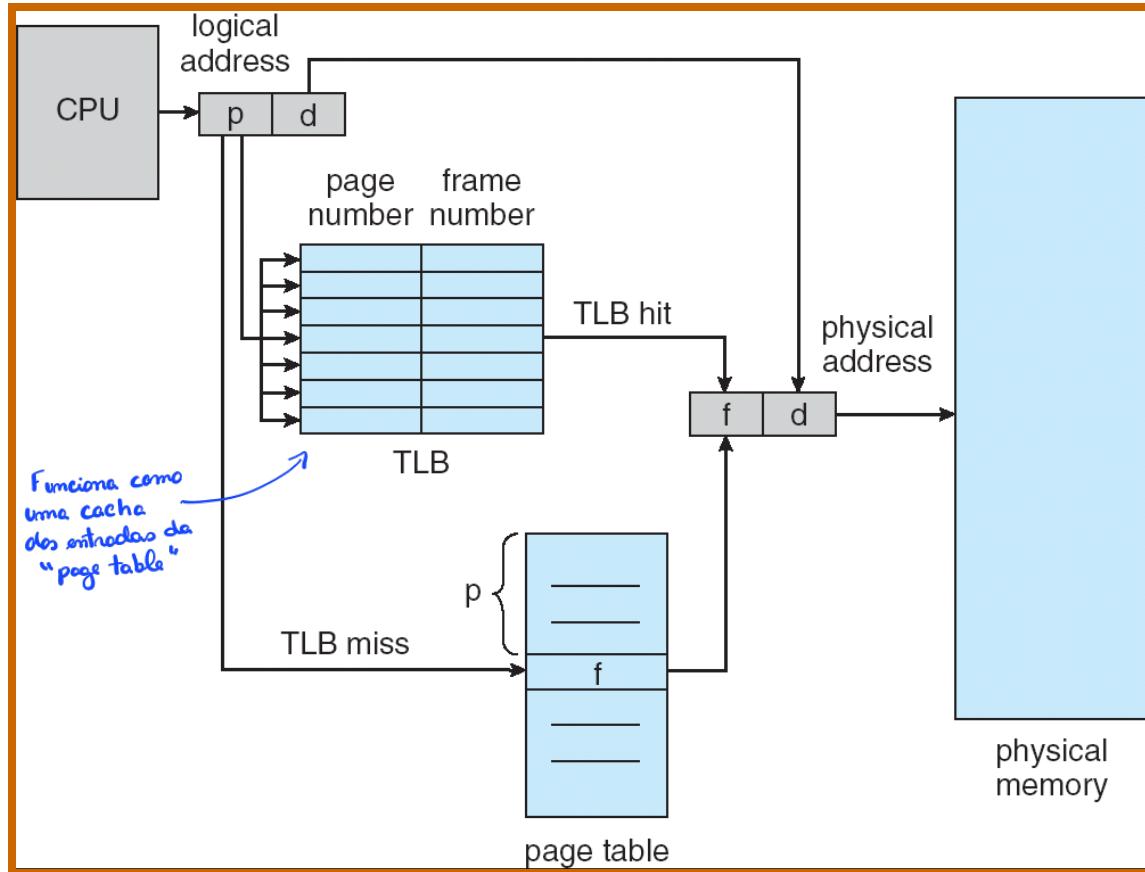
- TLB é uma memória associativa com procura paralela

Page #	Frame #

Tradução de endereços

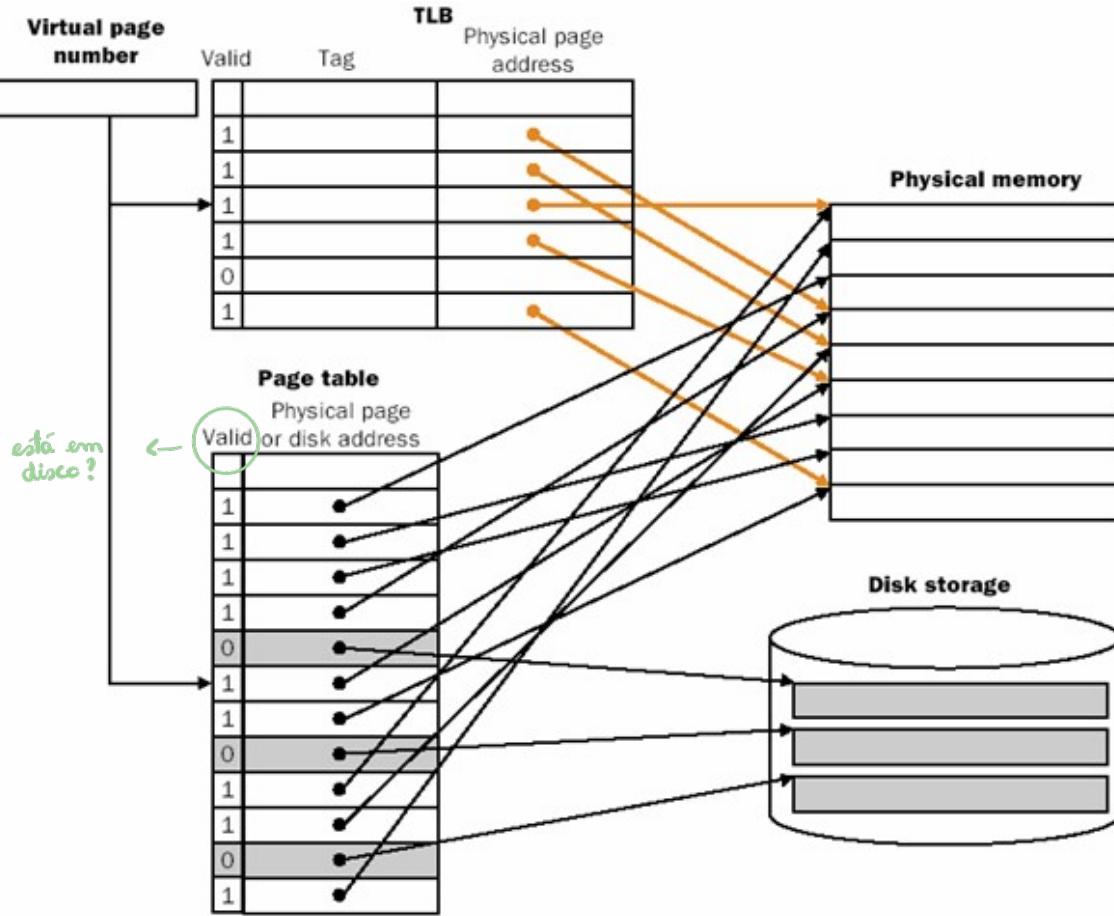
- Se p está na tabela associativa, determina # frame no TLB
- Senão procura # frame na tabela de página

Memória virtual com TLB



- TLB funciona como cache das entradas da tabela de página
- Páginas residentes em disco não são referenciadas no TLB *→ No TLB só estão endereços residentes em memória física*
- Tag, valid, etc

Memória virtual com TLB



- TLB funciona como cache das entradas da tabela de página
- Páginas residentes em disco não são referenciadas no TLB
- Tag, valid, etc

Tem ligação?
Está a ser usada?