

DIA 06 DE MARÇO

TURMA 2023/2024 - MATERIAIS

06 / 08:30 / 240

gerência de memória

- declarar uma variável como static armazena ela no DATA (dados estáticos), que não crescem de tamanho.

$$2^0 \cdot 2^0 \cdot 2^0; \text{ ex: } 16\text{GB} \rightarrow 2^4 \cdot 2^{30}$$

K M G

- int tem 32 bits (4 bytes)

- o kernel trata de proteção de memória; alocações também.

↳ deve ter memória dedicada só para ele (código, pilha ...)

- modelo 0 → em memória.

- single process; cada endereço usa 1 byte (8 bits)

- código: static int x = 0; compiler LOAD Adds rgs

x++;

- máquina de 16k endereços (do 0 ao 16k - 1)

- kernel usa 1024 endereços (do 0 ao 1k - 1)

- processo 0 pode usar os 1024 ao 16k - 1

- problema: por só armazenar em memória 1 processo por vez, fica muito lento o processamento, porque a cada troca de processo tem que ler/armazenar no disco. // SWAP OUT; SWAP IN

↳ mau desempenho.

modelo 1

- many process (in memory)

- fixed partition // cada processo tem tamanho fixo, o que torna o código muito mais simples para o programador do kernel, mas ruim para o usuário → pode haver desperdício de memória.

fragmentação interna!

- para reaprovar ao compilador de ms: cada processo deve rodar sempre na OBS: espaço de endereçamento diz respeito a todos os endereços que um processo pode usar.

- problema de tradução: as execuções/instruções de máquina dependem de onde o processo está alocado.

- nem tudo precisa passar pelo compilador → as instruções são geradas pelo kernel em tempo de execução.

mmu: memory manager unit

061	0820	24
-----	------	----

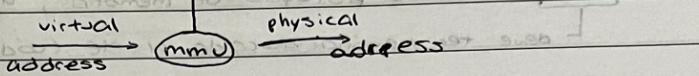
Anotações de aula:

→ se não for, mata o processo.

- garantia de proteção: para cada instrução (em tempo de exec.) deve-se checar se é válida, ou seja, se está no endereçamento daquele processo.

↳ uma solução: usar 2 registradores, um armazena o índice daquele processo na CPU e um para o final. → reg. base (endereço inicial) reg. limit (endereço final)

quem faz essa checagem é o hardware, o kernel só armazena os endereços.



- traduz endereços  
- garante proteção

agora após a mmu, o compilador utiliza endereços virtuais, por exemplo, para o primeiro dado, coloca 0. Esses endereços são abstrações criadas pelo compilador e que precisam ser traduzidos em tempo de execução.

ex: no compilador: LOAD 0x40000000 ; reg. base: 1024 ; reg. limit: 2047

então o MMU:  $1024 + 4 \rightarrow$  armazena no 1028 → 1028 é o endereço físico.

agora se posso redirecionar o programa para qualquer espaço na memória física.

— " —

08/08/24

Anotações de aula:

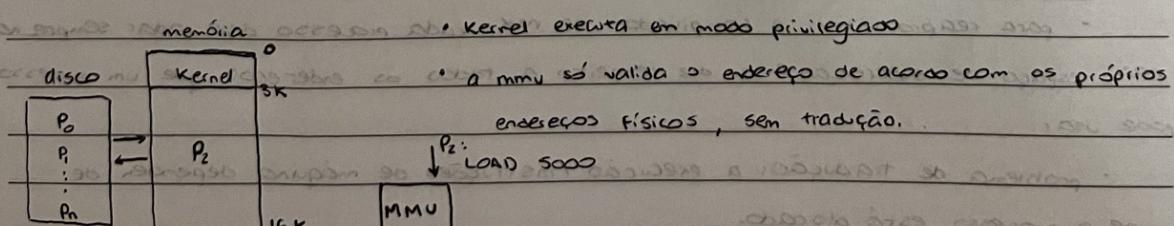
- funções do gerenciamento de memória:

- proteção (cada processo manipula apenas o seu pedaço de memória)

- alocação (dividir a memória entre os processos)

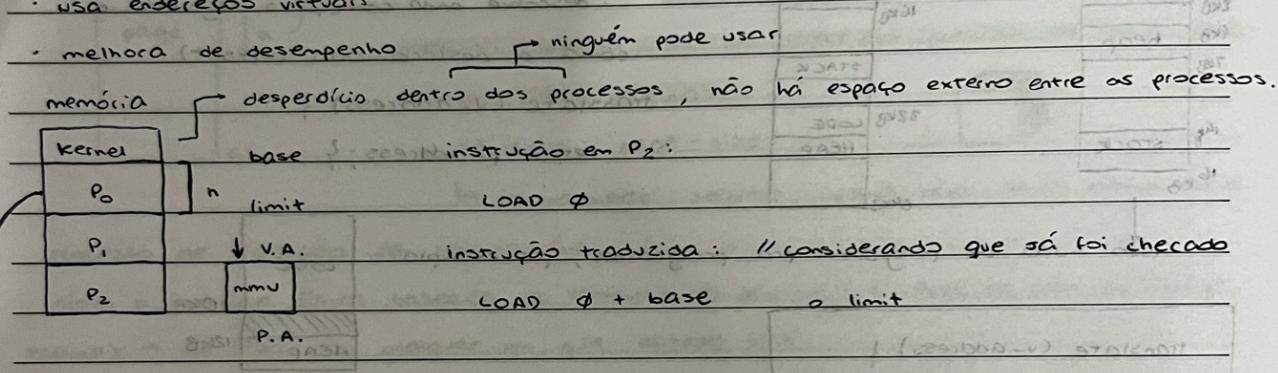
- tradução (endereço virtual → endereço físico)

- MMU:

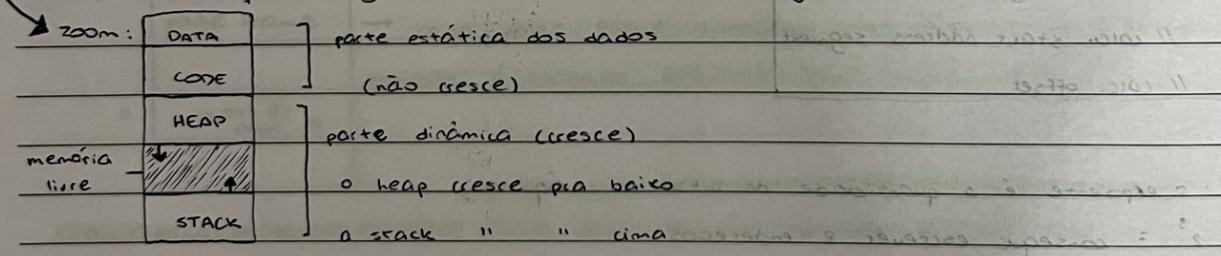


**FORONI:** fragmentação externa

- m<sub>1</sub>: // serve para máquinas de uso específico, ex: um supercomputador que calcula previsão de memória para mais de um processo, cada um com o mesmo tamanho. → tempo.
- alocação estática (n endereços fixos) // principal problema: fragmentação interna
  - ↳ se for muito pequeno não satisfaz os processos, não executam.
  - ↳ se for grande pode haver desperdício de memória, que é um recurso estático, além de ter menos processos em memória.
- 2 registradores na máquina: base e limit → quando o processo muda de ready para running o kernel atualiza os endereços nesses registradores para garantir proteção
- usa endereços virtuais



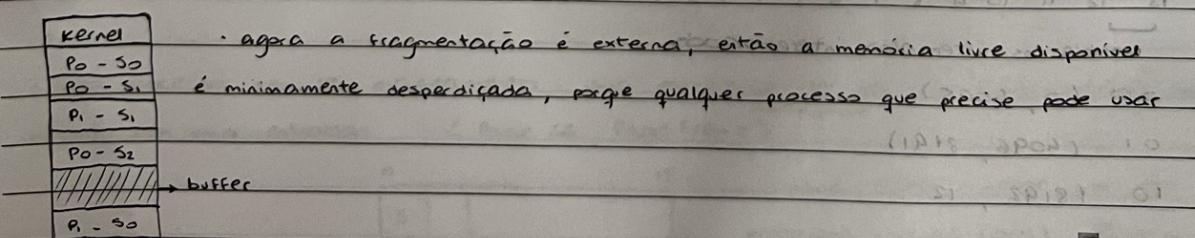
- SWAP: thread que carrega coisas do disco em background



- m<sub>2</sub>: modelo de segmentação

- crescimento dinâmico → muda o modo de alocação (era contínuo, ou seja, endereços em sequência) para memória segmentada, no segmento a memória é contínua.
- divide o processo em partes menores, os segmentos

memória:



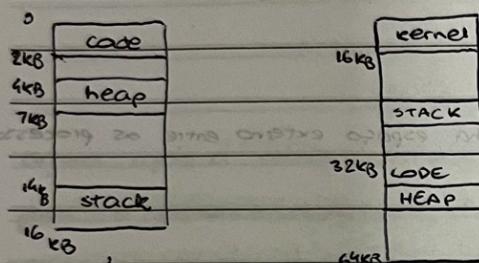
anotações de aula:

- o endereço virtual é o endereço gerado pelo compilador
- quanto mais endereços virtuais, mais memória
- espaço de endereçamento pode ser maior, menor ou igual à memória física
- limita a quantidade de endereços de um processo

ex:

considerando um processo de 16kB de endereços alocado dessa forma:

PROCESSO memoria



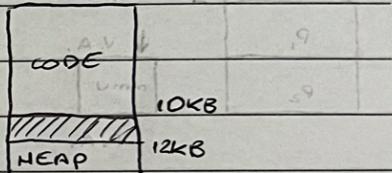
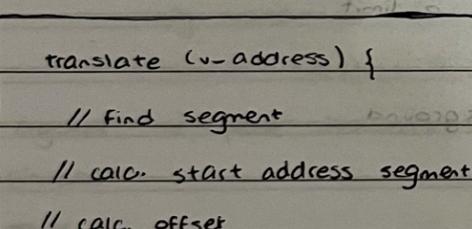
descreve onde começa o segmento e

só o deslocamento (offset)

translate (v-address) {

return p-address;

isso muda de programa pra programa, por exemplo:



o expoente é a quantidade de bits da máquina:

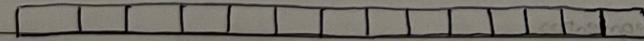
$2^3$  = consegue escrever 8 endereços → usa 3 bits

$2^4$  = 16 endereços

$2^{10}$  = 1024 → 1K

uma máquina de 16kB precisa de 16 bits

para obter a extensão de 16 bits →  $2^{10}$



16

16 bits por endereço

mostra os intervalos interessados:

00 (0, 4095)

01 (4096, 8191)

10 (8192, 12)

FORONI

• espaço de endereçamento é o conjunto de endereços virtuais → cada processo tem o seu.

- - -

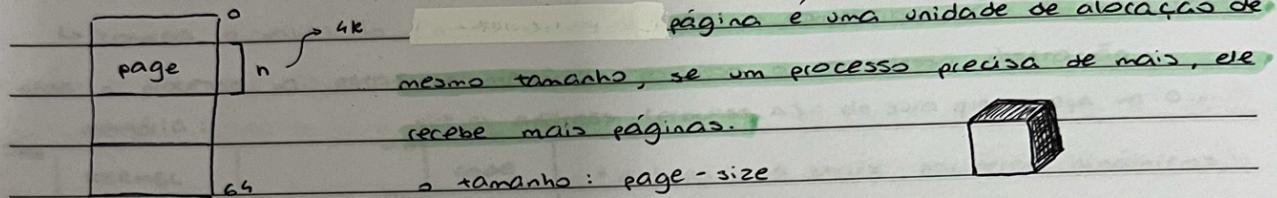
anotações de aula:

• problema: necessidade de diminuir mais os segmentos → complica muito o design

• novo modelo: volta a dividir segmentos de mesmo tamanho

\* mem. virtual

① de um processo: do ponto de vista do processo, é uma doração contínua.



↳ normalmente, regras, para não haver desperdício.

processo de tradução da paginação:

aloca a memória física baseada no tamanho da página: moldura de página (page frame) → vai alocar em qualquer um que esteja livre

memória física

page frame	→ cada page frame armazena uma página, é onde a memória é alocada: APENAS PAGE FRAMES INTEIROS.
5 process ① page 0	

• o espaço de endereçamento virtual se torna um conjunto de páginas.

translate (5000) {

// qual page? no exemplo, como cada page tem 4k, é na page 1

↳ (page - id) // no exemplo, 5000 é o 4º endereço:

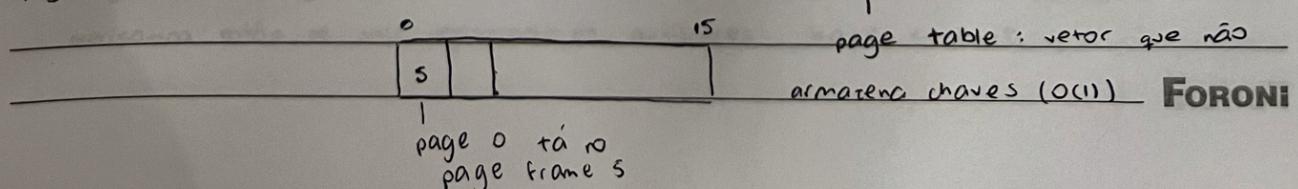
↳ (4096, 8191)

// acha o page frame

↳ (0, 4095) intervalo de endereços físico

// base + deslocamento

• cada processo tem seu mapeamento: uma estrutura de dados "dicionário", cada página tem uma entrada. ↳ Page Id, Page Frame



5 bits para armazenar o id de cada page frame ( $2^5 = 32$ ), com +1 para marcar se está ou não mapeado.

bit de presença 0: não alocado  $\rightarrow$  exception

dirty bit: indica se o processo foi alterado

o kernel mantém uma tabela por processo: muita complexidade espacial

as pages table ficam armazenadas em memória

resumo: gerência de memória

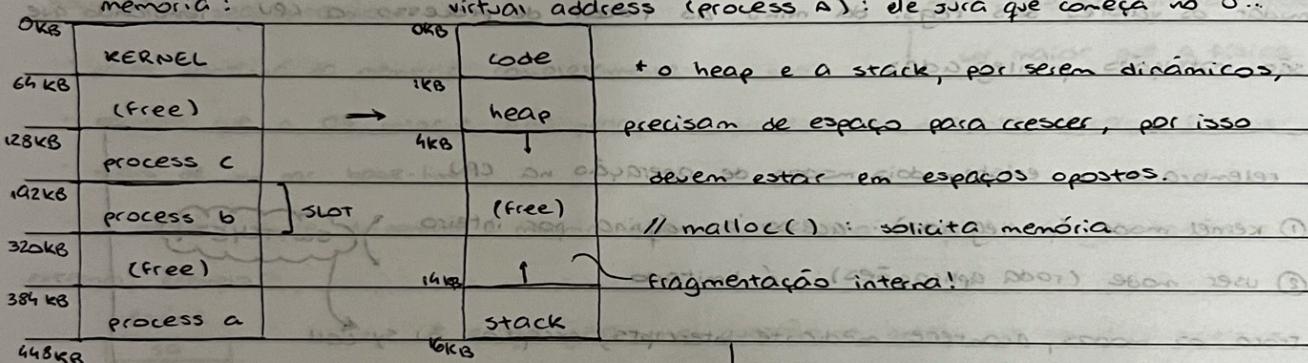
no modelo m<sub>1</sub> um único processo rodava por vez na memória, dessa forma, a troca de contexto de processo era muito lenta e havia "desperdício" de recurso (pois, consumindo toda memória era utilizada).

uma evolução (modelo m<sub>2</sub>) foi permitir o compartilhamento de memória para reduzir esse tempo de espera, o que exigiu novas demandas do SO, como proteção, ou seja, nenhum processo pode ler ou escrever no espaço de memória de outro.

↳ nesse modelo, o endereço de um processo contém todo ele // fixed slots

↳ começa a utilização de endereços virtuais: o compilador não tem como saber onde o processo vai estar na memória, por isso o uso dessa abstração.

memória:



ex: no compilador: LOAD 3

mmu → reg. base + 3 = endereço

384 KB + 3 = endereço 387 KB

= 396 287

o OS deve virtualizar a memória de modo invisível para o programa em execução: é função dele e do hardware traduzir esses endereços; além disso, o kernel também precisa garantir:

- eficiência de tempo e espaço

- isolamento entre os processos

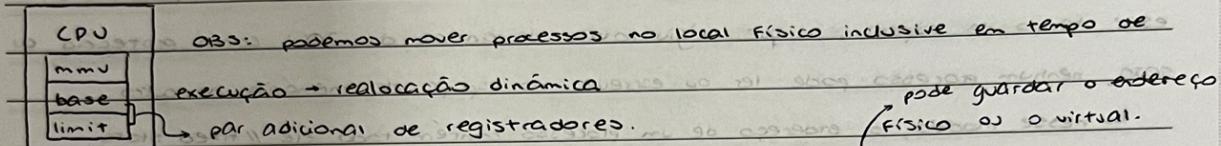
logo, entende-se que o SO garante que nenhuma aplicação acesse "qualquer" memória, mas que esteja livre de usar seus espaços como quiser. Para isso o address translation, ou seja, o hardware traduz cada acesso à memória, trocando o endereço virtual pela localização física.

\* manage memory: gerencia espaços livres e interveem como a memória é usada.

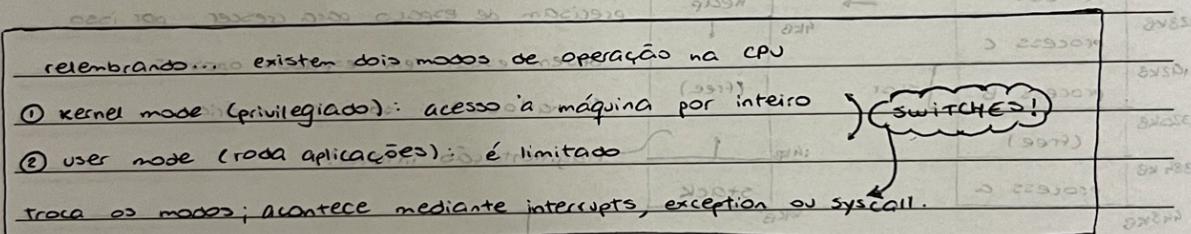
precisamos então de dois registradores auxiliares: base e limit.

esse par permite que o processo esteja em qualquer local da memória física e ainda assim saibamos traduzir seu endereço virtual. Quando um programa começa a rodar, o OS decide onde na memória ele deve ser alocado e seta a base no registrador.  
 → mmu:

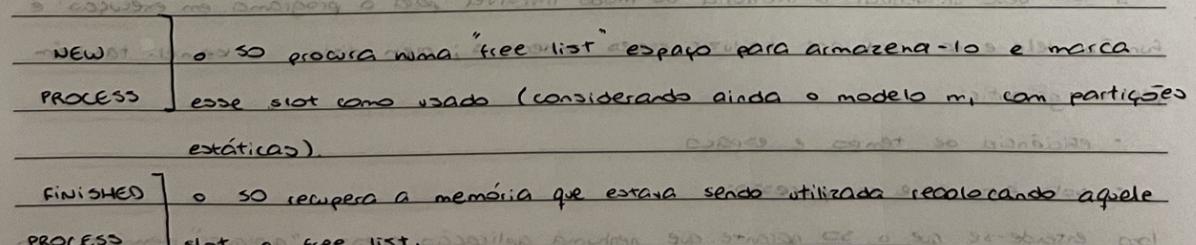
$$\text{physical address} = \text{virtual address} + \text{base}$$



- o registrador base serve para realocar e traduzir e o limit para proteção!
- o hardware deve prover um conjunto especial de instruções que permita o kernel mudar os valores dos regis. da mmu ao trocar de processo na CPU, lembrando que são instruções privilegiadas.



- existe apenas um par de base-limit em cada núcleo de processamento, quando há troca de processos o OS deve então salvar esses valores no PCB do processo e atualizar os registradores com os valores do próximo processo a ser executado.
- quando ocorre...



OBS: definição de endereço virtual → é o endereço gerado pelo compilador.  
 espaço de endereçamento é o conjunto de endereços virtuais de cada processo.

**FORONI**

O principal problema desse modelo é o da fragmentação interna: como os slots de memória são partições fixas, o processo inteiro é armazenado de maneira contínua num espaço reservado para ele, inclusive, sua área livre é armazenada. Acontece que, se esse processo cresce pouco, ou não cresce, aquele espaço disponível é desperdiçado, porque está reservado para um processo que não vai usar, ex:

memória

processo A	→	code	heap	stack	// desperdiça um pouco.
processo B	→	code	██████████		// " muitos.
processo C	→	code	heap	stack	// precisa de mais e não pode acessar.

BUT: não é culpa da MMU, é responsabilidade do sistema operacional.

SOLUÇÃO: segmentação!

OBS: memória virtual pode ser maior que a física.

- parte contínua da memória completamente em uso;
- permite que o OS mapeie cada pedaço do processo em diferentes espaços físicos;
- mais de um par base-limit
- a memória livre disponível é minimamente desperdiçada porque a fragmentação é externa; exemplo:

memória

so
code - P <sub>1</sub>
heap - P <sub>2</sub>
██████████
stack - P <sub>3</sub>
code - P <sub>4</sub>

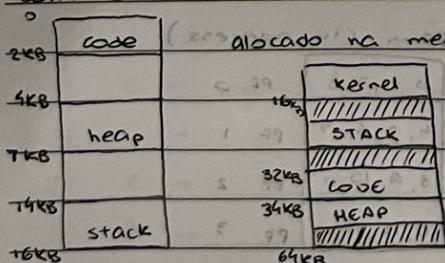
→ buffer

+ o que gera uma questão → antes, na memória contínua, a MMU só precisava somar o V.A. ao valor do registrador base, agora é preciso também identificar qual das bases / segmentos, porque o processo está espalhado pela memória.

1º forma de referenciar o segmento: identificar de qual reg. veio a instrução, ex: se foi do PC é no code, se foi do stack pointer é na stack...

2º forma: uso de offset (explicado no exemplo):

considerando um processo de 16KB:



para traduzir o endereço da instrução

"LOAD 4200"

1º converte o endereço p1 binário com 16 bits:  
= 01000001101000

os 2 primeiros

o restante dos bits é o offset

bits indicam o

com esse valor convertendo p1

segmento → 01 = heap

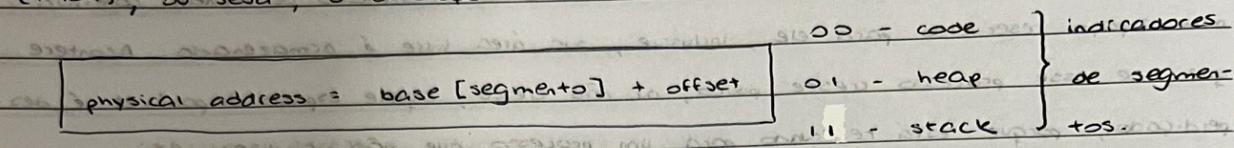
decimal → = 104

assim, sabemos que se trata do heap, agora pega a base dele: 34KB

FORONI

1935 080 24P1

$= 34\text{KB} = 34816$  (endereço onde ele começa), e soma isso ao valor do offset  
 $(+ 104)$ , ou seja, o endereço físico da instrução é: 34920



— " — 20108 — " —

anotações de aula: (imagine uma máquina:)

- 32 bits: significa que a máquina usa 32 bits para representar o endereço
- maior valor possível:  $2^{32} - 1$  (maior endereço)

· page 4K; → é armazenada na memória e um pedaço no cache (TLB)

ex) qual o tamanho da page table em bytes?

mechanismo de gerenciamento → estrutura de dados mantida pelo kernel para cada bloco de memória. processo em execução.

cada célula é uma entry

cada entrada armazena a informação relativa da página que se representa.

page table:

0	→ informações da page 0
1	→ informações da page 1
2	→ informações da page 2

exemplo de uma máquina pequena: 16 endereços, 4 endereços por página, 4 páginas totais:

memória virtual		page table	RAM (16 address)
0000	0, 3	page 0	0, 1, 2, 3 PF 0
0001	4, 6	11 1	4, 5, 6, 7 PF 1
0010	8, 10	0, 10	8, 9, 10, 11 PF 2
0011	12, 14	11 3	12, 13, 14, 15 PF 3
0100	13, 15		

(V.A) LOAD 10

//acha a page → page 2

//localiza a page e vê o endereço → 100 = PF 2

FORONI // se o 3º endereço na M.V. vai ser na RAM → 10

numero de entradas \* o tamanho de cada entrada = bytes da pte. por processo.

\* resposta do exemplo 1:

A quantidade de entradas = A qtd. de pages

$$\text{número de pages} = \frac{\text{qtd. v. address}}{\text{page size}} = \frac{2^{32}}{2^2 \cdot 2^{10}} = 2^{20} \text{ pages.}$$

O tamanho da entrada vai ser no mínimo 20 bits (para poder escrever os endereços to page frame) = 3 bytes

$$\text{tamanho da page table} = 2^{20} \cdot 3 \text{ bytes}$$

$$\text{mega} \cdot 3 \rightarrow 3 \text{ MB}$$

OBS: se o bit

mapped = 0  $\rightarrow$  page fault.

$\rightarrow$  mapped.

exemplo 2: máquina 64 bits, page 4KB (d)

$$\frac{\text{qtd. endereço}}{\text{page size}} = \frac{2^{64}}{2^{12}} = \frac{2^{52}}{2^2 \cdot 2^{10}} = 4 \text{ PB}$$

52 bits por pte

01111111111111111111111111111111

OBS: 1 barramento  $\uparrow$  variação de dados.

	ID (endereço)	page
000	0	15
000	0	14
000	0	13
000	0	12
111	1	11
000	0	10
101	0	9
000	0	8
000	1	7
000	1	6
011	1	5
100	1	4
000	1	3
110	1	2
001	1	1
010	1	0

exemplo: qual o P.A? 8196 = v.A. // para essa pte

64 KB = endereços

page 4KB -

$$64 = 16 \text{ pages} = 16 \text{ page frames} = 16 \text{ entradas}$$

$\downarrow$  ID  
(endereço)

$\downarrow$  page

com página tem 4096 (4K cada)

Anotações do page frame

page 0: 0 H (4096 - 1)

110 = page frame 6

page 1: 4096  $\leftrightarrow$  8192

page 2: 8192 H 12288  $\rightarrow$  está no page 2!

$\hookrightarrow$  é o 5º endereço.

PF 0  $\rightarrow$  começa em 0 . 4K da página

PF 1  $\rightarrow$  " em 1 . 4K "

PF 2  $\rightarrow$  " em 2 . 4K "

PF 3  $\rightarrow$  " em 3 . 4K "

PF 4  $\rightarrow$  " em 4 . 4K "

PF 5  $\rightarrow$  " em 5 . 4K "

PF 6  $\rightarrow$  começa em 6 . 4K = 24576 + 4 = 245810

Pode ser que o endereço, ou seja + 4 vai dar o 5º endereço

FORONI

