Sistemas Operacionais: Gerência de Memória

Prof. Maurício Aronne Pillon
Prof. Rafael R. Obelheiro

UDESC/CCT – Departamento de Ciência da Computação

{mauricio.pillon,rafael.obelheiro}@udesc.br

Joinville, outubro de 2017

Sumário

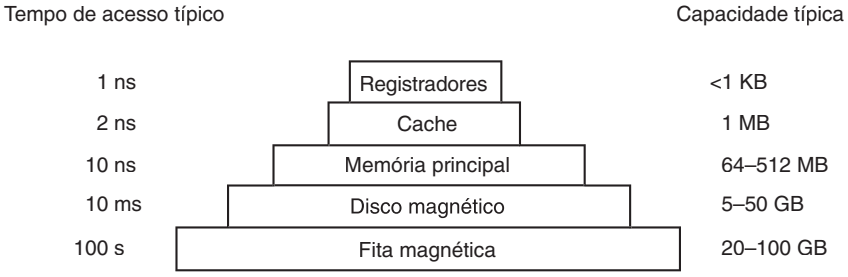
1. Conceitos básicos
2. Gerenciamento sem abstração de memória
3. Gerenciamento com espaços de endereçamento
4. Memória virtual
5. Gerência de memória no Linux

Introdução

- A maioria dos sistemas atuais é multiprogramada
  - vários processos concorrem pelo processador
  - para serem executados, os processos precisam estar residentes na memória
- Para ter eficiência na multiprogramação, é necessário gerenciar a memória de forma adequada
  - a gerência de memória influi no grau de multiprogramação atingível em um sistema, e por consequência no seu desempenho global
- A gerência de memória atinge importância maior na medida em que as aplicações aumentam o seu tamanho

Hierarquia de memória

- Um sistema típico possui várias memórias, diferindo em capacidade, tempo de acesso e custo
- A gerência de memória em um SO envolve memória principal e disco
  - sem o auxílio do disco
  - com o auxílio do disco: swapping e memória virtual

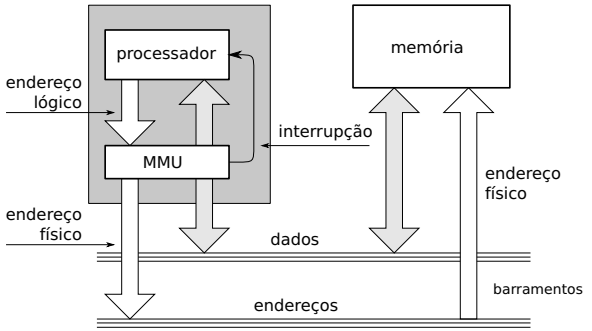


Memória lógica vs memória física (1/2)

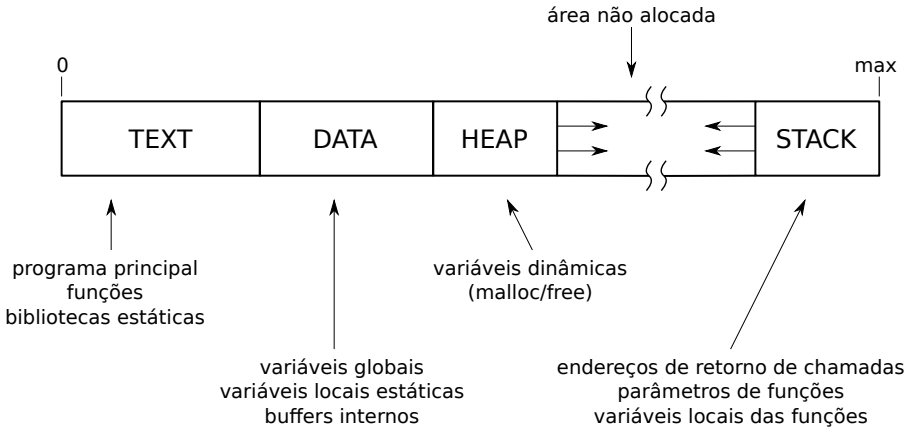
- Do ponto de vista do programador, a memória é um vetor de bytes endereçados individualmente
  - o tamanho máximo da memória é determinado pela largura do endereço em bits
    - b bits de endereço geram 2^b endereços diferentes $(0 \ldots 2^b - 1)$ $\Rightarrow 2^b$ bytes de memória
- Memória lógica: visão que um processo tem da memória
  - os endereços gerados por um processo são endereços lógicos
    - pertencentes à memória lógica
- Memória física: implementada pelos chips de memória
  - os endereços físicos correspondem a posições reais na memória
- Podem ser iguais ou diferentes
  - quando são diferentes, o mapeamento é feito em hardware com assistência do SO

Memória lógica vs memória física (2/2)

- O mapeamento entre endereços físicos e lógicos é realizado pela MMU (*Memory Management Unit*)
- O SO carrega tabelas de tradução na MMU



Layout típico da memória lógica

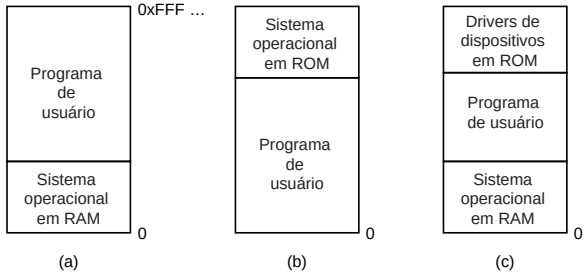


Sumário

1. Conceitos básicos
2. Gerenciamento sem abstração de memória
3. Gerenciamento com espaços de endereçamento
4. Memória virtual
5. Gerência de memória no Linux

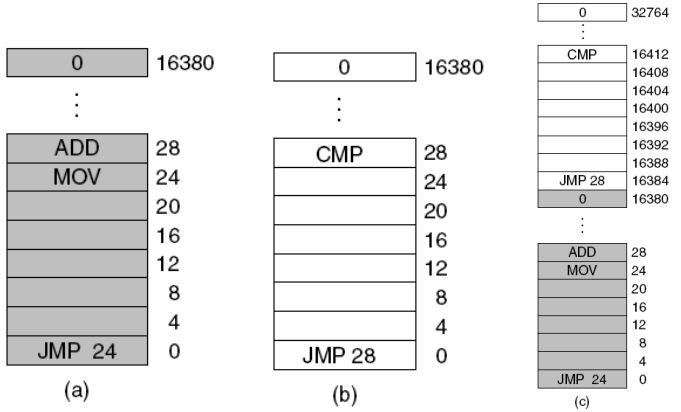
Gerenciamento sem abstração de memória

- Memória lógica = memória física
  - **endereçamento absoluto**
- Geralmente apenas um processo na memória por vez
  - **monoprogramação**
- Mecanismo **sem troca de processos** ou **paginação**
- Três maneiras simples de organizar a memória
  - um sistema operacional e um processo de usuário



Dificuldades com multiprogramação (1/3)

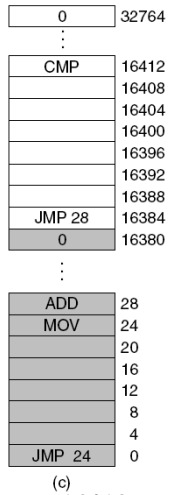
- Dois programas de 16 KB (a) e (b) são carregados em regiões adjacentes na memória (c)
  - quando (b) começa a executar, desvia para um endereço em (a)



Dificuldades com multiprogramação (2/3)

- Multiprogramação demanda proteção e relocação de memória
- IBM 360: memória dividida em blocos de 2 KB
  - cada bloco possuía uma chave de 4 bits, armazenada em um registrador específico
  - a PSW continha a chave do processo em execução
    - um processo que tentava acessar um bloco de memória com uma chave diferente era abortado → **proteção**
- Relocação precisava ser feita durante a carga → **relocação estática**
  - quando um processo era carregado na memória, todas as referências a endereços eram substituídas pelos endereços efetivamente usados
    - era preciso saber quais bytes no executável eram endereços e quais eram constantes

Dificuldades com multiprogramação (3/3)



- `JMP 28` seria relocado para `JMP 16412`
- `MOV AX,28` não seria alterado

Sumário

1. Conceitos básicos
2. Gerenciamento sem abstração de memória
3. Gerenciamento com espaços de endereçamento
4. Memória virtual
5. Gerência de memória no Linux

Espaços de endereçamento

- Endereçamento absoluto tem uma série de desvantagens
  - se não houver proteção, um processo pode corromper o SO
  - complica multiprogramação
- Uma solução melhor é recorrer à abstração de **espaço de endereçamento**
  - conjunto de endereços de memória que um processo pode usar
- Os programas são escritos considerando espaços de endereçamento privados, a menos que haja compartilhamento explícito
  - necessidade de proteção e relocação

Registradores de base e limite (1/2)

- Mecanismo simples para implementar proteção e relocação
- Um par de registradores, que só podem ser manipulados pelo SO
  - base = início do processo na memória física
  - limite = tamanho do processo na memória física
- Princípio de funcionamento

```
se EL < limite
    EF ← base + EL        /* relocação */
senão
    aborte                /* proteção */
```

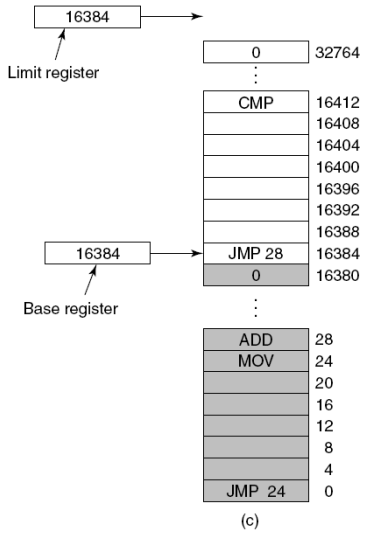
- Cada referência à memória exige uma adição e uma comparação

Registradores de base e limite (2/2)

(a) base = 0,
limite = 16384

(b) base = 16384,
limite = 16384

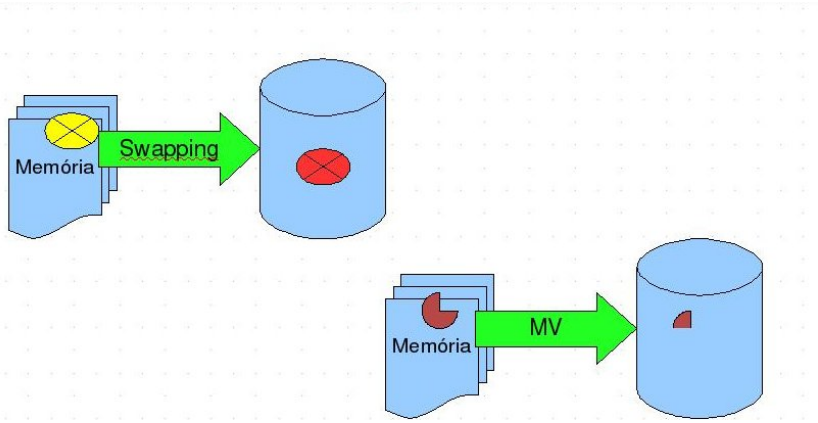
(c) se um programa de 8 KB fosse carregado logo após o prog. (b)?
base = ?
limite = ?



Swapping vs memória virtual (1/2)

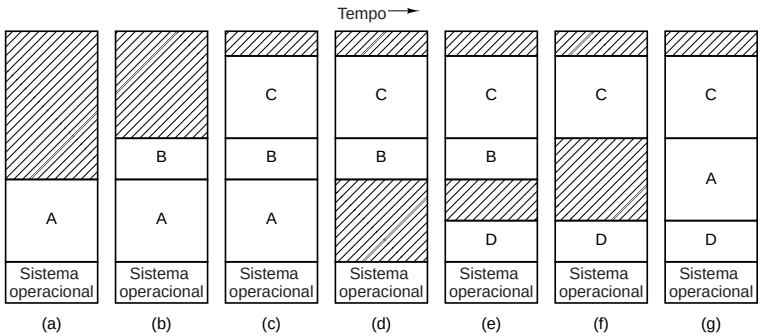
- Nem sempre há memória física suficiente para acomodar todos os processos ativos
- Uma solução é manter parte desses processos em disco
- Existem dois métodos básicos
  - **swapping (troca de processos)**: processos inteiros são trazidos da memória para o disco e vice-versa
  - **memória virtual**: os processos ativos estão parte na memória principal e parte no disco
    - particularmente útil se existem trechos de memória que não são efetivamente usados

Swapping vs memória virtual (2/2)



Exemplo de swapping

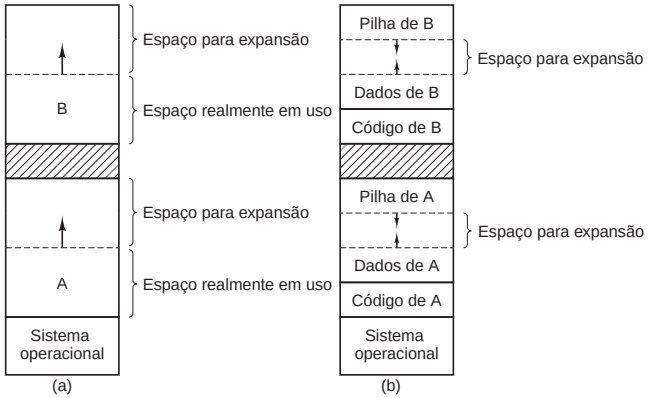
- Swapping com quatro processos A, B, C e D



áreas hachuradas = áreas livres (lacunas)

Dimensionamento de memória

- Normalmente é prudente alocar áreas de memória com uma certa folga para permitir alocação dinâmica
  - áreas de heap e pilha



(a) folga para dados (b) folga para dados e pilha

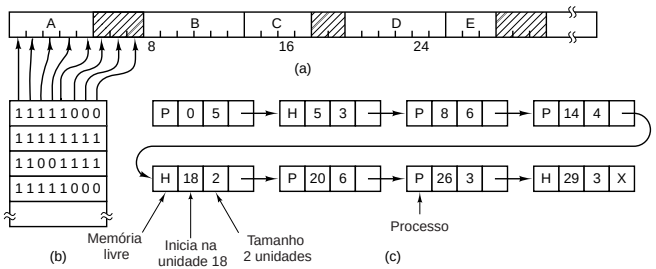
Problemas com swapping

- Como nem todos os processos usam a mesma quantidade de memória, ao longo do tempo ocorre **fragmentação externa**
  - grande número de pequenas lacunas que não podem ser usadas para alocar processos
  - uma solução é a **compactação de memória**
    - deslocar os processos e combinar as lacunas
    - overhead é grande
- Caso um processo precise alocar mais memória do que já possui, pode ser necessário movê-lo para outra região na memória física
  - se não houver uma lacuna grande o suficiente, outros processos também terão de ser movidos
- Se um processo usa apenas uma fração do seu espaço de endereçamento, muito tempo é gasto com transferências desnecessárias entre disco e memória

Gerência de espaço livre (1/2)

- O SO precisa controlar quais áreas da memória física estão ocupadas e quais estão livres
- Existem duas maneiras básicas de fazer isso
  - usando mapas de bits
    - problema: busca de 0s consecutivos no mapa para encontrar uma área disponível
  - usando listas encadeadas
    - ordenada por endereços de memória (atualização rápida e simples)

Gerência de espaço livre (2/2)

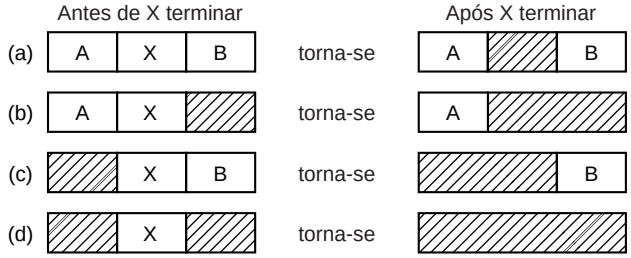


(a) Parte da memória com 5 segmentos de processos e 3 segmentos de memória livre
  - riscos simétricos denotam as unidades de alocação
  - regiões sombreadas denotam segmentos livres

(b) Mapa de bits correspondente

(c) Mesmas informações em uma lista encadeada

Gerência com listas encadeadas

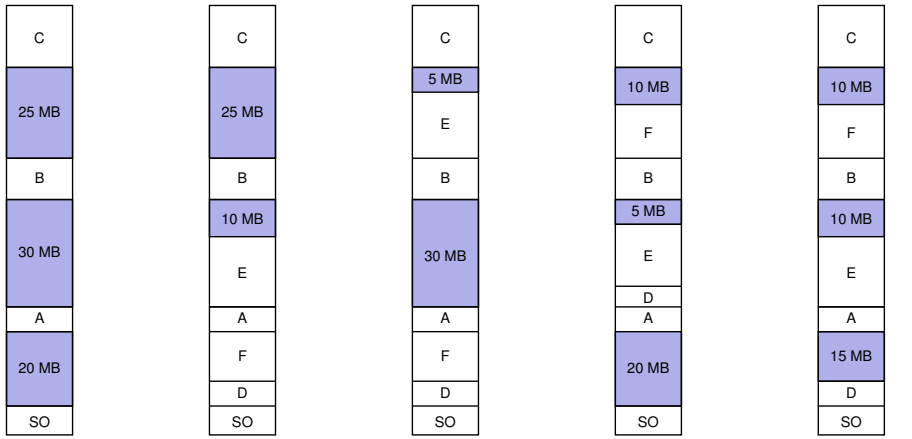


- Listas encadeadas de processos (P) e lacunas (H)
- Quando um processo termina, lacunas vizinhas são combinadas

Algoritmos para alocação de memória física (1/2)

- Como escolher onde alocar uma área contígua de memória?
- **First fit**: a primeira lacuna disponível é usada
- **Best fit**: a lacuna que melhor se ajusta é usada
  - tende a deixar lacunas muito pequenas
- **Worst fit**: a maior lacuna é usada
  - gera lacunas maiores, mas prejudica processos grandes
- **Next fit (circular fit)**: como o first fit, mas guarda a última posição na lista de lacunas
- **Quick fit**: mantém listas de tamanhos comuns de áreas de memória
  - pode ser dispendioso descobrir quais são os segmentos de memória disponíveis para poder concatená-los

Algoritmos para alocação de memória física (2/2)



novos processos: D (5 MB), E (20 MB), F (15 MB)

Problemas de alocação contígua

- Um dos problemas da alocação contígua de memória é a fragmentação
  - pode haver **memória livre indisponível** para processos
- Um segundo problema é quando os programas usados extrapolam a memória física disponível
  - hoje em dia, existem programas enormes com muitas funcionalidades intocadas durante boa parte do tempo
  - pode afetar processos individuais ou o seu conjunto
  - problema é agravado pela fragmentação

Sumário

1. Conceitos básicos
2. Gerenciamento sem abstração de memória
3. Gerenciamento com espaços de endereçamento
4. Memória virtual
5. Gerência de memória no Linux

Memória virtual

- Como lidar com programas maiores do que a memória física disponível?
- A primeira solução foi a introdução de **overlays**
  - o programador dividia o programa em módulos que eram carregados e descarregados da memória semi-manualmente
- A solução mais definitiva veio com a memória virtual
  - espaço de endereçamento lógico (dos processos) é mapeado em um espaço de endereçamento físico
  - mapeamento permite usar regiões não contíguas na memória física
  - nem todo o espaço de endereçamento lógico precisa estar mapeado na memória física em um dado instante
    - apenas as partes efetivamente usadas são mantidas na memória física, as demais são mantidas no disco e carregadas quando necessário

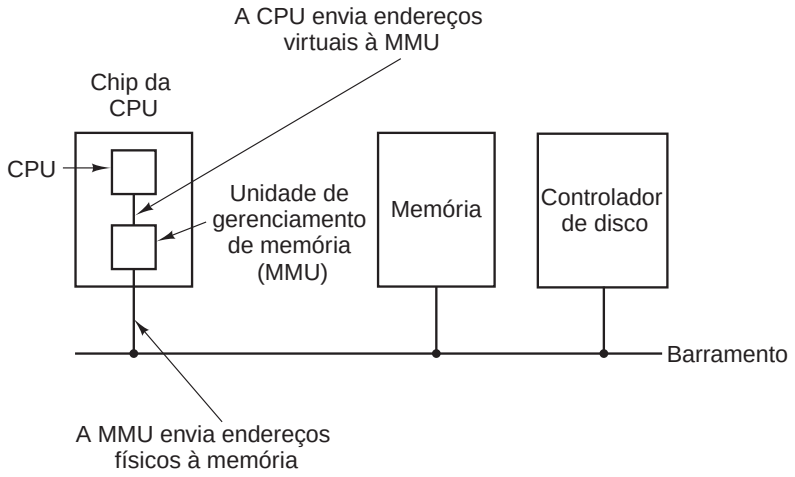
Variações de memória virtual

- Paginação
- Segmentação
- Segmentação paginada

Memória virtual com paginação

- A memória é dividida em blocos de tamanho fixo
  - memória lógica: páginas [virtuais]
  - memória física: molduras de página (*page frames*) ou páginas físicas
  - páginas têm tamanhos idênticos
- O mapeamento entre a memória lógica e a memória física é dado pela **tabela de páginas**
- A tradução de endereços lógicos em físicos é realizada pela MMU (*memory management unit*), que usa a tabela de páginas
- A paginação elimina a fragmentação externa e reduz a fragmentação interna

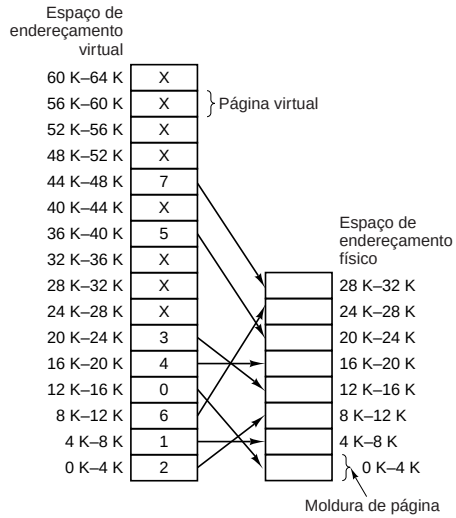
Localização e função da MMU



Endereçamento virtual vs físico

- Seja um sistema com as seguintes características
  - espaço de endereçamento virtual de 64 KB
  - espaço de endereçamento físico de 32 KB
  - páginas de 4 KB
    - tamanhos típicos de página variam entre 512 bytes e 64 KB
- Para esse sistema, são necessárias
  - 64 KB/4 KB = 16 páginas [virtuais]
  - 32 KB/4 KB = 8 molduras de página

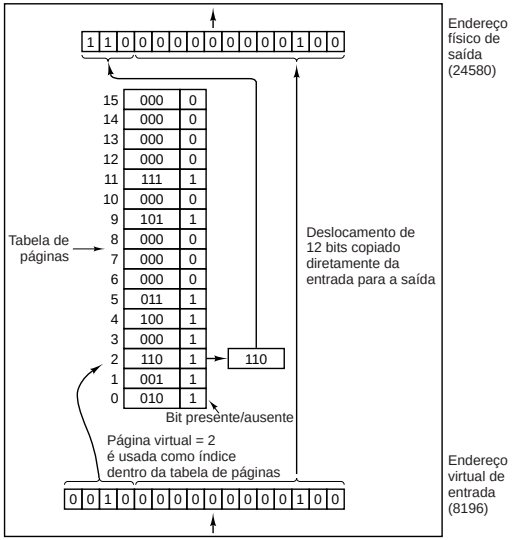
Endereçamento virtual vs físico



Falta de página

- Um bit de presente/ausente indica quais páginas lógicas estão presentes na memória física
- Quando um endereço referenciado pertence a uma página lógica que não está na memória física, ocorre uma **falta de página** (*page fault*)
  - MMU gera uma interrupção
  - SO carrega a página para a memória física
    - se memória cheia, escolhe uma página para remoção
    - algoritmos de substituição de páginas serão vistos mais adiante
  - SO atualiza a tabela de páginas e retorna ao processo
    - instrução interrompida é reiniciada

Operação interna da MMU



Tabelas de páginas (1/2)

- Endereços virtuais = número de página + deslocamento
  - deslocamento é posição dentro da página
- Endereços físicos = número da moldura + deslocamento
  - deslocamento é o mesmo
- A tabela de páginas mapeia páginas em molduras
- Para o exemplo anterior:
  - 12 bits para o deslocamento (páginas de 2^{12} bytes=4 KB)
  - 4 bits para o número de página (2^4=16 páginas)
  - 3 bits para o número da moldura (2^3=8 páginas)
  - endereços lógicos têm $12 + 4 = 16$ bits (EEL=2^{16} bytes=64 KB)
  - endereços físicos têm $12 + 3 = 15$ bits (EEF=2^{15} bytes=32 KB)

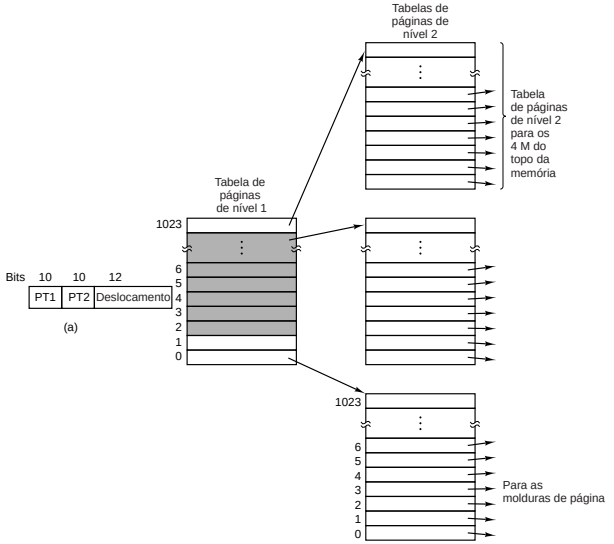
Tabelas de páginas (2/2)

- Um desafio para sistemas de paginação é o tamanho da tabela de páginas
  - quantas entradas são requeridas para um sistema com endereços de 32 bits e páginas de 4 KB?
  - qual o espaço ocupado por essa tabela?
- Outro desafio: o acesso à tabela deve ser rápido
  - usada em todo acesso à memória
- Soluções simplistas
  - tabela de páginas em memória
    - problema: desempenho
  - tabela de páginas em registradores
    - problemas: custo, desempenho nos chaveamentos de contexto

Tabela de páginas multiníveis (1/2)

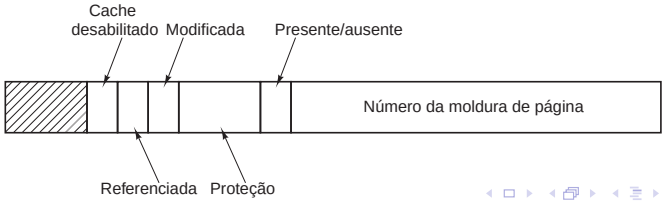
- Uma solução para tratar o tamanho de tabelas de páginas é o uso de tabelas de páginas multiníveis
- O número de página é subdividido em dois ou mais índices em tabelas de páginas distintas
  - ex: sistema de 32 bits com páginas de 4 KB
  - 20 bits estão disponíveis para o número de página
  - esses bits podem ser divididos em $10 + 10$ bits
    - tabela de páginas principal com $2^{10} = 1024$ entradas
    - cada entrada aponta para outra tabela com $2^{10} = 1024$ entradas
    - as entradas da 2^a tabela apontam para molduras de página
- Apenas as tabelas de páginas em uso precisam ficar residentes na memória

Tabelas de páginas multiníveis (2/2)



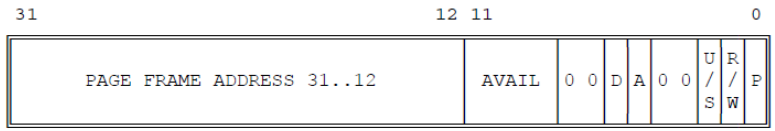
Entradas na tabela de páginas

- Tamanho comum de uma entrada: 32 bits
- Campos de uma entrada
  - número da moldura de página
  - bit presente (1) / ausente (0)
    - acesso a página com este bit em 0 causa uma falta de página
  - bit de proteção: leitura-escrita (0) / leitura (1)
    - esquemas mais sofisticados existem
  - modificada (sujo): (1) indica que a página foi escrita
  - referenciada: (1) indica que a página foi referenciada
  - cache desabilitado: (1) indica que a página não pode ser mantida em cache → E/S mapeada em memória



Exemplo: PTEs no 80386

Figure 5-10. Format of a Page Table Entry




```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```


Memória associativa (TLB)

- Tabelas de páginas invariavelmente são mantidas em memória devido a seu tamanho
- Isso, porém, tem impacto no desempenho
  - a cada referência a memória, dois acessos são necessários
  - a própria busca de uma instrução é afetada
- Felizmente, na prática os programas tendem a referenciar um conjunto reduzido de páginas em um dado período
- Isso permite armazenar as entradas mais usadas em um conjunto de registradores na MMU
  - TLB (*translation lookaside buffer*)

Funcionamento da TLB

- Uma página primeiro é buscada na TLB
  - se ocorre um acerto (*hit*), só é feito um acesso à memória
  - se ocorre um erro (*miss*), dois acessos são efetuados
    - nesse caso, a TLB é atualizada com os dados obtidos na tabela de páginas em memória
  - taxas de acerto típicas ultrapassam 99%

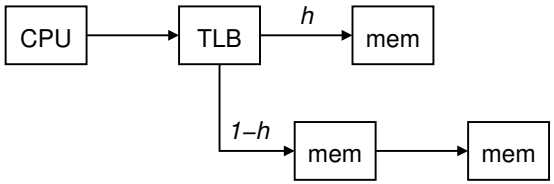
Exemplo de TLB

Válida	Página virtual	Bit modificada	Bits de proteção	Moldura de página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Impacto de TLB no desempenho (1/2)

- Seja t_{tlb} o tempo de acesso à TLB e t_{mem} o tempo de acesso à memória
- $t_{hit} = t_{tlb} + t_{mem}$
- $t_{miss} = t_{tlb} + t_{mem} + t_{mem} = t_{tlb} + 2\,t_{mem}$
- Para uma taxa de acerto h, o tempo médio de acesso à memória é

$$t_{ac} = h \cdot t_{hit} + (1 - h) \cdot t_{miss}$$



Impacto de TLB no desempenho (2/2)

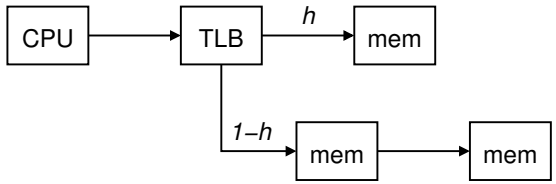
- Seja $t_{tlb} = 20$ ns e $t_{mem} = 100$ ns

  $t_{hit} = 20 + 100 = 120$ ns

  $t_{miss} = 20 + 2 \times 100 = 220$ ns
- Se a taxa de acerto for de 85% ($h = 0{,}85$):

  $t_{ac} = 0{,}85 \times 120 + (1 - 0{,}85) \times 220 = 135$ ns
- Se a taxa de acerto for de 99% ($h = 0{,}99$):

  $t_{ac} = 0{,}99 \times 120 + (1 - 0{,}99) \times 220 = 121$ ns



Algoritmos de substituição de páginas

- A falta de página força uma escolha
  - qual página deve ser removida
  - alocação de espaço para a página a ser trazida para a memória
- A página modificada deve primeiro ser salva
  - se não tiver sido modificada é apenas sobreposta
- Melhor não escolher uma página que está sendo muito usada
  - provavelmente precisará ser trazida de volta logo

Algoritmo *Ótimo*

- Substitui a página necessária o mais à frente possível
  - ótimo mas não realizável
- Estimada através de
  - registro do uso da página em execuções anteriores do processo
  - apesar disto ser impraticável

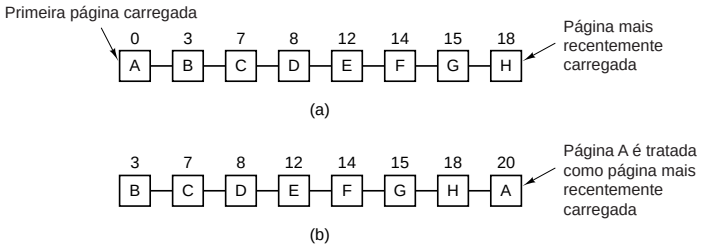
Algoritmo Não Usada Recentemente (NUR)

- Cada página tem os bits Referenciada (R) e Modificada (M)
  - bits são colocados em 1 quando a página é referenciada ou modificada
- As páginas são classificadas
  - classe 0: não referenciada, não modificada
  - classe 1: não referenciada, modificada
  - classe 2: referenciada, não modificada
  - classe 3: referenciada, modificada
- NUR remove página aleatoriamente
  - da classe de ordem mais baixa que não esteja vazia

Algoritmo FIFO

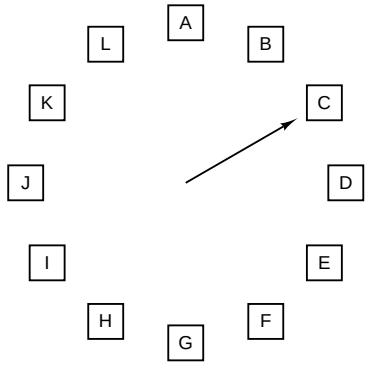
- Mantém uma lista encadeada de todas as páginas
  - página mais antiga na cabeça da lista
  - página que chegou por último na memória no final da lista
- Na ocorrência de falta de página
  - página na cabeça da lista é removida
  - nova página adicionada no final da lista
- Desvantagem
  - página há mais tempo na memória pode ser usada com muita frequência

Algoritmo Segunda Chance (SC)



- Operação do algoritmo segunda chance
  - lista de páginas em ordem FIFO
  - estado da lista em situação de falta de página no instante 20, com o bit R da página A em 1 (números representam instantes de carregamento das páginas na memória)
  - a referência considerada é aquela do intervalo de relógio anterior

Algoritmo do Relógio



Quando ocorre uma falta de página,
a página apontada é examinada.
A atitude a ser tomada depende do bit R:
R = 0: Retira a página,
R = 1: Faz R = 0 e avança o ponteiro.

Menos Recentemente Usada (MRU)

- Premissa do algoritmo ótimo (impraticável)
  - páginas referenciadas nas últimas instruções serão novamente referenciadas nas próximas instruções
- Reflexão (de modo oposto)
  - páginas que não foram referenciadas nas últimas instruções provavelmente não o sejam nas próximas

Menos Recentemente Usada (MRU)

- Assume que páginas usadas recentemente logo serão usadas novamente
  - retira da memória página que há mais tempo não é usada
- Uma lista encadeada de páginas deve ser mantida
  - página mais recentemente usada no início da lista, menos usada no final da lista
  - atualização da lista a cada referência à memória
- Alternativamente, manter contador em cada entrada da tabela de página
  - escolhe página com contador de menor valor
  - zera o contador periodicamente

Implementando MRU com uma matriz (hardware)

Página

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

0	0	0	0	0
1	0	1	1	1
2	1	0	0	1
3	1	0	0	0

(f)

0	1	1	1	
0	0	1	1	
0	0	0	1	
0	0	0	0	

(g)

0	1	1	0	
0	0	1	0	
0	0	0	0	
1	1	1	0	

(h)

0	1	0	0	
0	0	0	0	
1	1	0	1	
1	1	0	0	

(i)

0	1	0	0	
0	0	0	0	
1	1	0	0	
1	1	1	0	

(j)

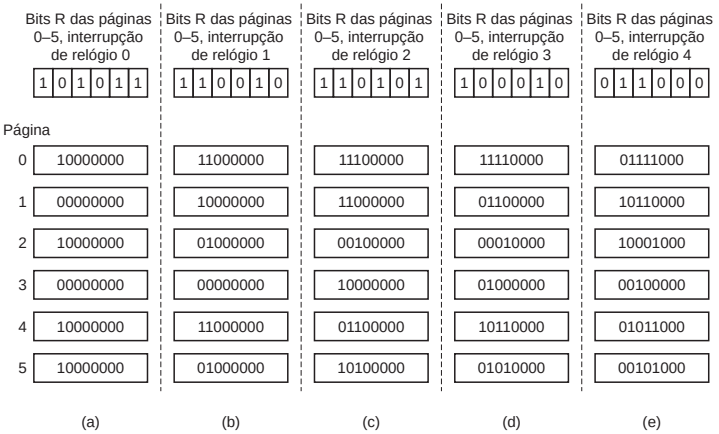
- ao acessar a página i, coloca a linha i em 1 e a coluna i em 0
  - a página correspondente à linha com menor valor é a mais antiga
- páginas referenciadas na ordem $(0, 1, 2, 3, 2, 1, 0, 3, 2, 3)$

Simulação do MRU em software (1/2)

- A primeira aproximação seria o algoritmo NFU (não frequentemente usado)
  - associa um contador a cada página, inicialmente 0
  - a cada interrupção de relógio, percorre a lista de páginas e adiciona o bit R ao contador da página
  - página com o menor contador é a mais antiga
- O problema do NFU é que ele tem longa memória
  - páginas intensamente referenciadas no passado têm contadores com valores elevados, mesmo que não sejam mais necessárias
  - páginas recém carregadas, mesmo que estejam sendo muito usadas, têm contadores mais baixos e acabam sendo escolhidas

Simulação do MRU em software (2/2)

- Uma solução é o algoritmo de envelhecimento (*aging*)
  - o contador é deslocado 1 bit para a direita
  - o bit R é copiado para o bit mais à esquerda



Revisão dos algoritmos

Algoritmo	Comentário
Ótimo	Não implementável, mas útil como um padrão de desempenho
NUR (não usada recentemente)	Muito rudimentar
FIFO (primeira a entrar, primeira a sair)	Pode descartar páginas importantes
Segunda chance	Algoritmo FIFO bastante melhorado
Relógio	Realista
MRU (menos recentemente usada)	Excelente algoritmo, porém difícil de ser implementado de maneira exata
NFU (não freqüentemente usada)	Aproximação bastante rudimentar do MRU
Envelhecimento (*aging*)	Algoritmo bastante eficiente que se aproxima bem do MRU
Conjunto de trabalho	Implementação um tanto cara
WSClock	Algoritmo bom e eficiente

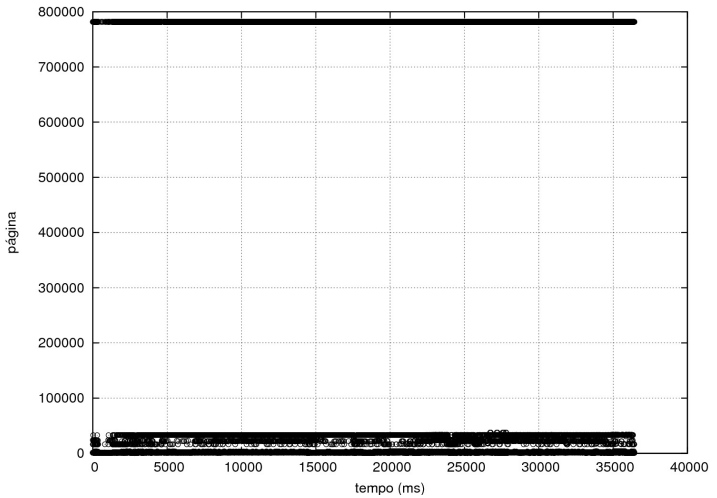
Paginação por demanda

- Na forma mais pura de paginação, um processo inicia sem nenhuma de suas páginas presentes na memória
- Quando a CPU vai buscar a primeira instrução, isso gera uma falta de página
  - outras faltas de página, para dados e pilha, geralmente se sucedem
- Depois de um certo tempo, todas as páginas necessárias à execução do processo estão na memória, e o número de faltas de página é bastante reduzido
- Essa estratégia é chamada de **paginação por demanda**

Localidade e conjunto de trabalho

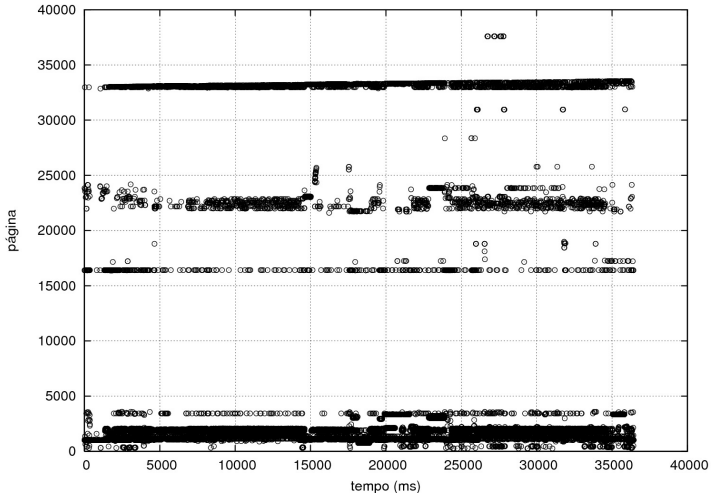
- A maioria dos processos apresenta **localidade de referências**: em um dado instante, as referências à memória se concentram em um conjunto reduzido de páginas
- O conjunto de páginas acessadas na história recente de um processo é o seu **conjunto de trabalho** (*working set*)
- O conjunto de trabalho evolui dinamicamente à medida em que o processo executa
- Se todas as páginas do conjunto de trabalho estão na memória, o processo executa com poucas faltas de página
  - apenas acessos a novas páginas geram faltas

Localidade de referências no gThumb (1/2)



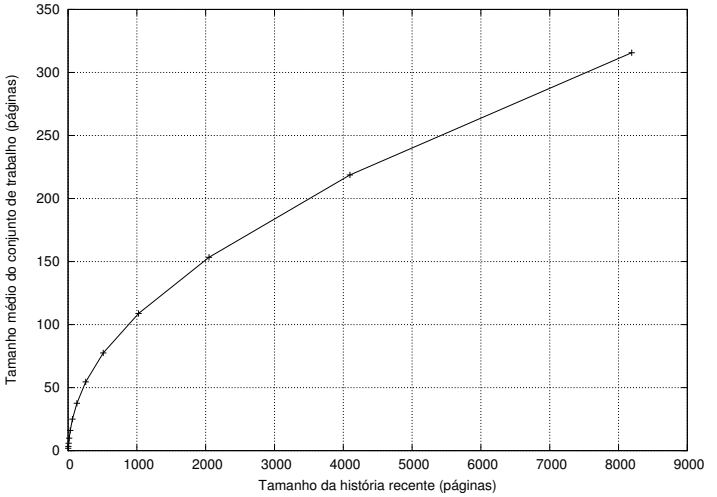
visão geral

Localidade de referências no gThumb (2/2)



visão da parte inferior (código, dados, pilha)

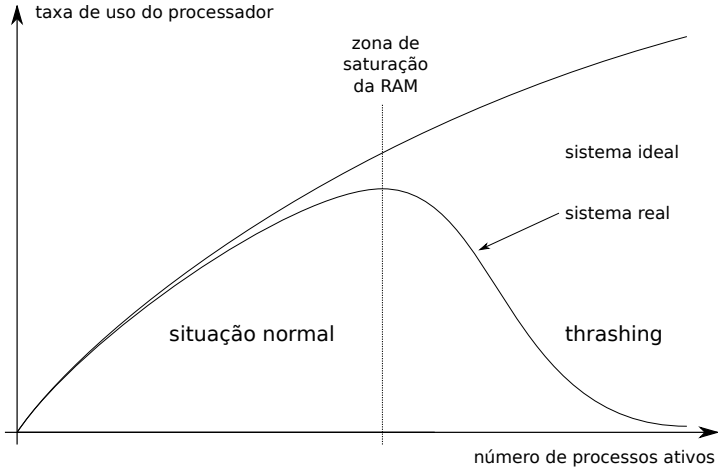
Conjunto de trabalho para o gThumb



Thrashing

- Caso a memória física não seja suficiente para armazenar o agregado do conjunto de trabalho de todos os processos, o número de faltas de página cresce rapidamente
  - o processo escalonado gera faltas de página para completar seu conjunto de trabalho
  - como não há memória física, é necessário usar páginas físicas de outros processos
  - quando um desses processos for escalonado, seu conjunto de trabalho também estará incompleto
- **Thrashing**: sistema passa boa parte do tempo (ou mesmo a maior parte do tempo) paginando em vez de fazer algo útil
  - solução é reduzir o número de processos no sistema

Thrashing



Políticas de alocação de páginas (1/3)

- Quando se escolhe uma página vítima, deve-se considerar todas as páginas na memória ou apenas as páginas do processo que causou a falta de página?
  - em outras palavras, como dividir a memória física entre os processos?
- **Alocação local**: considera apenas as páginas do processo
- **Alocação global**: considera as páginas de todos os processos

Políticas de alocação de páginas (2/3)

	Idade
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

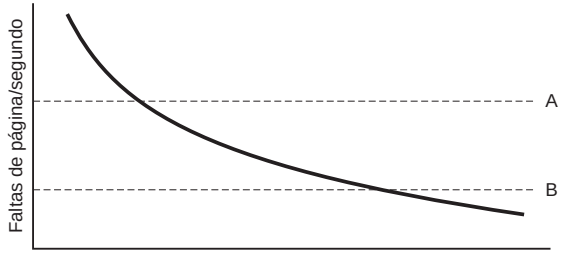
(a) original (b) alocação local (c) alocação global

Políticas de alocação de páginas (3/3)

- Alocação global usa melhor a memória disponível
  - se um processo usar toda a memória disponível para ele e estiver sendo usada alocação local, o processo pode entrar em thrashing, mesmo que haja memória livre
  - caso o conjunto de trabalho diminua, alocação local causa desperdício de memória
- Outra estratégia é alocar uma fração das páginas disponíveis para cada processo
  - geralmente proporcional ao tamanho do processo

Frequência de faltas de página (PFF)

- Algoritmo **frequência de faltas de página** (*page fault frequency*): tenta manter a taxa de faltas de página entre valores aceitáveis
  - se taxa $>$ A, aloca mais páginas físicas para processo
    - se não for possível manter todos os processos abaixo de A, envia um processo inteiro para o disco e reparte as suas páginas de memória entre os restantes
  - se taxa $<$ B, pode liberar páginas físicas do processo



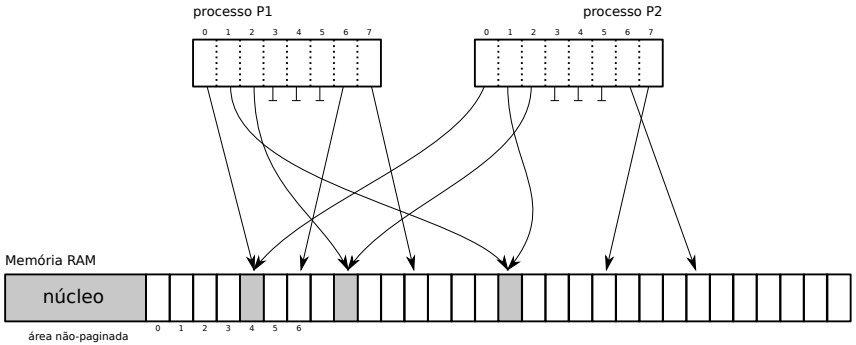
Tamanho de página

- Uma decisão de projeto importante é o tamanho da página
  - escolha de hardware ou do SO, dependendo da arquitetura
- Por que preferir páginas pequenas
  - menos desperdício de memória
    - fragmentação interna: 1/2 página por processo (média)
    - memória alocada porém não efetivamente usada
  - mais fácil manter o conjunto de trabalho de todos os processos na memória
- Por que preferir páginas grandes
  - tabelas de páginas menores
  - transferência mais rápida do disco
  - TLB cobre maiores áreas de memória
- Tendência tem sido o uso de páginas maiores

Compartilhamento de memória

- Em sistemas multiusuários e servidores, é comum ter várias instâncias do mesmo programa sendo executadas ao mesmo tempo
- Cada processo tem suas próprias áreas de código, dados, heap e pilha
- O código e os dados constantes podem ser compartilhados entre as várias instâncias $\rightarrow$ economia de memória
  - exemplo: programa com 200 MB (100 MB código + 100 MB dados)
    - 10 instâncias **sem** compartilhamento: 2000 MB
    - 10 instâncias **com** compartilhamento: 1100 MB
- O compartilhamento pode ser implementado mapeando páginas lógicas de processos distintos nas mesmas páginas físicas

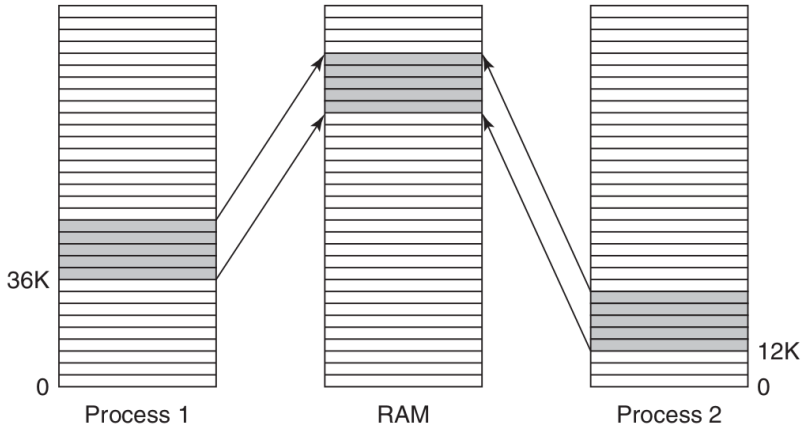
Compartilhamento de memória paginada



Bibliotecas compartilhadas (1)

- Programas executáveis precisam ter acesso a funções de biblioteca
  - `printf()`, `scanf()`, `qsort()`, ...
- Providenciar acesso às bibliotecas é tarefa do ligador (*linker*)
  - ligação estática: o executável incorpora o código e os dados necessários das bibliotecas
  - ligação dinâmica: o executável contém ponteiros para as bibliotecas que precisa, que são resolvidos durante a carga
    - bibliotecas compartilhadas no Unix, *dynamic-link libraries* (DLLs) no Windows
- Ligação estática consome mais espaço em disco e na memória, e exige recompilar/religar todos os executáveis em caso de alterações na biblioteca
- Com paginação sob demanda, as bibliotecas compartilhadas podem residir apenas parcialmente na memória

Bibliotecas compartilhadas (2)



Copiar ao escrever – *copy-on-write* (COW)

- Técnica usada para implementar `fork()` em sistemas modernos
- Em vez de duplicar o espaço de endereçamento do processo pai para criar o processo filho, apenas as entradas da tabela de páginas são copiadas
  - páginas protegidas contra escrita no pai **e** no filho
  - por default, todas as páginas são compartilhadas
- Quando um processo (pai ou filho) tenta escrever em uma página, a MMU gera uma exceção
  - SO aloca outra página física, copia o conteúdo para a nova página e ajusta a tabela de páginas → apenas no processo que escreve
    - página passa a ter permissão de escrita

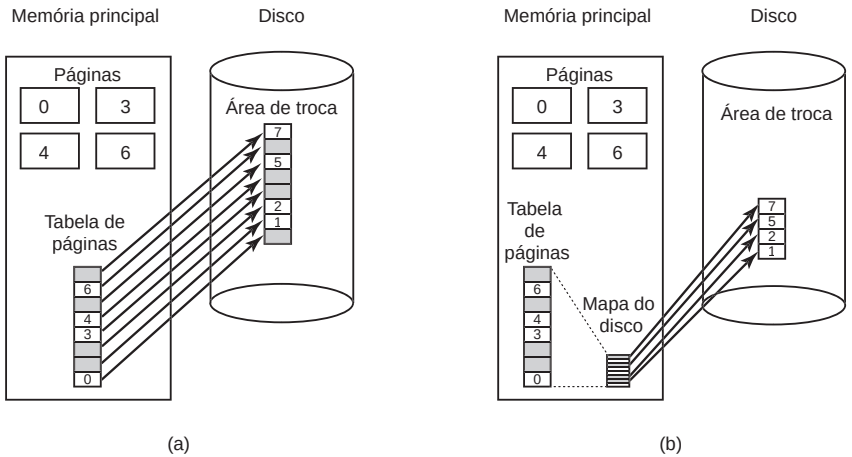
Travamento de páginas

- Determinadas páginas não podem ser retiradas da memória
  - buffers de E/S (especialmente com DMA), estruturas de dados usadas pelo núcleo, pilha do núcleo
- Essas páginas são travadas na memória
- O algoritmo de substituição de páginas precisa levar em conta quais páginas estão travadas

Memória secundária (1/2)

- Como gerenciar o armazenamento de páginas no disco?
- **Área de swap (troca)**: uma partição/disco dedicada para páginas
  - um espaço reservado na área de swap para cada processo criado
    - endereço da área do processo é mantido na tabela de páginas
    - processo deve ser copiado para área de swap na carga, ou páginas copiadas da memória quando necessário
  - para lidar com crescimento da memória, pode-se ter áreas de swap separadas para código, dados e pilha (possivelmente com múltiplas partes)
  - outra forma é alocar espaço apenas quando necessário
    - desvantagem é precisar mapear páginas nos seus endereços no disco
    - própria tabela de páginas pode armazenar endereços de disco nas entradas marcadas como inválidas
- Alguns sistemas (ex: Windows) usam um **arquivo de paginação**, com tamanho prealocado
  - para código pode-se usar o próprio executável como memória secundária

Memória secundária (2/2)



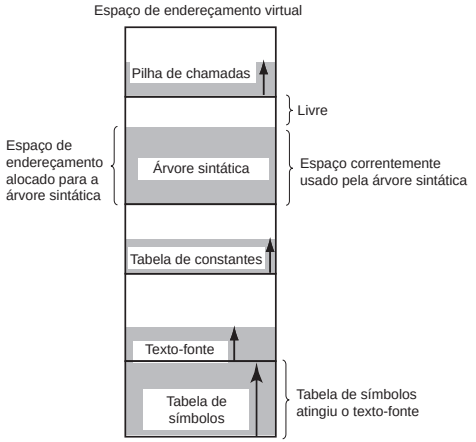
(a) área de swap estática

(b) área de swap dinâmica

Segmentação

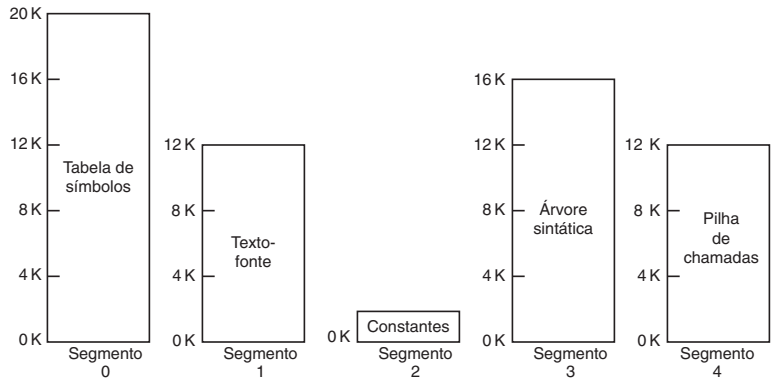
- Para alguns problemas, ter dois ou mais **espaços de endereçamento** é melhor do que um
- Em um compilador, o crescimento de uma **tabela de símbolos** é diferente do **texto fonte** (código)
- Fornecer espaços de endereçamento completamente independentes, **segmentos**
- Segmentos podem ter tamanhos diferentes
  - pode causar fragmentação externa
- Solução: segmentação paginada

Segmentação



- Espaço de endereçamento unidimensional com tabelas crescentes
- Uma tabela pode atingir outra

Segmentação



Permite que cada tabela cresça ou encolha, independentemente

Implementação de segmentação

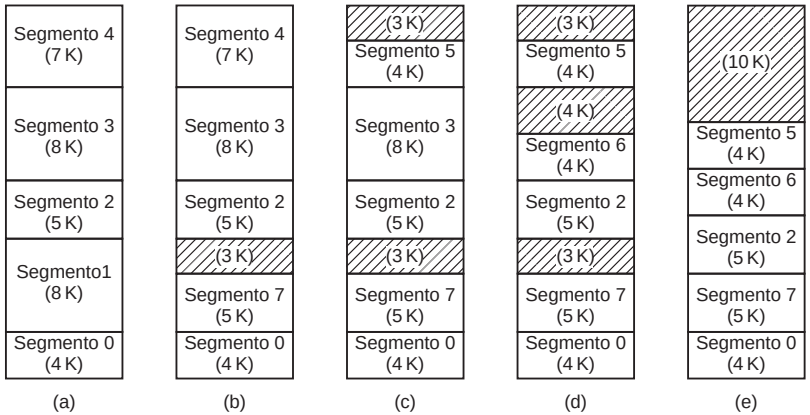
- O espaço de endereçamento de um processo é composto por um conjunto de segmentos, que podem ser de tamanhos diferentes
- Cada segmento pode ser descrito por um par de endereços base e limite
  - análogo ao usado com alocação contígua
- A descrição dos segmentos é armazenada em uma tabela de segmentos
- Um endereço lógico é um par (*segmento*, *deslocamento*)

Tabela de segmentos

segmento	base	limite
0	3400	5000
1	7000	11476
2	0	3200
3	5500	6500

- EL: $(0, 1400) \rightarrow$ EF: $3400 + 1400 = 4800$
- EL: $(1, 1400) \rightarrow$ EF: $7000 + 1400 = 8400$
- EL: $(2, 1400) \rightarrow$?
- EL: $(3, 1400) \rightarrow$?

Fragmentação externa na segmentação



- (a)–(d) Desenvolvimento de fragmentação externa
- (e) Remoção da fragmentação via compactação

Segmentação com paginação

- Combina segmentação e paginação
  - segmentação paginada
- Um segmento não é alocado contiguamente, mas é paginado
  - cada segmento possui uma tabela de páginas
- Um endereço lógico é dividido em três partes
  - número do segmento
  - número da página (referente ao segmento)
  - deslocamento dentro da página

  $\Rightarrow$ estrutura análoga à paginação multinível
- Elimina fragmentação externa na segmentação

Sumário

1. Conceitos básicos
2. Gerenciamento sem abstração de memória
3. Gerenciamento com espaços de endereçamento
4. Memória virtual
5. Gerência de memória no Linux

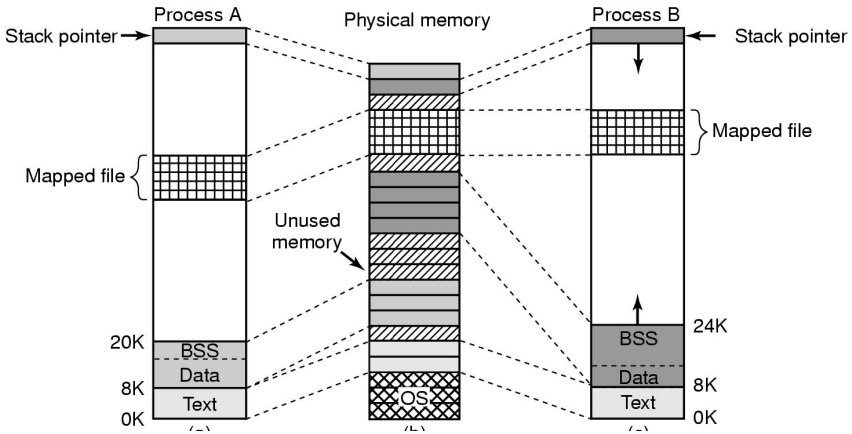
Gerência de memória no Linux

- A gerência de memória do Linux é bastante complicada
- Objetivo é ilustrar os principais conceitos
  - organização da memória lógica/física
  - alocação de memória
  - tabelas de páginas
  - substituição de páginas
  - caches de páginas e de swap
- Arquitetura considerada é Intel de 32 bits (x86-32)

Visão do usuário (1/2)

- Cada processo possui segmentos de código, dados e pilha
  - dados podem ser inicializados e não inicializados (BSS)
  - um arquivo executável contém código e dados inicializados, mais o tamanho do segmento BSS
- Sistema oferece suporte a
  - compartilhamento de memória
  - arquivos mapeados em memória
  - *copy-on-write*

Visão do usuário (2/2)



Chamadas de sistema

- As chamadas de sistema para gerência de memória não fazem parte do padrão POSIX
  - dependentes de arquitetura
  - padrão especifica o uso de `malloc()` et al.
- Principais chamadas do Linux
  - `brk`: ajusta o tamanho do segmento de dados (para mais ou para menos)
    - usada na implementação de `malloc()`
  - `mmap`: mapeia um arquivo em memória (no todo ou em parte)
  - `munmap`: desfaz um mapeamento de arquivo

Organização do espaço de endereçamento

- O espaço de endereçamento virtual na arquitetura x86-32 tem 4 GB
  - 0–3 GB (`0x00000000`–`0xbfffffff`): espaço do usuário
  - 3–4 GB (`0xc0000000`–`0xffffffff`): espaço do núcleo
    - o núcleo deve caber em 1 GB de RAM
    - as tabelas de páginas de todos os processos têm entradas idênticas para o espaço do núcleo
    - só pode ser acessado pelo próprio SO

Utilização da memória física

- O núcleo fica totalmente residente na memória física
  - não é paginado para o disco
- O restante da memória física é usado por páginas de usuários, cache de páginas, buffers

Descritores de página

- Um **descritor de página** é mantido para cada página física
- Estrutura de dados que descreve o estado da página
  - travada, suja, acessada, ativa/inativa, ...
- Contador de referências: em quantas tabelas de páginas esta moldura aparece?
- `struct page` em `<linux/mm_types.h>`

Organização da memória física

- Nem toda a memória é igualmente endereçável
  - tipos diferentes de memória precisam ser usados para propósitos distintos
- Linux divide a memória física em **zonas**
  - `ZONE_DMA`: alguns dispositivos de E/S só conseguem fazer DMA usando endereços < 16 MB
  - `ZONE_NORMAL`: memória regular, até 896 MB
  - `ZONE_HIGHMEM`: memória acima de 896 MB

Memória alta

- Na arquitetura x86-32, o núcleo não pode endereçar mais de 1 GB de memória
  - não há espaço de endereçamento virtual suficiente (3 GB/1 GB)
- A memória até 896 MB ($= 1\,\text{GB} - 128\,\text{MB}$) é endereçada diretamente
- Endereços virtuais entre 896 MB e 1 GB mudam constantemente, para endereçar memória física em endereços mais altos
- Em máquinas de 64 bits essa preocupação não existe
  - toda a memória é endereçável diretamente

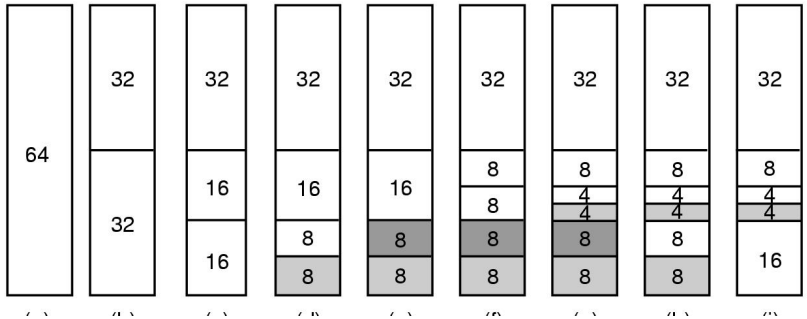
Alocação de memória no núcleo

- O núcleo possui três mecanismos de alocação de memória
  - alocador de páginas
  - alocador de fatias (*slab allocator*)
  - `vmalloc`

Alocador de páginas

- Aloca páginas de memória física para o núcleo
- Utiliza o algoritmo *buddy*
  - todas as páginas físicas da memória são agrupadas em listas
  - cada lista contém regiões (livres) com $2n$ páginas
    - Linux usa regiões com 1, 2, 4, 8, ..., 1024 páginas (4 MB)
  - quando é feita uma solicitação de memória, ela é arredondada para cima para $2k$ páginas, e busca-se uma entrada na lista correspondente
  - se não houver entrada disponível na lista de $2k$, uma entrada da lista de regiões maiores é quebrada em duas regiões, cada uma com $2k$ páginas
    - uma é alocada para o chamador, a outra é inserida na lista de $2k$
  - quando uma região é liberada, regiões contíguas podem ser combinadas recursivamente
  - fragmentação interna pode ser grande
- Listas separadas são mantidas para cada zona (`/proc/buddyinfo`)
  - `ZONE_DMA`, `ZONE_NORMAL`, `ZONE_HIGHMEM`

O algoritmo *buddy*



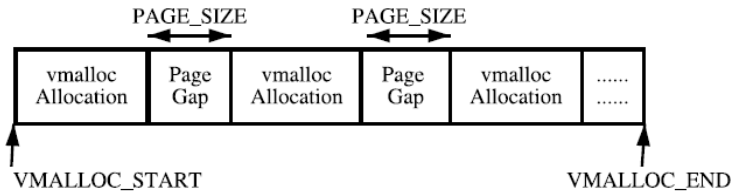
(a) Mem. com 64 págs
(b)–(d) Alocação de 8 págs
(e) Alocação de 8 págs
(f)–(g) Alocação de 4 págs
(h) Liberação de 8 págs
(i) Liberação de 8 págs

Alocador de fatias (*slab allocator*)

- Diversas estruturas de dados são frequentemente alocadas e desalocadas
  - descritores de processos, i-nodes, entradas de diretório, ...
- Em vez de alocar dinamicamente memória para essas estruturas, elas são alocadas em grupos e inseridas em uma lista de regiões de memória do mesmo tamanho → cache de objetos (*object cache*)
  - alocação de um objeto busca primeiro uma estrutura livre na lista
    - se não houver, o alocador de páginas é invocado
  - liberação de um objeto o devolve para a lista
    - se um conjunto de objetos livres constitui uma página, essa página pode ser recuperada
- `/proc/slabinfo`
- Múltiplas implementações do mesmo conceito básico
  - SLAB, SLOB, SLUB, SLAM, ...
  - `http://tinyurl.com/ckbbm5s`

vmalloc

- Aloca regiões de memória contíguas no espaço de endereçamento virtual, mas não na memória física
  - área compreendida entre `VMALLOC_START` e `VMALLOC_END`
- Memória alocada é arredondada para cima para um número inteiro de páginas
- Para reduzir riscos de acesso errôneo à memória, existe um intervalo de pelo menos uma página entre duas regiões alocadas com `vmalloc`
- Usado principalmente na carga de módulos dinâmicos do núcleo



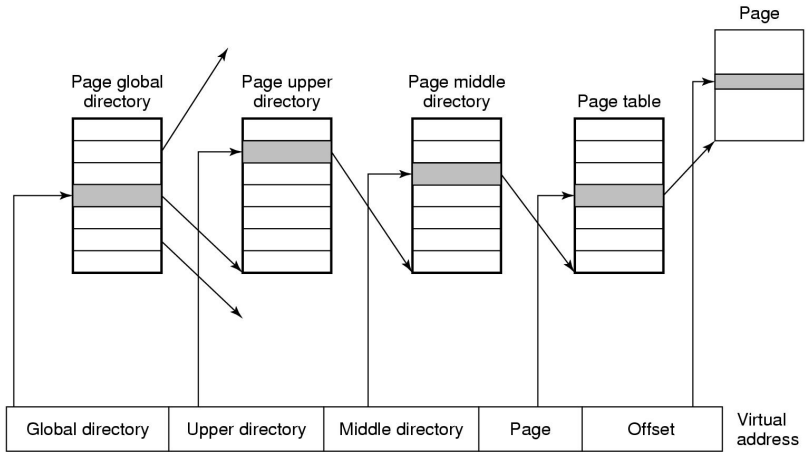
Representação do espaço de endereçamento virtual

- O espaço de endereçamento virtual é representado por um conjunto de **áreas** distintas
  - segmentos de código/dados/pilha, heap, arquivos mapeados em memória, ...
- Cada área é contígua, mas pode haver buracos entre as áreas
- Uma área é descrita por uma `struct vm_area`
  - o conjunto das áreas de um processo é organizado como uma lista encadeada e como uma árvore
  - contém bits de proteção, informações de compartilhamento e travamento, referência ao arquivo que serve como apoio de armazenamento, ...

Tabelas de páginas (1/2)

- Por portabilidade, o Linux usa tabelas de páginas de 4 níveis, independente da arquitetura
- Arquiteturas que não suportam 4 níveis precisam "emular" essa estrutura
  - em x86-32, os diretórios superior e intermediário têm tamanho um, e não "consomem" bits do endereço virtual
  - os 32 bits de endereço virtual são divididos em
    - 10 bits para indexar o diretório global
    - 10 bits para indexar a tabela de páginas
    - 12 bits de deslocamento

Tabelas de páginas (2/2)



O algoritmo de recuperação de molduras de página

- *Page frame reclaiming algorithm* (PFRA)
- Algoritmo usado para liberar páginas na memória física

Tipos de páginas

tipo	descrição	ação
Irrecuperáveis	páginas travadas, pilhas em modo núcleo, páginas livres, etc.	impossível
Trocáveis	páginas anônimas de usuário	gravar na área de troca
Sincronizáveis	pedaços de arquivos; páginas de usuário mapeadas	gravar no arquivo se necessário
Descartáveis	páginas sem uso	nada a fazer

Páginas mapeadas e páginas anônimas

- Uma **página mapeada** é parte de um arquivo
  - corresponde a um bloco no sistema de arquivos
- Frequentemente, é apenas de leitura, e não pode estar suja
- Um **página anônima** não corresponde a nenhum arquivo
  - pode ser parte do segmento de dados ou pilha
  - precisa ser salva na área de troca

Princípios da recuperação

- Primeiro são recuperadas páginas que não estão associadas a nenhum processo
  - não exigem mudanças nas tabelas de páginas
- Praticamente todas as páginas de usuário são recuperáveis
- Para páginas compartilhadas, é preciso limpar as entradas em todas as tabelas de páginas de uma vez
- Busca-se recuperar apenas páginas sem uso, i.e., aquelas que não foram referenciadas recentemente

Quando recuperar memória

- Pouca memória disponível
- Hibernação
- Periodicamente
  - garante que haja memória disponível quando for solicitada
  - importante para evitar deadlocks no núcleo

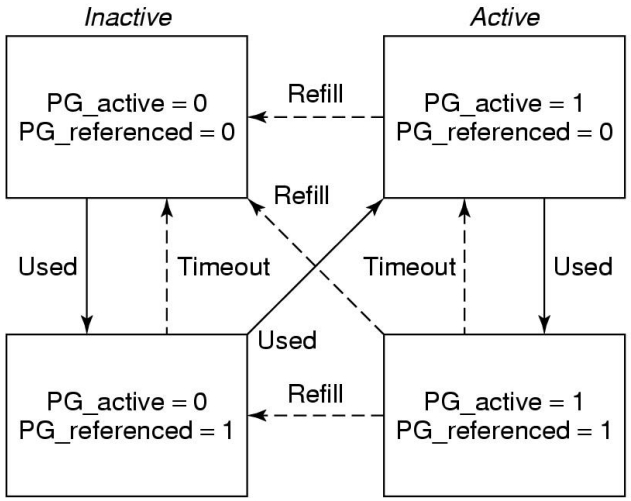
Listas LRU (*least recently used*)

- Cada processo possui duas listas de páginas, **ativas** e **inativas**
- O desejável é recuperar páginas inativas primeiro
- Se uma página não foi referenciada recentemente, ela migra para a lista de inativas
- Quando uma página é referenciada, ela **não entra** na lista de ativas **imediatamente**

Ativação em duas etapas

- A primeira vez que uma página é acessada, a flag `PG_referenced` é ligada
- A próxima vez que a página for acessada ela é movida para a lista de ativas (`PG_active` $\leftarrow 1$)
  - são necessários dois acessos para que uma página seja declarada ativa
  - mais especificamente, dois acessos em intervalos distintos
- Se o segundo acesso não ocorrer dentro de um período, a flag `PG_referenced` é desligada

Diagrama de estados de páginas



Estados de acesso

- Quando uma página é referenciada, ela passa para um estado mais ativo
- Depois de dois acessos, ela é movida para a lista de ativas
- Quando uma página não é usada durante um tempo, ela passa para um estado menos ativo
- Depois de dois timeouts, ela é movida para a lista de inativas
- Se o sistema estiver com pouca memória, as páginas podem ser rebaixadas independente do seu estado de atividade

Repreenchimento (*refilling*) de memória

- Periodicamente, tenta tornar páginas recuperáveis
- Se for muito agressivo, muitas páginas ativas são recuperadas
- Se for muito conservador, pouca memória estará disponível
- Estratégia adaptativa
  - aumenta ou diminui a taxa de repreenchimento conforme a necessidade de memória do sistema
  - objetivo é que o número de páginas ativas seja $\approx 2/3$ do número de páginas inativas

Gravação de páginas sujas

- Normalmente, poucas páginas sujas são salvas no disco a cada vez
- A escrita ocupa o disco, e pode ser inútil
  - se for antecipatória, a página pode nunca ser recuperada
  - a página pode ser suja novamente antes de ser recuperada
- Em situações de pouca memória, um número mais significativo de páginas pode ser salvo
  - ex: falha de um pedido de alocação de memória

Recuperação periódica

- Uma thread de núcleo `kswapd` executa periodicamente para recuperar memória
- Alguma memória **precisa** estar livre durante todo o tempo
  - muitas vezes é preciso alocar memória no tratador de interrupções, que não pode bloquear
- Se houver pouca memória na zona ($<$ `pages_min`), tenta liberar mais memória
- Se houver muita memória livre ($>$ `pages_high`), não faz nada
- `kswapd` recupera até 32 páginas e entrega a CPU para o escalonador

Cache de páginas

- A cache de páginas mantém em memória páginas recentemente lidas do disco
- Existem basicamente quatro tipos de páginas na cache
  - páginas carregadas para atender a uma falta de página em um arquivo mapeado em memória
  - blocos lidos de um dispositivo de bloco ou sistema de arquivos
    - agrupados em páginas de buffer (*buffer pages*)
    - número de blocos por página é dependente de arquitetura
  - páginas anônimas que ocupam uma área especial → cache de swap
  - páginas pertencentes a regiões de memória compartilhada são tratadas como páginas anônimas
    - a diferença é que as páginas compartilhadas são adicionadas ao cache de swap e têm espaço reservado no armazenamento de suporte (*backing storage*) imediatamente após a primeira escrita na página
- As caches de páginas e de blocos (*buffer cache*) são unificadas
  - **todas as operações com arquivos passam pela cache de páginas**

O *swapper*

- O *swapper* escreve páginas na área de troca (*swap*) e lê outras páginas
  - o nome *swapper* é incorreto, pois isso é paginação
- Ele também gerencia as áreas de troca no disco
  - geralmente uma partição separada, mas pode ser um arquivo regular
    - partição tem melhor desempenho
- Mantém o mapeamento entre os endereços virtuais de cada processo e os endereços dos blocos de disco
  - endereços de blocos são armazenados nas entradas de tabela de páginas que se referem a páginas no disco

Distribuição de páginas

- Tenta manter páginas em regiões contíguas da área de troca
  - redução no tempo de seek
- Quando existem múltiplas áreas de troca, elas são priorizadas por velocidade de acesso
  - sistema pode ter até 32 áreas de troca ativas
- Round-robin é usado entre áreas de troca com a mesma velocidade

Cache de swap

- A transferência de páginas de/para a área de troca está sujeita a diversas condições de disputa
  - dois processos tentam trazer, ao mesmo tempo, uma mesma página compartilhada para a memória física
  - um processo tenta ler uma página que está sendo enviada para o disco
- A cache de swap (*swap cache*) é usada como "dona" intermediária das páginas
  - na verdade uma área da cache de páginas
  - todas as tentativas de modificar o status de uma página passam pela cache
    - permite o controle da concorrência, eliminando a disputa

Flusher threads

- Conjunto de threads do núcleo responsável por salvar páginas sujas da cache de páginas no disco
  - função anteriormente desempenhada pelas threads `pdflush`
- As *flusher threads* são acordadas em duas situações
  1. periodicamente (500 ms), para manter a consistência
  2. quando a memória disponível cai abaixo do mínimo
- É usada uma thread por dispositivo de armazenamento
  - independe do número de partições do dispositivo

Bibliografia Básica

Andrew S. Tanenbaum.
Sistemas Operacionais Modernos, 3ª *Edição*. Capítulo 3.
Pearson Prentice-Hall, 2010.

Abraham Silberchatz, Greg Gagne e Peter Baer Galvin.
Fundamentos de Sistemas Operacionais, 6ª *Edição*.
LTC - Livros Técnicos e Científicos Editora S.A., 2004.

Carlos Alberto Maziero.
Sistemas Operacionais. Capítulo V – Gerência de Memória.
`http://dainf.ct.utfpr.edu.br/~maziero/doku.php/so:livro_de_sistemas_operacionais`