

Memória Virtual

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho



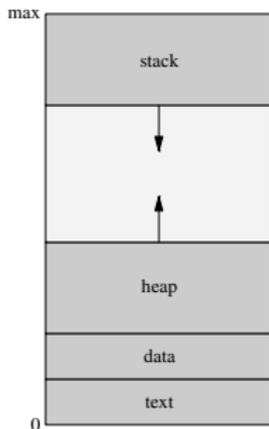
Memória virtual

- Separar o conceito de memória lógica/virtual que os processos manipulam da sua existência em memória física
- Apenas parte do programa necessita de estar em memória em cada instante
- Espaço de endereçamento virtual manipulado pode ser muito maior do que a memória física
- Memória secundária (disco) usada para guardar partes da memória virtual
- Pode ser implementada via paginação ou segmentação
- Implementada eficientemente via paginação **on demand**



Espaço de endereçamento lógico/virtual de processo

- Não pode ser prevista à partida memória necessária
- Grande espaço de endereçamento com stack “longe” do heap
- E se espaço de endereçamento maior do que memória física?
- Processo não usa todo espaço de endereçamento; e.g. arrays definidos por excesso; rotinas não usadas



Benefícios de processo parcialmente em memória

- Tamanho do programa não seria restringido pela memória física:
 - programadores escrevem para espaço virtual grande
 - programas funcionam em máquinas com diferente memória
- Cada programa necessita de menos memória:
 - mais processos ao mesmo tempo
 - maior utilização de CPU
- Menos I/O a carregar ou fazer swap; programas correm mais rápido



Partilha de memória e ficheiros

Sistemas de memória virtual permitem tipicamente:

- Partilha de bibliotecas entre processos via objectos partilhados mapeados na mesma memória física
- Partilha de memória para comunicação entre processos
- Partilha de páginas na criação de processos com fork



Demand paging

- Usado em sistemas de memória virtual
- Em vez de fazer swap-in de todas as páginas do processo ...
- Trazer página para memória apenas quando necessária:
 - menos I/O
 - menos memória necessária
 - resposta mais rápida
 - menos memória por processo; mais processos em memória
- **Pager: lazy swapper** – só traz página para memória se necessária
- Página endereçada; possibilidades
 - página inválida – abortar programa
 - página não presente – trazer para memória



Bit de válido/inválido na tabela de páginas

- Cada entrada na tabela de páginas tem bit válido/inválido
- Bit é aproveitado para implementar memória virtual:
- Válido: se endereço válido e página mapeada em memória física
- Inválido se:
 - página inválida
 - página em disco; não mapeada em memória física
- E se endereçada página com bit inválido?



Page fault

- Quando endereçada página com bit inválido – trap de endereçamento: **page-fault**
- Sistema operativo consulta outra tabela e decide:
 - endereço inválido: aborta programa
 - página não presente em memória: trazer página
- Passos para trazer página:
 - encontrar e alocar frame livre
 - escalonar leitura da página em disco para a frame
 - executar outro processo
 - mais tarde, aquando interrupção leitura terminada:
 - modificar tabela de páginas marcando página válida;
 - recomeçar execução de instrução interrompida
- Estes passos são transparentes ao processo, que recomeça como se a página sempre estivesse em memória



Suporte a demand paging

- O suporte de hardware para demand paging é:
 - tabela de páginas; com bit válido/inválido
 - memória secundária – **swap space**: área de disco dedicada
 - recomeço de execução de instrução aquando page-fault
- O recomeço de execução de instruções pode ser complicado:
 - instruções de transferência de blocos
 - e se houver sobreposição fonte-destino?



Performance de demand paging

- Quando há page-fault, é necessário:
 - servir interrupção
 - ler página
 - recomençar processo
- Trazer página demora; e.g. 8ms
- Mesmo podendo executar outro processo entretanto:
 - se este novo processo causa page-fault . . .
 - pode conduzir a todos os processos à espera das páginas
- Performance dependente do **page-fault rate** p



Performance de demand paging

- Com acesso à memória m e a trazer página de disco d
- Tempo de acesso efectivo:

$$TAE = (1 - p) \times m + p \times d$$

- Se $m = 200ns$, $d = 8ms$ e $p = 1/1000$:
TAE = $8200ns = 41 m$
- Essencial que p seja muito baixo
- Com valores m e d acima, para não haver degradação de desempenho necessário $p < 1/40000$



Copy-on-write

- Permite processos pai e filho partilhar páginas inicialmente
- Página só é copiada quando um deles tenta modificar página
- Página é marcada na tabela de páginas para permitir detecção; e.g. bits de permissão
- Permite criação eficiente de processos via fork



E se não houver frames livres?

- Ao trazer páginas de disco, eventualmente memória cheia
- Quando não há frames livres, necessário libertar frame
- Necessário escolher página para libertar
- Como escolher?
 - páginas que não estejam a ser muito usadas
 - necessário algoritmo de substituição de páginas (**page replacement**)
- Páginas alternam entre memória e disco; podem ser trazidas várias vezes de disco



Substituição de páginas

- Rotina de tratamento de page-faults modificada para incluir libertar páginas
- Páginas em memória que não tenham sido modificadas não necessitam de ser escritas em disco
- Como saber se foi modificada?
 - via **modify (dirty) bit** - colocado a 1 quando página modificada
 - e se não houver suporte de hardware?
- Páginas modificadas são escritas em disco
- Páginas libertadas são marcadas como inválidas



Substituição de páginas - passos básicos

- 1 Encontrar página em falta no disco
- 2 Encontrar uma frame livre:
 - se houver livre – usar
 - caso contrário:
 - 1 correr algoritmo para seleccionar vítima
 - 2 marcar página inválida
 - 3 transferir para disco se modificada
- 3 Trazer página para a frame libertada
- 4 Actualizar tabelas de páginas e frames
- 5 Recomeçar processo



Algoritmos de substituição de páginas

- Objectivo: minimizar a frequência de page-fault
- Como avaliar algoritmos?
 - correr algoritmo sobre sequência de referências à memória
 - e.g. 1,2,3,4,1,2,5,1,2,3,4,5
 - contar o número de page-faults ocorrido
- Vários algoritmos possíveis:
 - FIFO
 - Óptimo
 - LRU
 - Bit de referência
 - Second-chance
 - Via Countadores



FIFO

- Expulsa a página há mais tempo em memória
- Pode ser implementado com lista ligada de páginas
- Lista apenas actualizada em page-faults
- Simples mas mau algoritmo
- Exemplo: com referências 1,2,3,4,1,2,5,1,2,3,4,5
- Com 3 frames, 9 page-faults:

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

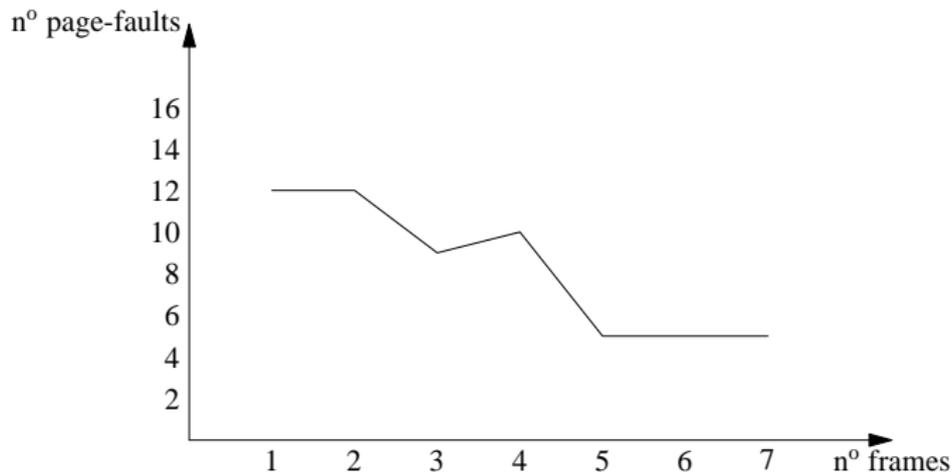
- Com 4 frames, 10 page-faults:

1	1	1	1	5	5	5	5	4	4
	2	2	2	2	1	1	1	1	5
		3	3	3	3	2	2	2	2
			4	4	4	4	3	3	3



Anomalia de Belady

- Esperado que com mais frames menos page-faults ocorram
- Nem sempre é verdade; e.g. no FIFO
- Anomalia de Belady: mais frames \rightarrow mais page-faults



Algoritmo óptimo

- Expulsa a página que não será usada durante mais tempo
- Exemplo: com referências 1,2,3,4,1,2,5,1,2,3,4,5
- Com 4 frames, 6 page-faults:

1	1	1	1	1	4
	2	2	2	2	2
		3	3	3	3
			4	5	5

- Impossível de implementar: necessita conhecimento do futuro
- Serve como referência do melhor que se pode fazer para avaliar outros algoritmos



LRU – least recently used

- Expulsa a página que foi usada menos recentemente
- Exemplo: com referências 1,2,3,4,1,2,5,1,2,3,4,5
- Com 4 frames, 8 page-faults:

1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2
		3	3	5	5	4	4
			4	4	3	3	3

- Melhor do que FIFO
- Mais complexo de implementar: necessita registar acessos



LRU – least recently used

- Implementação com contadores:
 - cada página tem contador
 - quando referenciada, copiar relógio para contador
 - expulsar a página com contador mais antigo
- Implementação com lista duplamente ligada:
 - página referenciada – mover para um extremo
 - página a expulsar – do outro extremo
 - não necessita pesquisa
 - complexo: actualização de apontadores em memória por cada acesso



Aproximações a LRU

- LRU é interessante em teoria mas complexo
- Aproximações ao LRU são usadas
- Bit de referência:
 - inicialmente: bit=0
 - quando acedida página: bit=1
 - expulsar página com bit=0
 - aproximação grosseira: pode haver muitas ou nenhuma com bit=0
- Histórico do bit de referência:
 - mantido byte com histórico do bit de referência
 - periodicamente feito right-shift para bit mais significativo
 - expulsa página com menor valor



Aproximações a LRU

- Second-chance ou clock:
 - bit de referência + FIFO
 - se bit=0, expulsar
 - se bit=1, marcar bit=0 e avançar para próxima
 - implementado com lista circular
- Second-chance melhorado:
 - second chance + bit modificado
 - classificar páginas segundo bits de referência e modificado:
 - (0,0)
 - (0,1)
 - (1,0)
 - (1,1)
 - Expulsar a próxima da classe mais baixa



Substituição baseada em contadores

- Manter contador de referências à página
- Least frequently used (LFU):
 - substituir a página com contador menor
 - e se foi acedida há muito tempo?
 - fazer right-shift do contador para provocar esquecimento exponencial
- Most frequently used (MFU):
 - substituir página com contador maior
 - racional: páginas trazidas recentemente, que ainda devem ser usadas, têm contador baixo
 - pouco interessante
- Algoritmos de implementação difícil e fracas aproximações ao ótimo; pouco usados



Buffer de páginas libertadas

- Deixar acabar frames livres prejudicial: processo que necessita de frame tem que esperar pela substituição
- Solução: quando atingido limiar, começar a rejeitar páginas
- Páginas rejeitadas vão para lista de livres
- Melhoria:
 - manter associação página–frame na lista de livres
 - em page-fault ver primeiro se está na lista de livres
 - pode ser adicionada a diferentes algoritmos de substituição
 - e.g. usada com FIFO em VAX/VMS
 - e.g. usada com second-chance em alguns UNIX
- Cuidado com transições: e.g. página rejeitada mas ainda não escrita em disco



Alocação de Frames

- Processos necessitam de várias frames para correrem
- Arquitectura dita número mínimo: para todas as instruções poderem recomeçar
- Muito poucas frames provocam rácio de page-faults elevado
- Vários processos competem por memória
- Como alocar frames?
 - alocação fixa versus prioridade
 - alocação local versus global



Alocação fixa

- Distribuir um número de frames fixas a cada processo
- Equalitária:
 - o mesmo número de frames a cada processo
 - e.g. 1000 frames, 10 processos: 100 frames a cada processo
 - problema: necessidades de memória variam muito
- Proporcional:
 - atribuir número de frames proporcional à memória virtual necessária ao processo
 - sendo v_i memória virtual do processo i
 - e $V = \sum_i v_i$ memória virtual de todos os processos
 - e f número de frames total
 - alocação ao processo i : $f_i = f \times v_i / V$



Alocação por prioridade

- Processos de mais alta prioridade devem correr mais rápido
- Alocação pode ter em conta prioridade do processo
- Alocação proporcional à prioridade
- Esquema combinado de tamanho e prioridade dos processos



Alocação local versus global

- Alocação local:
 - a cada processo é atribuído número de frames
 - substituição de páginas local ao processo
 - comportamento não depende de outros processos
- Alocação global:
 - não é definido à partida frames por processo
 - algoritmo de substituição de páginas global: vítima escolhida pode ser do mesmo ou de outro processo
 - comportamento depende do comportamento de outros processos
 - otimiza melhor uso global da memória; melhora throughput; método mais usado

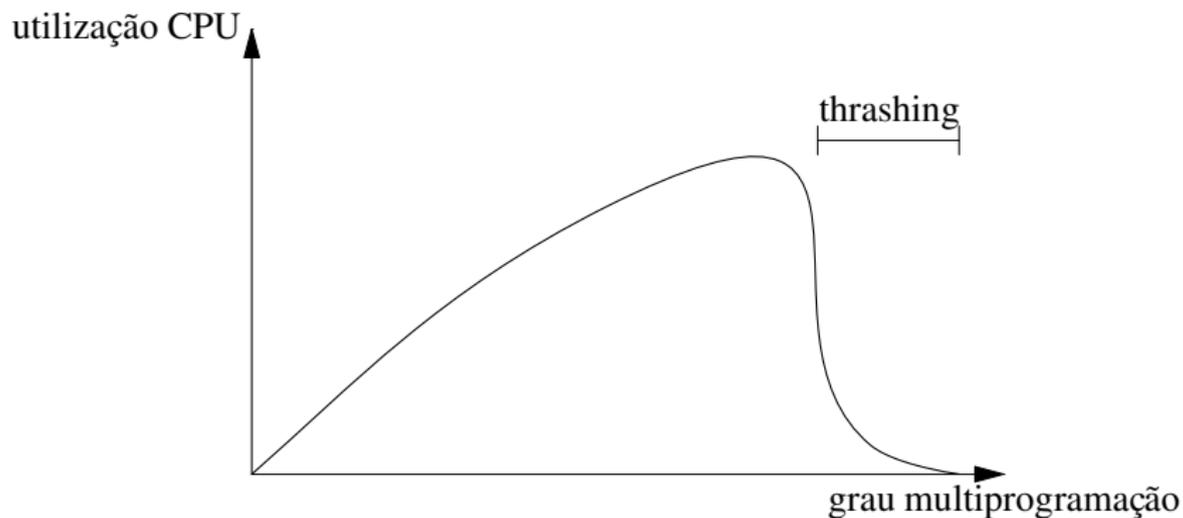


Thrashing

- Se um processo tiver muito poucas frames atribuídas:
 - quando há page fault, página é substituída ...
 - página substituída necessária logo em seguida ...
 - nova page-fault; page-fault rate elevadíssimo
 - processo passa mais tempo em paging do que a executar
- Diz-se que o processo está em **thrashing**
- Causas de thrashing:
 - demasiados requisitos de memória virtual devido a elevado grau de multiprogramação
 - pode ser causado por escalonador
 - alocação global com prioridades: thrashing em processos de baixa prioridade



Thrashing e grau de multiprogramação



Cenário conduzente a thrashing

- Uma combinação de factores pode levar inesperadamente a thrashing:
 - substituição global de páginas
 - escalonador que olha apenas à utilização de CPU
- Sequência de eventos:
 - processo entra em nova fase e necessita páginas novas
 - começam page-faults
 - frames são obtidas retirando páginas a outros processos
 - estes processos necessitam páginas, causando mais page-faults
 - todos estes processos ficam na fila de espera de I/O
 - a ready queue esvazia-se; diminui a utilização de CPU
 - o escalonador decide aumentar grau de multiprogramação
 - novo processo retira frames aos existentes; mais page-faults
 - aumenta fila de I/O devido a paging
 - diminui ainda mais utilização de CPU
 - ciclo vicioso . . .



Prevenção de thrashing

- Quando o cenário anterior acontece a solução é diminuir o grau de multiprogramação
- O escalonador não pode olhar cegamente apenas para a utilização de CPU
- Alocação local de frames solução apenas parcial:
 - confina thrashing a um processo
 - mas mesmo num processo, thrashing prejudica o sistema todo
- Solução: dar a cada processo que corra um número apropriado de frames, ainda que limitando o grau de multiprogramação
- Como saber quantas frames são necessárias?

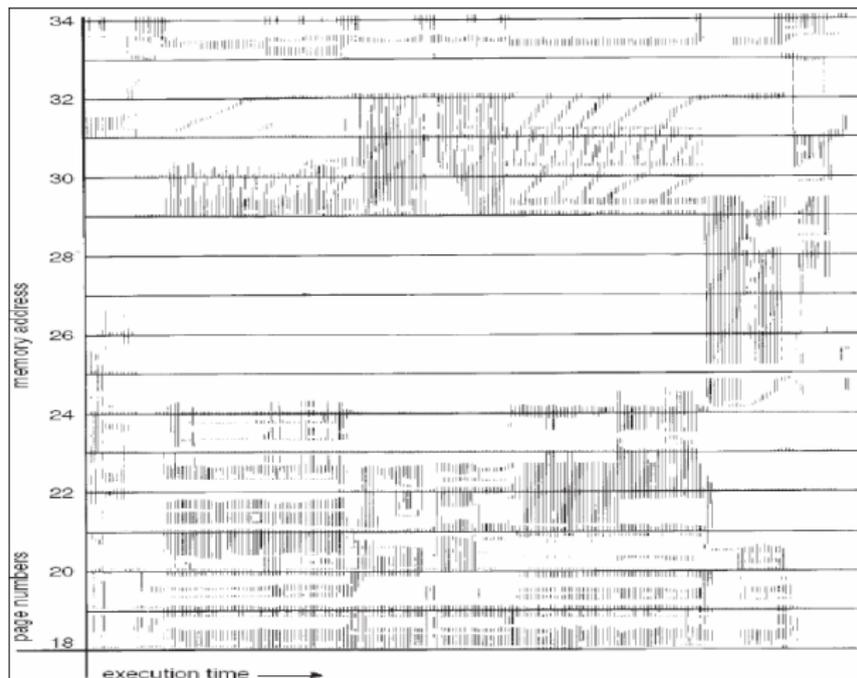


Localidade espacial e temporal

- Memória virtual viável porque processos não acedem à memória aleatoriamente
- Localidade espacial: se um endereço é usado, endereços próximos serão usados
- Localidade temporal: se um endereço é usado, provavelmente continuará a ser usado nos próximos instantes
- Motivação:
 - processo executa função: usa código e stack das variáveis locais
 - ciclos levam a acesso repetido aos mesmo endereços
 - travessia de arrays leva a acessos a endereços próximos
- Resultado: um processo, em cada instante apenas acede a um conjunto (pequeno) de páginas, uma **localidade**
- Processos vão passando por diferentes localidades, tipicamente sobrepostas



Localidade num padrão de referências



Modelo do working-set

- **Working-set**, para uma janela temporal Δ , é o conjunto das páginas que foram acedidos nos mais recentes Δ acessos

- Exemplo, para $\Delta = 10$:

2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3

t_1 t_2

$$WS_{\Delta}(t_1) = \{1, 2, 5, 6, 7\} \quad WS_{\Delta}(t_2) = \{3, 4\}$$

- Working-set é uma aproximação da localidade
- Δ não pode ser demasiado pequeno nem demasiado grande
- Working-set pode ser aproximado:
 - usando um temporizador e bit de referência
 - bit referência copiado para memória e apagado
 - shift-register em memória guarda p períodos mais recentes
 - página no working-set se estiver a 1 no bit de referência ou shift-register



Evitar thrashing via working-set

Thrashing é evitado:

- dando a cada processo frames para caber o seu working set
- o working-set de cada processo varia com o tempo
- aumentar o grau de multiprogramação enquanto soma dos working-sets não exceder memória
- se soma dos working sets aumentar e exceder memória:
 - escolher um processo para suspender (swap-out)
 - alocar frames desse processo pelos restantes
 - quando necessidades diminuírem fazer swap-in do processo



Evitar thrashing via frequência de page-faults

- Thrashing significa elevada frequência de page-faults
- Medir frequência de page-faults permite controlar thrashing mais directamente
- Com mais frames: frequência de page-faults diminui
- Abordagem:
 - estabelecer limite inferior e superior de frequência
 - se limite superior ultrapassado: dar frame ao processo
 - se limite inferior ultrapassado: retirar frame ao processo
 - se não houver mais frames: suspender um processo
 - quando houver frames: trazer processo de volta



Estrutura dos programas e memória virtual

- Conhecimento do funcionamento da memória virtual pode ajudar programadores a melhorar desempenho de programas
- Exemplo:
 - dado páginas de 4KB, inteiros de 4B e `int a[1024,1024];`
 - cada linha do array ocupa uma página

- Programa 1:

```
for (j = 0; j < 1024; j++)  
    for (i = 0; i < 1024; i++)  
        a[i, j] = 0;
```

- Programa 2:

```
for (i = 0; i < 1024; i++)  
    for (j = 0; j < 1024; j++)  
        a[i, j] = 0;
```

- Com poucas frames por processo:

- programa 1 pode causar $1024 \times 1024 = 1048576$ page-faults
- programa 2 pode causar 1024 page-faults



Pre-paging

- Começar um processo sem páginas mapeadas leva a grande número de page-faults
- Solução: fazer **pre-paging** de páginas que o processo necessite
- Outra situação:
 - conhecendo o working-set de um processo
 - quando necessário suspender processo, memorizar working-set
 - quando é reactivado, fazer pre-paging do working-set
- Vantagens do pre-paging:
 - menos serviço de interrupções de page-fault
 - possível leitura em bloco do disco; mais eficiente



Influência do tamanho de página

- Páginas pequenas:
 - menor fragmentação interna
 - maior resolução: menos memória para armazenar localidade
- Páginas grandes:
 - menor tabela de páginas
 - transferência de disco mais rápida (por unidade de memória)
 - menos page-faults por gama de memória necessária
- Tendência é para maiores tamanhos e escolha por software



Ficheiros mapeados em memória

- Ficheiros mapeados em memória permitem acesso a ficheiros ser feito como acesso a memória
- Mecanismo de memória virtual é usado para mapear espaço de endereçamento do ficheiro em endereços virtuais
- Ficheiro pode ser trazido e acedido via demand paging
- Pode existir chamada ao sistema explícita (e.g. `mmap`) para mapear ficheiro em memória
- Certos sistemas operativos (e.g. Solaris) mapeiam ficheiros em memória mesmo quando acedidos via `open`, `read`, `write`
- Permite partilha: vários processos podem mapear o mesmo ficheiro num espaço partilhado



Alocação de memória no kernel

- Memória do kernel é tratada de modo diferente
- É útil que dispositivos possam aceder directamente a memória física (e.g. DMA) sem passar por paginação
- Motiva gamas contíguas de memória física
- Sistemas operativos podem não submeter memória do kernel à paginação
- Kernel usa mecanismos próprios de alocação de memória:
 - buddy
 - slab



Sistema buddy

- Aloca memória de segmento contíguo de tamanho fixo
- Memória alocada em blocos com tamanho potência de 2:
 - pedido é arredondado para próxima potência de 2
 - espaço total, blocos alocados e livres são todos potência de 2
 - bloco livre grande é sucessivamente dividido em 2 buddies com metade do tamanho, até ao tamanho arredondado
- Fusão de blocos é simples:
 - se buddy do bloco está livre, fundir com buddy
 - repetir com o bloco resultante
- Desvantagem: grande fragmentação interna



Alocação slab

- Terminologia:
 - **slab**: uma ou mais páginas fisicamente contíguas
 - **cache**: um ou mais slabs
- Cache própria para cada tipo de estrutura de dados do kernel
- Cada cache contém objectos: instâncias da estrutura de dados
- Na criação: cache preenchida com objectos marcados livres
- Cada slab pode estar como:
 - cheio: todos os objectos estão usados
 - livre: todos os objectos estão livres
 - parcial: alguns objectos usados



Alocação slab

- Quando pedido de alocação de objecto:
 - devolvido objecto em slab parcial, senão
 - devolvido objecto em slab vazio, senão
 - novo slab alocado à cache
- Quando libertação de objecto: objecto marcado como livre
- Benefícios:
 - devolvida memória exacta para cada objecto
 - quase ausência de fragmentação
 - alocação e desalocação muito rápida
- Linux usava alocador buddy; usa slab desde o kernel 2.2

