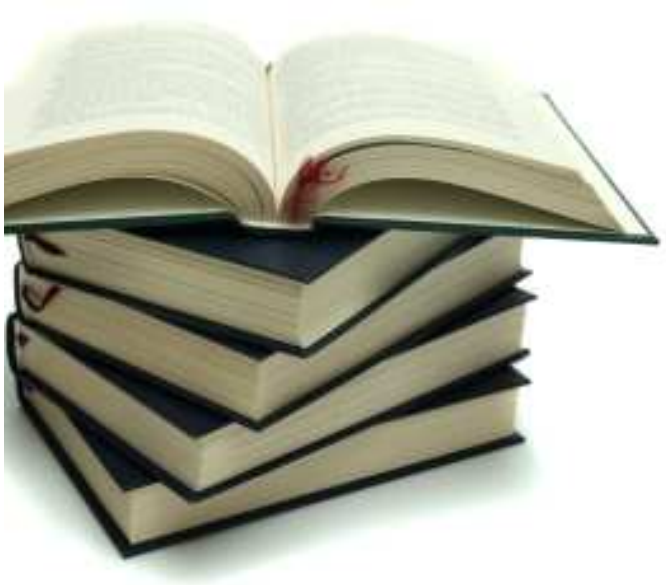
SO2
kernel crunch

Cursul 9



9

Gestiunea memoriei (II)

30 aprilie 2009

Test Curs 8

- Creați tabela de pagini (Intel) care să mapeze următorul spațiu de adresă virtual: [0x10000000 – 0x10000400], [0x4000000 – 0x40000400], [0xc0000000 – 0xc000400]
- Care sunt avantajele folosirii paginilor de 4M?
- De ce este o schimbare de context între două thread-uri mai puțin costisitoare decât o schimbare de context între două procese

Gestiunea memoriei (II)

- Gestiunea memorie fizice în Linux
- Gestiunea memoriei in Windows
- Gestiunea memoriei virtuale în Linux
- Gestiunea memoriei virtuale în Windows

Bibliografie

- UTLK: capitolele 8, 9
- WI: capitolul 7

Gestiunea memoriei fizice în Linux

- Algoritmi şi structuri de date ce mențin starea memoriei fizice
- Se face la nivel de pagină
- (Relativ) independentă de gestiunea memoriei virtuale
- Fiecare pagina fizică are asociată un descriptor: struct page
- În zona lowmem se ține un vector de astfel de descriptori

Descriptorii de pagini (fizice)

- Contor de utilizare al paginii
- Adresa virtuală a paginii (în kernel-space) – pentru paginile din highmem
- Zona de care aparține această pagină
- Coada de aşteptare asociată paginii
- Flaguri
- În anumite cazuri: poziția în swap sau în fişier, buferele conținute de pagină, poziția în „page cache”

Flaguri asociate paginilor fizice

- PG_locked
  - Pagina este folosita pentru un transfer I/O in curs
- PG_error
  - Eroare I/O la transfer
- PG_referenced
  - Accesată pentru o operație I/O
- PG_uptodate
  - Operația I/O s-a terminat
- PG_dirty
  - Pagina modificată
- PG_lru
  - Prezentă în lista lru (activa sau inactiva)
- PG_active
  - Prezentă in lista lru activă
- PG_slab
  - Folosită de slab
- PG_skip
  - Nefolosită
- PG_highmem
  - Pagină din zona high memory

Zone (fizice) de memorie

- Zona DMA: 0 - 16Mb
- Zona Normal (LowMem): 16Mb - 896Mb
- Zona HighMem: 896Mb – 4Gb/64Gb
- Non-Uniform Memory Access
  - Memoria fizică este împartită între mai multe noduri
    - există un spațiu comun de adrese fizice
    - accesul la memoria locală este mai rapidă
  - Fiecare nod
    - are procesorul propriu
    - are zone de memorie proprii: DMA, NORMAL, HIGHMEM

Alocarea de pagini

- alloc_pages(gfp_mask, order)
  - Alocă 2^order PAGINI contignue şi întoarce un descriptor de pagină pentru prima pagină alocată
  - alloc_page(gfp_mask)
- Functii mai folositoare (întorc adresa lineară a paginii/primei pagini)
  - __get_free_pages(gfp_mask, order)
  - __get_free_page(gfp_mask)
  - __get_zero_page(gfp_mask)
  - __get_dma_pages(gfp_mask, order)

Măştile GFP

- __GFP_WAIT
  - Kernelul poate bloca procesul curent pentru a aştepta pagini libere
- GFP_HIGH
  - Prioritate mare; kernelul are voie să folosească pagini libere rezervate pentru ieşirea din situații critice (atunci când există foarte puțină memorie)
- __GFP_IO
  - Kernelul are voie să facă transferuri I/O pe pagini normale pentru a elibera memorie (necesar pentru evitare deadlock-uri)
- __GFP_HIGHIO
  - Kernelul are voie să facă transferuri I/O pe pagini din high memory pentru a elibera memorie (necesar pentru evitare deadlock-uri)
- __GFP_FS
  - Kernelul are voie să execute operații VFS low level (necesar pentru evitare deadlock-uri)
- __GFP_DMA
  - Paginile trebuie alocate din zona DMA
- __GFP_HIGHMEM
  - Paginile trebuie alocate din zona HIGHMEM

Măştile GFP (2)

- GFP_ATOMIC
  - __GFP_HIGH
- GFP_KERNEL
  - __GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS
- GFP_USER
  - __GFP_HIGH | __GFP_WAIT | __GFP_IO | __GFP_HIGHIO | __GFP_FS

De ce alocare în puteri ale lui 2?

- Apar probleme de fragmentare externă
- Soluție: paging
  - Uneori kernelul chiar are nevoie de memorie fizică contignua (pentru DMA)
  - Dacă s-ar folosi paging, tabela de pagini s-ar modifica => penalizări la accesul la memorie
  - Dacă s-ar folosi paging nu s-ar mai putea folosi paginare extinsă (pagini de 4Mb/2Mb)
- Soluție: folosirea unui algoritm care să fragmenteze cât mai puțin memoria
  - Algoritmul buddy

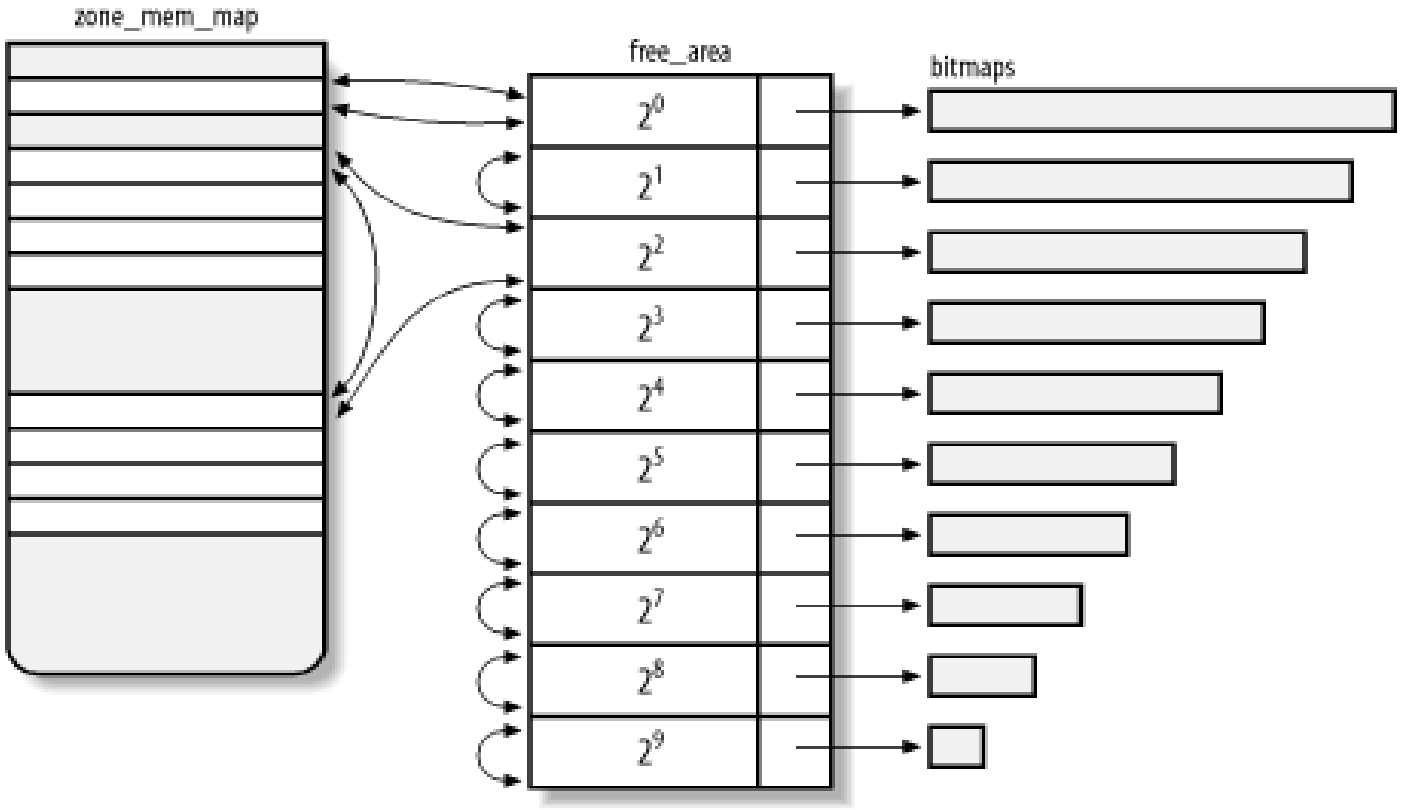
Algoritmul buddy

- Blocurile sunt grupate în liste de dimensiune fixă
  - Blocurile au ca dimensiune puterile lui 2, aliniate corespunzător cu dimensiunea
- Alocarea se face numai în blocuri de puteri ale lui 2
- La alocarea unui bloc (dimensiune N)
  - Dacă există un bloc de dimensiune N se alocă
  - Dacă nu, se sparge un bloc de dimensiune 2N în două blocuri de dimensiune N, unul se alocă, altul se pune în lista cu blocuri de dimensiune N
- La dealocarea unui bloc (dimensiune N)
  - Dacă există blocuri adiacente în memoria fizică de aceaşi dimensiune N, iar primul este aliniat la 2N, cele două blocuri se “coaguleaza” într-un bloc de dimensiune 2N
  - Se încearcă iterativ să se coaguleze cât mai multe blocuri
- Se observă că
  - Algoritmul reduce fragmentarea externă
  - Există cazuri în care se întoarce dublul memorie cerute => fragmentare internă

Implementarea algorimului buddy în Linux

- Sunt folosite 10 liste pentru blocuri de 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 PAGINI
- Fiecare zonă de memorie are alocatorul buddy propriu
- Fiecare listă de blocuri are asociată un vector de biți pentru a indica care blocuri sunt libere şi care ocupate
- Optimizare: pentru că dacă două blocuri adiacente sunt libere ele sunt automat “coagulate” este necesar un singur bit pentru fiecare pereche de “buddies”

Un exemplu:




```
typedef struct free_area_struct {
        struct list_head free_list;
        unsigned long *map;
} free_area_t;
```

Alocarea de blocuri cu dimensiune mică

- Sistemul buddy este folosit în kernel pentru a aloca pagini
- Multe componente din kernel au nevoie de blocuri de dimensiune mult mai mică decât dimensiunea unei pagini
  - Soluție: folosirea de blocuri mai mici cu dimensiune variabile
    - Probleme: fragmentare externă
  - Soluție: folosirea unor blocuri de dimensiune fixă, dar mai mici
    - Probleme: cât de mica/mare sa fie dimensiunea ?
    - Soluție: crearea mai multor zone cu blocuri de mai multe dimensiuni, distribuite geometric: 32, 64, ..., 131 056

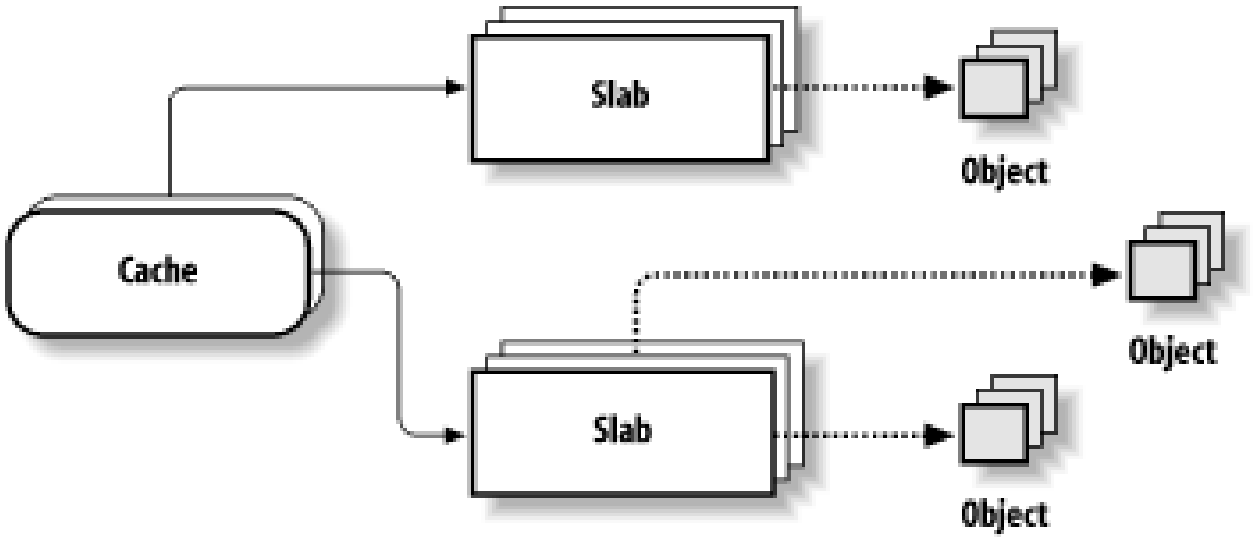
Alocatorul slab

- Zonele de memorie alocate/dealocate sunt văzute ca “obiecte”:
- Fiecare tip de “obiect” are un constructor şi un destructor
- Obiectele dealocate sunt păstrate într-un cache, astfel că la urmatoarea folosire a lor nu mai trebuie reinițializate şi nu mai trebuie apelat algoritmul buddy
- În Linux nu (prea) se folosesc constructori/descructori din motive de eficiență

De ce slab?

- Kernelul tinde să aloce/dealoce succesiv acelaşi tip de structuri de date (de exemplu PCB-uri); folosirea cache-ului din slab reduce frecvența operatiilor (mai costisitoare) de alocare/dealocare
- Multe cereri de alocare frecvente folosesc aceaşi dimensiune pentru fiecare alocare: pentru aceste tipuri se pot crea zone speciale, care să conțină blocuri de dimensiunea dorită => se reduce astfel fragmentarea interna
- Pentru cereri ce nu folosesc aceaşi dimensiune la alocare (mult mai puțin frecvente) se poate folosi în continuare abordarea geometrică (şi cache)
- Reduce foot-print-ul pentru alocare/dealocare pentru că nu se mai caută pagini libere ci sunt luate direct din cache-uri (cache-urile de obiecte sunt concentrate într-o zonă de memorie şi sunt mult mai mici decât structurile folosite de buddy)
- Distribuie obiectele în memorie astfel încât să acopere uniform liniile de cache

Arhitectura alocatorului slab



- Alocatorul slab este format din
  - Mai multe cache-uri
  - Fiecare cache are mai multe slab-uri
  - Fiecare slab menține mai multe obiecte alocate/libere

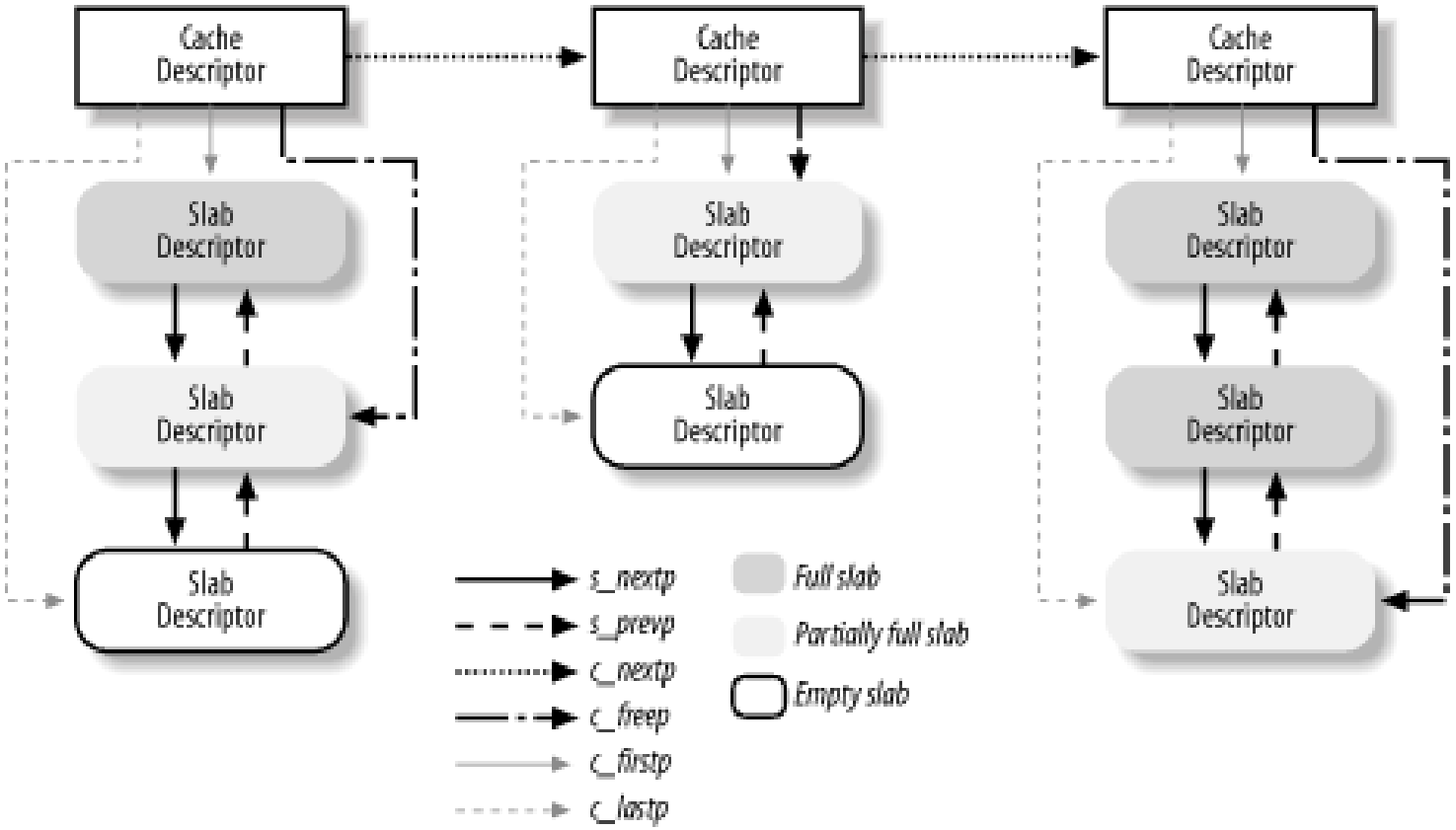
Descriptorii de cache

- Un nume folosit pentru informații către user-space
- Funcții pentru inițializarea / deinițializarea obiectelor cache-ului
- Dimensiunea obiectelor
- Diverse flaguri statice / dinamice
- Dimensiunea slab-urilor (structurile care conțin efectiv obiectele) în pagini (puteri ale lui 2)
- Măști GFP pentru alocarea dintr-o zonă specifică
- Mai multe slab-uri: pline, libere sau parțial pline

Descriptorii de slab

- Numărul de obiecte alocate
- Zona de memorie în care se țin obiectele
- Zona de memorie către primul obiect liber
- Descriptorii de slab sunt ținuti fie
  - În cadrul slab-ului ce îl descriu (dacă dimensiunea obiectelor e mai mică de 512, sau dacă fragmentarea internă lasa loc pentru descriptorul de slab)
  - În cadrul unor cache-uri generale folosite de către alocatorul slab

Arhitectura detaliată slab

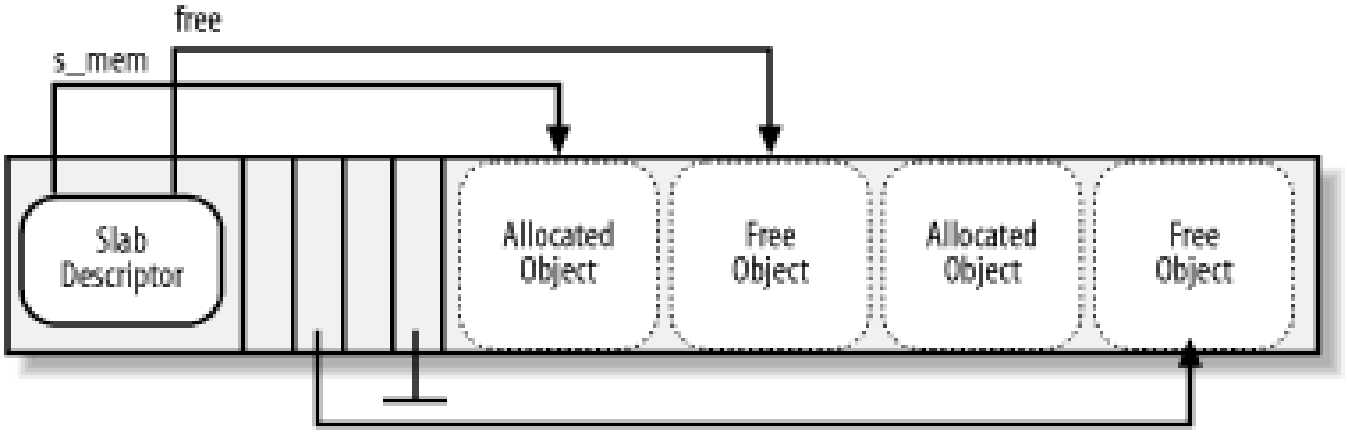


Cache-urile generale / specifice

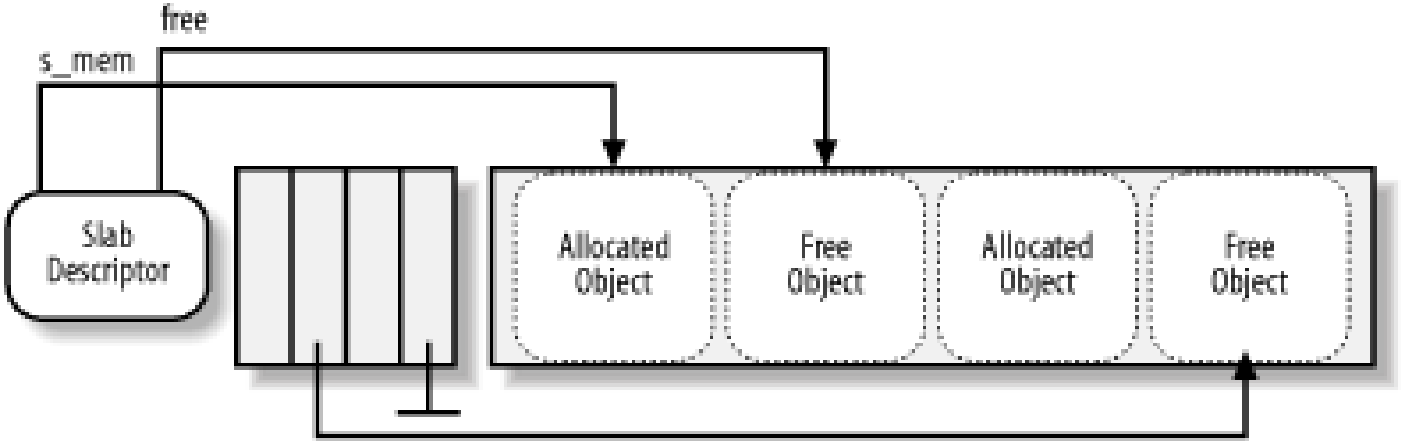
- Cache-urile generale sunt folosite de către alocatorul slab pentru
  - A ține descriptorii de cache pentru celelalte cache-uri
  - A ține 26 de cache-uri generale cu dimensiunea obiectelor de 32, 64, 128, ..., 131 072 (unul pentru zona normală şi unul pentru zona DMA); kmalloc() alocă memorie din aceste cache-uri
- Cache-urile specifice sunt create de către restul kernelului la cerere

Descriptorii de obiecte

Slab with Internal Descriptors



Slab with External Descriptors



Descriptorii de obiecte

- Descriptorul de obiect este folosit doar atunci când obiectul este liber
- Este de fapt un întreg care indică următorul obiect liber
- Ultimul obiect liber are valoarea BUFCTL_END
- Descriptori de obiecte:
  - Interni - ținuți în slab
  - Externi - ținuți în cache-urile generale

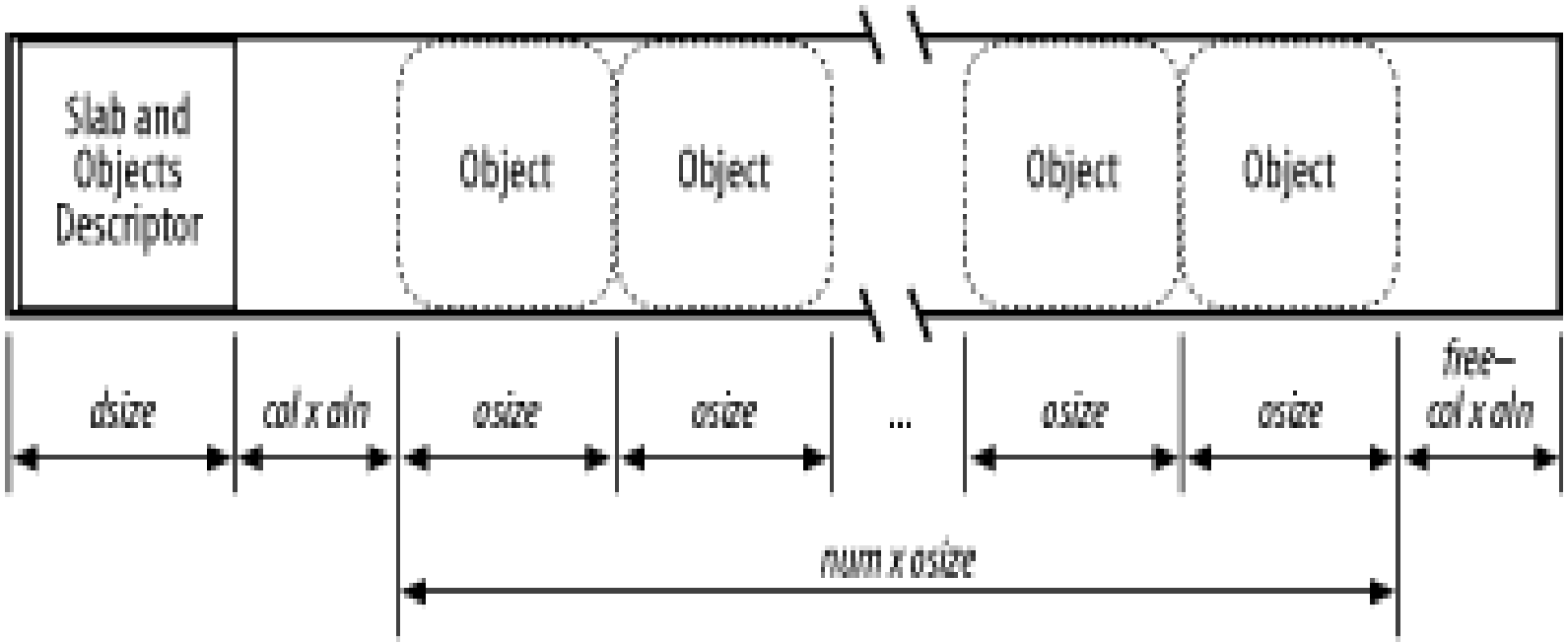
“Colorarea” slab-ului

- Colorarea este folosită pentru a alinia obiectele din diversele slab-uri din cache la offset-uri diferite (dar constrângerile de aliniere ale obiectelor din cache sunt păstrate)
- Fie
  - Aln - constrângerea de aliniere; adresa obiectului trebuie să fie un multiplu de acest număr
  - Num - numărul de obiecte care pot fi stocate în slab
  - Objsize - dimensiunea obiectelor, inclusiv octeții nefolosiți dar necesari pentru aliniere
  - Dsize - dimensiunea descriptorului slab-ului + dimensiunea descriptorilor de obiecte
  - Free - numărul de octeți neutilizați în slab

“Colorarea” slab-ului (2)

- Atunci
  - slab_length= (num x objsize) + dsize + free
  - Numărul de culori = (free/aln) + 1
  - Numărul de culori reprezintă de fapt numărul maxim de posibilități de plasa primul obiect în slab prin mutarea unei bucăți din zona free de la sfârşitul slab-ului la începtul acestuia
  - Cum toate definițiile date sunt comune unui cache, înseamnă că alocatorul slab poate aloca “culori” diferite fiecărui slab

“Colorarea” slab-ului (3)



Gestiunea memoriei fizice în WNT

- Paged Pool
  - alocări cu dimensiuni variabile
  - are rezervată un interval de adrese în spațiul kernel
  - paginile din această zonă pot fi evacuate
- NonPaged Pool
  - alocări cu dimensiuni variabile
  - are rezervată un interval de adrese în spațiul kernel
  - paginile din această zonă sunt permanent rezidente
  - o zonă de uz general
  - o zonă pentru cazuri de urgență (4 pagini)

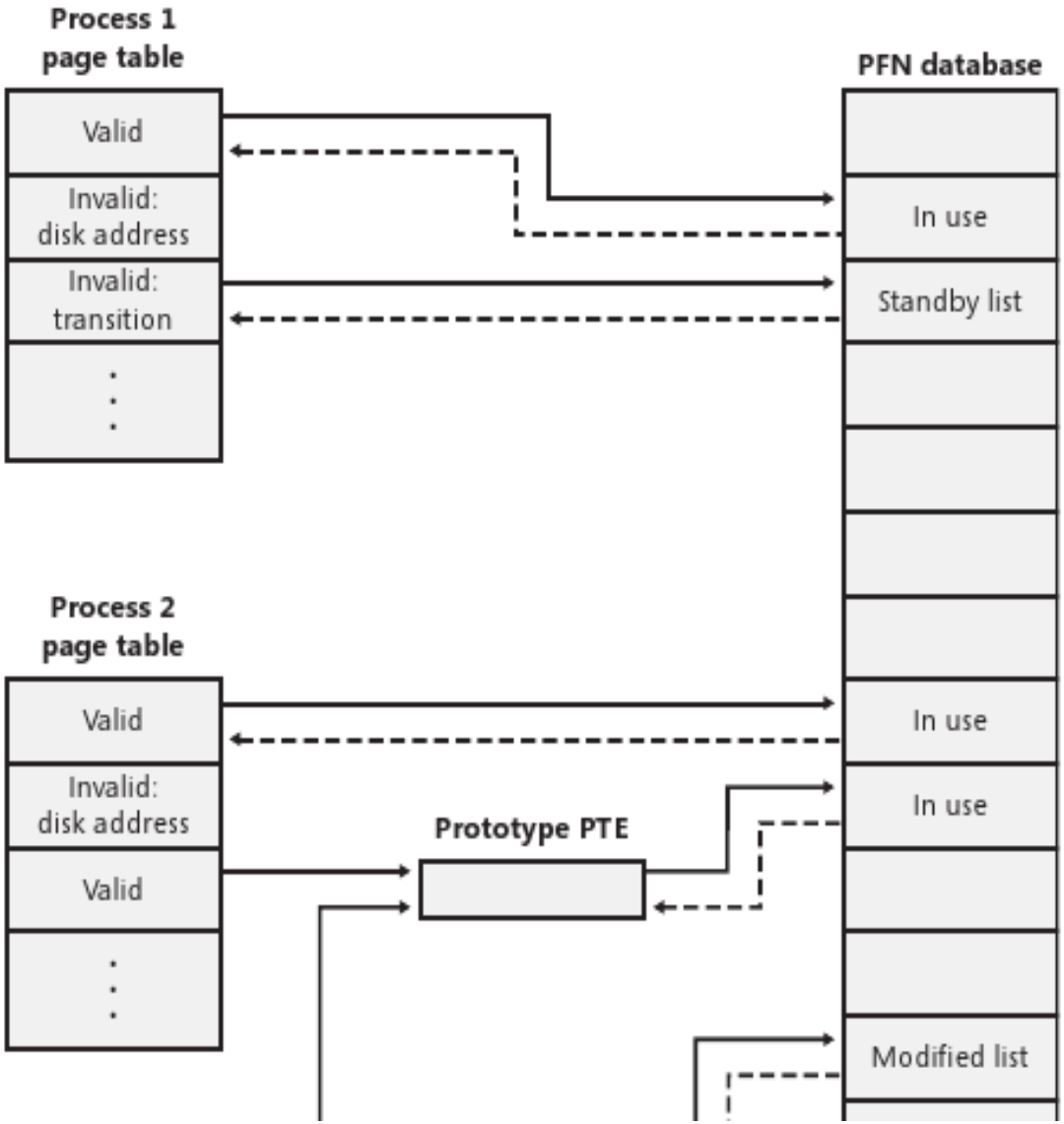
Look-Aside Lists

- Modalitate de alocare rapidă:
  - ExInitialize{NPaged|Paged}LookasideList
- Se alocă blocuri de dimensiune fixă
- Similar cu slab
  - există mai multe asemenea liste, fiecare funcționând ca un cache
  - pentru dimensiuni < 256 poolurile generale folosesc liste special construite

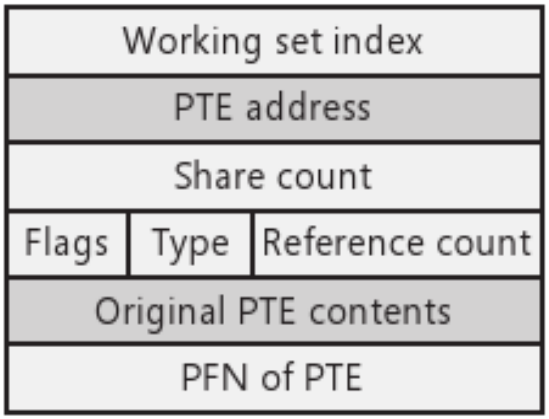
Alocare din (Non)PagedPool

- Dacă dimensiunea de alocate este suficient de mică se alocă dintr-un LookAsideList predefinit
- În caz contrar se alocă un număr de pagini putere a lui 2 – algoritm buddy

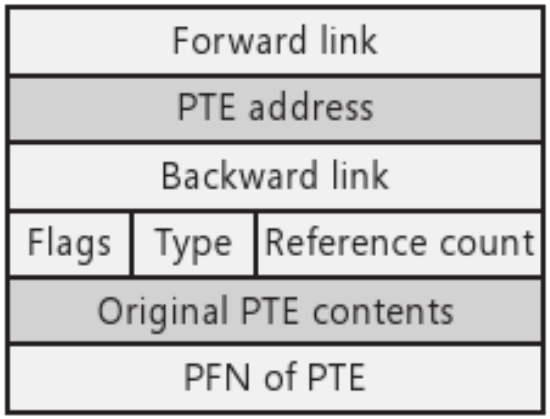
Page Frame Number Database



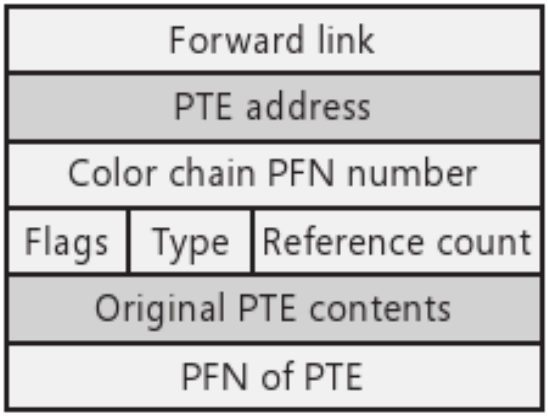
Page frame descriptor



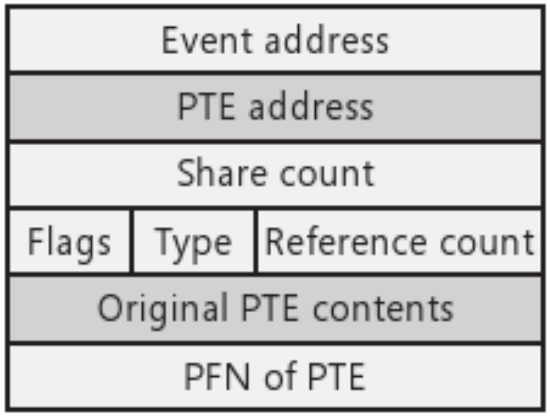
PFN for a page in a working set



PFN for a page on the standby or the modified list



PFN for a page on the zero or free list

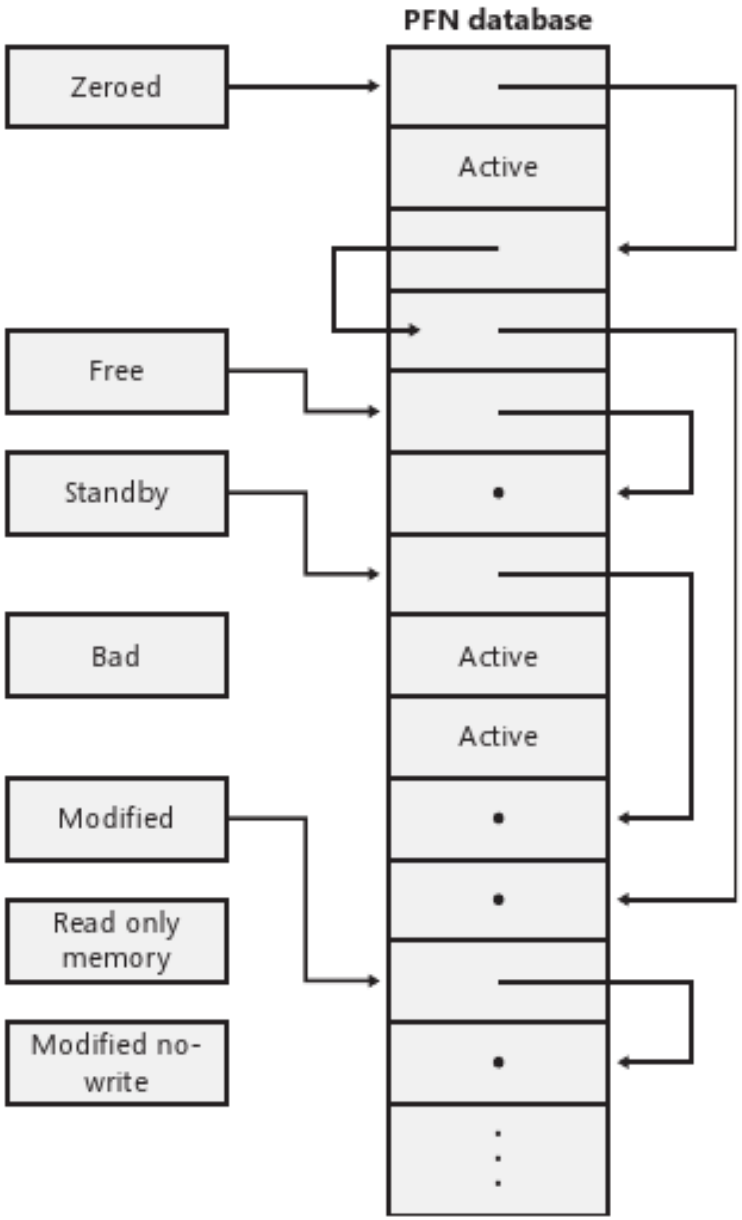


PFN for a page with an I/O in progress

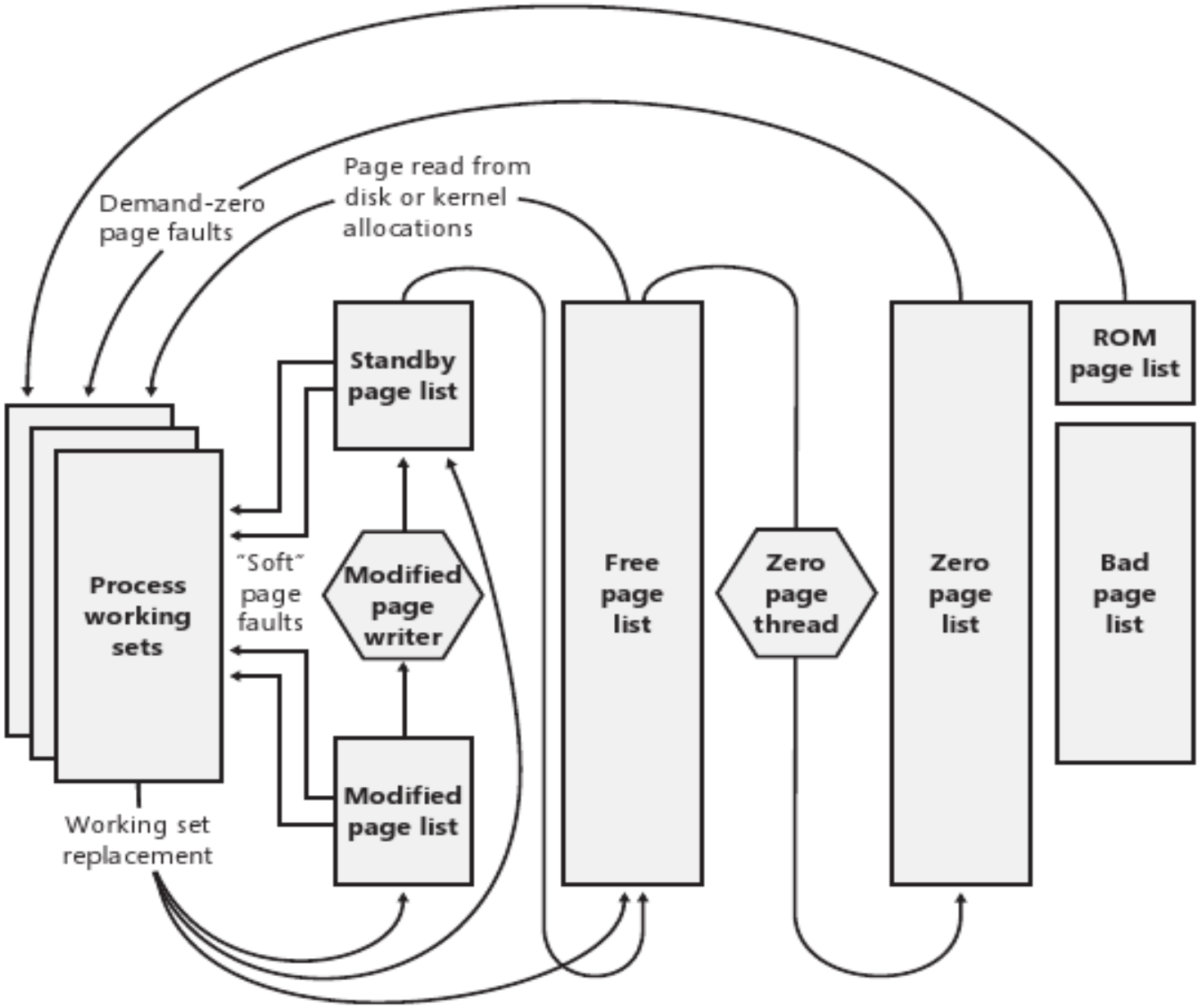
Starea paginii fizice

- Active - folosită
- Transition: se efectuează o operație de I/O asupra paginii
- Free: pagini libere, dar care nu au fost curățate
- Zeroed: pagini libere şi curățate
- Standby: pagini care au fost demapate dintr-un spațiu de adresă (user sau kernel); PTE-ul din spațiul de adresă pointează încă către pagina fizică, dar este marcat ca invalid; pagina nu a fost modificată
- Modified: analog cu standby doar că paginile au fost modificate (şi înainte de a fi eliberate trebuiesc scrise pe disc)
- Modified no write: analog cu modified doar că paginile nu pot fi încă scrise pe disc

Liste cu paginile fizice



Dinamica paginilor în WNT



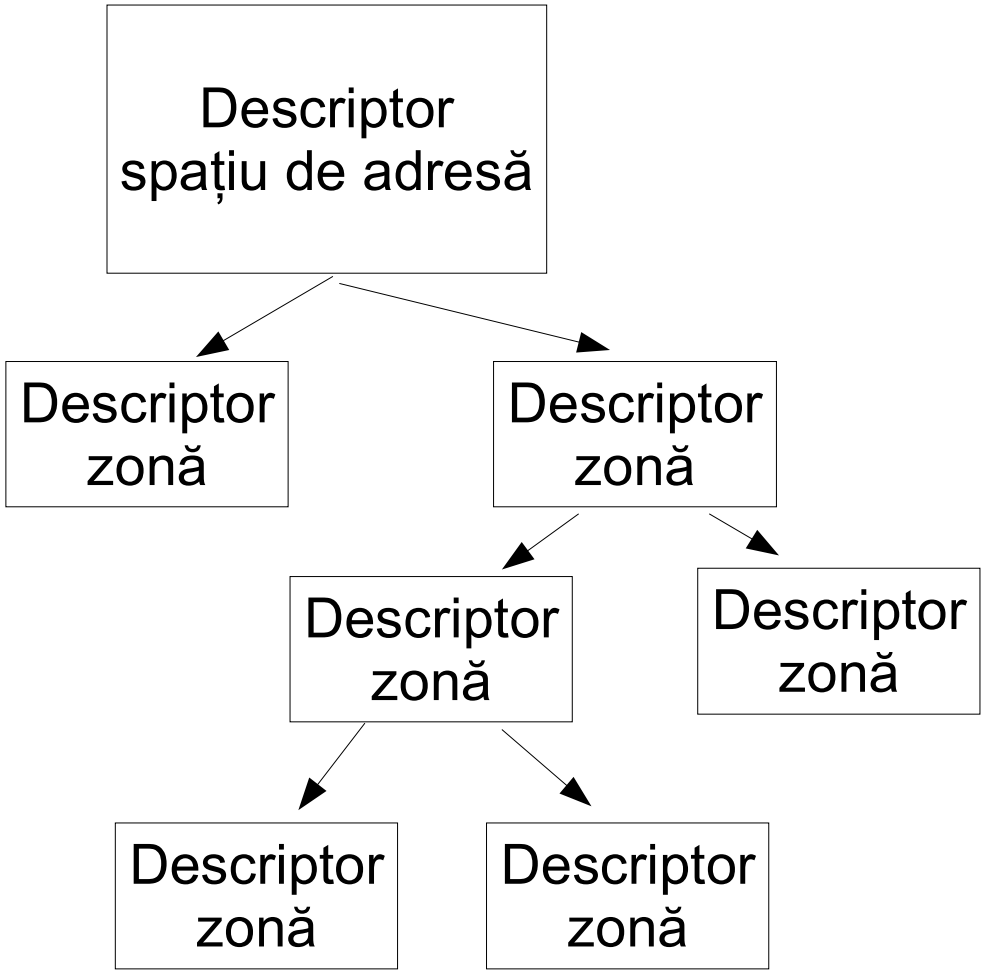
Dinamica paginilor în WNT (2)

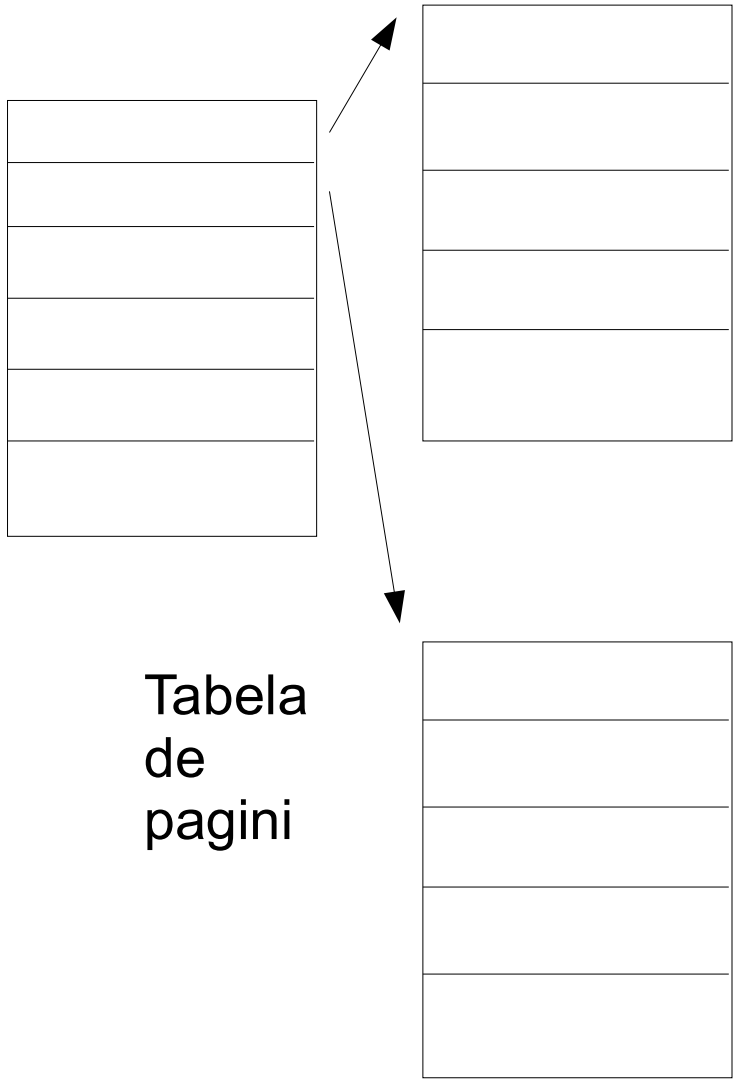
- Zero page thread
  - curăța paginile
  - prioritate 0, rulează doar dacă sunt mai mult de 8 pagini libere
- Modified page writer
  - se apelează când:
    - numărul de paginile modificate depăşesc o limită
    - numărul de pagini libere scade sub o limită
  - 2 thread-uri: MiModifiedPageWriter, MiMappedPageWriter
  - încearcă să scrie cât mai multe pagini cu o singură operație

Gestiunea memoriei virtuale

- Memoria virtuală este folosită
  - în user-space
  - în kernel-space
- Alocarea unei zone de memorie în spațiul de adresă (kernel sau user) implică
  - alocarea unei pagini fizice
  - alocarea unei zone din spațiul de adresă
    - în tabelele de pagini
    - în structuri interne, menținute de sistemul de operare

Descrierea spațiului de adresă





Descrierea spațiului de adresă (2)

- Tabela de pagini este folosită de procesor sau la procesoarele RISC de către un layer low level (de obicei scris în assembler) care caută o pagină şi o adaugă în TLB
- Descriptorul spațiului de adresă este folosit de sistemul de operare pentru a menține informații mai high level
- Fiecare descriptor de zonă specifică dacă zona este mapată peste un fişier, este mapată read-only, este mapată copy-on-write, etc.

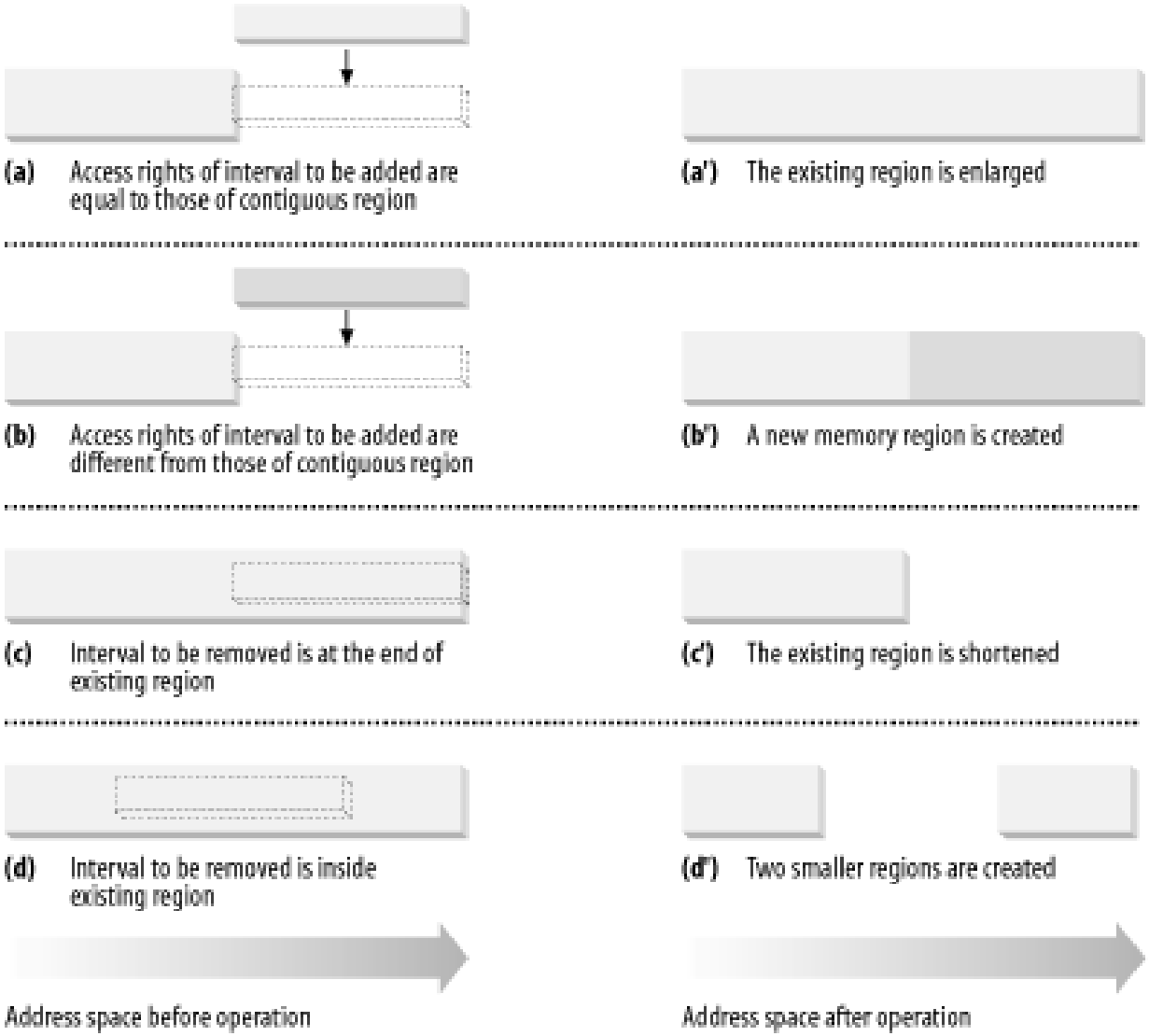
Alocarea de memorie virtuală

1. Căutarea unei zone libere în descriptorul spațiului de adresă
2. Alocarea de pagini fizice pentru descriptorul zonei
3. Inserarea în descriptorul spațiului de adresă a descriptorului zonei
4. Alocarea de pagini fizice pentru tabelele de pagini
5. Inițializarea tabelei de pagini astfel încât să indice către zona alocată
6. Alocarea la cerere de pagini fizice pentru zona alocată (şi eventual pentru noi tabele de pagină)

Dealocarea de memorie virtuală

1. Ştergerea descriptorului zonei din descriptorul spațiului de adresă
2. Eliberarea paginii fizice asociată cu descriptorul zonei
3. Invalidarea tuturor întrarilor din tabelele de pagini asociate cu zona de dealocat
4. Invalidarea TLB-ului pentru zona delocată
5. Eliberarea paginilor asociate cu tabelele de pagini construite pentru zona dealocată

Operații asupra spațiului de adresă



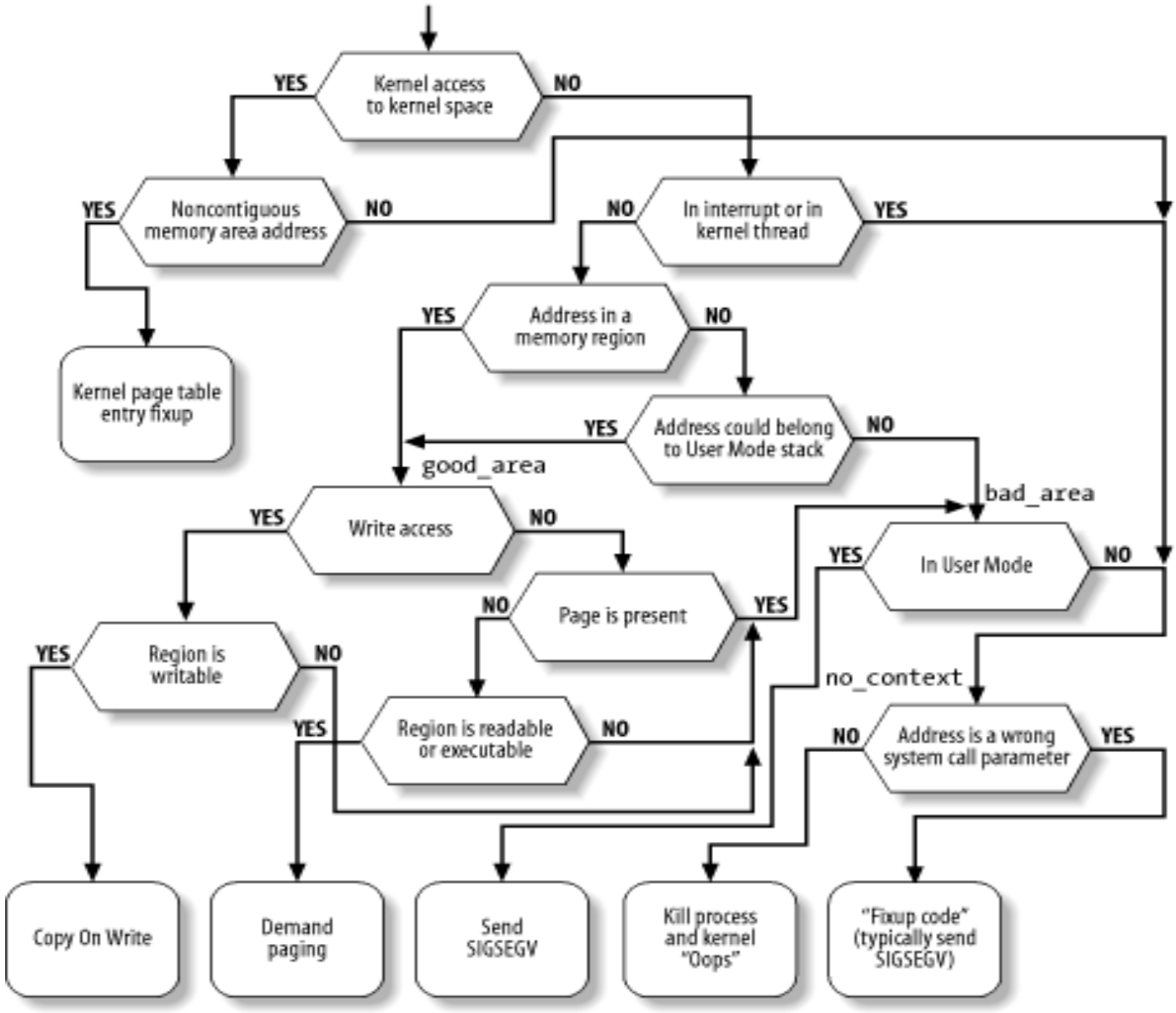
Gestiunea memoriei virtuale în WNT

- Descriptorul de zonă: virtual address descriptor (VAD)
  - zona din spațiul de adresă
  - protecție: read-only, read-write, copy-on-write
  - Moştenire (partajare spațiului de adresă între procese)
- Descriptorul spațiului de adresă: arbore binar balansat (AVL)

Gestiunea memorie virtuale în Linux

- În kernel-space
  - pentru apeluri vmalloc
  - descriptorul de zonă: struct vm_struct
  - descriptorul spațiului de adresă: listă simplu înlănțuită de descriptori de zonă
- În user-space
  - descriptorul de zonă: struct vm_area_struct
  - descriptorul spațiului de adresă: arbore red-black

Tratarea page fault-urilor în Linux



Tratarea page-fault-urilor în WNT

- Similar Linux doar că
  - în loc să se trimită SIGSEGV se generează o excepție (access violation)
  - în loc de „kernel oops” avem „blue screen of death”
  - nu există fixup-uri pentru că
    - în general device driverele nu ating direct memoria utilizatorului
    - se pot folosi primitivele __try, __exception pentru protejarea unei zone de cod de execepții
  - în plus, accesarea unei pagini în tranziție (lista stanby, modified sau modified-no-write) va aduce pagina în working-set

Evacuarea paginilor în Linux

- Algoritm de tip LRU
  - se mențin doua liste: cu paginile active şi cu paginile inactive
  - atunci când sistemul are nevoie de pagini, se eliberează pagini, în ordine, din:
    - diversele cache-uri: buffer cache, dcache, icache
    - lista de pagini inactive
    - lista de pagini active
- kswapd
  - kernel thread care evacuează paginile
  - este activat atunci când numărul de pagini libere scade sub o limită

Evacuarea paginilor în WNT

- Se foloseşte un algoritm de tip Working-Set
- Ajustarea working-setului are loc în thread-ul sistem KeBalanceSetManager
- Pentru fiecare proces există un working set
- Există un working set şi pentru kernel-space

Intrebări

?