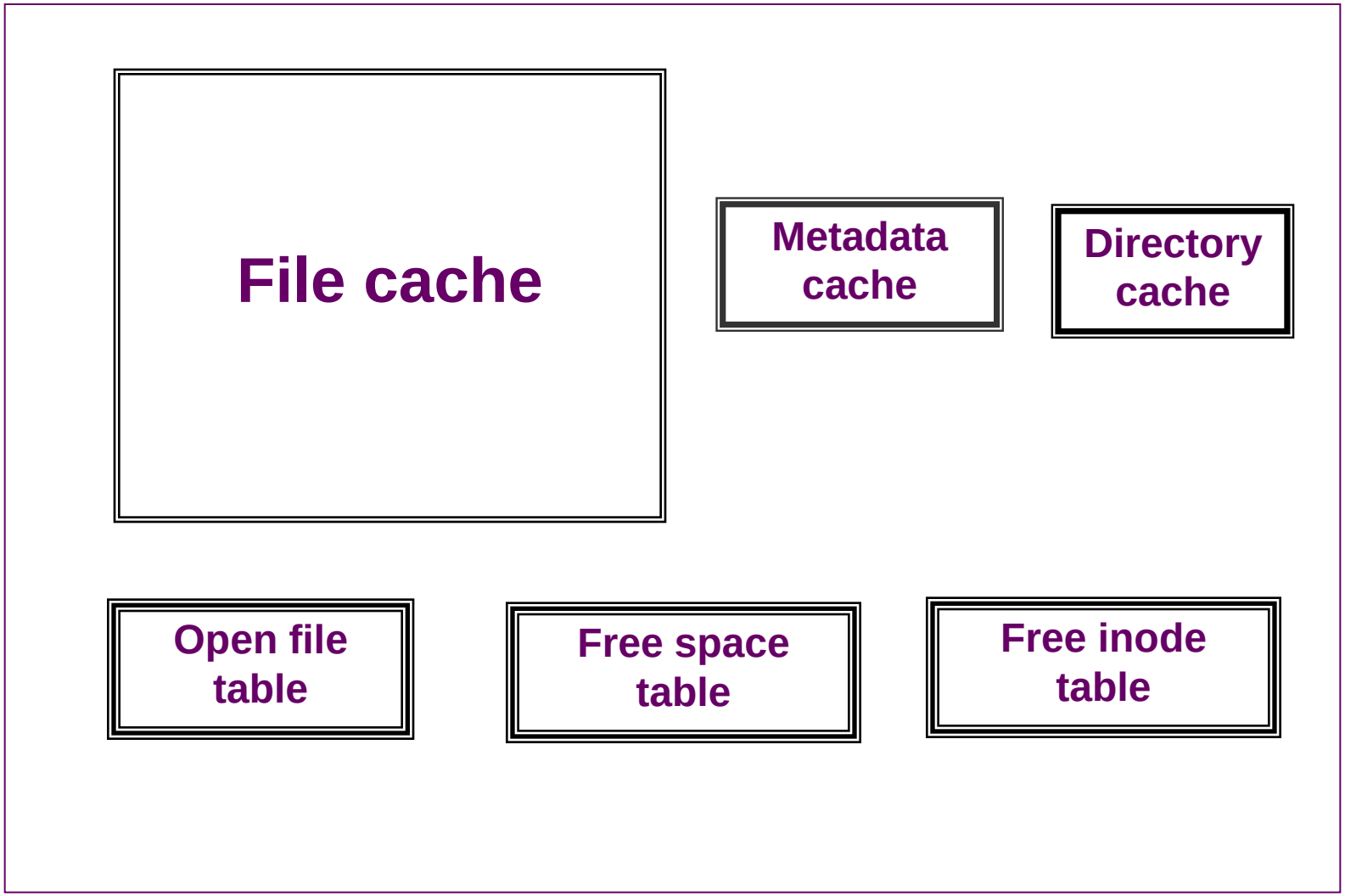
File-cache splitting

- Data Cache
- Metadata cache
- Directory cache

tag memory

data cache memory

Podela keša


File cache
Metadata cache
Directory cache
Open file table
Free space table
Free inode table

File-caching

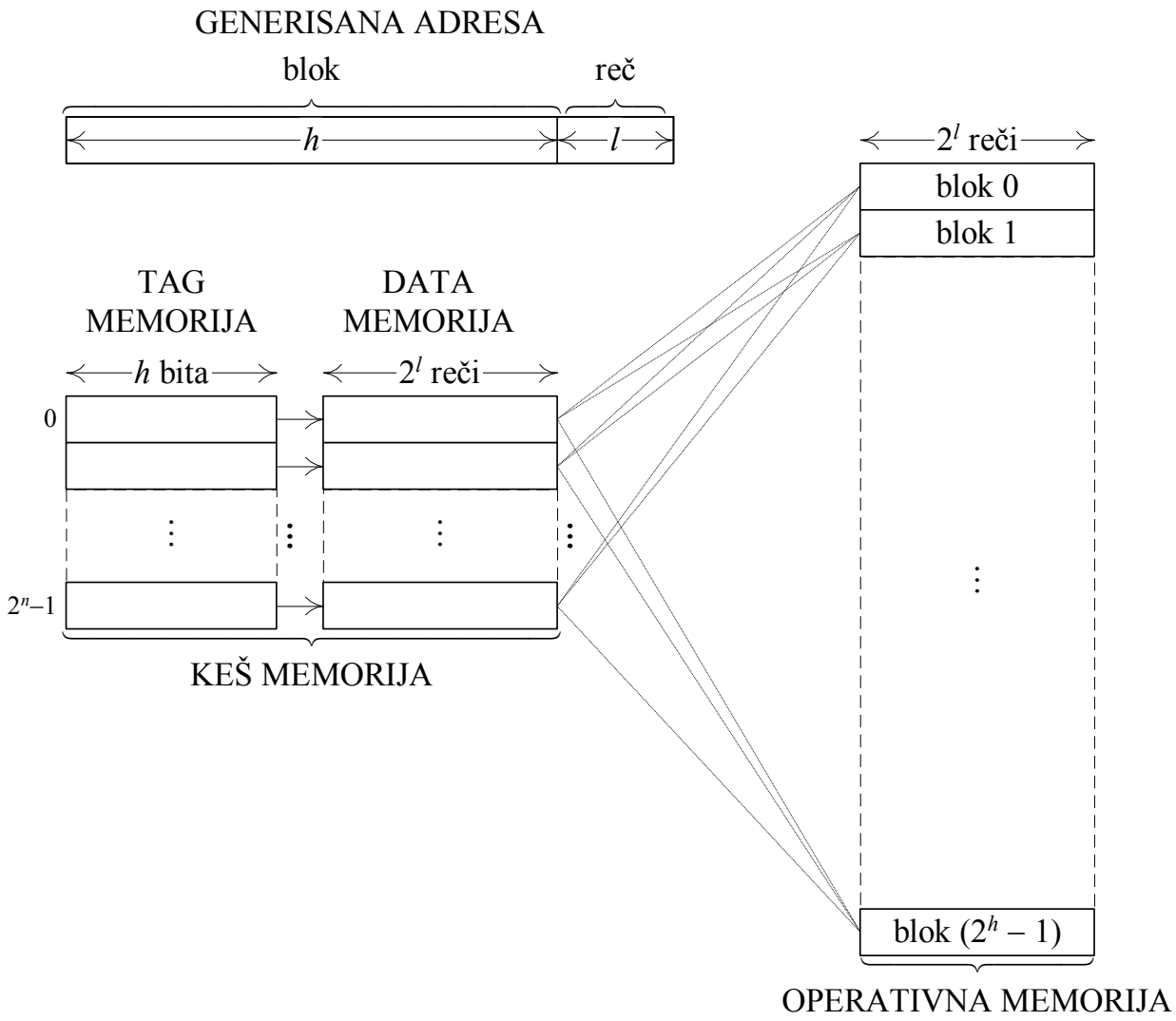
- **memory-style caching**
  - **Fully-associative**
  - **Direct mapped**
  - **Set-associative**
- **Hashing**

Fully-associative Cache

- U **pitanju je ekstremna keš organizacija gde**
  - **svaka linija u keš memoriji**
  - može da prihvati **bilo koju liniju sa diska**,
  - **zato što je adresni deo linije (tag) nezavistan** i
  - sadrži adresu popunjene linije sa diska.

- Algoritam za ovakvu organizaciju je složeniji,
  - vreme pristupa veliko, osim ako se ne koristi asocijativna tag memorija
  - ali
  - **ispoljava visok odnos pogodaka/promašaja**

Fully-associative Cache



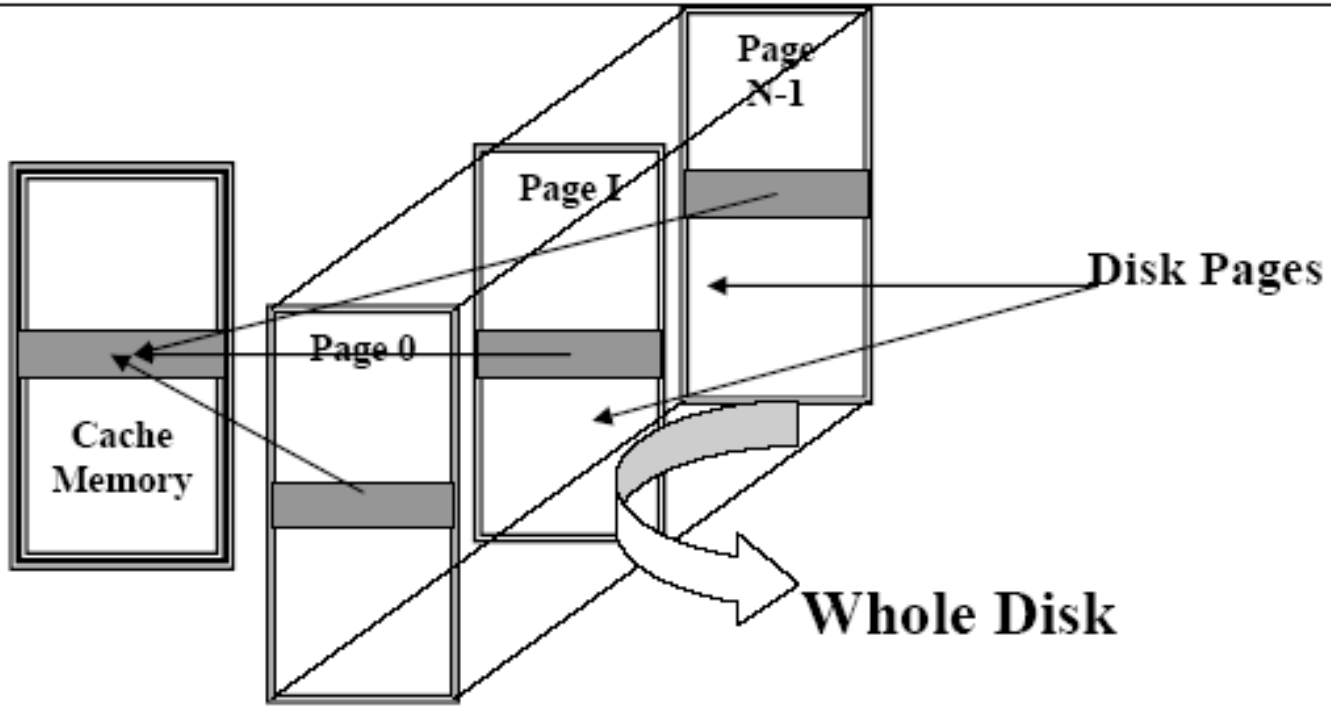
Direct-mapped Cache

- U pitanju je još jedna ekstremna keš organizacija
  - gde svaka linija u keš memoriji
  - može da prihvati
  - **samo određenu strogo definisanu liniju sa diska**.

- Algoritam za ovakvu organizaciju je **jednostavniji**,
  - samim tim ubrzava vreme pristupa,
  - **ali ispoljava niži odnos pogodaka/promašaja**,
  - **zato što pati od česte razmene konkurentnih podataka u kešu**
  - (konflikti).

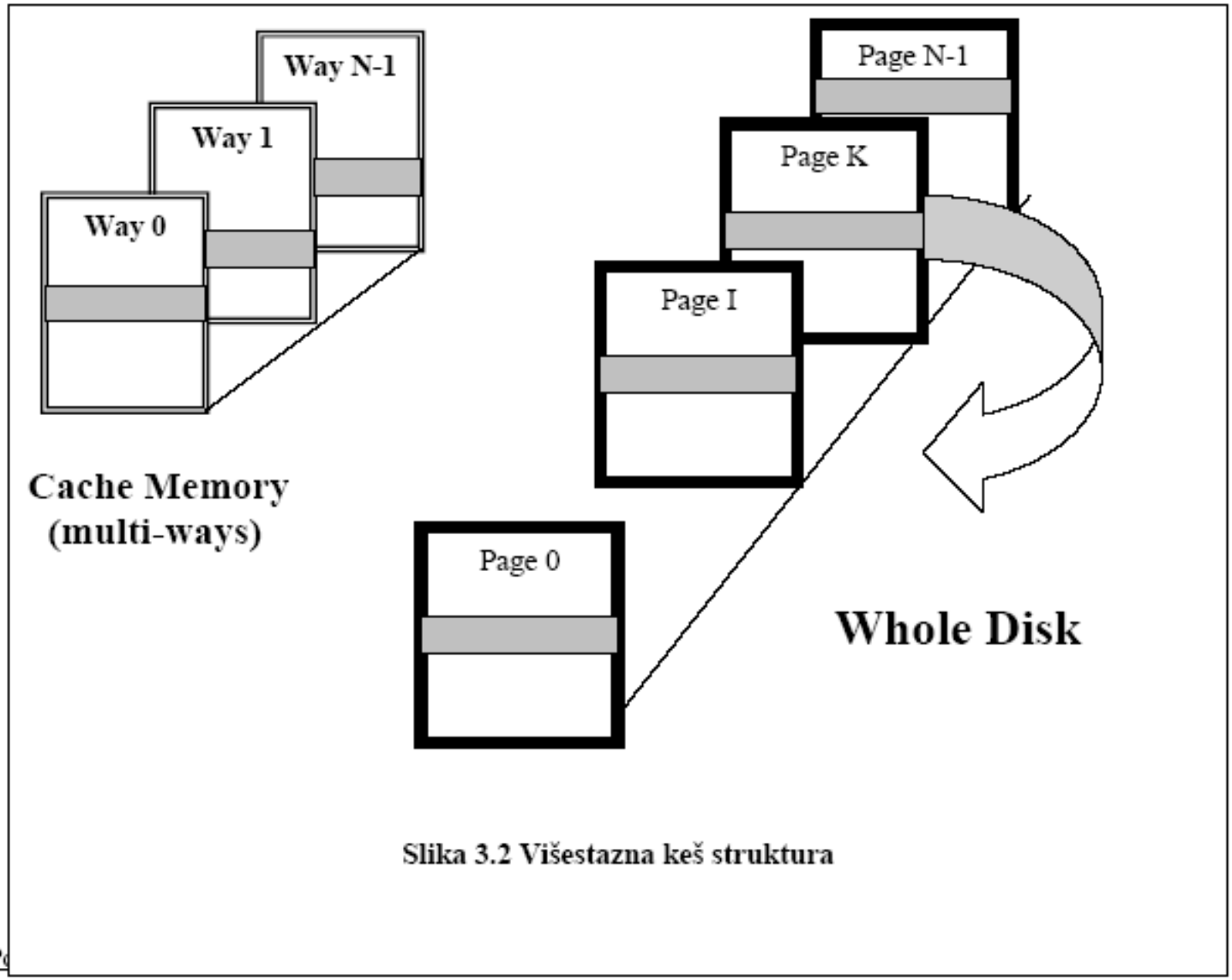
Direct-mapped Cache

predstavlja lokaciju u keš memoriji i njene korespodentne lokacije na disku na različitim stranicama.



Slika 3.1 Direktno mapirana keš struktura

Multi-way concept- set associative



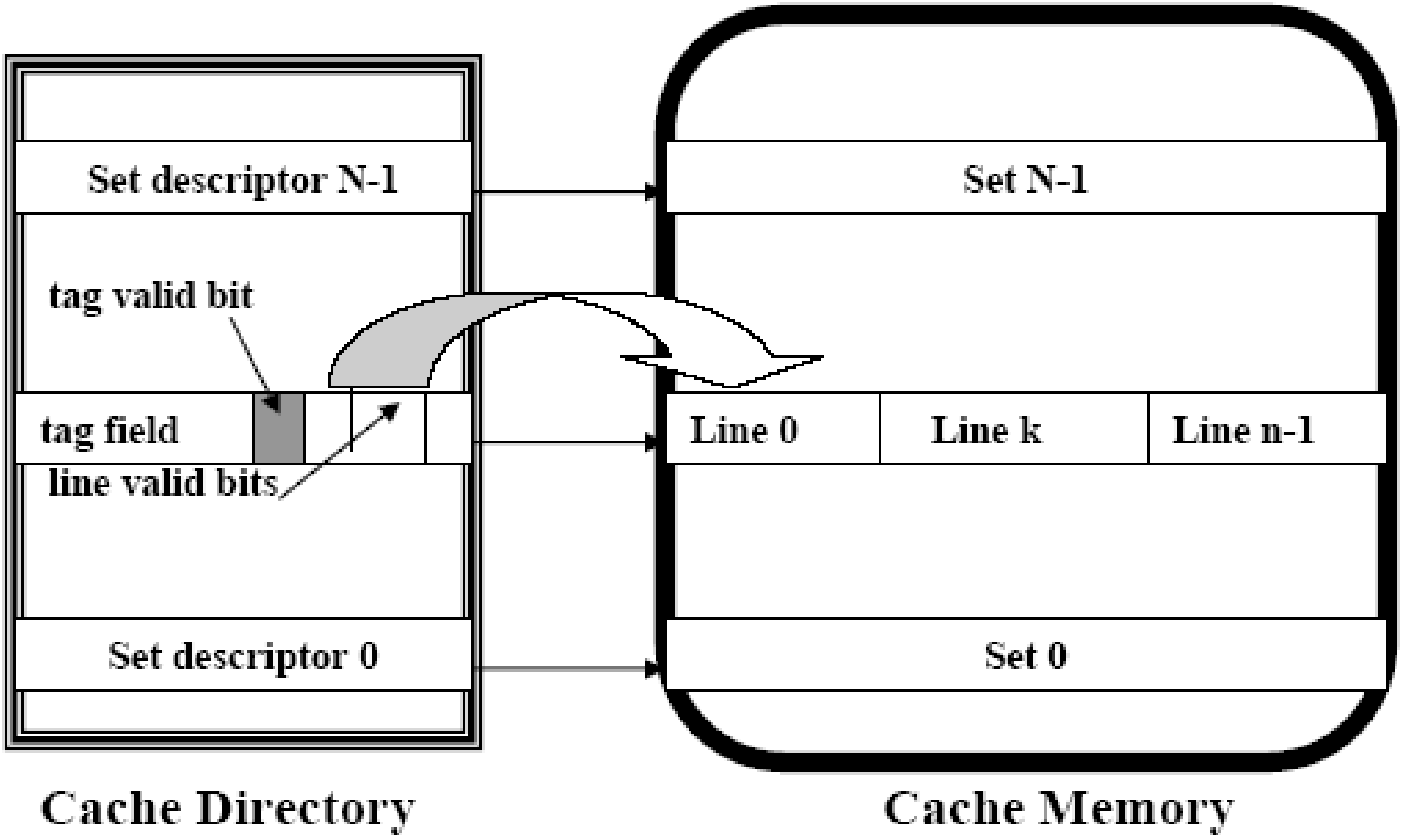
Slika 3.2 Višestazna keš struktura

Set-associative Cache

- **Set-associative Cache**
- Ova stuktura **ima set organizaciju**,
  - **setovi su identične keš memorije**
  - pri čemu se **set sastoji od više linija**,
  - za koje je karakteristično da imaju **isti viši deo adrese (tag),**
  - svaka linija ima svoj bit validnosti,
  - a takođe bit validnosti postoji za ceo set.

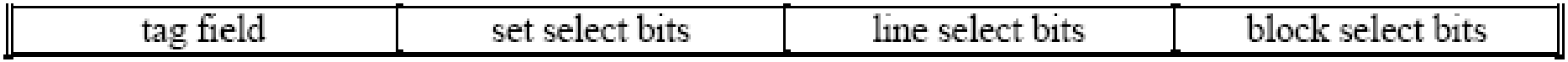
- To je **kompromisna keš organizacija sa stanovišta**
  - **kompleknosti** i
  - **odnosa pogodaka/promašaja**

Set-associative Cache



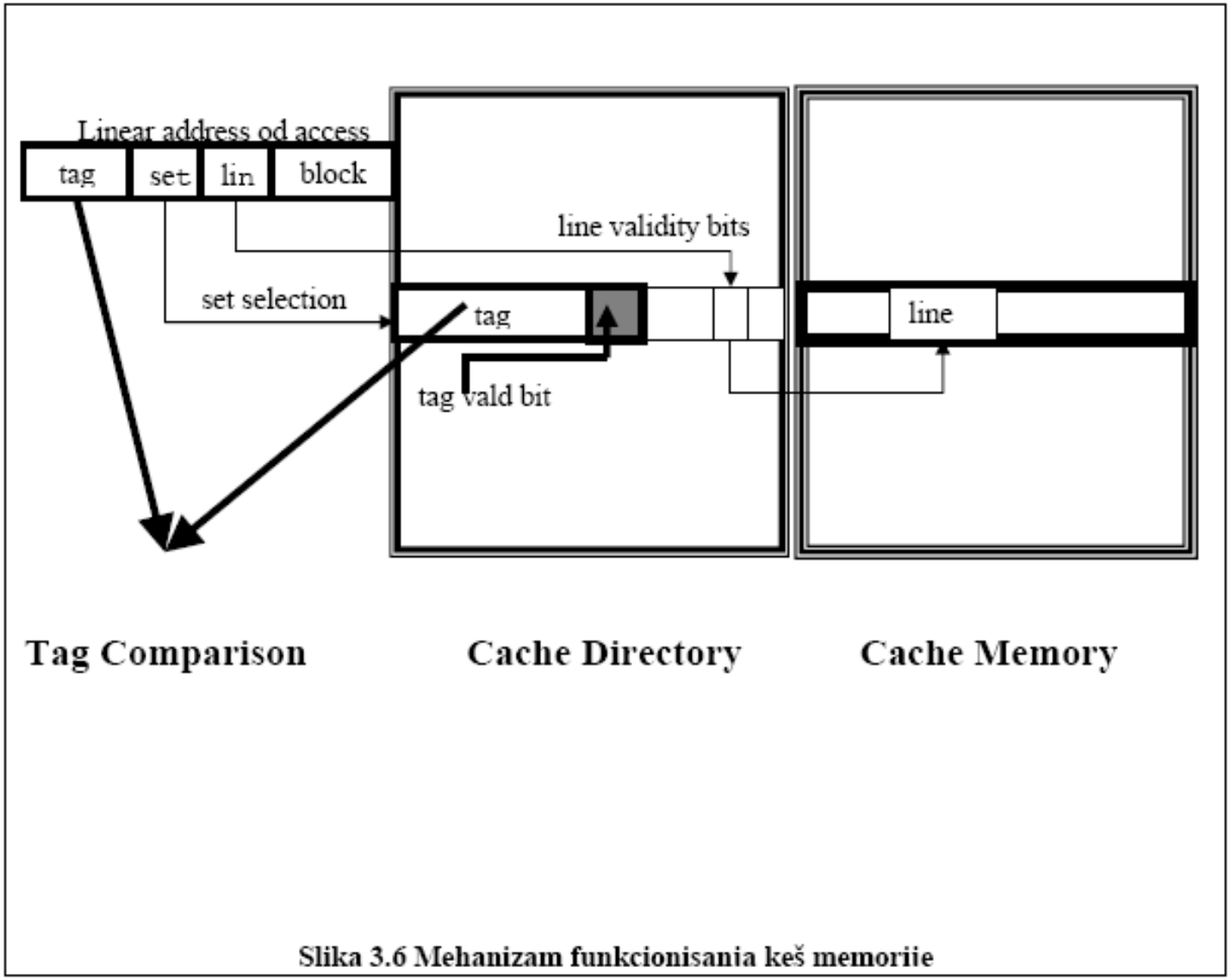
Slika 3.3 Principska organizacija keš memorije

Logical address in the cache context



Slika 3.4 Interpretacija linearne blok adrese sa stanovišta keš algoritma

Tag comparison

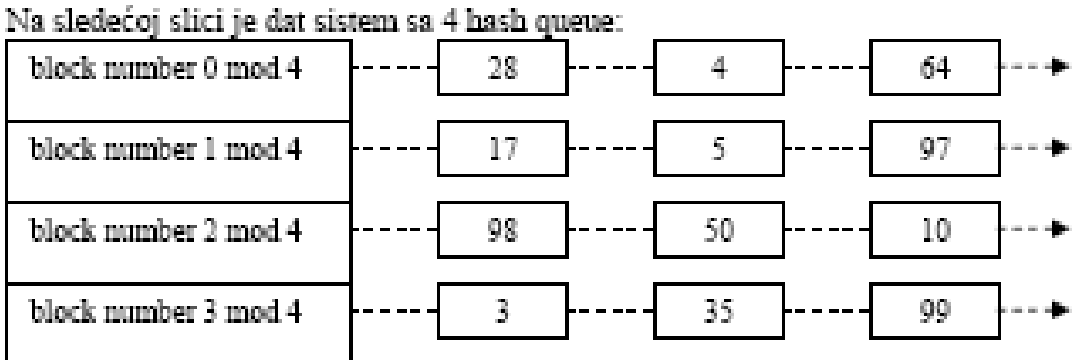


Slika 3.6 Mehanizam funkcionisania keš memoriie

HASHING

- **Hash function**

- **1. Block searching**
  - ☞ **Cache hit**
  - ☞ **Cache miss**
- **2. Block allocating (on the miss)**



Buffer cache

- **Kernel pokušava** da **smanji broj disk pristupa** preko **baferskog keša,**
  - a to je **skup internih bafera podataka**
  - koji sadrži podatke **nedavno korišćenih disk blokova**.
- **(read)** Kada se proces obraća datoteci za čitanje,
  - kernel pokušava da je locira u kešu,
  - ako je u kešu ne čita je sa diska (**read hit**),
  - a ako nije prvo se dovede u keš (**read miss**),
  - čuvajući u kešu one podatke za koje algoritam smatra da su najkorisniji.
- **(write)** Po pitanju upisa, keš je vrlo beneficijalan,
  - može da **kompenzuje višestruka upisivanje** ili
  - da **potpuno elimiše upis** na **disk**, (ako usledi proces brisanja datoteke)
  - takođe više odloženih upisa se mogu kombinovati u optimalnom redosledu.
- Keš algoritam izdaje instrukcije za
  - pre-cache (read-ahead) i
  - delayed-write da bi se maksimizovao keš efekat.

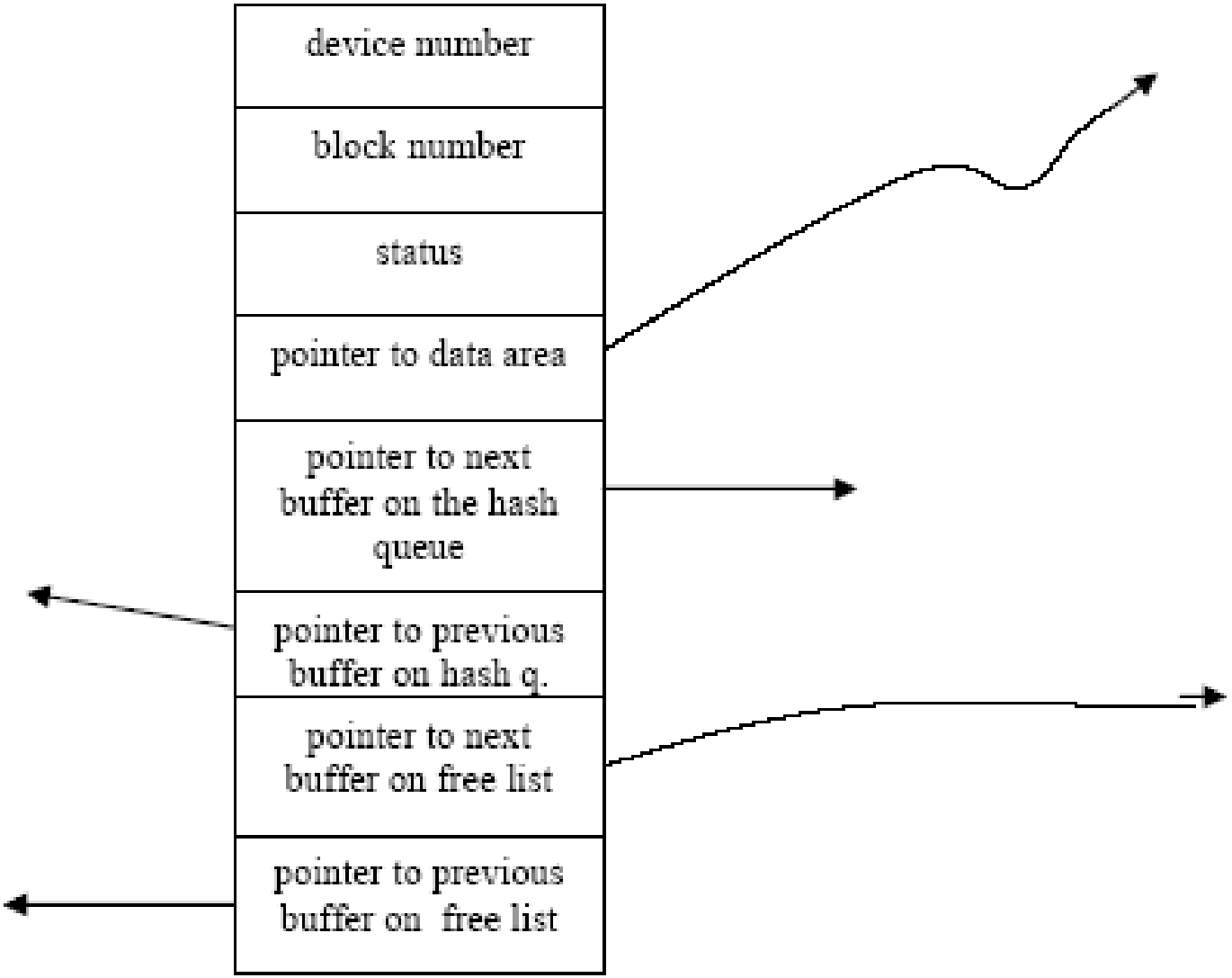
Baferi (Buffers)

- Za vreme **sistemske inicijalizacije**,
  - kernel alocira prostor za **određeni broj bafera (total cache size)**,
  - a taj prostor zavisi od **memorijske veličine** i **sistemskih performansi**.
- Bafer se sastoji od **2 dela-komponente**:
  - **bafer:** memorijsko polje koje sadrži podatke, na nivou logičkog bloka FS
  - **bafersko zaglavlje (header)** koje identifikuje bafer
- **Kernel ispituju zaglavlja** kako bi **analizirao sadržaj keš bafera**.

Baferska zaglavlja (Buffer headers)

- **Bafer zaglavlje** sadrži **sledeća polja**:
- **device number**
- **block number**
- **data pointer:** ukazivač na blok iz data-area (mora da pokrije sve blokove iz data area)
- **status**
  - **locked**: bafer je trenutno locked što znači da je zauzet (puni se ..)
  - **valid**: bafer sadži validne podatke
  - **delayed write**: kernel mora upisati sadržaj bafera na disk pre nego što obavi ponovnu dodelu bafera
  - **in reading/writing:** kernel trenutno čita ili piše po baferu
  - **wait for be free:** proces čeka da bafer postane slobodan
- **set pointera** za alokaciju bafera koje koristi keš algoritam

Buffer header



Struktura bafera (Buffer pool)

- **Baferi se opslužuju po LRU algoritmu**:
  - kada se bafer dodeli disk bloku,
  - taj blok će ostati tu
  - sve dok ne postane **najstariji** u baferu po pitanju obraćanja.

- **Kernel održava listu slobodnih** bafera u **LRU redosledu**.

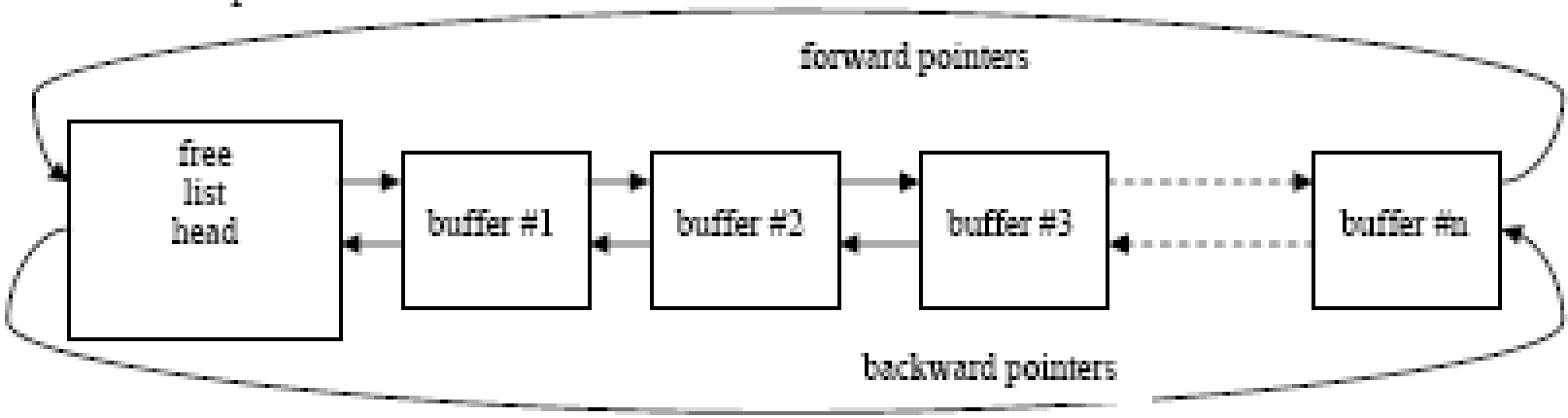
- **Slobodna lista** je **duplo linkovana cirkularna lista** sa **header-om na početku.**

- U **početku**, **svi su baferi slobodni**:

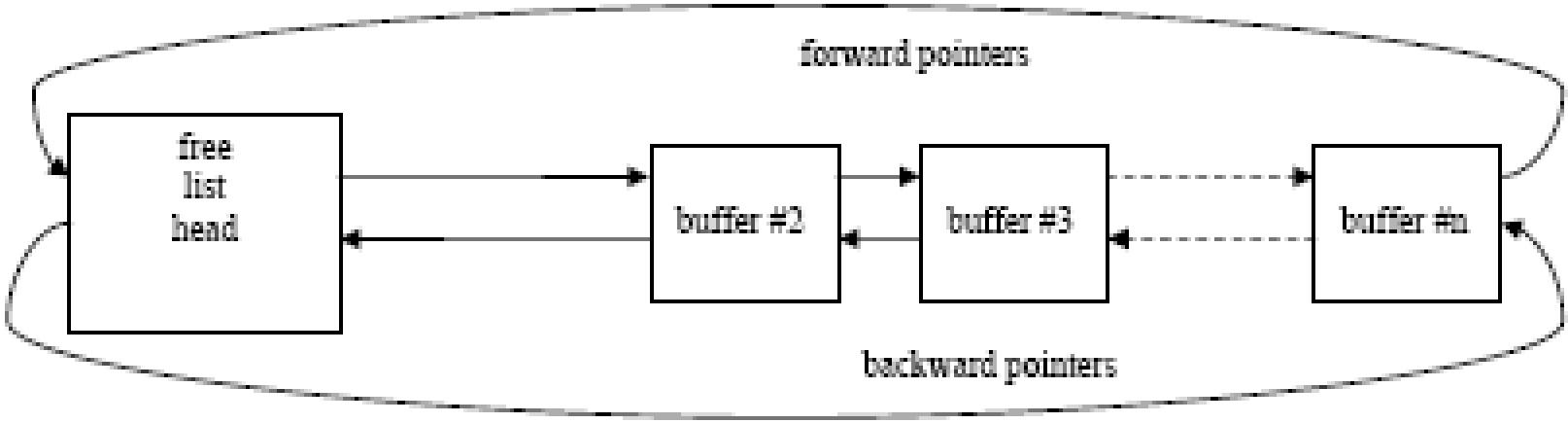
- kernel uzima bafer koji je na početku free liste
  - tada ga izbacuje iz slobodne liste,
  - a po povratku ga stavlja na kraj free liste.

Struktura bafera (Buffer pool)

Slobodna lista pre dodele bafera #1



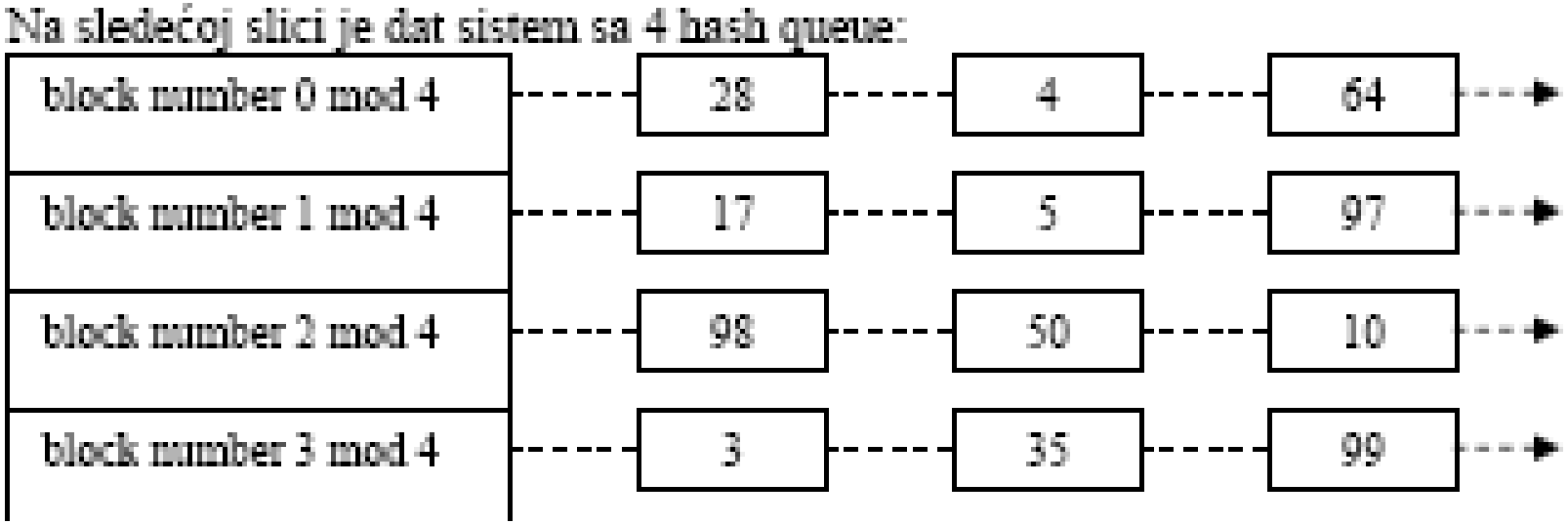
Slobodna lista posle dodele bafera #1



Struktura bafera (Buffer pool)

- Kada kernel prisitupa disk bloku,
- **on pretražuje sve headers** za odgovarajuću kombinaciju
- **<device, block number>**
- **Da ne bi pretraživao sva zaglavlja**,
  - kernel organizuje cache-bafer u odvojene redove (**hash queue**)
  - na bazi hash funkcije od **device-block** broja.
- Kernel povezuje bafere na duplo linkovani hash redove čekanja, u sličnu strukturu kao za listu slobodnih blokova.
- **Broj bafera** za jedan **hash queue** je **promenljiv** i zavisno od obraćanja disku.
- Kernel korsiti **hash funkciju** za raspodelu baferu između hash redova, koja mora biti dovoljna brza.
- **Broj hash redova** se **određuje prilikom podizanja sistema** i to može po default-u ili sistem administrator određuje.

hash queue example



- Na levoj strani su headeri, blokovi se raspoređuju u **hash redove** po funkciji LBA mod 4.
- Isprekidane linije ukazuju na hash pointere u oba smera.
- Svaki bafer je u nekom od hash queue, ali može se dogoditi da bude i u slobodnoj listi (never allocated).

Tehnike za dobijanje podataka iz bafera

- **Keš algoritam** upravlja baferskim kešom. **Za svako čitanje sa diska kernel** mora prvo da odredi da li je blok u kešu (cache buffer pool) i ako nije tamo mora da mu **dodeli slobodan bafer**. To se **isto** dešava i **prilikom upisa**, oba algoritma i za čitanje i za upis koriste čuveni UNIX **getblk** algoritam koji fukcioniše na sledeći način.
- Algoritam **getblk** ima **5 mogućih scenarija** prilikom dodele baferu u kešu:
  - #1 **Kernel pronalazi** bafer u njegovom **hash queue** i bafer je slobodan **(hit&free)**
  - #2 Kernel **ne nalazi** bafer u njegovom **hash queue**, zato može da alocira bafer iz free liste **(miss&there is free)**
  - #3 Kernel **ne nalazi bafer** u njegovom hash queue, zato može da alocira bafer iz free liste ali taj bafer ima atribut delayed write, što znači da mora prvo da se upiše na disk **(miss&there is free, dirty)**
  - #4Kernel **ne nalazi bafer** u njegovom hash queue, ali i **free lista je prazna (miss&there is no free)**
  - #5 Kernel nalazi bafer u njegovom hash queue, **ali je taj bafer locked (busy),** neko drugi ga je vec da ga čita **(hit&busy)**
- Pažnja: blok je free, ako je u free listi, koja znači da ga niko ne koristi pri čemu može biti u hash listi ili ne..

getblk (block is in the cache)

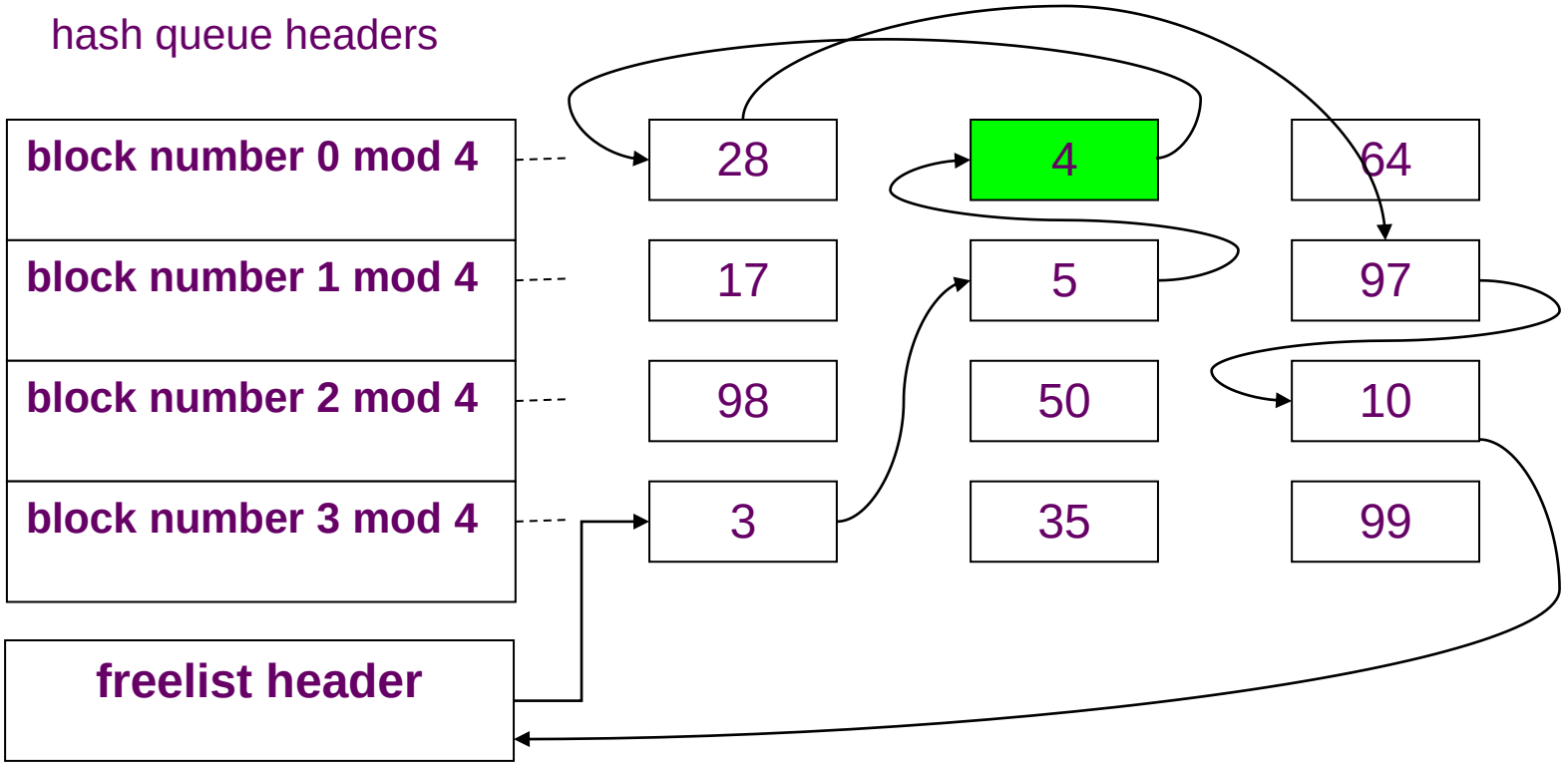
- **algoritam getblk**
- input: **FS number, block number**
- output: **locked buffer** that can now be used for block
- {
- while (buffer not found)
- {
- if (**buffer in hash queue**)
- {
- if **(buffer busy**) /***scenario 5***/
- {
- sleep (**event buffer becomes free**);
- continue; /* back to while lop*/
- }
- **mark buffer busy**; /***scenario 1***/
- **remove buffer** from **free list**; /*lock*/
- return buffer;
- }

getblk (block is not in the cache)

- else /***block not on hash queue***/
- {
- if (**there are no buffers on free list**) /***scenario 4***/
- {
- sleep (**event any buffer becomes free**);
- continue; /* back to while lop*/
- }
- remove buffer from free list; /*lock*/
- if (**buffer marked for delayed write**) /***scenario 3***/
- {
- asynchronous write buffer to disk;
- continue; /* back to while loop*/
- }
- /* **scenario 2 – found a free buffer***/
- **remove buffer** from **old hash queue**;
- **put buffer** onto **new hash queue**;
- return buffer;
- }/*else*/ }/*while*/ }/*main*/.

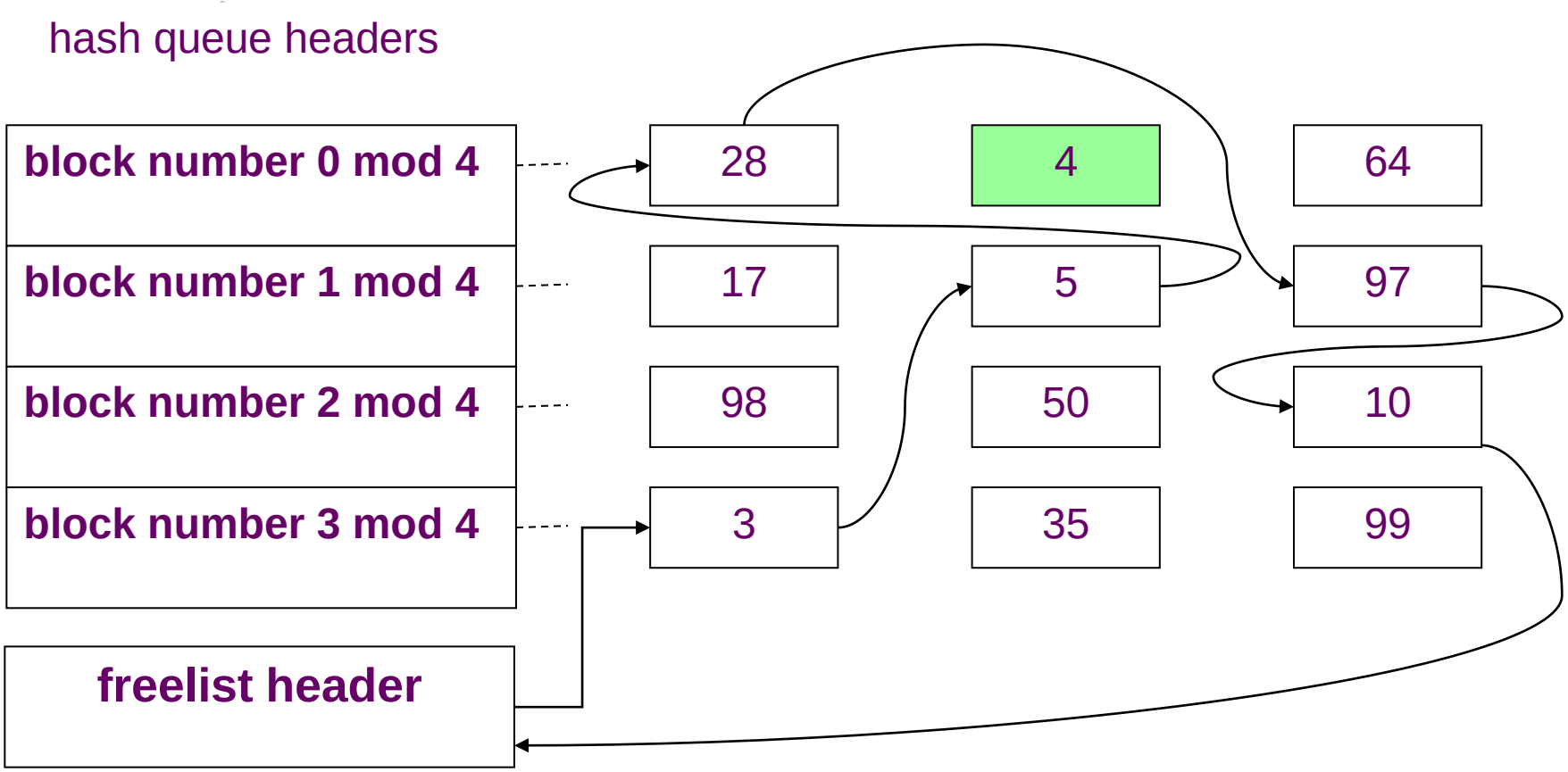
scenario #1 (searching)

- Tražimo blok 4, blok 4 se nalazi u hash queue 0 i slobodan je



scenario #1 (allocating)

- **Blok 4 je nađen, izbacuje se iz free liste**



after allocating

- Pre nego što objasnimo ostale scenarije, demonstrirajmo šta se dešava kada se bafer dodeli.

- Svaki proces posle **getblk** dobija bafer **koga kernel je locked** samo za njega.
- Iza toga blok može da se čita da se upisuje i

- **sve dok mu kernel drži lock**, nijedan proces **ne može pristupiti tom baferu**.

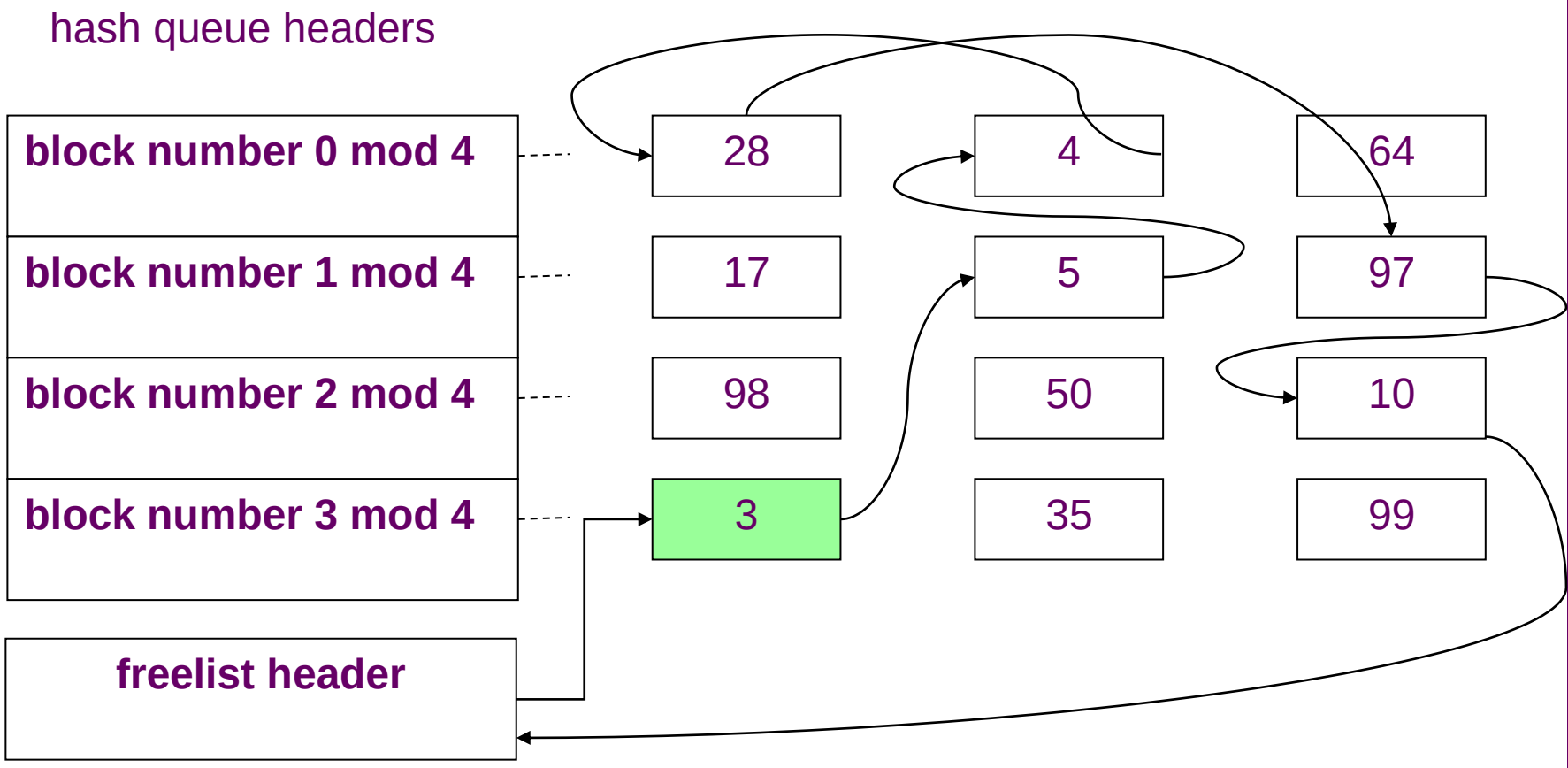
- Kada proces obavi svoje, **bafer se mora osloboditi** preko algoritma **brelse**.

- Uvek **kada se ažurira free lista**, prekidi **moraju biti blokirani**.

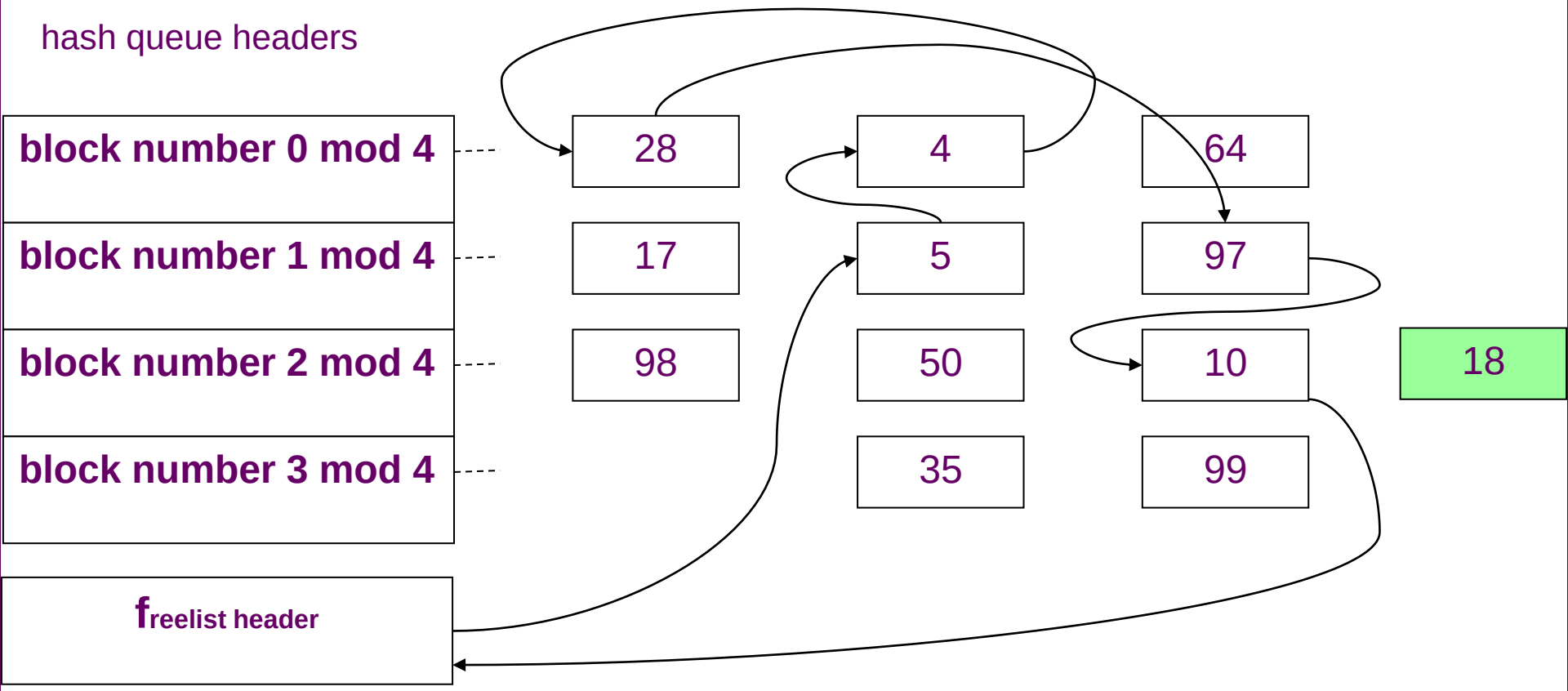
algoritam brelse

- **algoritam brelse**
- **input: locked buffer**
- **output: none**
- {
- **wakeup all procs**: event, **waiting for any buffer** to become free;
- **wakeup all procs**: event, waiting **for this buffer** to become free;
- **raise CPU execution** level **to block interrupts**;
- **if (buffer contents valid and buffer not old)**
- enqueue buffer at the **end of free list**
- else
- enqueue buffer **at** the **beginning of free list**
- lower CPU execution level to **allow interrupts**;
- **unlock(buffer);**
- }/*main***/.**

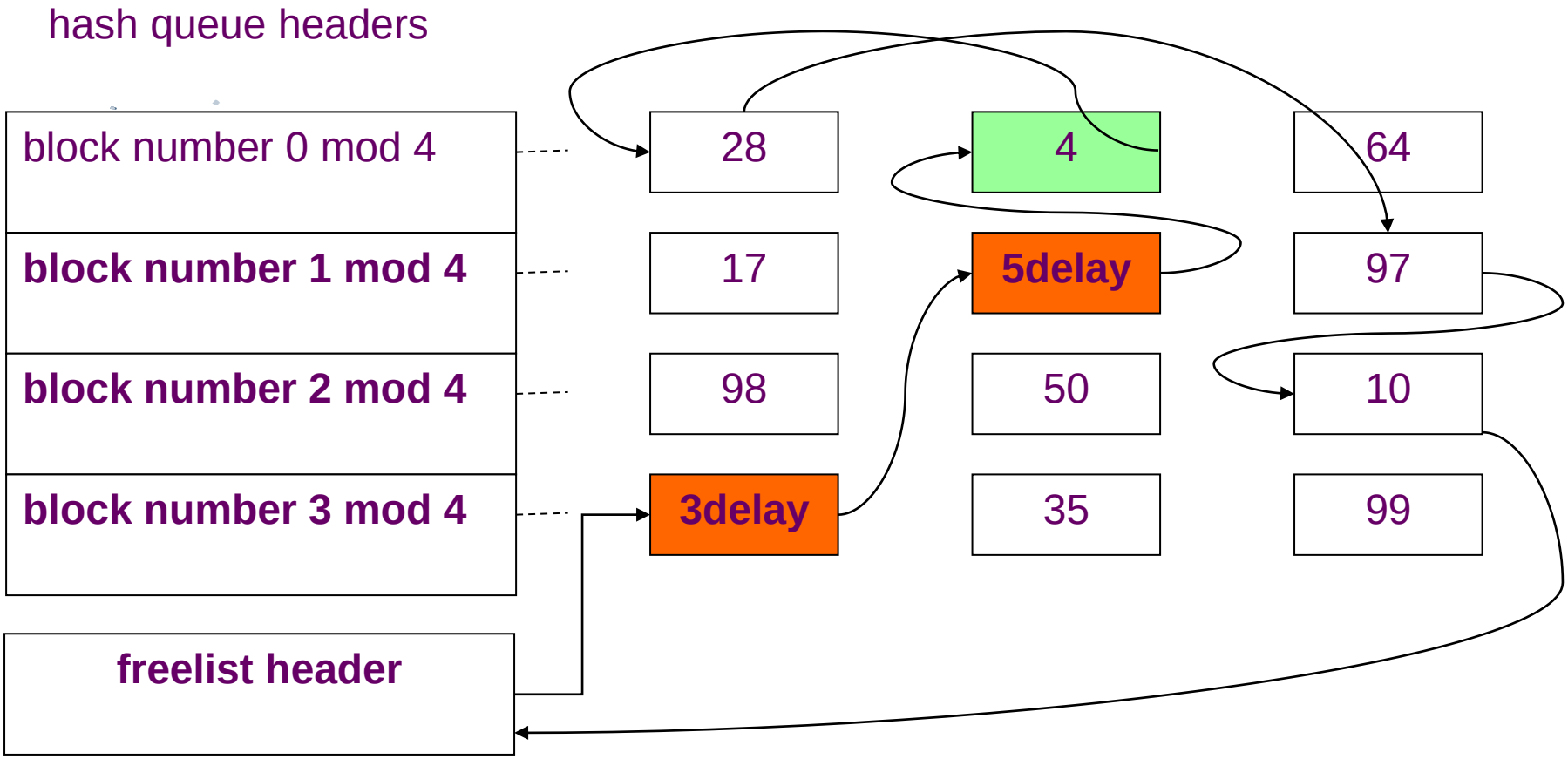
scenario 2. a) Tražimo blok 18, nije u kešu



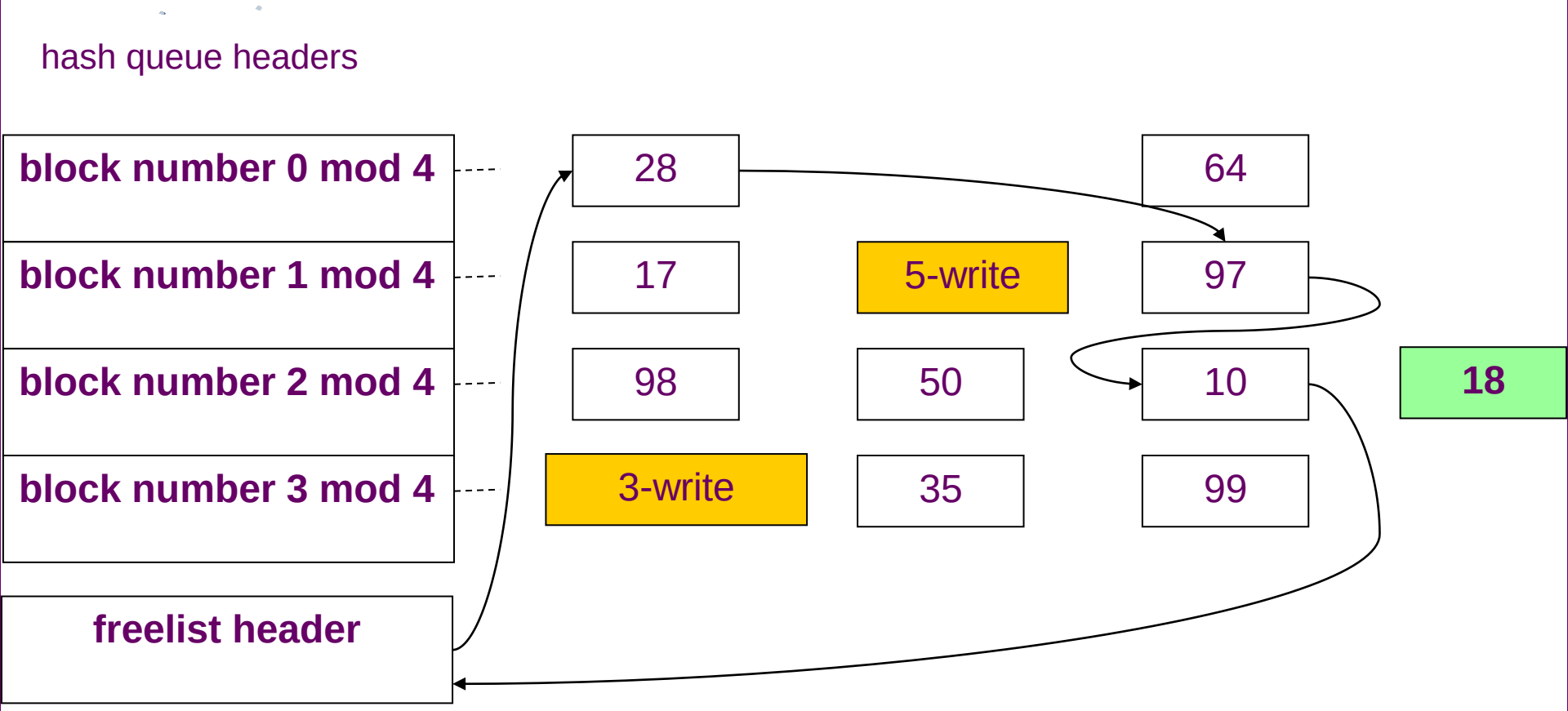
scenario 2. b) Uklanja se prvi blok iz free liste i dodeljuje novom bloku 18



scenario 3. a) Tražimo blok 18, nije u kešu, ali 3 i 5 moraju da se upišu na disk

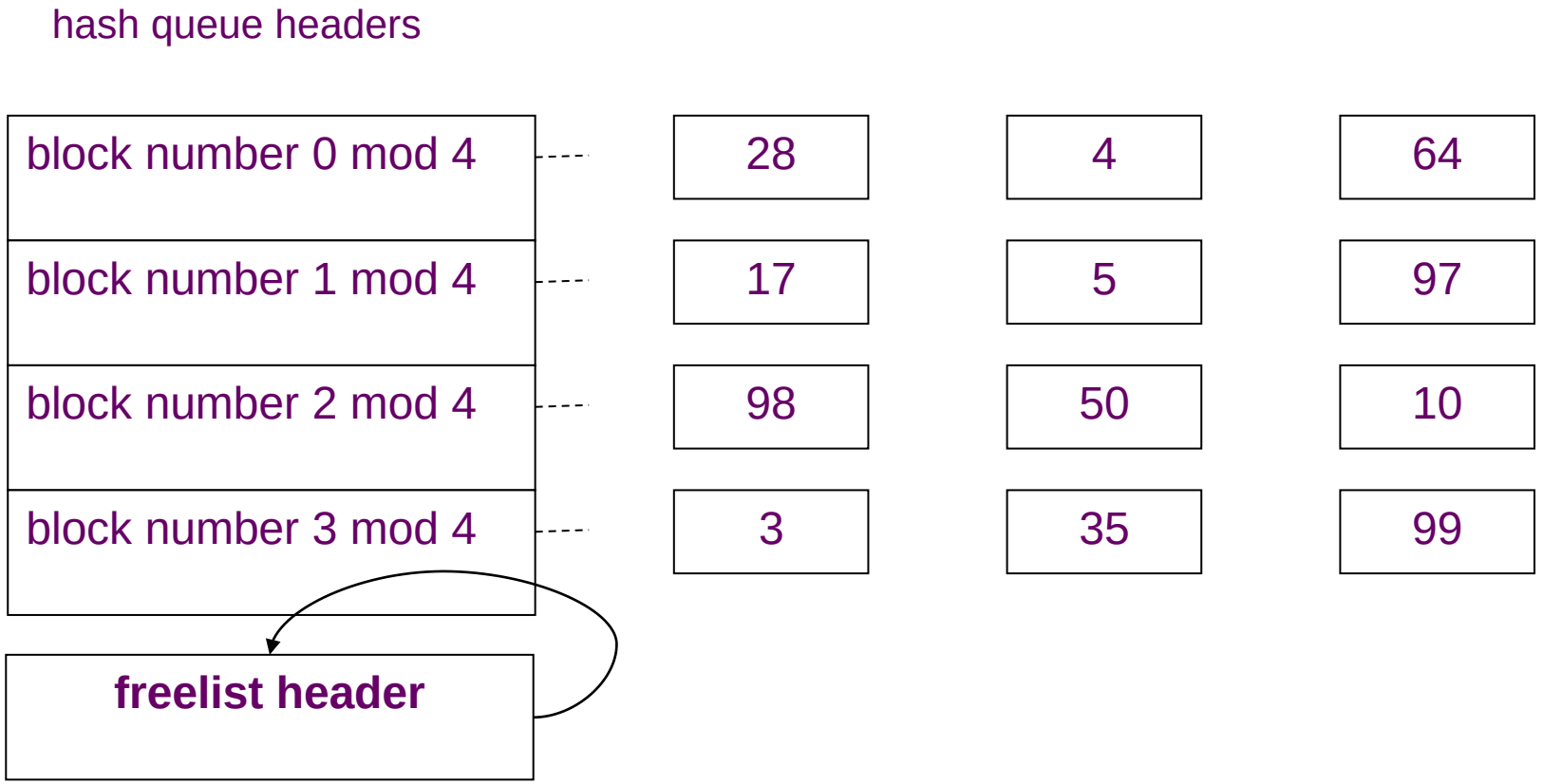


cenario 3. b) Blokovi 3 i 5 se upisuju a prvi slobodni se dodeljuje za 18, a to je 4



scenario 4. a) Tražimo blok 18, nije u kešu, nema ni jednog slobodnog bloka

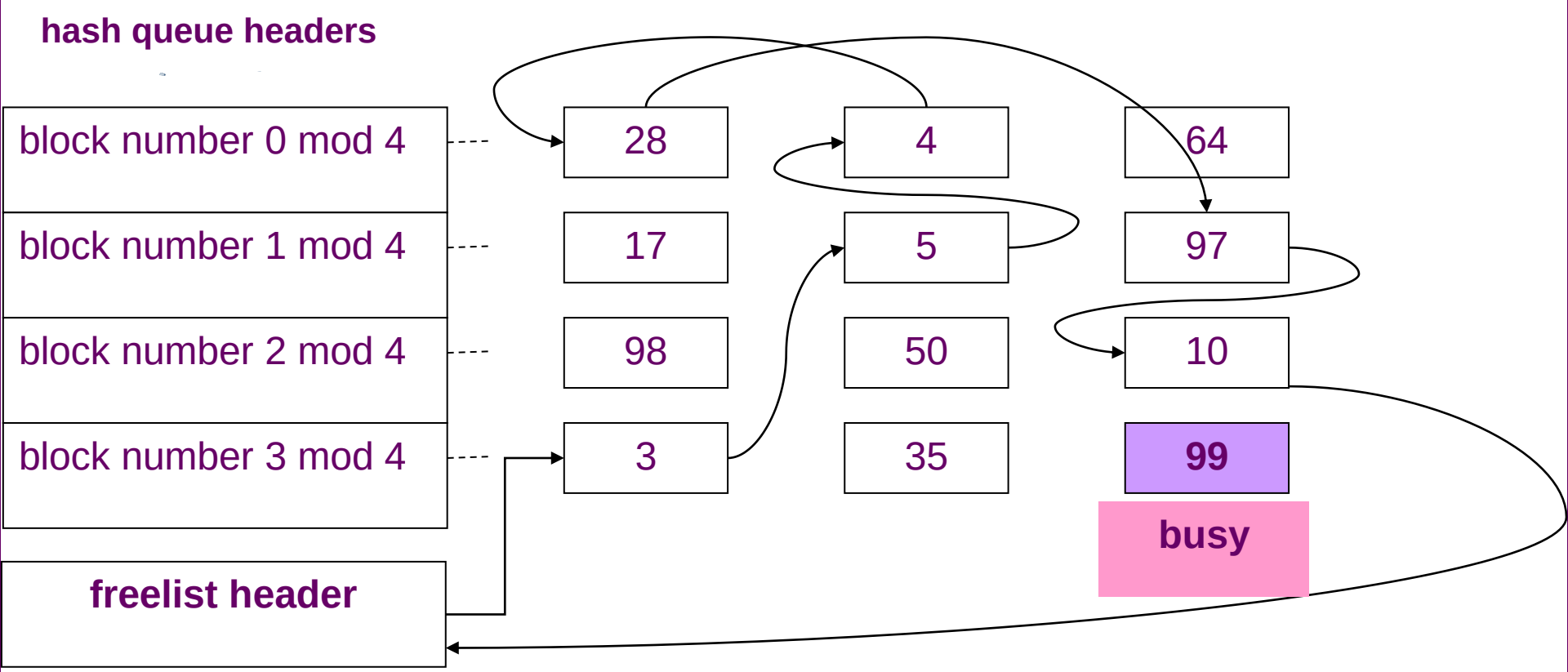
- free list je prazna



- Proces mora otići na spavanje sve dok neko ne obavi algoritam brelse koji će osloboditi bar jedan bafer.
- Čak i da je u kešu hit, proces mora na spavanje.

scenario 5. a) Tražimo blok 99, u kešu je, ali je trenutno zauzet (locked)

- Proces mora da čeka da blok bude oslobođen.



Čitanje i upis kroz bafer bread

- Algoritam za čitanje je bread.
- Kod njega su 2 glavne karaktersitike, a **cache hit** i **cache miss**.

- **algorthm bread()** /* block */
- input: file system block number
- output: buffer containing data
- {
- get buffer for block (algorithm **getblk**);
- if (**buffer data valid**) return (buffer); **/*hit*/**
- else
- **initiate disk read**; **/*miss*/**
- sleep (event disk read complete);
- return (buffer);
- }

read-ahead (breada)

- **Za povećanje performansi koristi se read-ahead tehnika:**

- **algoritam breada** /* block read and read ahead*/
- input:
  - (1) file system block number for immediate read
  - (2) file system block number for asynchronous read
- output: buffer containing data for immediate read
- {
- if (**first block not in** the **cache**) /*cache miss*/
- {
- **get buffer for first block** (algorithm **getblk**);
- if (buffer data not valid) **initiate disk read**;
- }
- if (**second block not in** the **cache**)
- {
- **get buffer for second block** (algorithm **getblk**);
- if (buffer data valid) release buffer (algorithm **brelse**)
- else **initiate disk read**;
- }
-

read-ahead (breada)

- if (first block **was originally** in the cache)
- {
- read first block (algorithm bread);
- return buffer;
- }

- sleep (event first buffer contains valid data)
- return buffer

- }/*main*/
- **miss (2 read=read normal + read ahead)**
- Ako **prvi blok nije u kešu** odvija se **sinhrono disk čitanje** sa sleep-om, a ako je u kešu zadaje je neposredno prosleđivanje bafera.
- Odmah se obavlja **asinhroni read-ahead**, proveri se da li je sledeći sekvencijalni blok u kešu.
- Ako jeste nikom ništa, ali ako nije nađe se blok u kešu (getblk) i ako je dodeljeni bafer prazan, inicira se asinhrono disk čitanje.

- **hit (no disk reading)**
- Ako dodeljeni bafer već ima validne podatke read-ahead se ne izvršava, već se bafer otpušta (to je hit za read-ahead).

upis kroz bafer (bwrite)

- Algoritam za upis je **bwrite**. 2 glavne karaktersitike, cache hit i cache miss.
- **algorthm bwrite()** /* block */
- input: **buffer to be written**
- output: none
- {
- initiate disk write; /* ovaj I/O možda se obavlja ili sihrono ili delayed */
- **if(I/O synchronous**)
- {
- sleep (event I/O complete)
- release buffer (algorithm **brelse**);
- }
- else if (buffer marked for **delayed write**);
- **mark buffer** to **put** at **head** of **free list**;
- }

upis kroz bafer (bwrite)

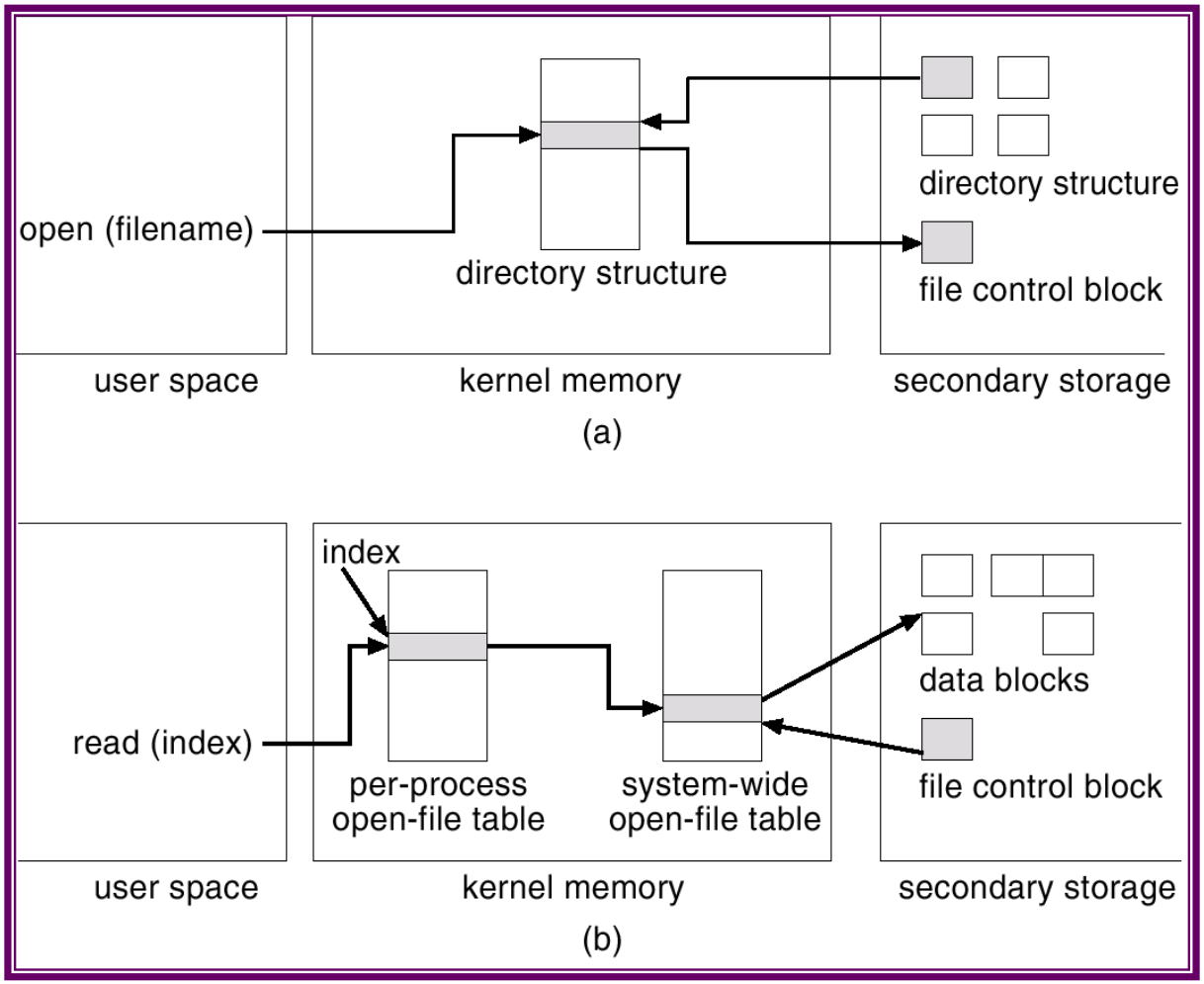
- Pretpostavimo da je upis **hit(in_cache)** i kernel daje nalog za upis na disk.
- Taj nalog može biti sinhroni i DW.
- Ako je **sinhroni write**, kernel upisuje blok na disk i čeka da se I/O završi.
- Ako **delayed write**, to se markira za taj bafer i ne vrši se I/O, a bafer se oslobođa.
- Upis se inicira u **getblk** algoritmu po scenariju 3 kada je DW blok u free listi, a dodeljen je novi blok iz free liste.
- Ti DW baferi se iniciraju za write, odnosno **pražnjenje**.
- DW blokovi se potiskuju na početak free liste kako bi se praznili na svaki keš miss.

Prednosti i nedostaci baferskog keša

- **Prednosti:**
  - **Smanjuje disk trafic**
  - DW je izrazita prednost
  - **keš je kernelska memorija** koja je zaštićena i dobro sinhronizovana, **nije podložna programerskim** ili **korisničkim greškama**

- **Nedostaci:**
  - **međutransfer**, sa **diska u keš**, pa u **user buffer**
  - **DW** može dovesti do **gubitka podataka**, što se ublažava sa journaling tehnikom

In-Memory File System Structures



Directory cache

- **Directory**
- **=**
  - **special file or Structure**
  - **contain FCBs**

- **Directory cache = cache for directory blocks**

- **Contains the most recently used directory blocks**

Metadata cache

- Cache for metadata area
- Contains the most recently used metadata fragments

- UNIX
  - Open-file tables
  - In-core inode table

- MS Windows
  - Fragments of FAT
  - Fragments of NTFS, MFT

INODES

- **Definicija**
- struktura koja potpuno opisuje datoteku
- sve osim imena

- Postoje 2 strukture:
- **disk inode**
  - to su inodes u statičkoj formi na disku (in inode table)
  - pasive files
- **in-core inode**.
  - disk inode kernel čita u memorijsku strukturu koja se naziva nazvati in-core inode
  - active files

disk-inode

- **Disk inode** se sastoji od **sledećih polja**:
  - **Vlasništvo** se deli na 2 komponente, na korisničko vlasništvo i grupno vlasništvo i ta dva identifikatora određuju vlasničke odnose grupe prema datoteci
    - **vlasnik datoteke** (**UID identifikator**).
    - **grupa** kojoj pripada datoteka (**GID identifikator**)
  - **tip datoteke**: datoteka može biti regularna, direktorijum, karakter ili blok specijalna datoteka, FIFO (pipe)
  - **prava pristupa za datoteku**: definišu se preko
    - 3 vlasničke kategorije (**owner, group, other**)
    - 3 prava pristupa za njih **(read, write, execute**) koja imaju precizno značenje, a nezavisno se deklarišu
  - **vremena pristupa datoteke**: ima 3 karakteristična vremena (vreme poslednje modifikacije, vreme poslednjeg pristupa, i vreme kad je inode poslednji put modifikovan)
  - **broj linkova na datoteku**: predstavlja broj različitih imena za isti prostor na disku
  - **tabela koja opisuje alokaciju** datoteke **na disku**
  - **veličina** datoteke
- Inode ne opisuje path imena već svaka path komponenta ima poseban inode.

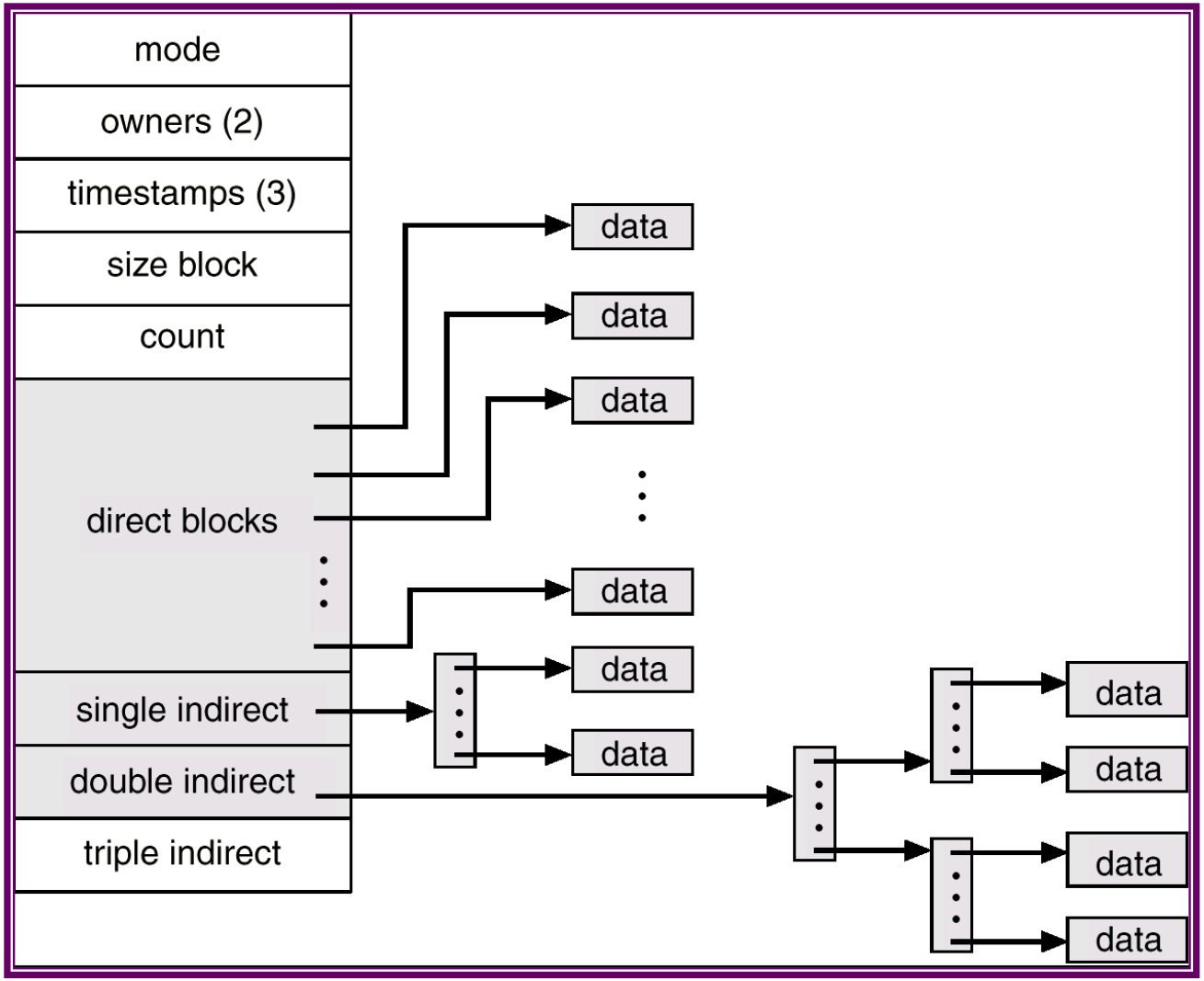
Disk inode example

- **U ovom primeru** imamo regularnu datoteku veličine 6030 bajtova čiji je vlasnik user mjb sa pravima pristupa rwx, datoteka pripada grupi os i svi članovi grupe imaju r i x pravo bez w prava, dok svi ostali useri imaju r i x pravo bez w prava. Poslednji put je neko pristupao i to samo čitao datoteku 23. oktobra 2004 u 1:45 PM, a posledni put je neko upisao nešto u datoteku 22. oktobra 2004 u 10:30 AM. Inode je promenjen zadnji put 23. oktobra u 1:30 PM.
- **Postoji razlika** između **upisa u datoteku** i **upisa u inode**, **svaka promena** u **datoteci se reflektuje u inode**, dok postoje promene koje idu samo u inode a nemaju veze sa upisom datoteku, kao što je promena vlasništva, grupe, prava pristupa ili link setovanje.

owner mjb
group os
type regular file
perms rwxr-xr-x
accessed Oct 23 2004 1:45 P.M.
modified Oct 22 2004 10:30 A.M.
inode Oct 23 2004 1:30 P.M.
size 6030 bytes
disk adresses

UNIX Metadata cache

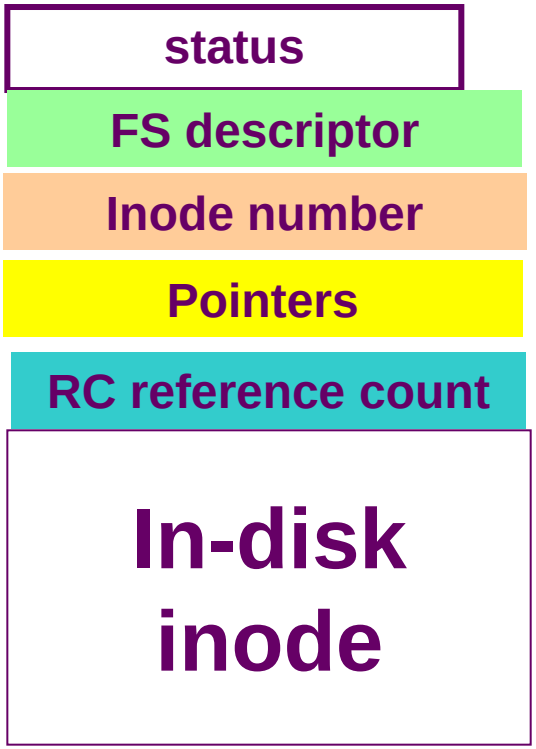
- **Disk inode**



in-core inode

- **in-core inode** sadrži dodatne informacije u odnosu na **polja disk inode**:
- **Status in-core inoda** koji pokazuje:
  - da li inode zaključan (**locked**)
  - da li proces čeka da inode postane otključan (**unlocked**)
  - da li se **in-core inode razlikuje od svoje disk kopije** kao rezultat **promene polja u inode**
  - da li se **in-core inode** razlikuje od svoje disk kopije **kao rezultat promene** podataka **u datoteci**
  - da li je ta datoteka postala **mountpoint**
- **FS descriptor**(opis FS): u vidu logičkog broja koji sadrži tu datoteku, odnosno iz kog FS je taj inode
- **inode broj: in-core** inode poseduje i ovo polje zato što **na disku se pozicija inode** određuje u polju **fiksnog formata**, zna se offset, a **u memoriji polje nije fiksno** i mora da se zna koji je inode u **in-core tabeli**
- **ukazivači** na druge **in-core inodove**. Kernel povezuje **in-core inode-ove** u **hash queue liste** i **slobodne liste** na sličan način kao kod baferskog keširanja. **Hash queue se idetifikuje** n**a osnovu FS** i **inoda broja**. Kernel sadrži najviše jednu kopiju disk inoda, a ona može biti ili u hash queue ili u slobodnoj listi.
- **broj referenci**, pokazuje broj aktivnih instanci na tu datoteku, broj procesa koji su otvorili datoteku, odnosno taj inode

In-core inode



in-core inode v buffer cache

- **Mnoga polja** u in-core(inode) su **analogna poljima** u baferskom zaglavlju
- **upravljanje incore inodovima** je slično kao i kod baferskog keša.
- Kada se **inode zaključa**, ostali procesi ne mogu da ga otvore, a ostali procesi postave svoje flagove da su zainteresovani za taj inode i da ih treba probuditi kada se inode otključa.
- Kernel postavlja i **druge flagove** koji ukazuju na razliku između disk inode i njegove in-core kopije, kako bi na bazi tih flagova sravnjuje(destage) stanje inoda na disku.
- **Najveća razlika** između incore inoda i bafer zaglavlja je u **broju referenci** za **in-core inode** koga **uopšte nema kod baferskog keš zaglavlja**.
- **Svaki proces** koji pristupa **in-core inodu** povećava mu **reference count**
- **inode može biti u free listi samo ako je RC=0**.
  - Samo takav inode može napustiti in-core,
  - dok kod **baferskog keša**, **bafer je free listi samo ako je otključan** (unlocked)

Prefetching: Case a study

- Traditional
- or
- Application controlled

Traditional prefetching

- **Almost sequential prefetching**

- **1. One-block-look-ahead**
  - **If block K and K+1 have been referenced, prefetch block K+2**

- **2. Extent based prefetching**
  - When a **cache miss happens**
    - **fetch not only** the **requested block**
    - but also a **number of adjacent blocks (whole extent)**

Traditional replacement

- LRU
- or
- LRU modifications

New approach: ACFS

- **Application-controlled File System**
- **Application-controlled caching**
- \+
- **Prefetching of N next requests**

- **Based on knowledge of future access pattern**

- **Incorporate disk scheduling** **Limited batch scheduling**

Two-level cache management

- **Two-level cache management strategy**
- **kernel decides**
  - **how many cache blocks each process can use**
- **each process decides**
  - **how to use its block for caching** and **prefetching**
- **1. Kernel** have **global allocation** policy
  - for deciding how allocate cache blocks to processes
- **Process** has a **local policy**,
  - **for deciding how to use its blocks**
- **2. Local policy**
- **Default-kernel based**:
  - **LRU + one-block look-ahead prefetching**
- **Smart:**
  - **caching + controlled-aggressive prefetching + disk scheduling**

LRU-SP

- **swapping + place holder**
  - **If block A is at the end of LRU list** and
  - the **user process chooses to replace block B instead**,
  - **kernel swaps** the **positions of A and B in LRU list**
  - **then kernel builds a record = placeholder for B, pointing to A**
  - **placeholder will remember** the **process's choice**
- **miss of B**
  - **If a user process miss on the block B**, and **placeholder for B exists**
  - **then** the **block pointed to by that placeholder is replaced**
  - Otherwise,
    - **process that owns the block at the end of LRU list**
    - **is chosen to give up a block**
- **hit of A**
  - **If** a **user process hits on block A, placeholder pointing to A is deleted**

LRU-SP

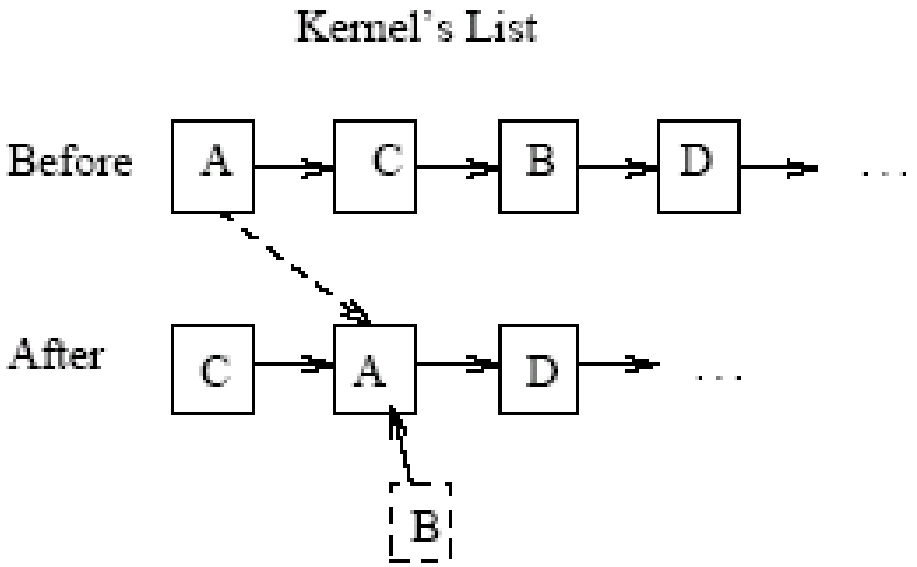


Figure 2: How LRU-SP keeps the "LRU" list. Here block A is at the "least recently used" end of the list. The process that owns A decides to replace block B. The figure shows the list before and after the replacement decision.

Implementation of ACFS

BUF handles the file accesses to the buffer cache, does bookkeeping, and implements the cache block allocation policy. ACM implements the application-controlled caching interface calls and acts as a proxy for the user-level managers. The PCM module implements the application prefetching interface and issues prefetch requests. It interfaces with the BUF, ACM and disk driver modules to optimize prefetching performance.

Prefetching Control Module

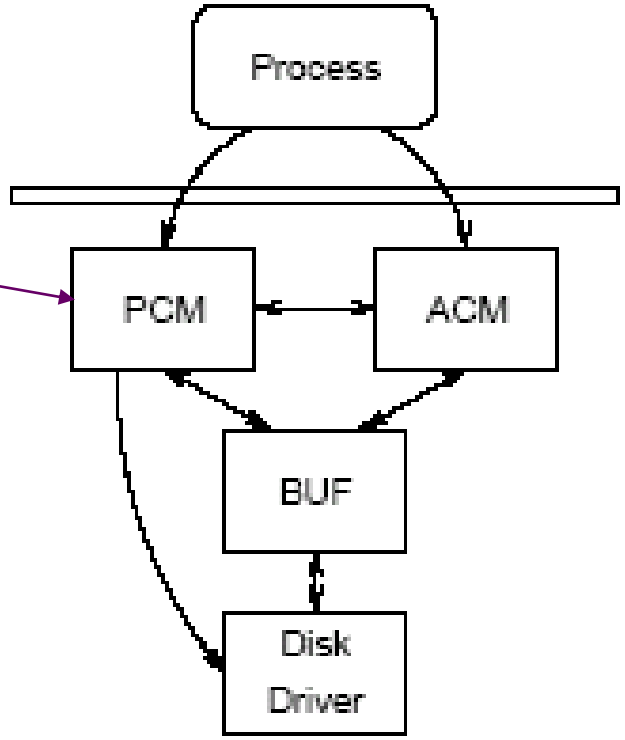


Figure 3: Structure of our implementation. Cache management is handled by three modules: BUF manages the allocation of blocks to processes, ACM implements the replacement policy on behalf of each application, and PCM manages prefetching.

Performance measurement

- **1. Global LRU:**
  - **global LRU**
  - **with one-block look-ahead sequential prefetching on file blocks**

- **2. Global LRU + prefetching**:
  - **baseline FS with addition** of **application-controlled prefetching**:
  - **application can kernel what to prefetch**;
  - **kernel uses default LRU** for **each application's replacement policy**

- **3. Global LRU + prefetching + scheduling**:
  - the same as previous
  - **with disk scheduling added**

Performance measurement

- **4. AC caching (replacement):**
  - baseline FS
  - with the addition of two-level cache allocation
  - and application-controlled caching

- **5. AC caching + prefetching:**
  - an **integration of application-controlled caching and prefetching**,
  - **using controlled aggressive cache management policy**

- **6. AC caching + prefetching + scheduling**:
  - the **fully-integrated system**
  - **which incorporated all the technique**

Single process performances (7 benchmarks)

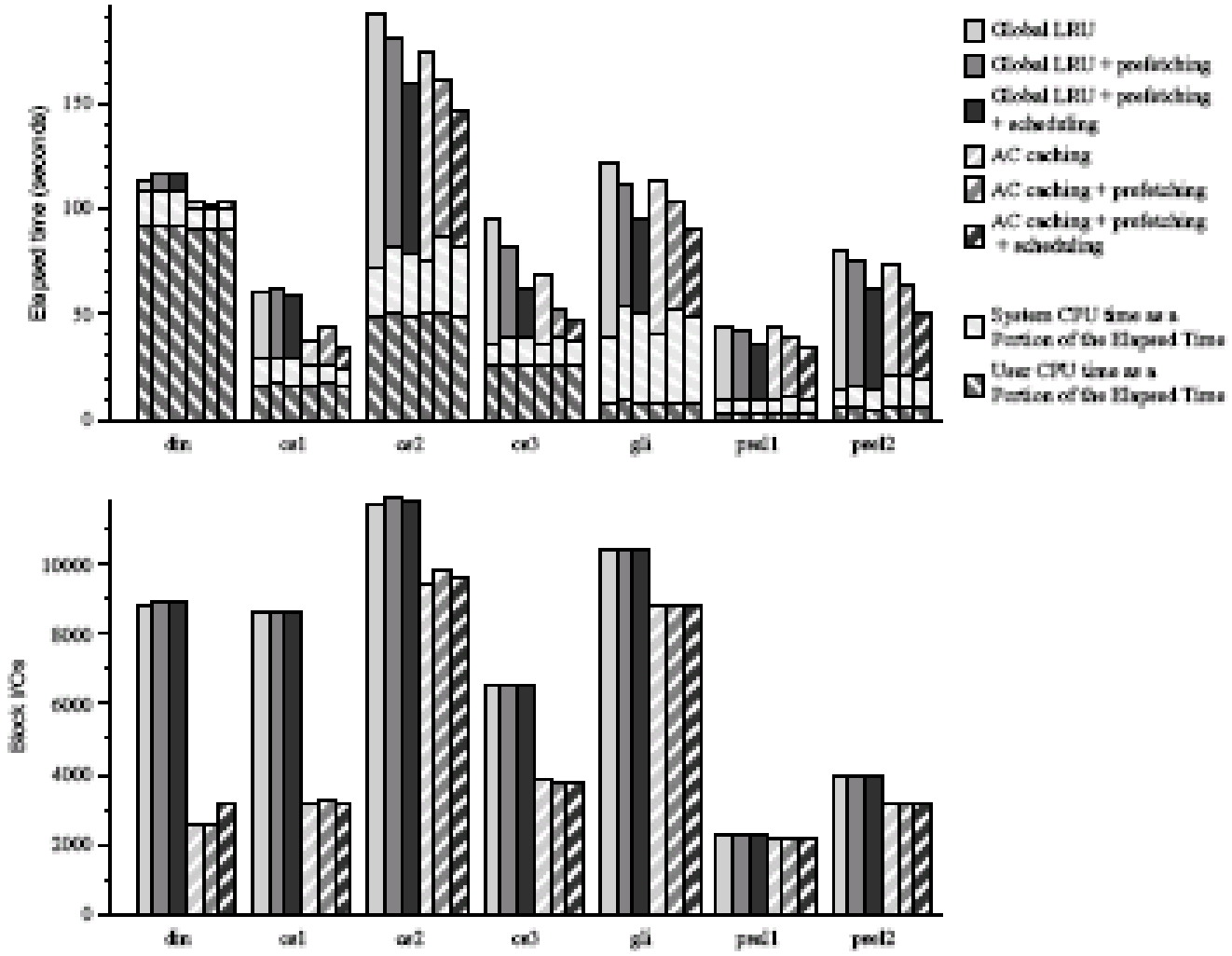


Figure 4: Single application running time, and number of block I/Os, with a 6.4MB file cache.

LRU-origin

- LRU = Least Recently Used
  - replace the page
  - that was least recently accessed
  - or
  - used

- Until the early 80's, LRU was the choice in nearly all cases.

- database systems usually have access patterns
- in which
- LRU performs poorly

- As a result
- there has been considerable interest
- in buffer management algorithms
- that perform well in a database system.

Contribution of LRU-K

- **LRU does not discriminate well**
  - **between frequently** and **infrequently referenced pages**,
  - **it is necessary to have some other agent provide hints** of **one kind or another.**

- The **contribution** of the **LRU-k is to derive**
  - a **new self–reliant page–replacement algorithm**
  - **that takes into account more of the access history for each page**,
  - **to better discriminate pages that should be kept in buffer**.

- **This seems** a **sensible approach**
  - **since** the **page history used by the LRU algorithm is quite limited**:
  - **simply** the **time of last reference**.

LRU Modifications

- 2Q
- LRU-K
- LRFU

Two tunable parameters

Correlated Reference Period

- **At the same time**,
  - **multiple accesses within** a **correlation period**
  - **count only as** a **single access**
  - **from** the **point of view of replacement priority**.

Retained Information period

- **which is**
- the **length of time**
- a **page's access history is remembered**
- **after it is ejected from** the **buffer**.

2Q

- **Instead of cleaning cold pages from** the **main buffer**,
- **2Q admits**
- **only hot pages to** the **main buffer-Am**.

- **As with LRU/2**, **2Q tests pages for their second reference time**

- Simplifying slightly,
- **on** the **first reference to a page**,
- **2Q places it in a special buffer**, the **A1 queue**,
- **which is managed as** a **FIFO queue**.

A1
FIFO based

- **If** the **page is re-referenced during its A1 residency**,
  - **then it is probably** a **hot page**.
  - **So, if a page in A1 is referenced**,
  - the **page is moved to** the **Am queue,**
  - **which is managed as** an **LRU queue**.

Am
LRU based

- **If** a **page is not referenced while on A1**,
  - **it is probably** a **cold page**,
  - **so 2Q removes it from the buffer**.

Conclusions

- **2Q is** a **good buffering algorithm,**
  - **giving** a **5-10% improvement**
  - **in hit rate over LRU for** a **wide variety of applications** and **buffer sizes and never hurting,**
  - **having constant time overhead, and**
  - **requiring little or no tuning**.
- **It works well for** the **same intuitive reason that LRU/2 works well**:
  - it bases buffer priority on sustained popularity
  - rather than on a single access.
- **2Q seems to behave as well as LRU/2 in our tests** (slightly better usually, in fact)
  - **can be implemented in constant time using conventional list operations**
  - **rather than in logarithmic time using a priority queue**,
  - **both analysis and experiment suggest it requires little or no tuning**.
- Finally, it can potentially be combined with buffer hint-passing algorithms of the DBMIN family.

LRU-K

- **The basic idea:**
  - **keep track of the times**
    - of the **last *K* references**
    - to **popular database pages,**
  - **using this information**
    - **to statistically estimate** the **inter-arrival time**
    - of **such references on** a **page by page basis**.

- **LRU–K approach incurs little book-keeping overhead**

LRU-K

- **Idea**
  - of **taking into account** the **history of** the **last two references**,
  - or **more generally the last K references**, $K \geq 2$

- LRU–2 **takes into account knowledge** of
  - the **last two references to a page is named**, and
  - the **natural generalization is the LRU–K algorithm**;
  - **classical LRU algorithm** within **this taxonomy as LRU–1**.

- **It turns out that**,
- **for K > 2**, the **LRU–K algorithm provides**
  - **somewhat improved performance over LRU–2**
  - **for stable patterns** of **access,**
  - **but is less responsive to changes in access patterns**,
  - an **important consideration for some applications**.

Backward K–distance $b_t(p,K)$

- **Given a reference string known up to time t**,
- **r1, r2, . .. , r_t,**
- **backward K–distance b_t(p,K)**
  - **is** the **distance backward**
  - **to** the **K^{th} most recent reference to** the **page p**:
- **$b_t(p,K) = x$**,
  - **if r_t–x has the value p** and
  - **there have been exactly K–1 other values I**
  - **with $t - x < i < t$, where $r_i = p$,**
- **= ∞,**
  - **if p does not appear at least K times in r1, r2, . . . , r_t**

Zipfian Random Access Experiment

B	LRU–1	LRU–2	A_0
40	0.53	0.61	0.640
60	0.57	0.65	0.677
80	0.61	0.67	0.705
100	0.63	0.68	0.727
120	0.64	0.71	0.745
140	0.67	0.72	0.761
160	0.70	0.74	0.776
180	0.71	0.73	0.788
200	0.72	0.76	0.825
300	0.78	0.80	0.846
500	0.87	0.87	0.908

OLTP Trace Experiment

B	LRU–1	LRU–2	LFU
100	0.005	0.07	0.07
200	0.01	0.15	0.11
300	0.02	0.20	0.15
400	0.06	0.23	0.17
500	0.09	0.24	0.19
600	0.13	0.25	0.20
800	0.18	0.28	0.23
1000	0.22	0.29	0.25
1200	0.24	0.31	0.27
1400	0.26	0.33	0.30
1600	0.29	0.34	0.31
2000	0.31	0.36	0.33
3000	0.38	0.40	0.39
5000	0.46	0.47	0.44

LRFU

- Two group of policies
- LRU
  - based on recency of references
- LFU
  - based on frequency of references

LFU

- **LFU policy keeps the reference count for each block**
- **Block selected** for **replacement** = **block** with **smallest Reference Count**

- **Drawback**
- **LFU considers all references in** the **past**
- **LFU may replace currently hot blocks**
- **instead of currently cold blocks**
  - ☞ **(that have been hot in past)**

- **Implementation**
- **Heap data structure** = **Balanced binary tree**
- **Heap to order** the **blocks according to their RC**

- **Time complexity of implementation** = $O(\log_2 n)$

LRU

- **LRU considers the time of** the **most recent reference**
- **Block selected for replacement**
- =
- **block that has not been referenced for** the **longest time**

- **Drawback**
- **LRU cannot discriminate well**
  - between
  - **frequently** and **infrequently referenced blocks**
- LRU may replace **hot blocks from past time**

- **Implementation**
- **Single linked list** = a **list that orders**
  - **the blocks according**
  - **to** the **times of their most recent references**

- **Time complexity of implementation = O(1)**

LRU-K

- **LRU considers the distance of** the **Kth reference**
  - (**Kth to last non-correlated reference**)
- **Block selected for replacement**
- =
- **block that has** the **longest distance of** the **Kth reference**
- **When the K is large**,
  - **LRU-K can discriminate well between frequently and infrequently referenced blocks**
- **When** the **K is small**,
  - **LRU-K can remove cold block quickly**
- **Drawback**
- **LRU-K does not combine**
  - **recency** and **frequency**
  - **in** a **unified manner**
- **Implementation**
- **Space complexity = O(K)**
- **Time complexity of implementation** = $O(\log_2 n)$

LRFU Least Recently/Frequently Used

- **LRFU policy associate** a **value with each block**
- **It is CRF (Combined Recency** and **Frequency)**
- **Each reference contributes in CRF with weighing function**
- **LRFU policy replaces** the **block with minimum CRF value**

Definition 1 *Assume that the system time can be represented by an integer value and that at most one block may be referenced at any one time. The CRF value of a block b at time* t_{base}, *denoted by* $C_{t_{base}}(b)$, *is defined as*

$$C_{t_{base}}(b) = \sum_{i=1}^{k} \mathcal{F}(t_{base} - t_{b_i})$$

where $\mathcal{F}(x)$ *is a weighing function and* $\{t_{b_1}, t_{b_2}, \cdots, t_{b_k}\}$ *are the reference times of block b and* $t_{b_1} < t_{b_2} < \cdots < t_{b_k} \leq t_{base}$.

LRFU

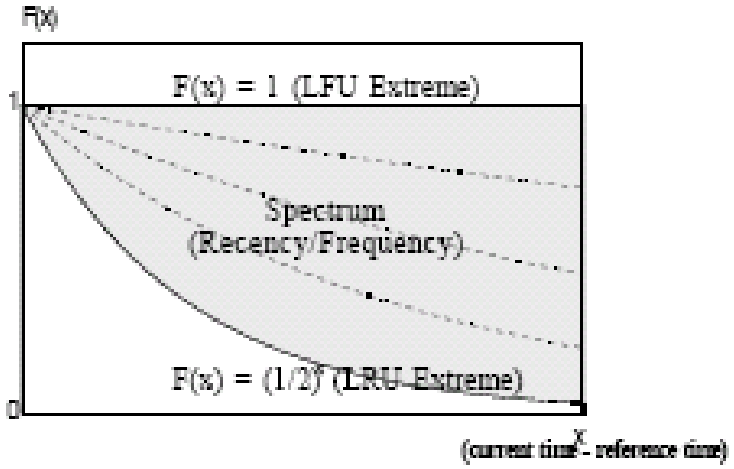


Fig. 1. Spectrum of LRFU according to the function $\mathcal{F}(x) = (\frac{1}{2})^{\lambda x}$ where x is (current_time − reference_time).

Spectrum of LRFU implementation

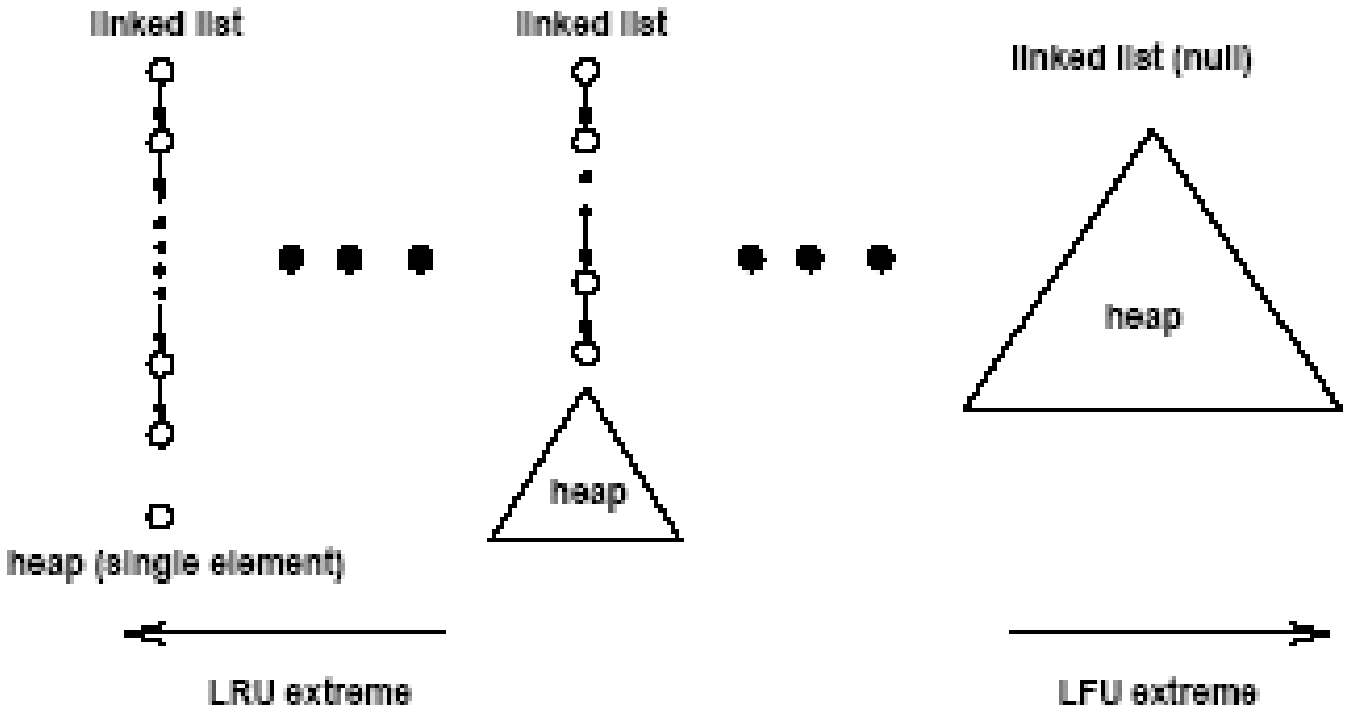
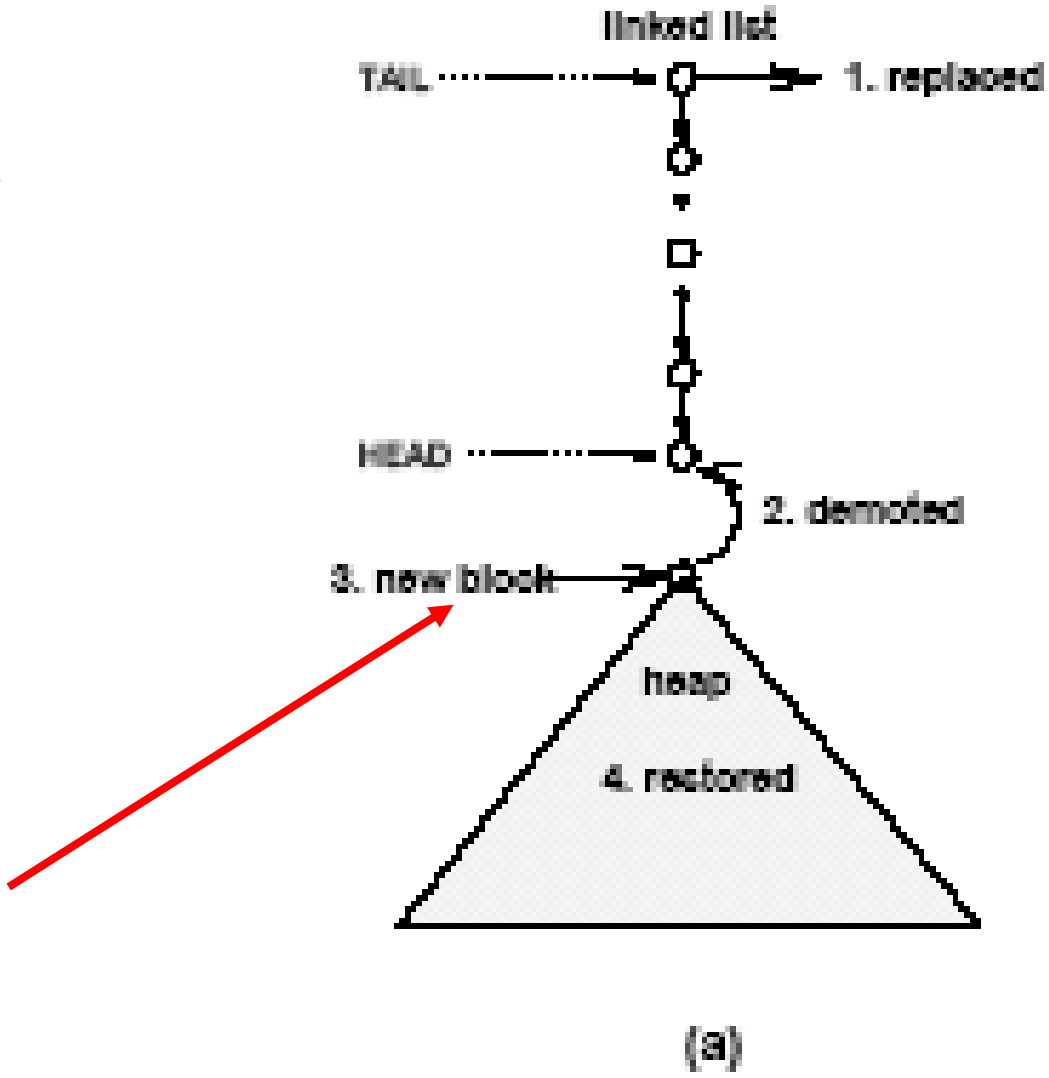
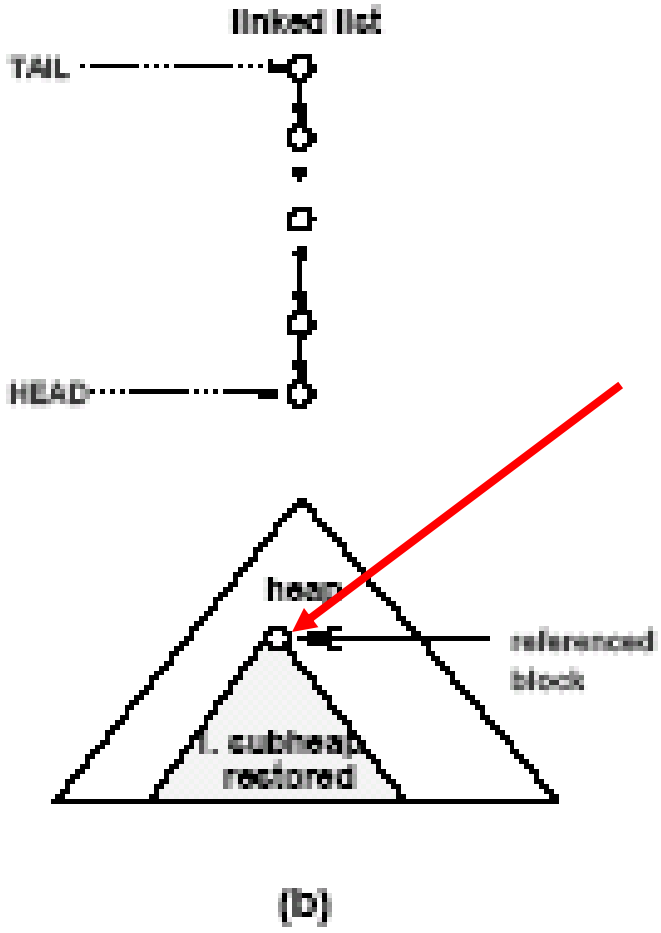


Fig. 4. Spectrum of the LRFU implementations.

LRFU miss

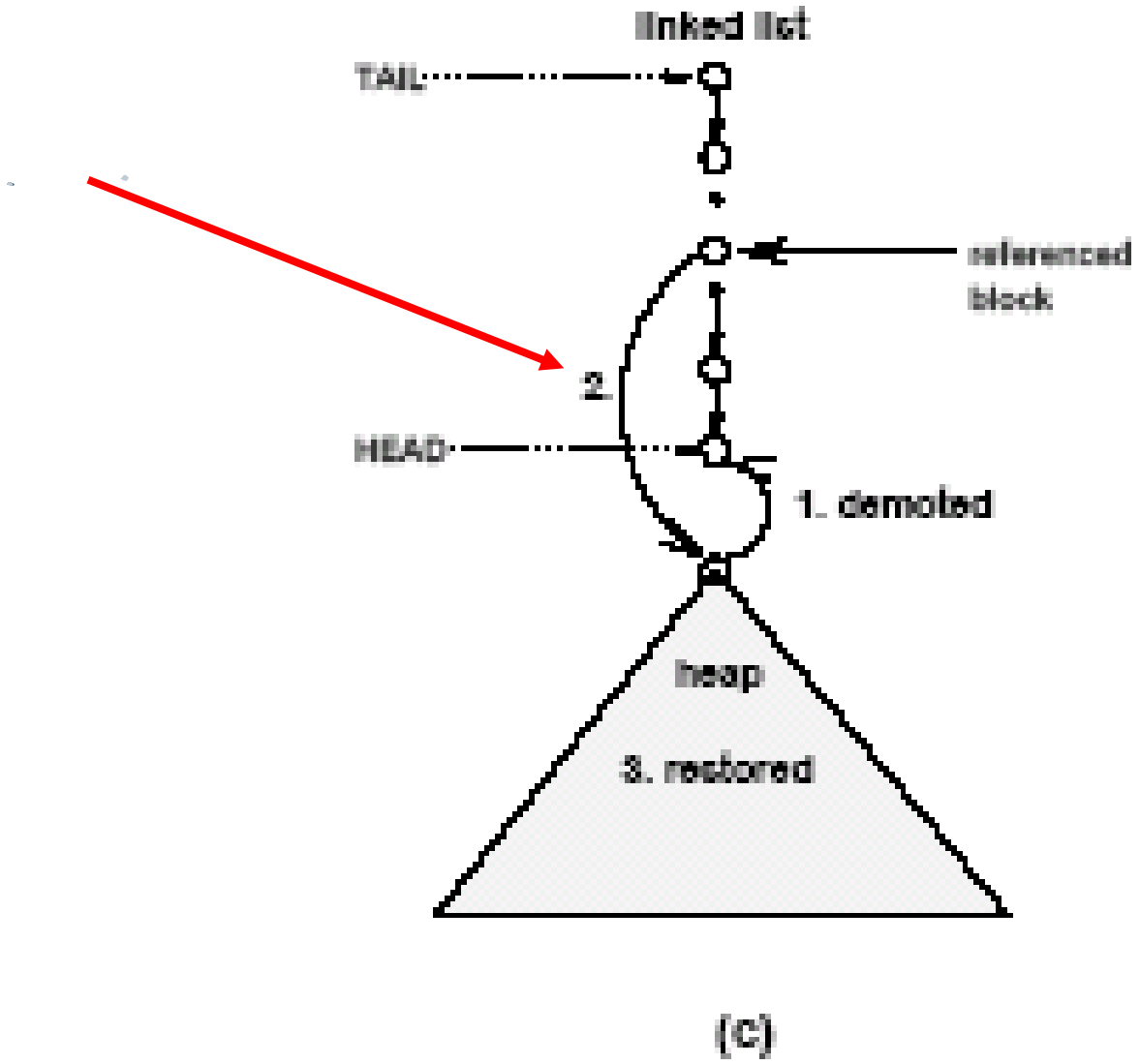

linked list
TAIL
1. replaced
HEAD
2. demoted
3. new block
heap
4. restored
(a)

LRFU hit (in the heap)

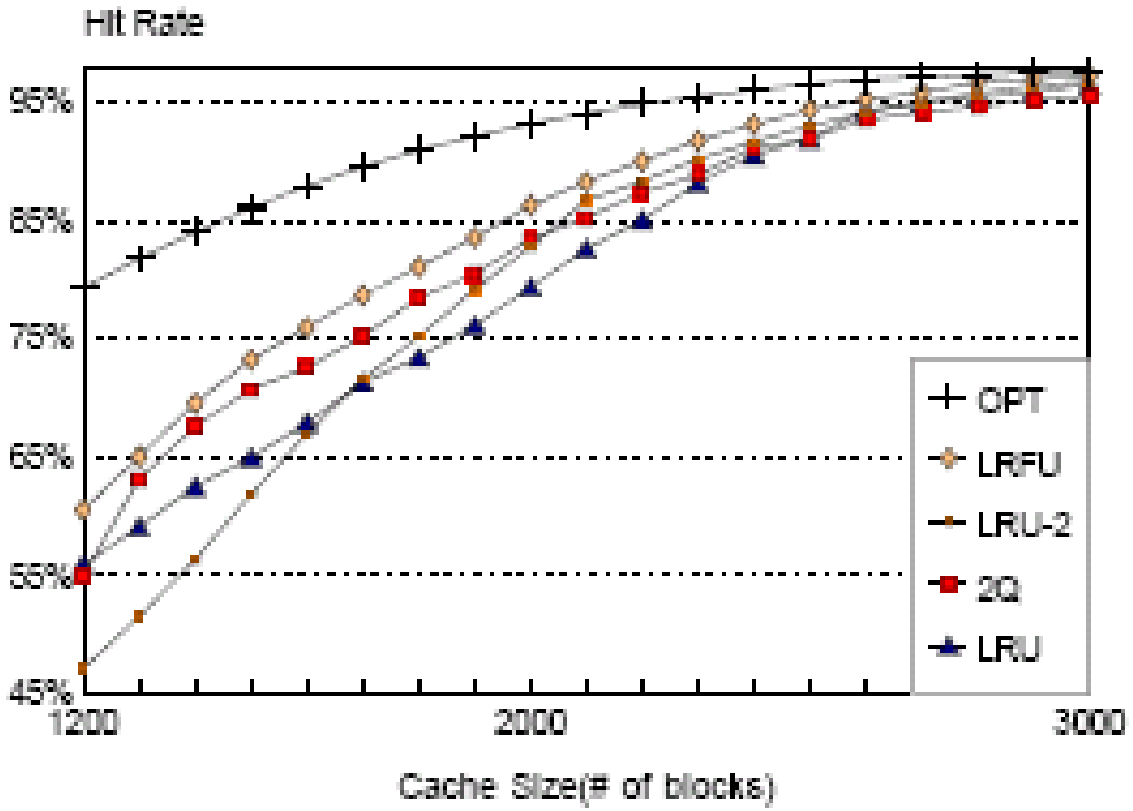


(b)

LRFU hit (in the linked list)

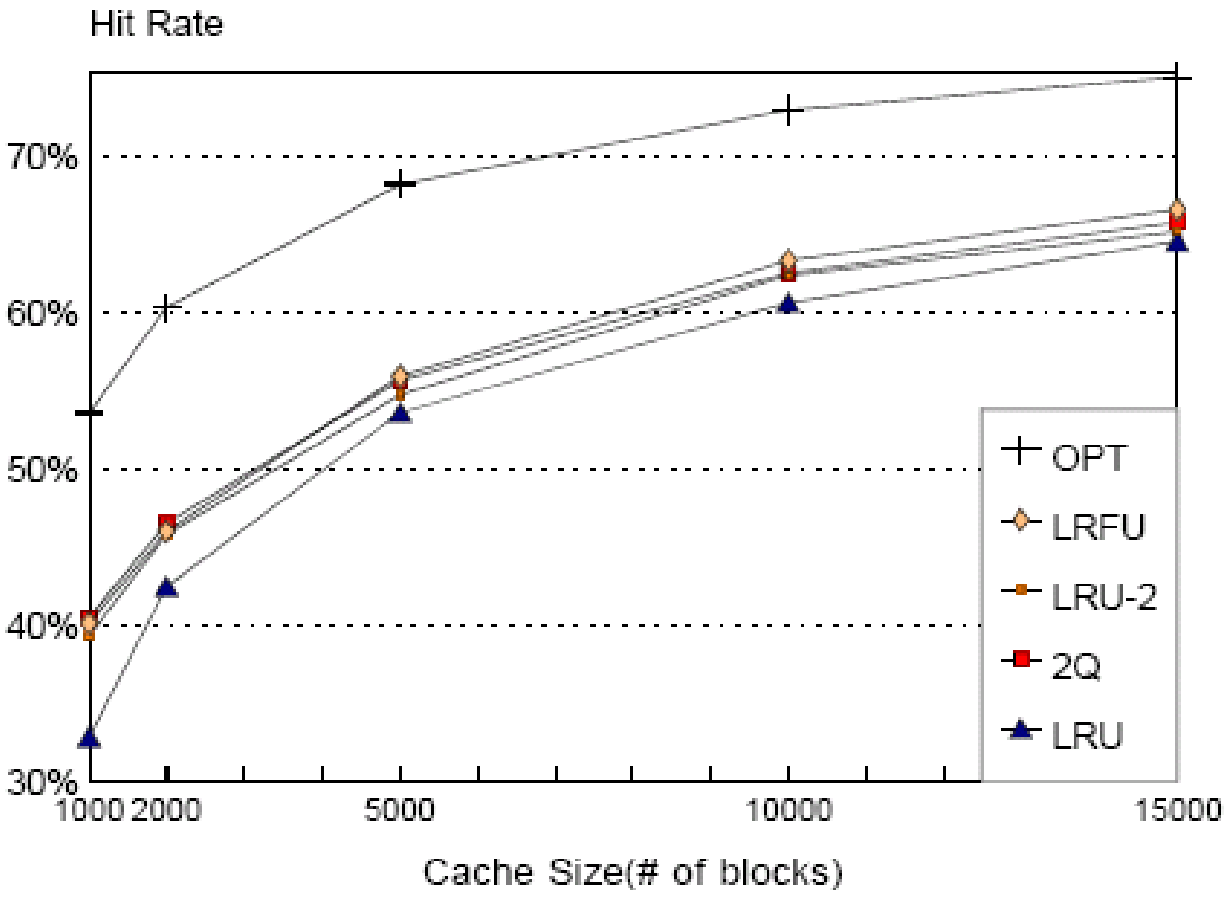


Comparison of LRFU with other



(a) Client workstation 54 in the Sprite trace

OLTP


Hit Rate
70%
60%
50%
40%
30%
1000
2000
5000
10000
15000
Cache Size(# of blocks)
OPT
LRFU
LRU-2
2Q
LRU

(b) **OLTP**

Write techniques

- Cache without writing = low performance
- Two methods
  - Write-through (synchronous write)
    - Safe system
    - no data loss
    - but low performances
  - Write-back (delayed write)
    - High performances
    - but data loss
    - on the system crash
- Write-back topics
  - Write on eject, only
  - Flushing
  - Soft updates
  - Log-approach, Log for meta data, only = JFS
  - Total log = LFS

Delayed-write modification

- Flushing
- Write on close
- Battery-backup
- Soft-updates
- Cache logging (Journaling)

Flushing

- Variation –
- scan cache at regular intervals
- and
- flush blocks that have been modified since the last scan.
- UNIX
  - Flushing at 30sec

Write-on-close

- **Variation –**
- **write-on-close**,
- **writes data back to the server**
- **when**
- the **file is closed.**
- **Best for files**
  - **that are open for long periods**
  - and
  - **frequently modified**.

Caches with Battery Backup

- **One way to make file caches more reliable**
  - **is to store them in memory**
  - **with battery backup.**

- **This would allow the contents of** the **caches** to **survive power outages**.

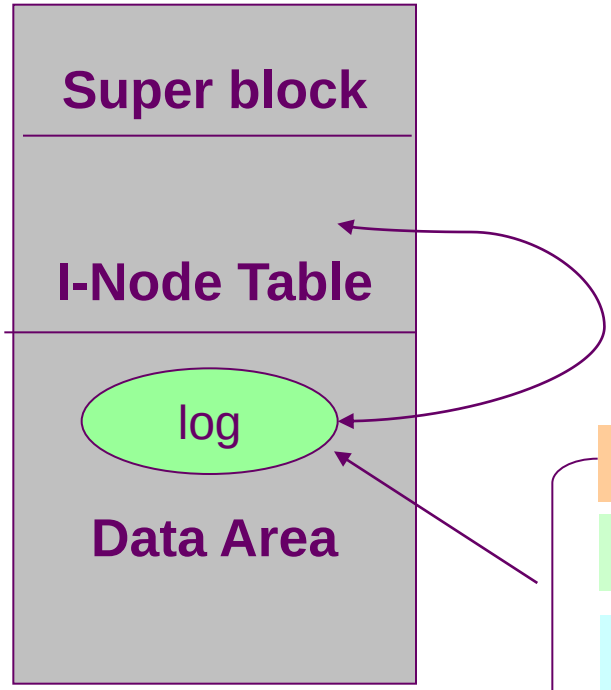
- However,
  - **the battery backup**
  - **is only the first of several steps**
  - **needed to ensure the reliability of cache data.**

- Even so,
  - **battery backup would not be sufficient by itself**:

- the **file cache would have**
  - **to be able to survive operating system crashes**
  - **as well as power outages.**

Journaling –LOG approach

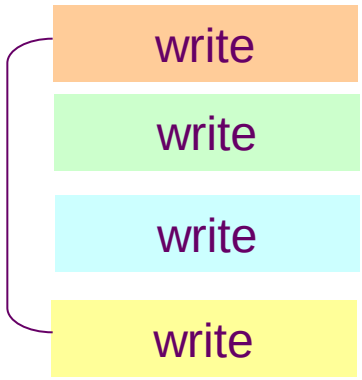
- **all metadata updates (t) → one large log update**



1. **transactions are written** to a **log**

2. **transactions** in the log are **asynchronously written to the file system**.

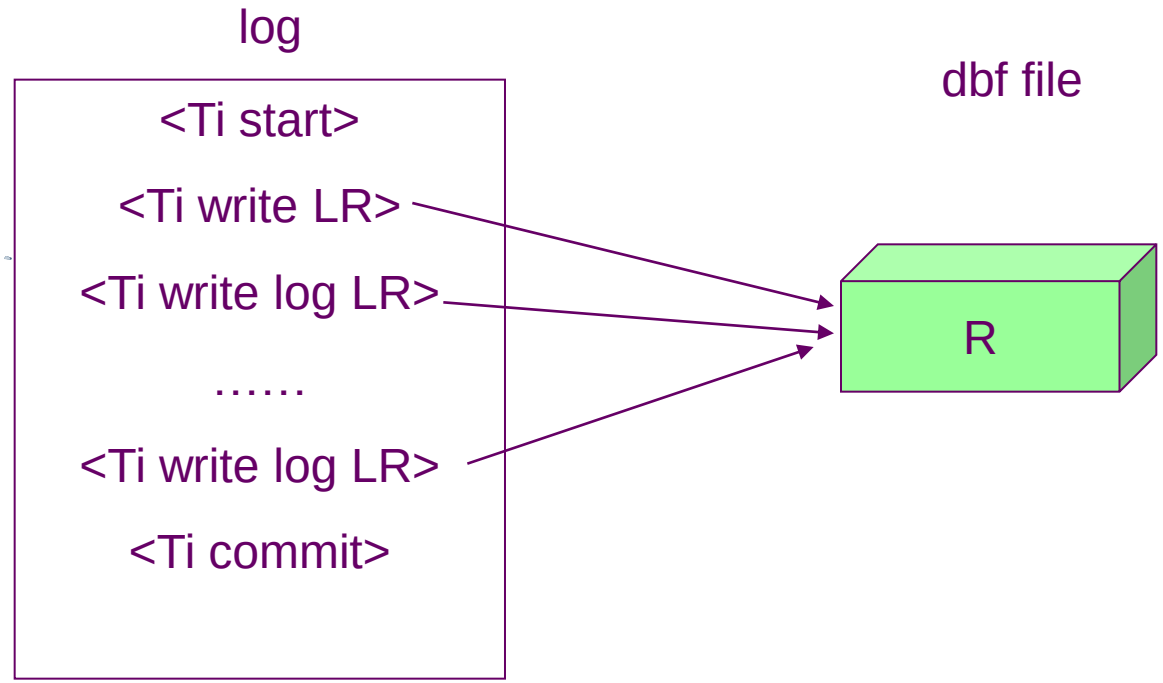
3. **When** the **file system is modified**, the **transaction is removed from the log**.



key design issues in the design of LOG

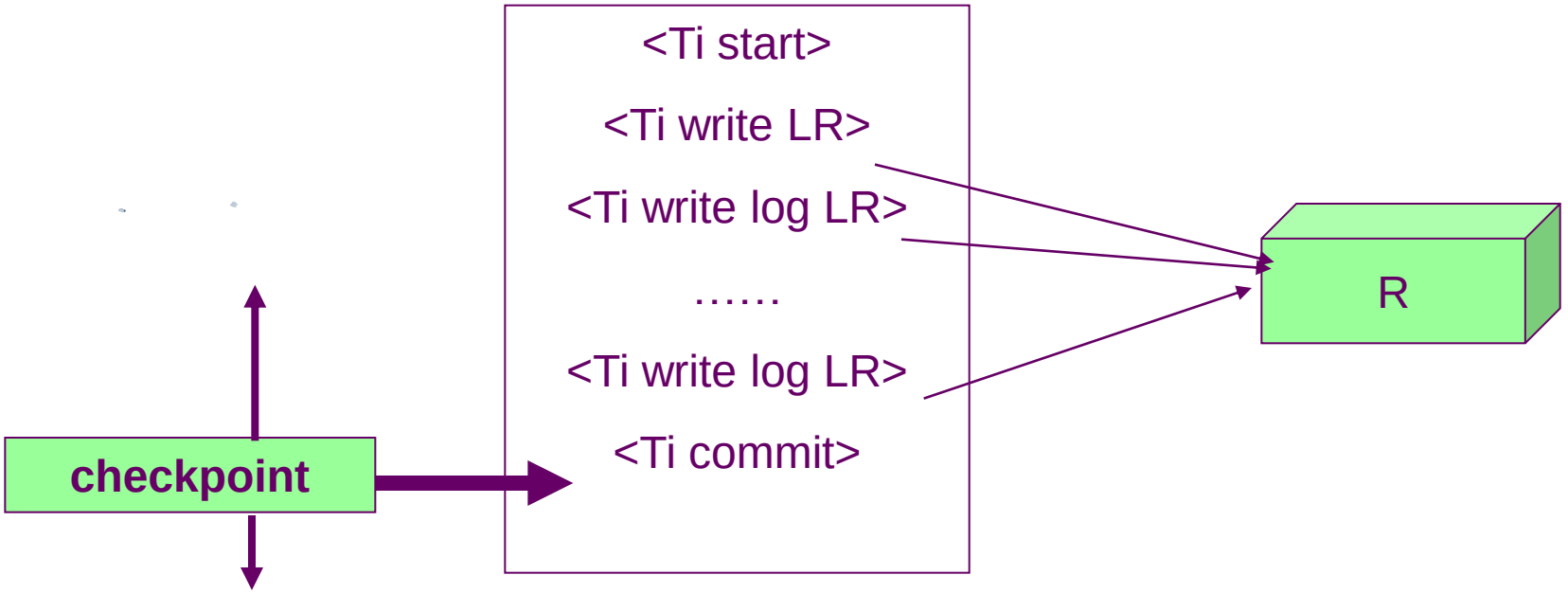
- **Location of the log**
  - **Log as file**
  - **Log as FS (separated partition on disk)**
- **Management of the log**
  - **space reclamation**
  - **check pointing**
- **Integration** or **interfacing between** the **log and** the **main FS**
- **Recovering** the **log.**

Atomic Transactions-Log



- **After system crash => Atomic transaction**
  - **undo(T) if not exist <commit T>, back to state of R before Ti**
  - **redo (T) not exist <commit T>, write last values of T to R**

Atomic Transactions-Checkpoint



- **Checkpoint = all successful Transactions**
- **after crash -> look at checkpoint**
  - **for all Tk with <Tcommit> make** a **redo(Tk)**
  - **for all Tk without <Tcommit> make** a **undo(Tk)**

Time to recovery

- **Without journaling**
- **LONG**
  - **All metadata structures must be check**
  - **FCB, free lists**

- **With journaling**
- **FAST**
  - **Transactions after check point, only**

Journaling

- In this section, we describe two different implementations of journaling applied to the fast file system.
- The first implementation (LFFS-file) maintains a circular log in a file on the FFS, in which it records journaling information. The buffer manager enforces a write-ahead logging protocol to ensure proper synchronization between normal file data and the log.

- The second implementation (LFFS-wafs) records log records in a separate stand-alone service, a writeahead file system (WAFS). This stand-alone logging service can be used by other clients, such as a database management system, as was done in the Quicksilver operating system.
- LFFS-wafs implements its log in an auxiliary file system that is associated with the FFS. The logging file system (WAFS, for Write-Ahead File System) is a simple, free-standing file system that supports a limited number of operations: it can be mounted and unmounted, it can append data, and it can return data by sequential or keyed reads.

Soft Updates

- Soft Updates attacks the meta-data update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/Os.
- In general, a Soft Updates system must maintain *dependency information*, or detailed information about the relationship between cached pieces of data.
- For example, when a file is created, the system much ensure that the new inode reaches disk before the directory that references it does. In order to delay writes, Soft Updates must maintain information that indicates that the directory data block is dependent upon the new inode and therefore, the directory data block cannot be written to disk until after the inode has been written to disk. In practice, this dependency information is maintained on a per-pointer basis instead of a per-block basis in order to reduce the number of cyclic dependencies.

Micro benchmark results

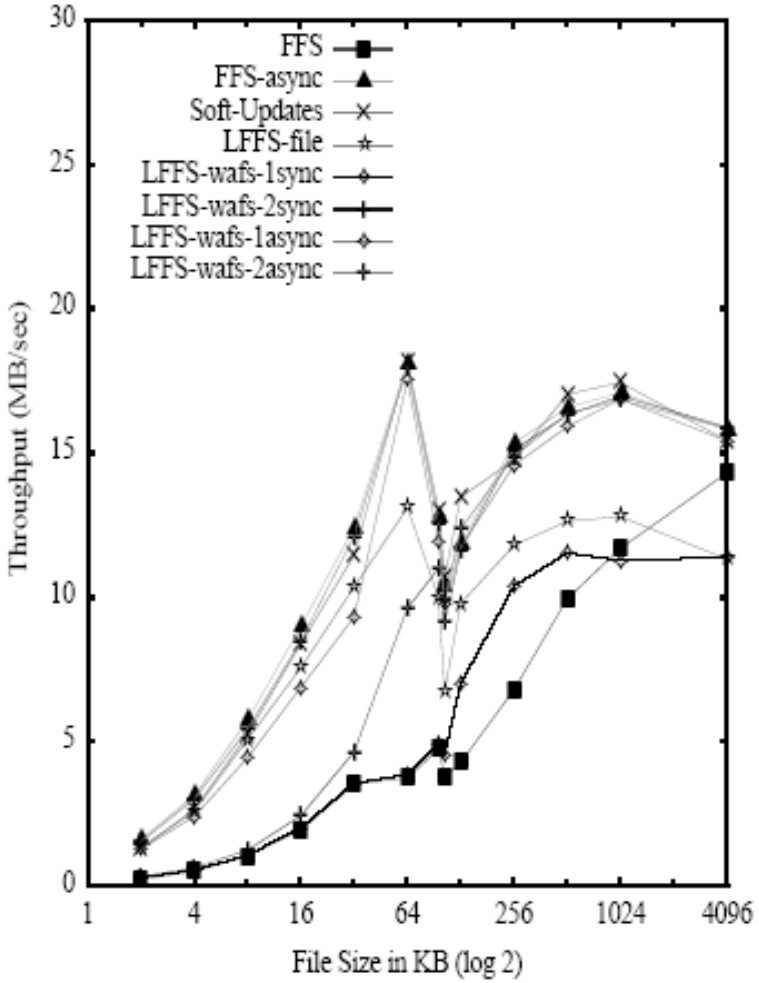


Figure 1. Create Performance as a Function of File Size.

Postmark results

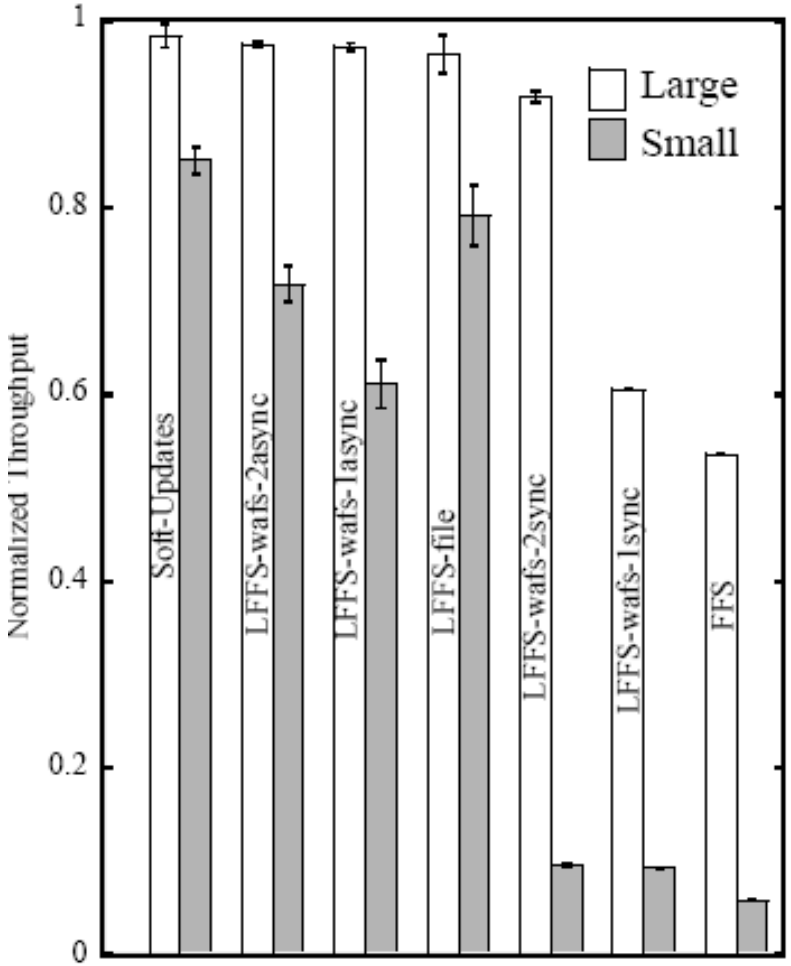


Figure 5. PostMark Results. The results are the averages of five runs; standard deviations are shown with error bars.