Chapter 13
File and Buffer Management

Mijin An
meeeeejin@gmail.com

VLDB Lab.

Introduction: File Manager

- One of the most important **resource managers in a transaction processing system**

- **Purpose** of the file manager:
  - To **organize data kept on durable external storage** such that it can **easily be processed** by other resource managers and by application programs

Abstractions Provided by the File Manager

- **Device independence**
  - To turn the large variety of external storage devices into simple abstract data types
  - To support certain access operations, such as read and write on records
- **Allocation independence**
  - To do its own space management for storing the data objects presented by the client
- **Address independence**
  - To provides mechanisms for associative access (e.g., SELECT)

External Storage vs. Main Memory (1)

- **Capacity**
  - Main memory is usually limited in its addressing capabilities to a size that is smaller than what large databases need
  - External storage provides the large storage capacities

- **Economics**
  - The cost per byte of main memory is considerably higher than for disks
  - External storage holds large volumes of data at reasonable cost

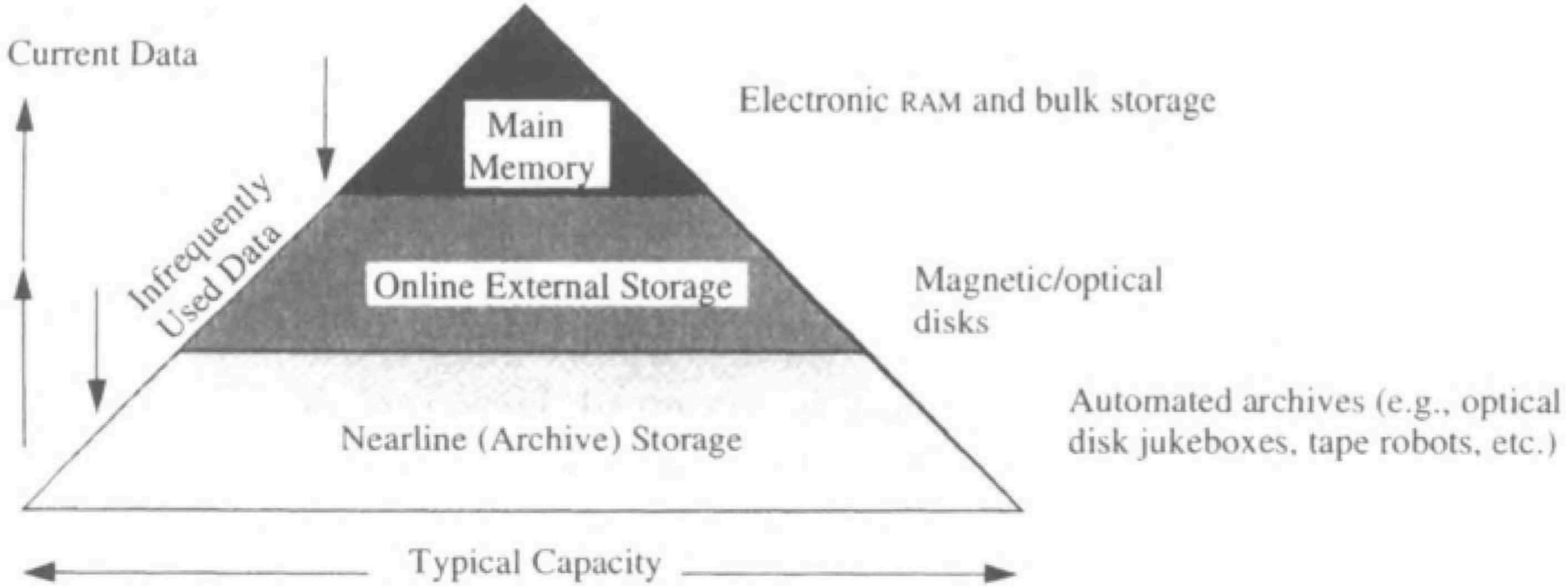
- **Durability**
  - Main memory is volatile
  - External storages are inherently durable and proper to store persistent objects

External Storage vs. Main Memory (2)

- **Speed**
  - External storage devices are slower than main memory
  - Thus, it is more costly, both in terms of latency and path-length, to get data from external storage to the CPU than to load data from main memory

- **Functionality**
  - Data cannot be processed directly on external storage
  - For data to be processed, they must first be transferred into main memory

A Storage Hierarchy

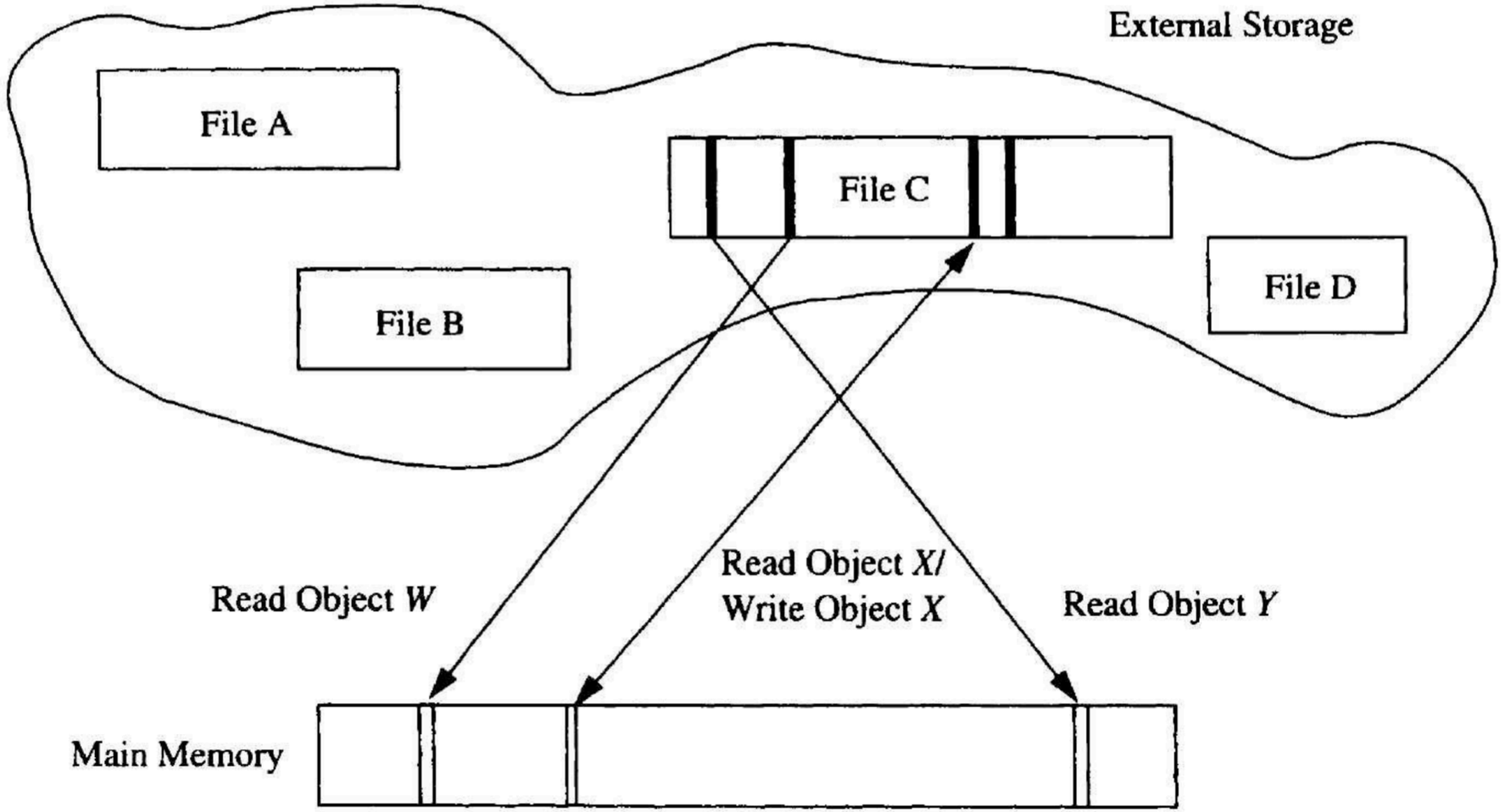


The Abstract View of External Storage

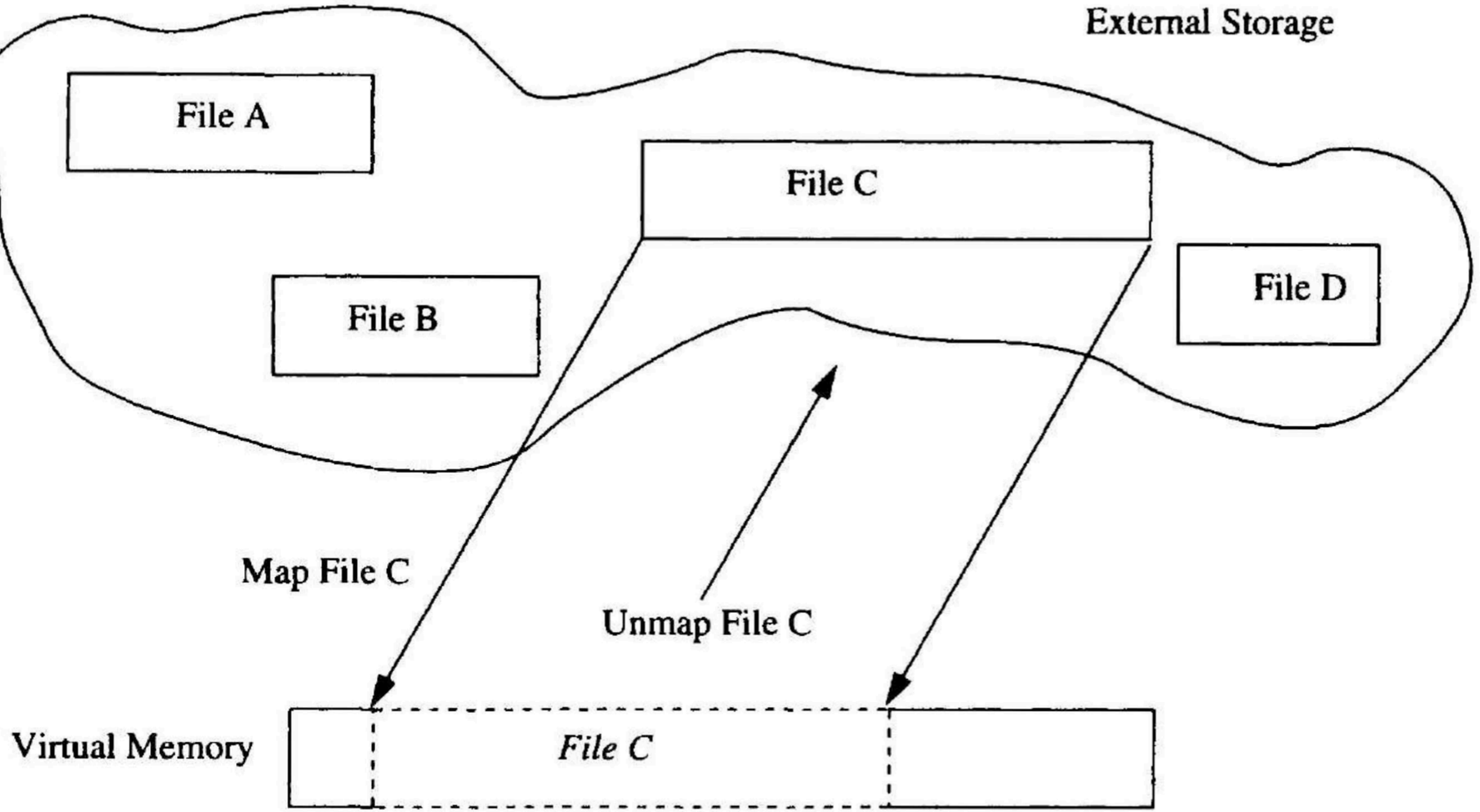
- **External storage** is presented to the applications and resource managers **not as a variety of devices, but as files** with permanent and unique names

- Given this abstract view of external storage as a collection of files, there are three structurally different methods to make them accessible in main memory:
  1. Read/write mapping
  2. Memory mapped files
  3. Single-level storage

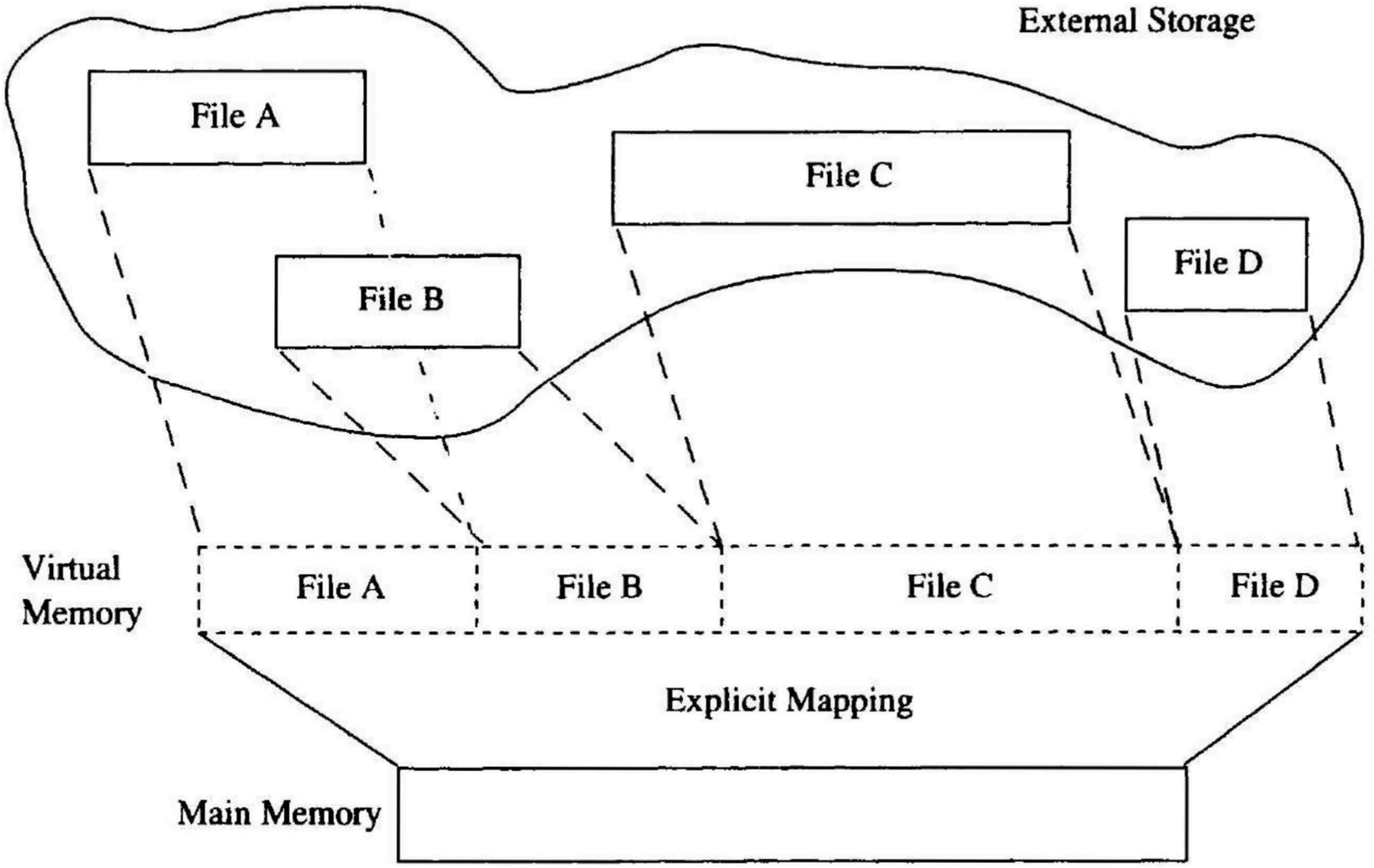
1. Read/Write Mapping



2. Memory-Mapped Files



3. Single-Level Storage

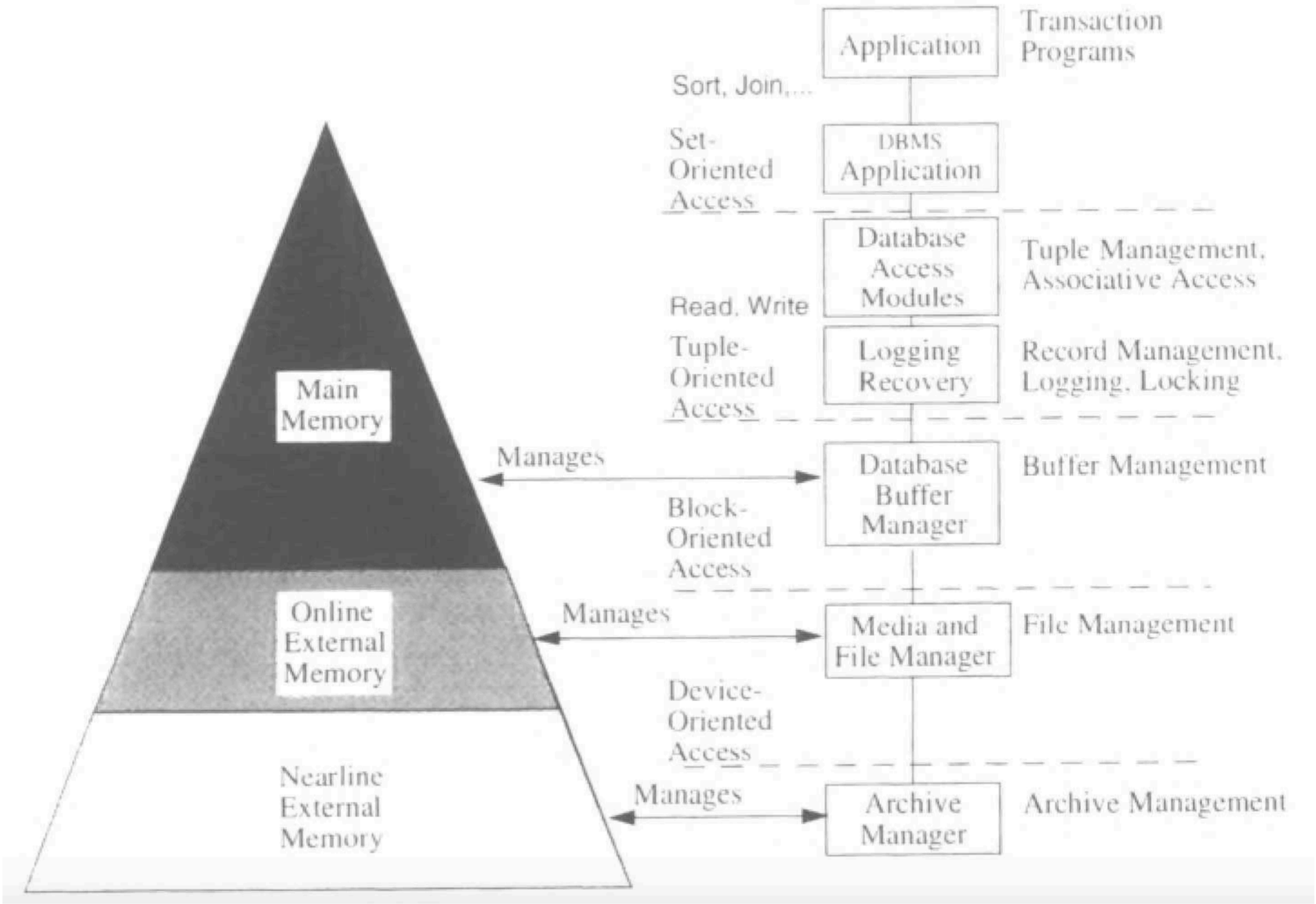


Locality and Caching

- A file manager **does not know** which data are active and which are not

- The data are distinguished by observing the actual reference pattern and by interpreting it under the principle of **locality**:
  - **Locality of active data**:
    - Data that have recently been reference will very likely be referenced again
  - **Locality of passive data**:
    - Data that have not been referenced recently will most likely not be referenced in the future

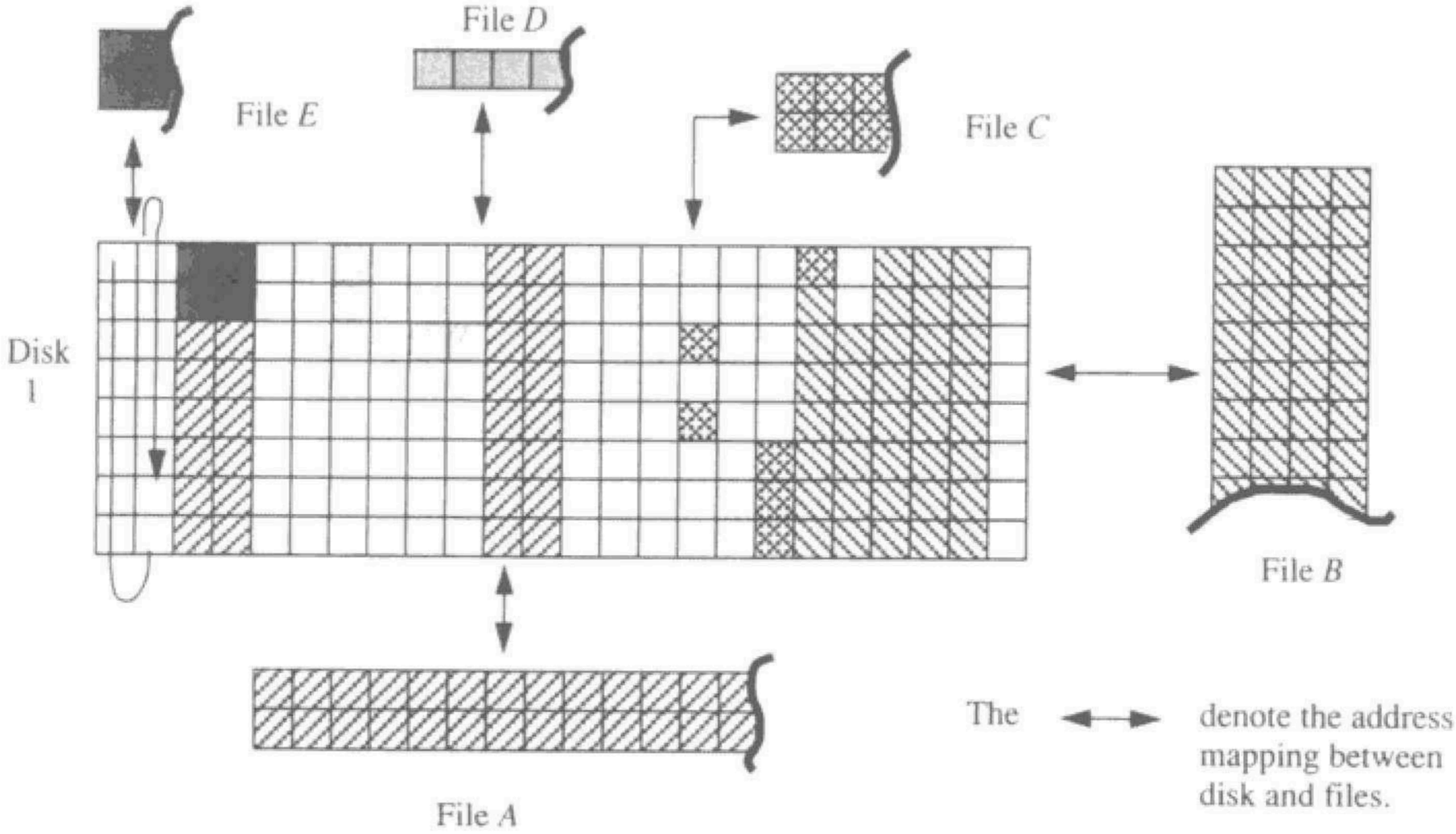
Components of the Physical Storage Organization in a Transactional Database System



Operations of the Basic File System

```
STATUS create(filename, allocparmp)
STATUS delete(filename)
STATUS open(filename, ACCESSMODE, FILEID)
STATUS close(FILEID)
STATUS extend(FILEID, allocparmp)
STATUS read(FILEID, BLOCKID, BLOCKP)
STATUS readc(FILEID, BLOCKID, blockcount, BLOCKP)
STATUS write(FILEID, BLOCKID, BLOCKP)
STATUS writec(FILEID, BLOCKID, blockcount, BLOCKP)
```

Mapping Files to Disk



Issues in Managing Disk Space

- **Initial allocation**
  - When a file is created how many contiguous slots should be allocated to it?
- **Incremental expansion**
  - If an existing file grows beyond the number of slots currently allocated, how many additional contiguous blocks should be assigned to that file?
- **Reorganization**
  - When and how should the free space on the disk be reorganized?

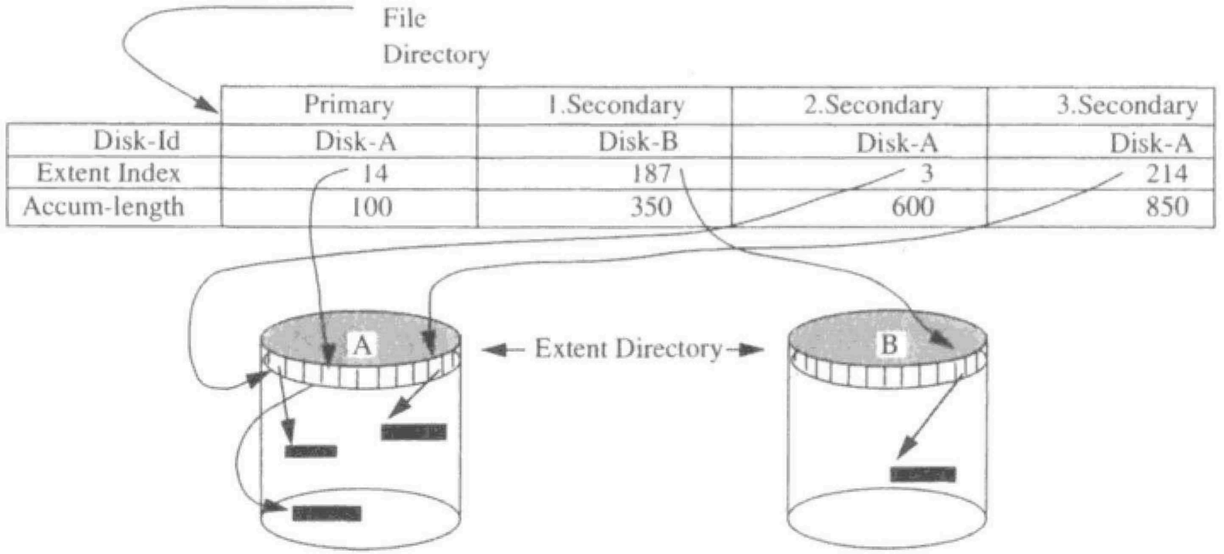
Static and Contiguous Allocation

- At file creations time, the total number of blocks is **allocated at one time in contiguous slots**
- Easy address translation
  - Block number of 0: S_b
  - Block number of k: S_b+k
- The only mapping information needed is **the complete address of the first slot** and **the total number of blocks** in the file
- But, such files **cannot grow** and **cannot shrink** either

Extent-based Allocation

- It is impossible to allocate the maximum of all future space requirements at file-creation time

- Therefore, **do multiple contiguous allocations to extend/remove space dynamically**
  - The file gets space in chunks of contiguous slots called extents



Single-Slot Allocation

- Extent-based allocation with a fixed extent size of one slot
  - **Each block of the file gets its own slot, irrespective of where the other are**

- For very large files, **random** access performance gets **poor** because of the skew in the addressing structure

- Two conclusions drawn from this method:
  1. Space allocation must use a technique that allows for **very fast address translations from block numbers into disk addresses**
  2. Space allocation should be done in **large units of physically adjacent disk blocks** in order to support **fast sequential access**

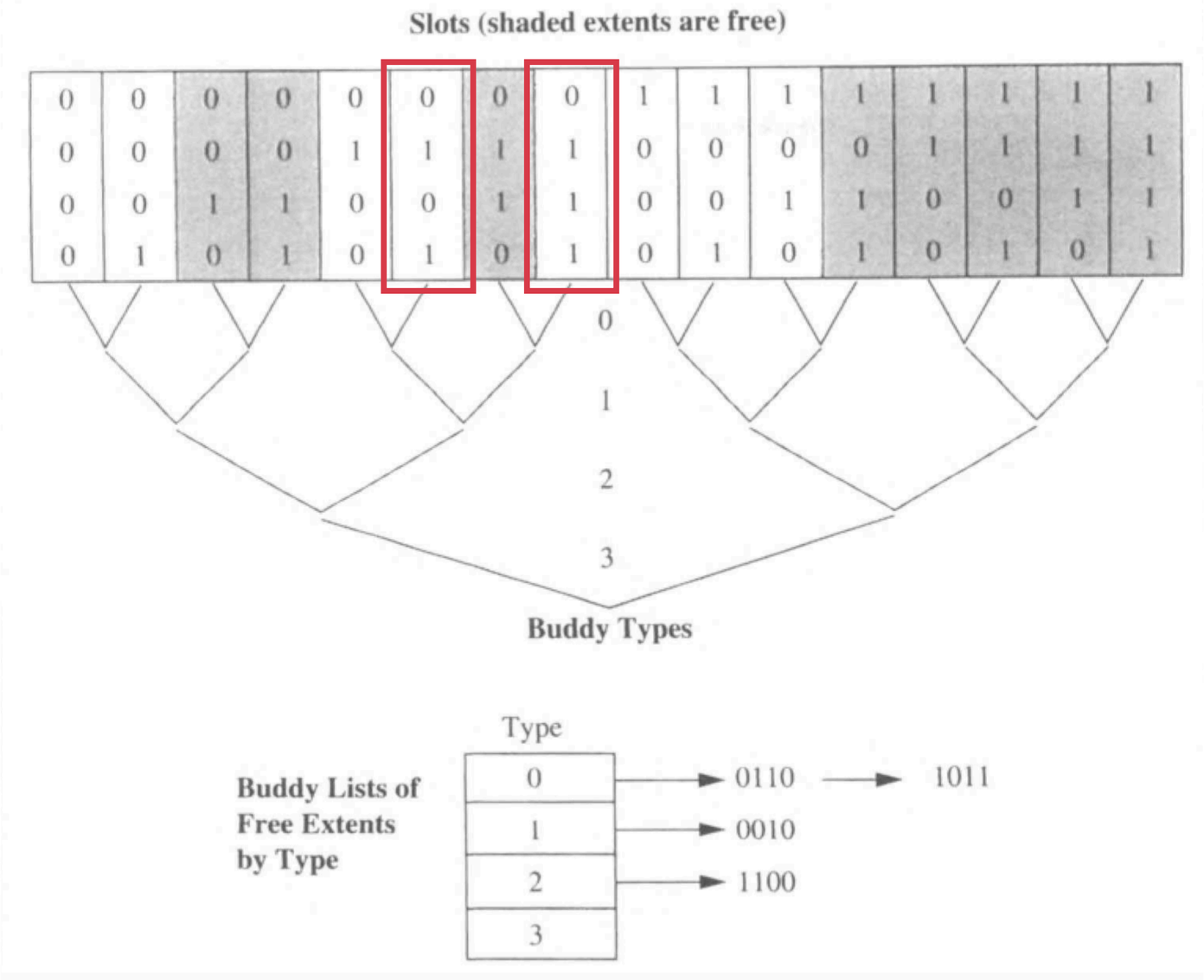
Buddy Systems

- This method allows only certain **predefined extent sizes**
  - Assumption: the number of blocks per extent always is a power of 2
  - An extent is said to be `of type i` if it contains 2^i contiguous slots

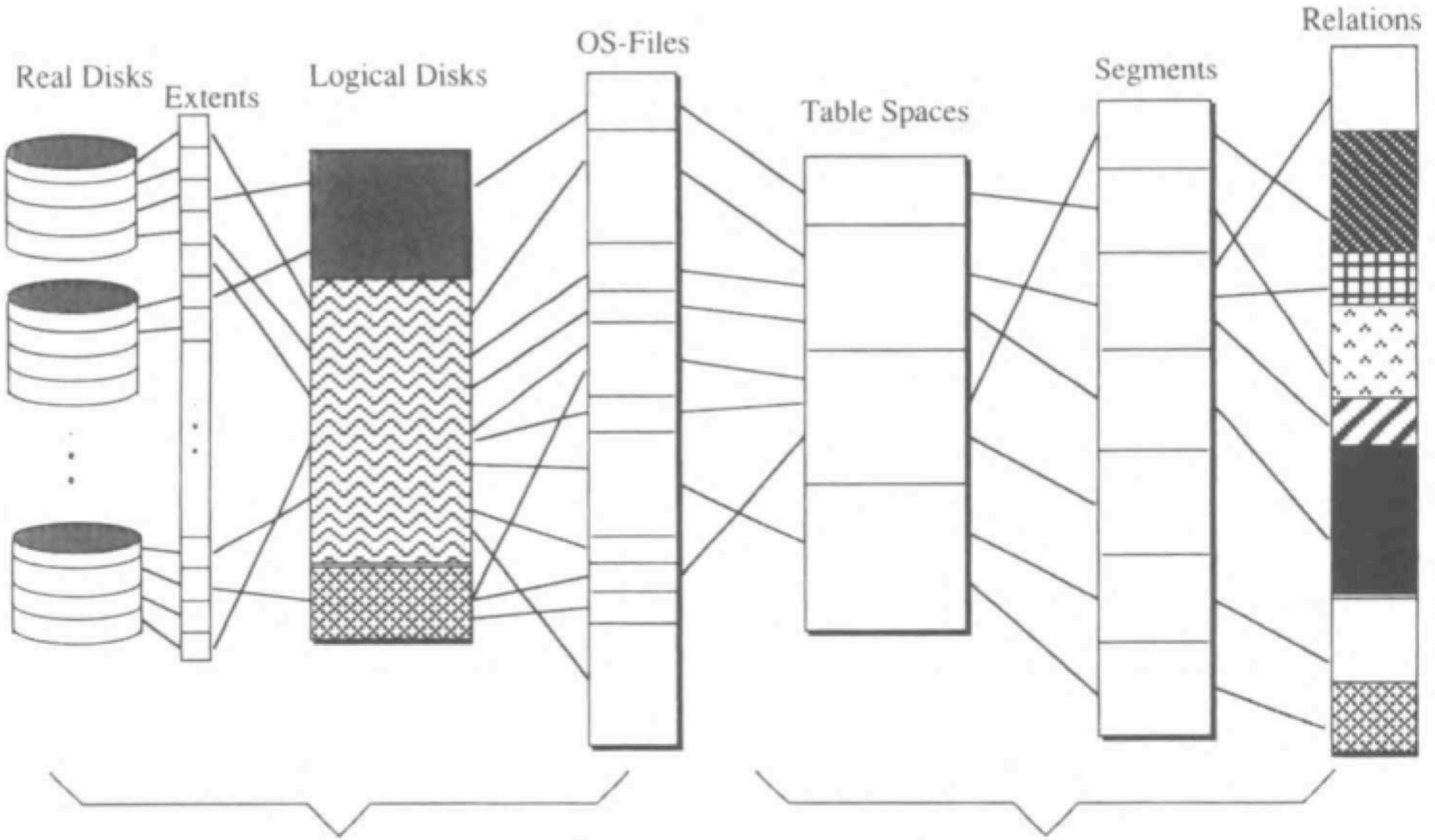
- Free extents of the same size are linked together in a **free list**
  - If no free extent is available, the free list for the next bigger type is inspected

- If an extent is freed, this triggers a check of **whether its buddy is also free**
  - Whenever an extent and its buddy are both free, both are made on extent of the next-larger type
  - The merging is repeated until no free buddy is found

Buddy Systems



Multiple Layers of Mapping of Relations to Disks



(AIX)

(Oracle)

Buffer Management

- The buffer manager's main **purposes**:
  1. To make the **pages addressable in main memory**
  2. To **coordinate the writing of pages to disk** with the log manager and the recovery manager
  3. To **minimize the number of actual disk accesses** for doing above things

Design Options for the Buffer Manager

- **Buffer per file**:
  - Each file has its own private buffer pool
- **Buffer per page size**:
  - In systems with different page sizes, there is at least one buffer for each page size
- **Buffer per file type**:
  - There are files like indices which are accessed in a different way from other files
  - Some systems dedicate buffers to files depending on the access pattern and try to manage each of them in an optimal way for the respective file organization

C Code for Page Identifier and Buffer

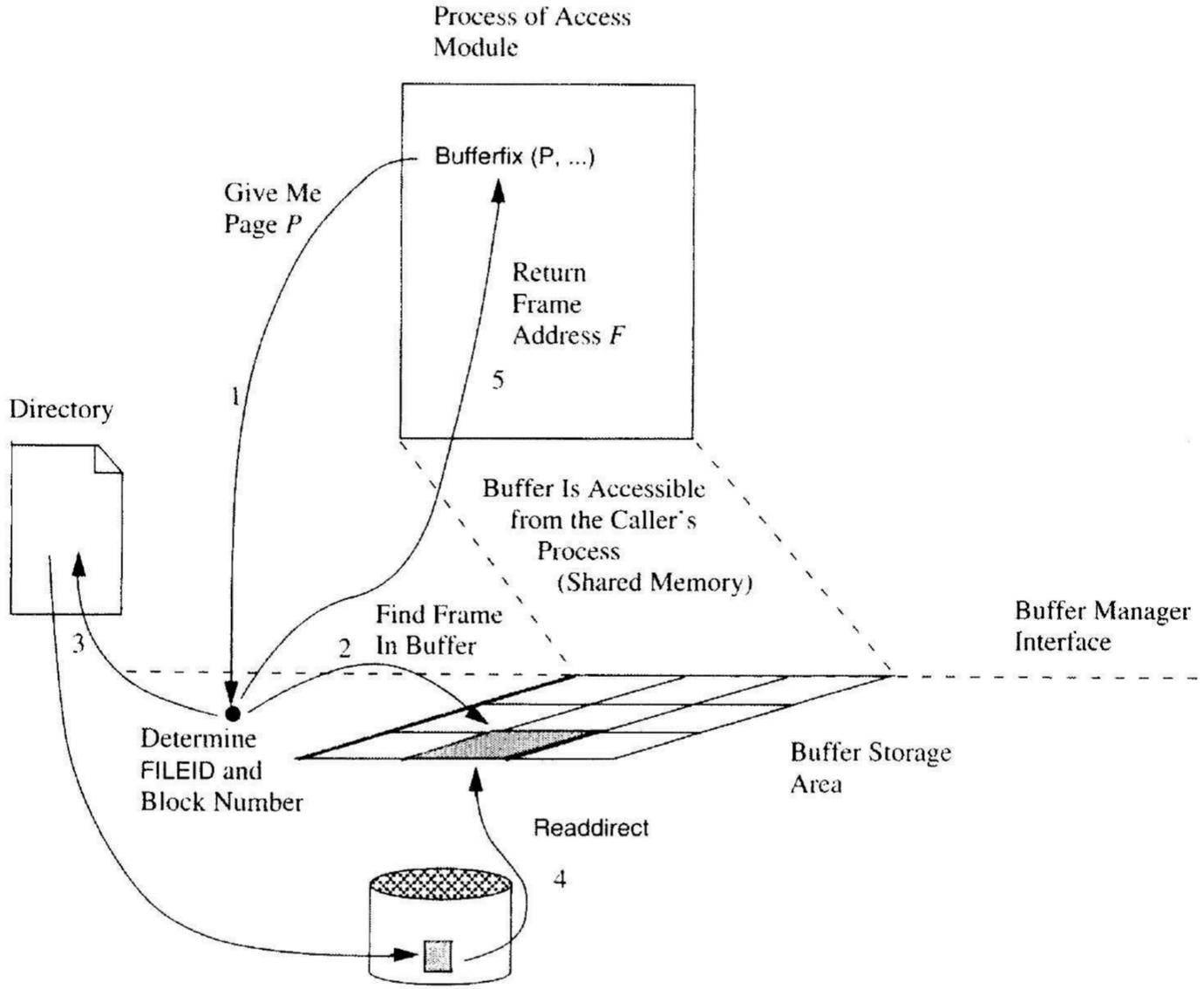
- A page identifier:

```
typedef struct {
   FILENO       fileno;          /* file to which the page belongs */
   unsigned int pageno;          /* page number in the file */
} PAGEID, *PAGEIDP;
```

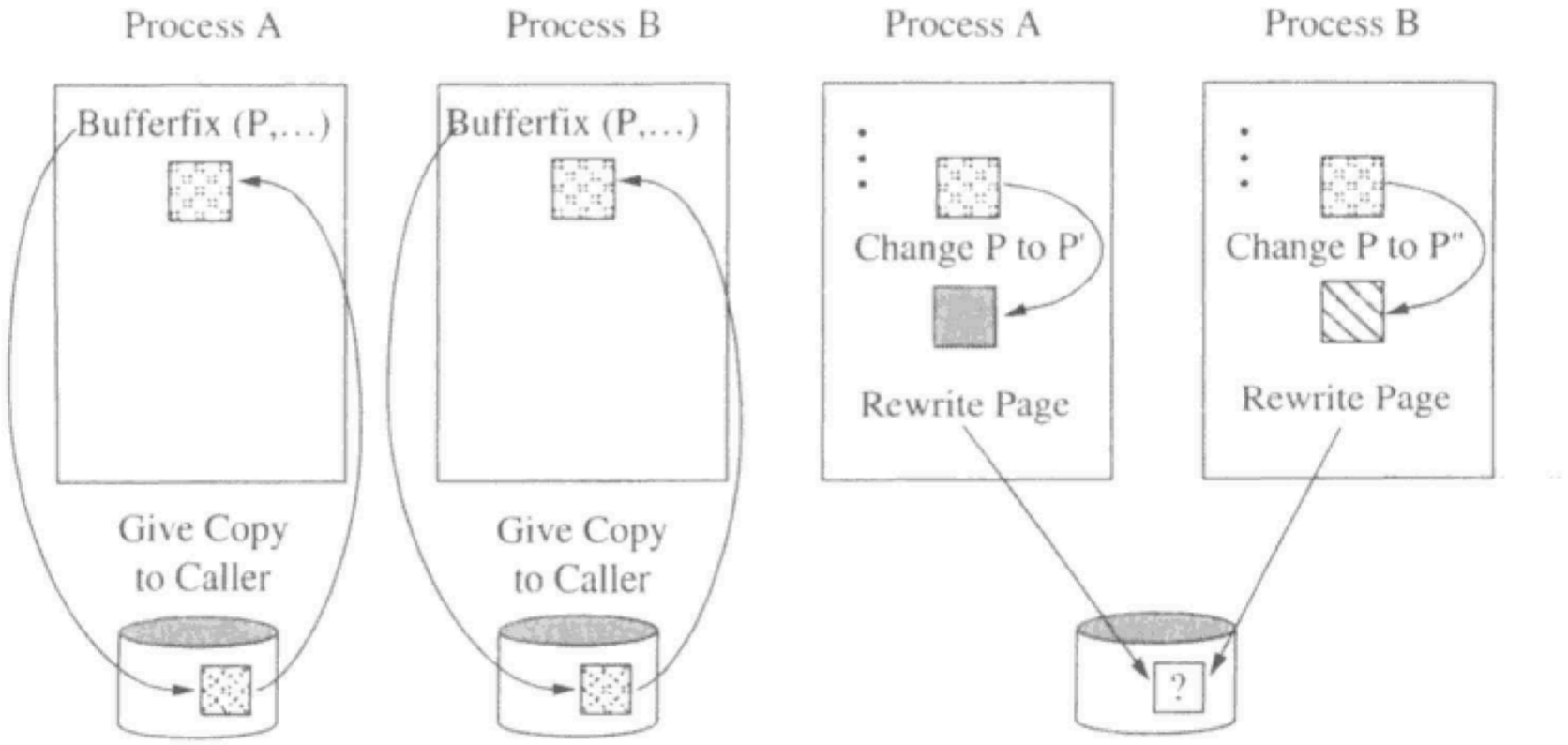
- A database buffer:

```
struct {
   PAGE_HEADER page_header;      /* header of a page in a file */
   char        page_body[];      /* payload of the page */
} bufferpool[buffersize];        /* fixed size is assumed for the buffer */
```

Work Flow of Buffer Manager



Lost Update Anomaly



a) Access module in process A requests access to page P; gets private copy.

b) Access module in process B requests access to page P; gets private copy.

c) Both processes try to rewrite an updated version of the page, but the versions are different. Only the version written last will be on disk; this is the "lost update" anomaly.

What the Buffer Manager Does for Synchronization

- **Sharing**
  - Pages are made addressable to all processes that run the database code
- **Addressability**
  - Each access module is returned an address in the buffer pool
- **Semaphore protection**
  - Each requestor gets the address of a semaphore protecting the page
- **Durable storage**
  - The access modules inform the buffer manager if their page access has resulted in an update of the page

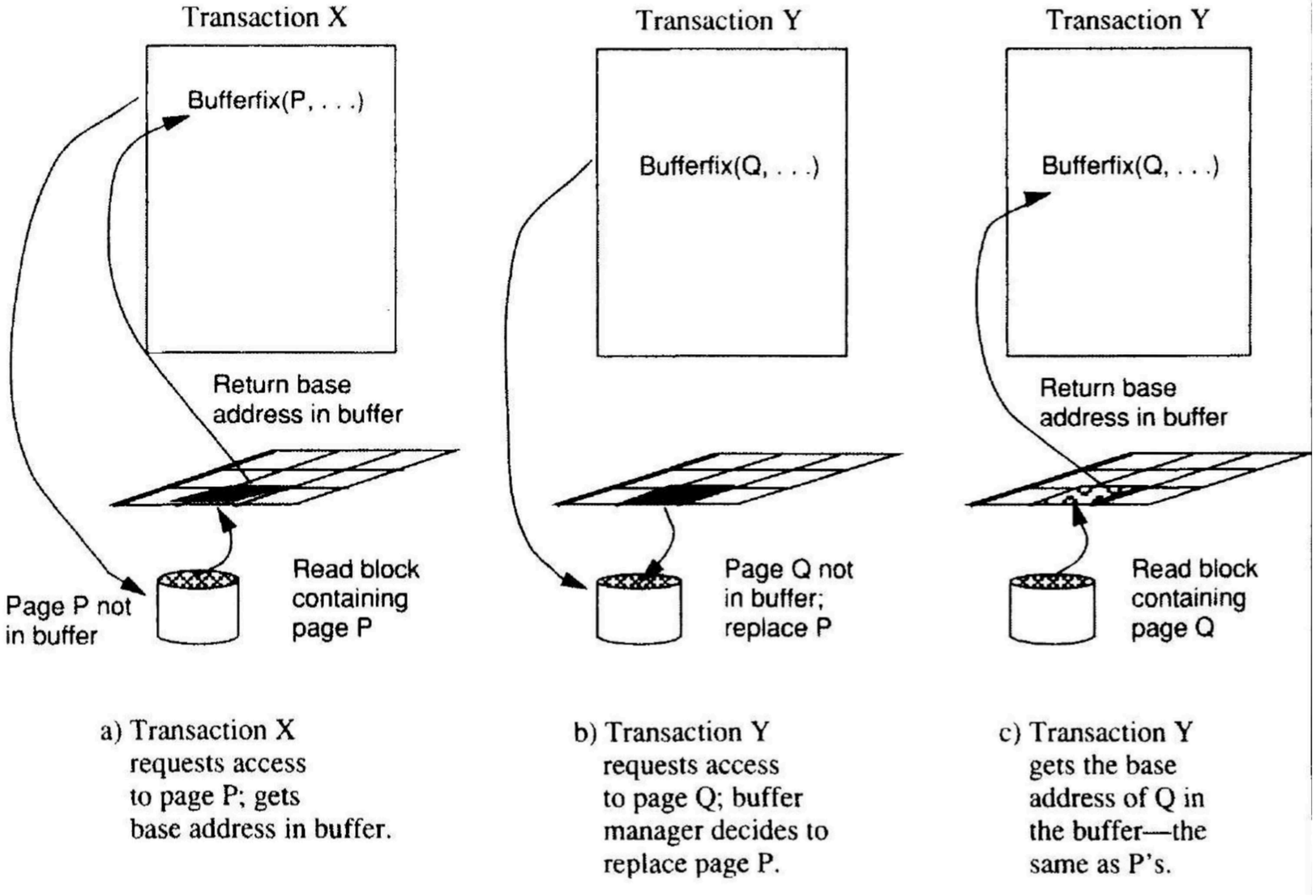
Buffer Access Control Block

- A buffer access control block:

```
typedef struct {
   PAGEID      fileno;              /* id of page in file */
   PAGEPTR     pageaddr;            /* base address of page in buffer pool */
   int         index;               /* record within page (used by caller) */
   semaphore * pagesem;             /* pointer to the semaphore for the page */
   boolean     modified;            /* flag says caller modified page */
   boolean     invalid;             /* flag says caller destroyed page */
} BUFFER_ACC_CB, *BUFFER_ACC_CBP; /* control block for buffer access */
```

  - Control block tells the caller **what it has to know for accessing the requested page in the buffer**

The Need for Fix and Unfix



a) Transaction X requests access to page P; gets base address in buffer.

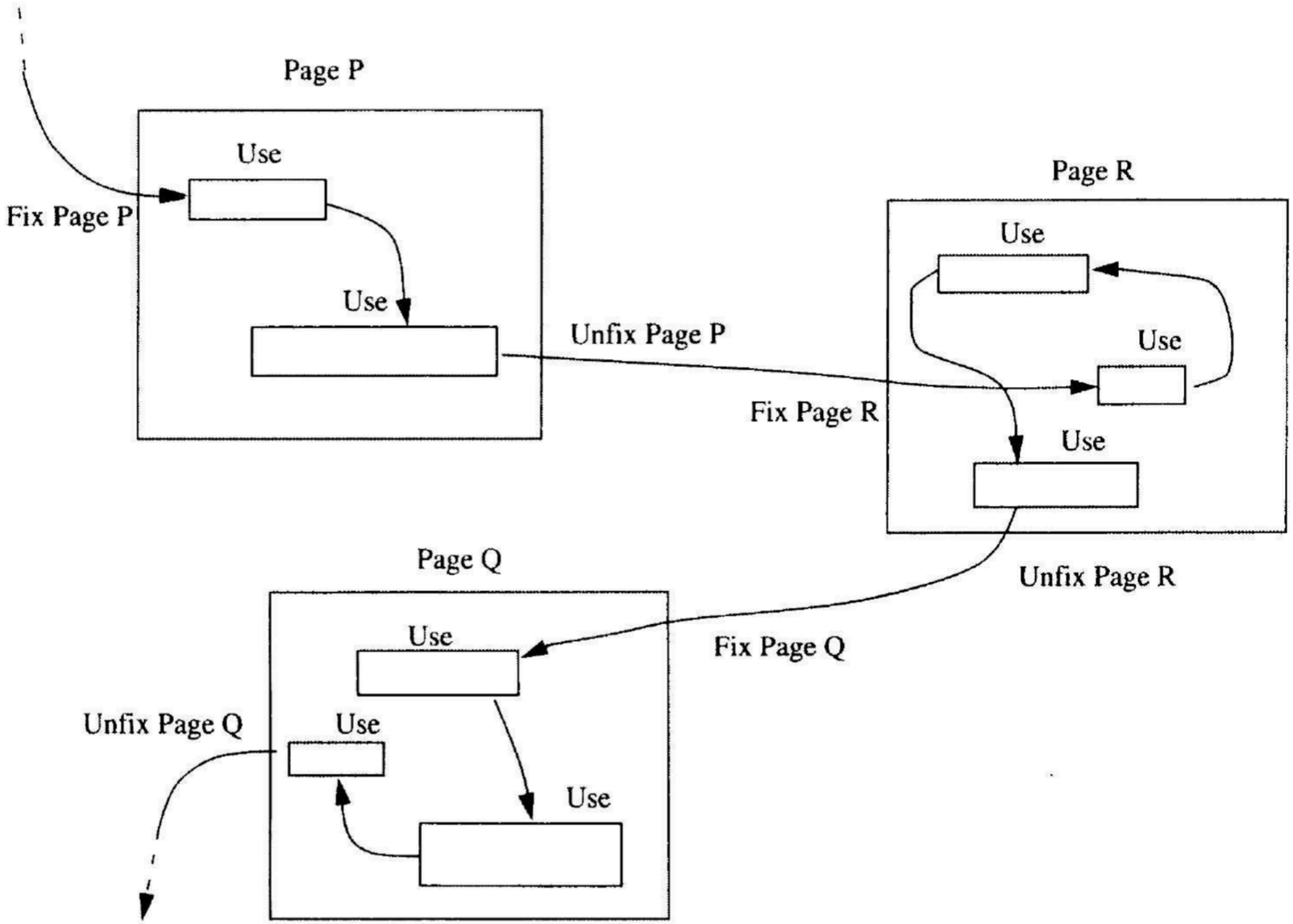
b) Transaction Y requests access to page Q; buffer manager decides to replace page P.

c) Transaction Y gets the base address of Q in the buffer—the same as P's.

The FIX-USE-UNFIX Protocol

- **FIX**
  - The client request access to a page using the `bufferfix` interface
- **USE**
  - The client uses the page with the guarantee that the pointer to the frame containing the page will remain valid
- **UNFIX**
  - The client explicitly waives further usage of the frame pointer
  - It tells the buffer manager that it no longer wants to use that page

The Strict FIX-USE-UNFIX Protocol



Data Structures Maintained by the Buffer Manager

- The key data structure is an array of frame; **buffer pool**
- A **hash function** is used for associative access to the buffer pool via `PAGEID`
- Each **buffer control block** contains all the information about a page in the buffer pool

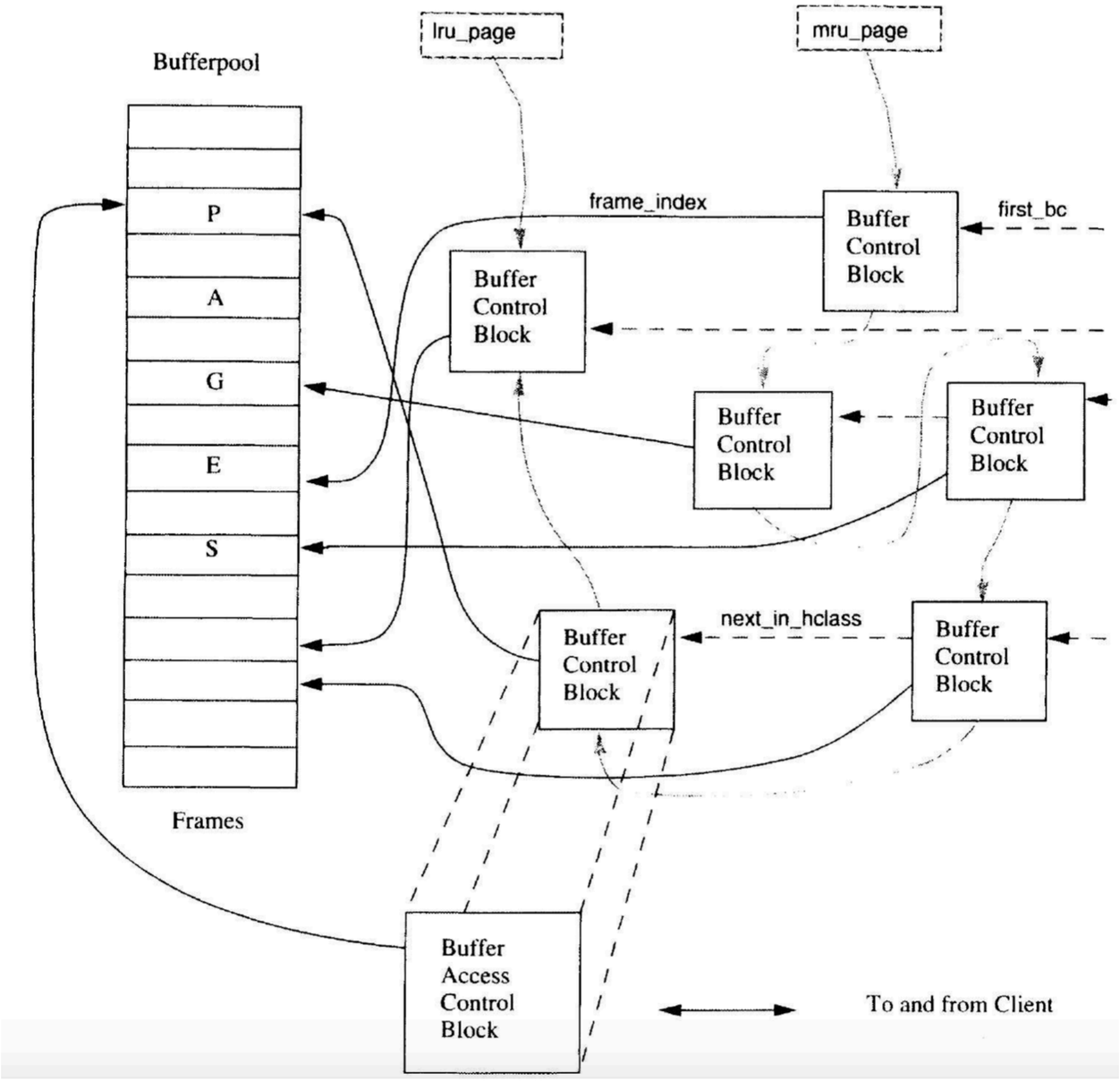
```
/* The buffer pool array is statically allocated.
   Each entry contains storage space for one page. */

struct {
   PAGE_HEADER page_header;   /* header of a page in a file */
   char        page_body[];   /* payload of the page */
} bufferpool[buffersize];     /* fixed size is assumed for the buffer */

struct {
   Uint free_index;           /* index of a free frame */
} free_frames[buffersize];    /* table of free frame indexes */

Ulong     no_free_frames;     /* current number of free frames */
semaphore free_frame_sem;     /* semaphore to protect the free_frames list */
```

Data Structures Maintained by the Buffer Manager

- **Buffer control block**: The administrative data structure to access the buffer pool and to keep track of how the pages have been used

```
typedef struct {
   PAGEID     pageid;          /* no. of page in buffer */
   FILEID     *in_file;        /* handle of file where page is stored */
   Uint       frame_index;     /* buffer pool index where page now stored */
   semaphore  pagesem;         /* semaphore to protect the page */
   boolean    modified;        /* TRUE if page is modified, FASLE else */
   int        fixcount;        /* number of page fixes */
   LSN        forminlsn;       /* LSN of first fix for update */
   BUFFER_CBP prev_in_LRU;     /* previous page in LRU – chain */
   BUFFER_CBP next_in_LRU;     /* new page in LRU – chain */
   BUFFER_CBP next_in_hclass; /* hash overflow chain forward pointer */
} BUFFER_CB, *BUFFER_CBP;   /* table of free frame indexes */
```

Structure of the Buffer Manager



Initialization of the Buffer Manager

1. The buffer pool itself must be made consistent by making all frames and semaphores as free
2. The control blocks must be made consistent with the data structure they refer to
3. The linked lists chaining together control blocks must be set up

- There are three groups of control blocks that need to be initialized by the buffer manager:
  - The `buffer_hash` table together with buffer control blocks
  - Buffer access control blocks
  - The list of open files

The `get_frame()` Routine (1)

- The routine `get_frame()` **returns a pointer to a buffer control block that is empty**

- How it works:
  1. Check the list of unused frames
  2. If there are no free frames, one must be freed by replacing another page; **replacement victim**
  3. Once the page to be replaced had been determined, the buffer manager checks whether or not that page has been updated while in buffer (`modified`)
  4. If so, the page must be **written back** to the file
     - Before that, the corresponding log entries must be on durable storage (`log_flush`)

The `get_frame()` Routine (2)

- If the page suggested by the LRU chain has another semaphore, we do **not** want to **wait** for it to become available
  - Rather, we consider the next-younger page from the LRU chain

- The routine doest **not hold a semaphore while doing I/O**

Logging/Recovery from the Buffer's Perspective

- The buffer manager must make sure not to violate the ACID properties
- This requires synchronization with both the log manager and the transaction manager

State of Transaction *TA*	Page *A*	Page *B*	State of the Database
Aborted	In Buffer (Old)	In Buffer (Old)	Consistent (Old)
Aborted	In Buffer (Old)	On Disk (New)	*Inconsistent*
Aborted	On Disk (New)	In Buffer (Old)	*Inconsistent*
Aborted	On Disk (New)	On Disk (New)	*Inconsistent*
Committed	In Buffer (Old)	In Buffer (Old)	*Inconsistent*
Committed	In Buffer (Old)	On Disk (New)	*Inconsistent*
Committed	On Disk (New)	In Buffer (Old)	*Inconsistent*
Committed	On Disk (New)	On Disk (New)	Consistent (New)

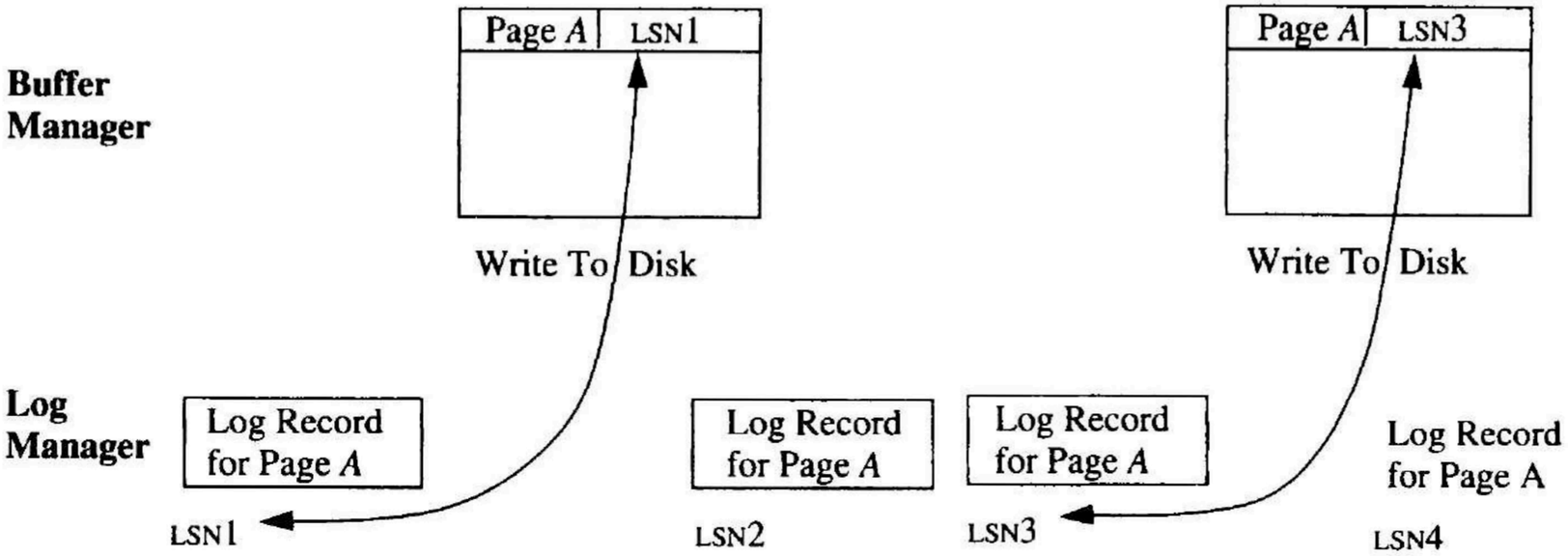
Recovery Operations Applied to the Wrong States of a Page

- If the recovery operation is not synchronized with the state of the page to be recovered, such operations generally result in an inconsistent database

State of Transaction *TA*	State of Page *A* In Database	Result of Recovery Using Operation Log
Aborted	Old	Wrong Tuple Might Be Deleted
Aborted	**New**	Inverse Operation Succeeds
Committed	Old	Operation Succeeds
Committed	**New**	Duplicate Of Tuple Is Inserted

The Log and Page LSNs

- In order to guarantee the correct execution of recovery actions during restart, the buffer manager and the log manager have to exchange information at run time
  - The **Log Sequence Number (LSN)**
- The state of a page on disk is recorded by storing the LSN of the most recent log record pertaining to that page every time the log manager writes the page to disk



Steal vs. No-Steal Buffer Management

- **Steal** policy:
  - **Pages can be written from the buffer even if the transaction having modified the pages is still active**
  - Rollback of a transaction (UNDO) is required during recovery

- **No-Steal** policy:
  - **All dirty pages are retained in the buffer pool until the final outcome of the transaction has been determined**
  - Rollback is not needed; UNDO is not needed
  - A buffer pool large enough for accommodating all the dirty pages of all concurrent transactions is needed

Force vs. No-Force Buffer Management

- **Force** policy:
  - **At end of transaction (COMMIT), all modified pages are forced to disk in a series of synchronous write operations**
  - This policy avoid any REDO recovery during restart
  - But, **more I/Os** for pages occur during their life span in the buffer pool

- **No-Force** policy:
  - **No modified page is forced during COMMIT; Only if it becomes the replacement victim will it be written to disk**
  - REDO recovery is needed
  - All page writes from the buffer can be done **asynchronously**; thus, it can keep response time low

Checkpointing

- Checkpointing guarantees that only a fixed portion of the log has to be considered for REDO purposes, independent of how long the system has been up

- The basic checkpoint algorithm:
  - **Quiesce the system**
    - Delay all incoming update DML requests until all fixes with exclusive semaphore have been released
  - **Flush the buffer**
    - Write all modified pages
  - **Log the checkpoint**
    - Write a record to the log, saying that a checkpoint has been generated
  - **Resume normal operation**

Pre-Fetching and Pre-Flushing

- **Pre-fetching** tries to capitalize on **locality** by hypothesizing that if a transaction has accessed $page_i$, there is a good chance that it will also access $page_{i+1}$
  - Therefore, **read it before there is a** `bufferfix` **request for it**

- **Pre-flushing** is the technique that allow exploiting the performance potential of the no-force policy (**asynchronous** process)
  - **Write a modified page** at time when there is **nothing else is to do** and when no activity is waiting for the page

Further Possibilities for Optimization

- **Transaction scheduling** and buffer management can take **hints form the query optimizer**:
  - This relation will be scanned sequentially
  - This is a sequential scan of the leaves of a B-tree
  - This is the traversal of a B-tree, starting at the root
  - This is a nested-loop join, where the inner relation is scanned in physically sequential order

Reference

[1] Jim Gray and Andreas Reuter, “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann, San Mateo, CA (1993)

[2] “TRANSCRIPT OF FILES AND BUFFER MANAGER CHAPTER 15. JIM GRAY, ANDREAS REUTER TRANSACTION PROCESSING - CONCEPTS AND…”, FDOCUMENTS, https://fdocuments.in/document/files-and-buffer-manager-chapter-15-jim-gray-andreas-reuter-transaction-processing-concepts-and-techniques-wics-august-2-6-1999-2-abstractions.html