2. Atomicity & Durability Using Shadow Paging

CSEP 545 Transaction Processing for E-Commerce

Philip A. Bernstein

Introduction

- To get started on the Java-C# project, you need to implement atomicity and durability in a centralized resource manager (i.e. a database).
- We recommend you use ***shadowing.***
- This section provides a quick introduction.
  - It's described in the textbook: Chapter 7, Section 6.
- A more thorough explanation of the overall topic of database recovery will be presented in a couple of weeks.

Review of Atomicity & Durability

- Atomicity - a transaction is all-or-nothing
- Durability – the results of a committed transaction will survive failures
- Problem
  - The only hardware operation that is atomic with respect to failure and whose result is durable is "write one disk block"
  - But the database doesn't fit on one disk block!

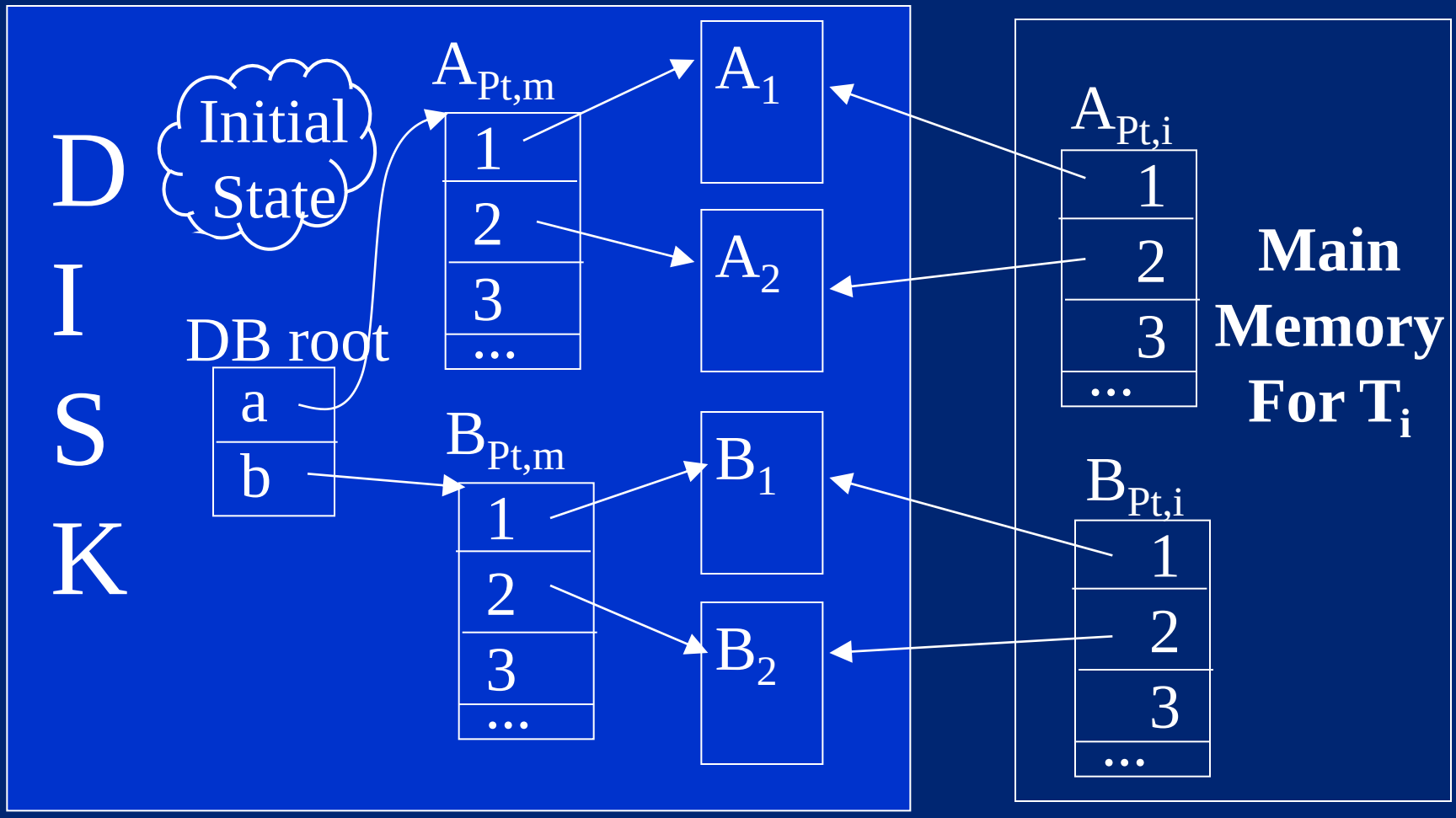
Shadowing in a Nutshell

- The database is a tree whose *root* is a single disk block
- There are two copies of the tree, the *master* and *shadow*
- The root points to the master copy
- Updates are applied to the shadow copy
- To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow
  - Before overwriting the root, none of the transaction's updates are part of the disk-resident database
  - After overwriting the root, all of the transaction's updates are part of the disk-resident database
  - Which means the transaction is atomic and durable

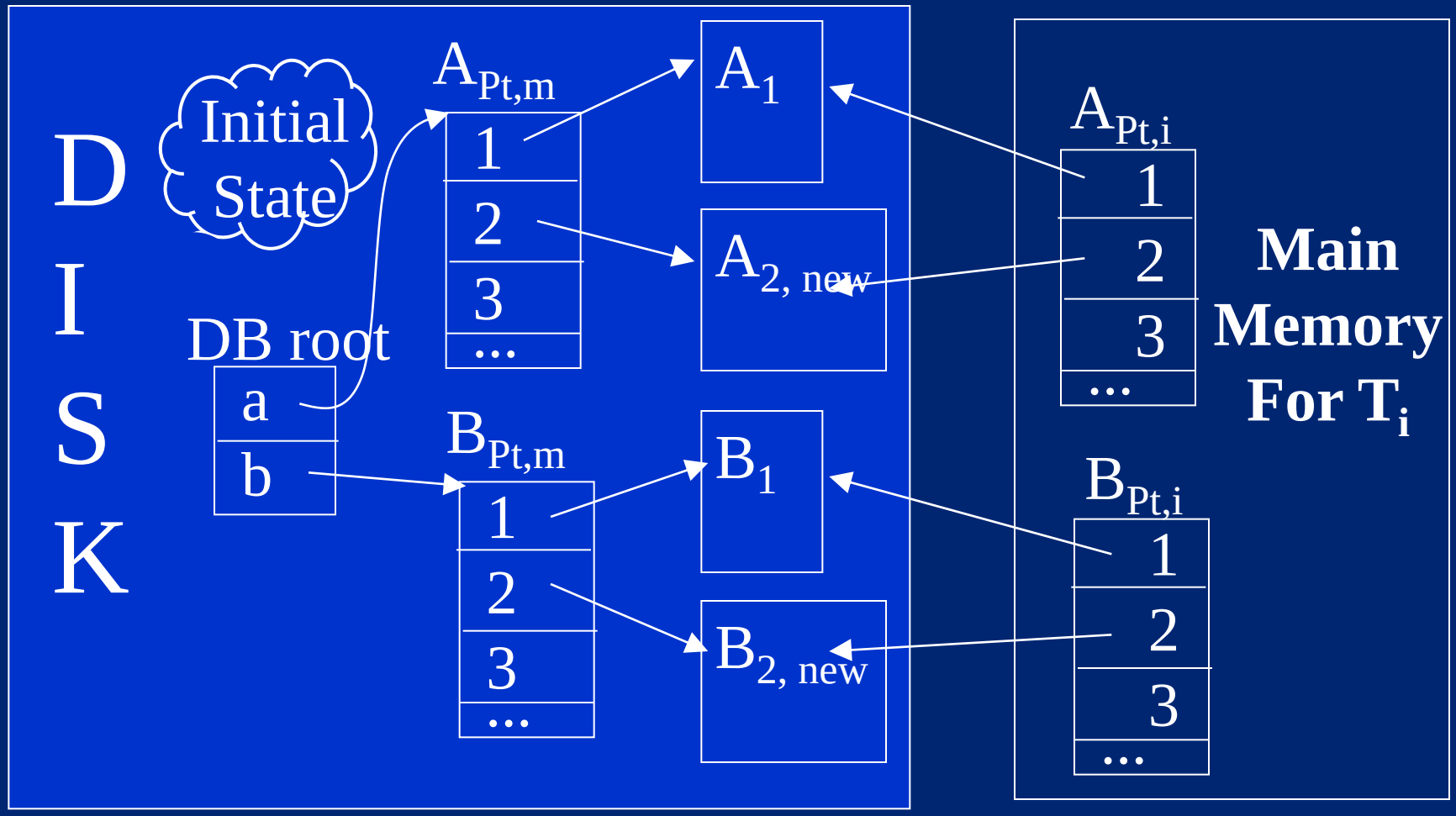
More Specifically …

- The *database* consists of a *set of files*.
- Each file F consists of a *page table* F_{Pt} and a *set of pages* that F_{Pt} points to.
- A *database root page* points to each file's *master page table*.
- To start, assume that
  - Transactions run serially.
    I.e., at most one transaction runs at any given time.
  - For each page table, the transaction has a private shadow copy in main-memory.

Initial State of Files A and B and Transaction T_i



Without Recovery Support: T_i Overwrites A_2 and B_2



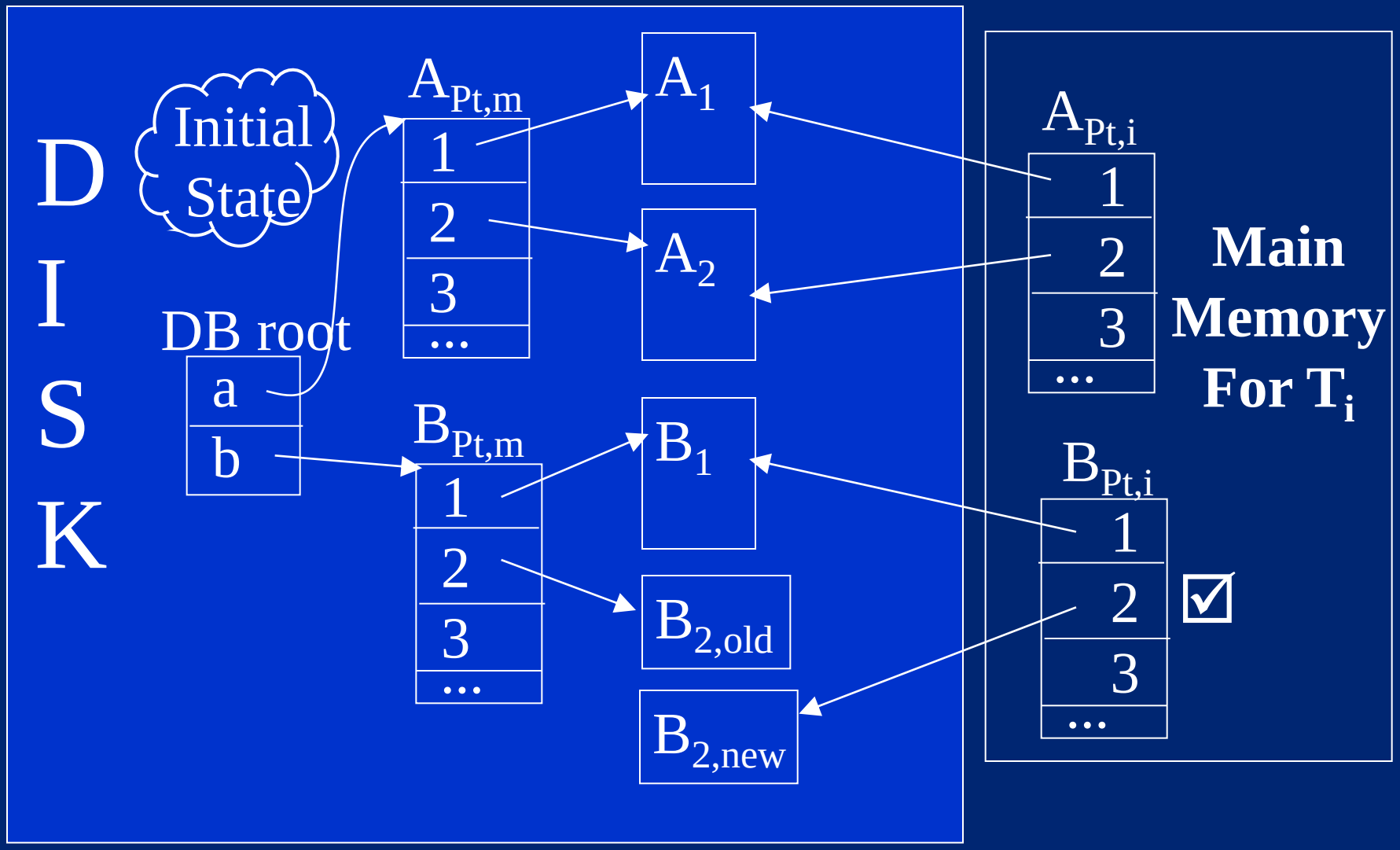
What Could Go Wrong?

- Atomicity violation: A failure while T_i is running can leave the disk state as $[A_{2,new}, B_2]$ or $[A_2, B_{2,new}]$
  - Could be a failure of T_i or hardware or the OS
- This could corrupt a multi-page data structure, making it unintelligible.
- Even if the state is $[A_{2,new}, B_{2,new}]$, readers can't tell whether T_i completed.
  - If T_i completed, maybe it would have re-written one of those pages, or have written a third page B_3

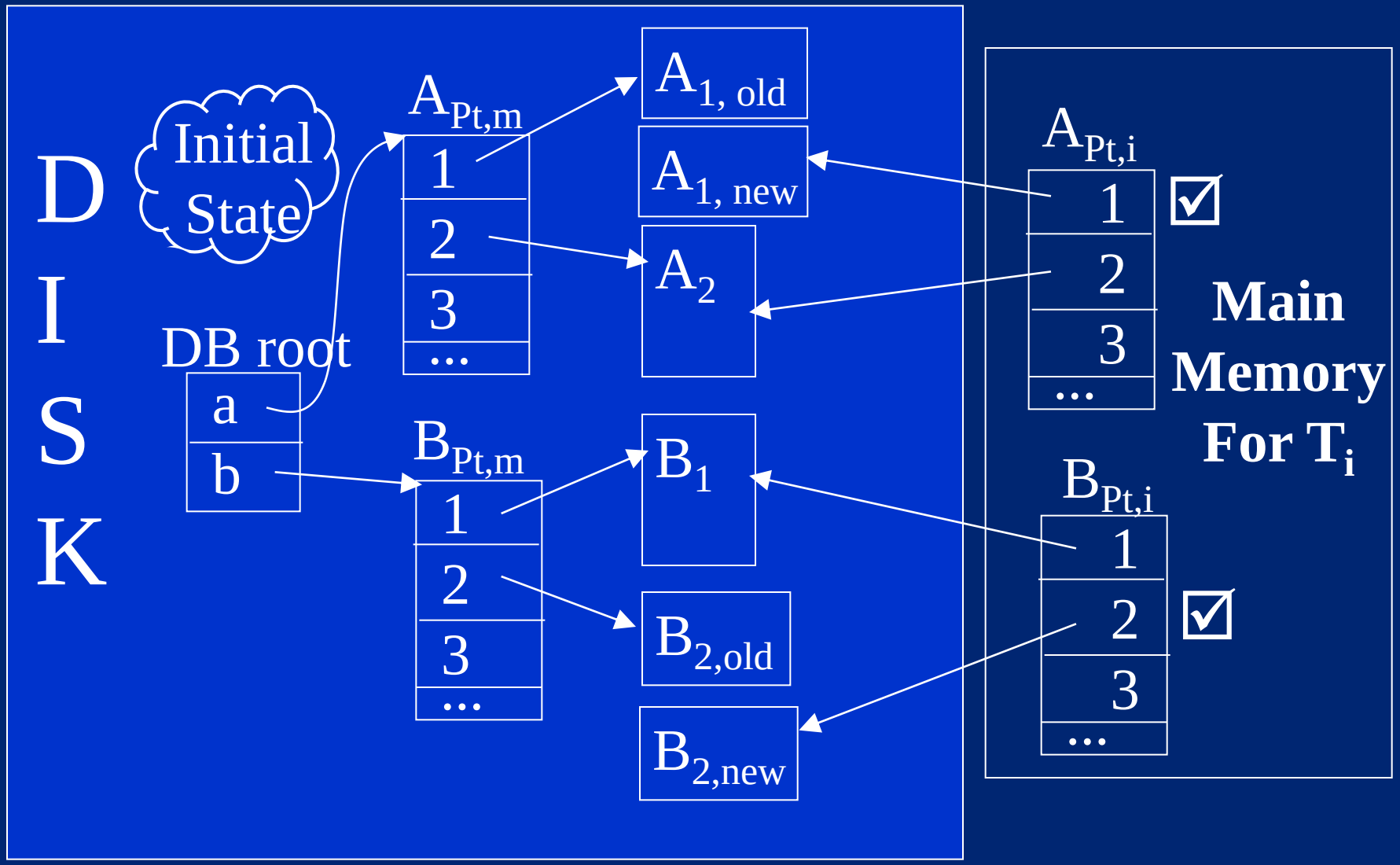
To Write a Page P

- Transaction writes a shadow copy of page P to disk (i.e. does not overwrite the master copy).
- Transaction updates its page table for P's file to point to the shadow copy of P.
- Transaction marks P's entry in the page table (to remember which pages were updated).

After Writing Page B_2



After Writing Page A_1

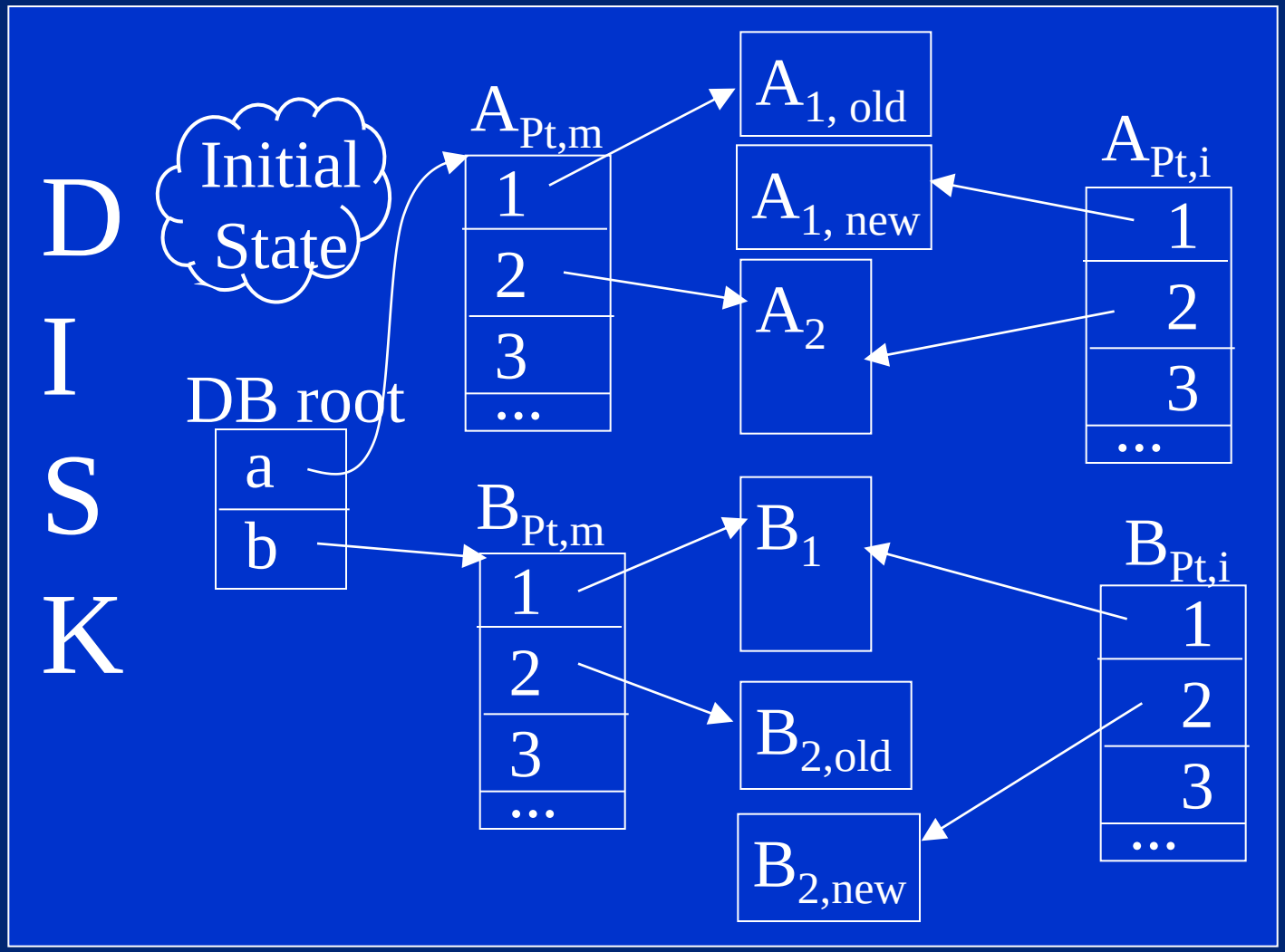


What if the System Fails?

- Main memory is lost
- The current transaction is effectively aborted
- But the database is still consistent

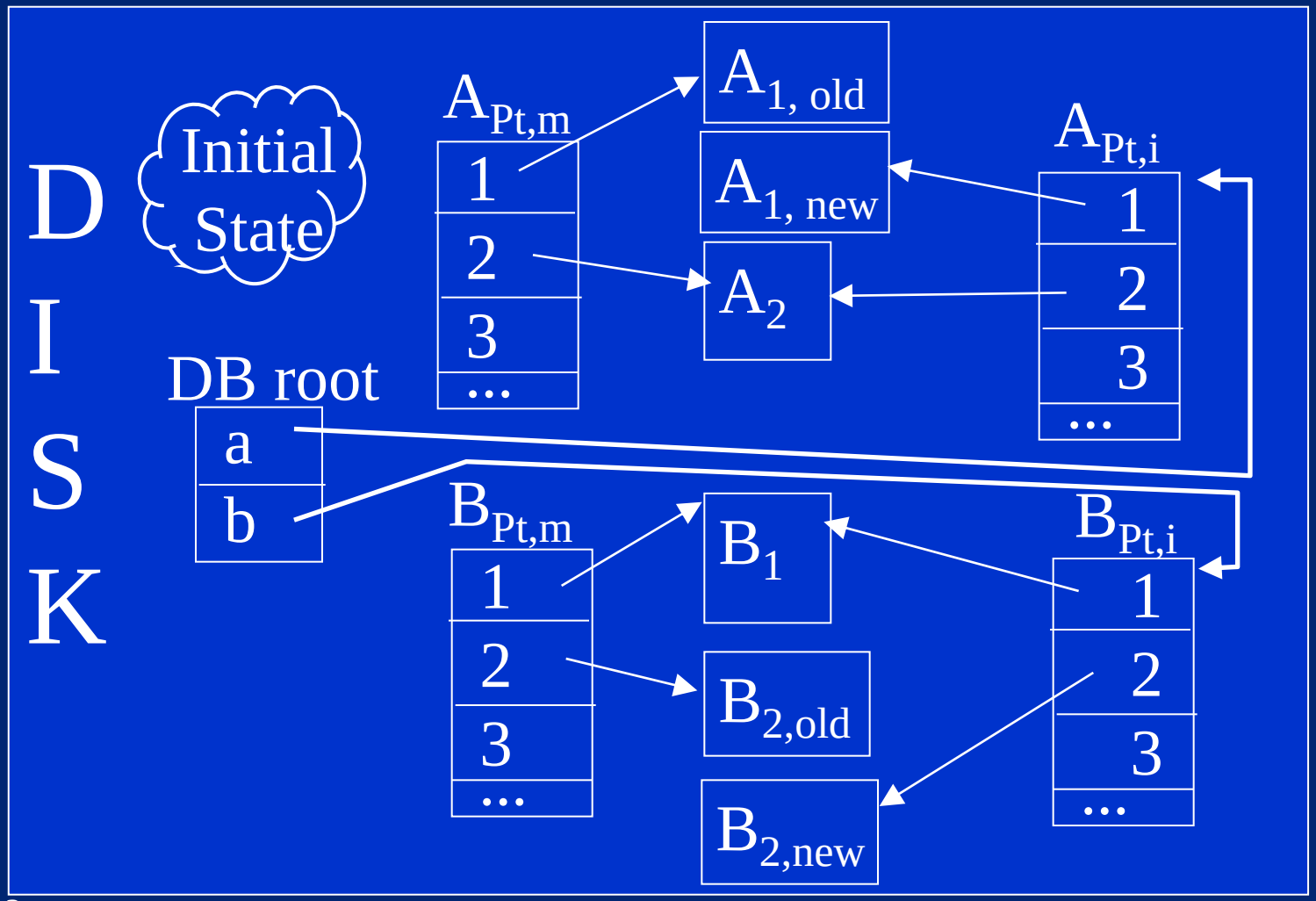
To Commit T_i

1. First copy $A_{Pt,i}$ and $B_{Pt,i}$ to disk



To Commit T_i (cont'd)

2. Then overwrite DB root to point to the new Pt's.
   - This is the atomic hardware operation that commits T_i.



Shadow Paging with Shared Files

- What if two transactions update different pages of a file?
  - If they share their main-memory shadow copy of the page table, then committing one will commit the other's updates too!
- One solution: File-grained locking (but poor concurrency).
- Better solution: use a private shadow-copy of each page table, per transaction. To commit T, do the following *within a critical section* :
  - For each file F modified by T
    - Get a private copy C of last committed value of F's page tbl.
    - Update C's entries for pages modified by T.
    - Store C on disk.
  - Write a new master record, which swaps page tables for the files updated by T, thereby installing just T's updates.

Managing Available Disk Space

- Treat the list of available pages, Avail, like another file
- The DB root points to the master Avail
- When a transaction allocates a page, update its shadow Avail list
- When a transaction commits, write a shadow copy of Avail to disk
- Committing the transaction swaps the master Avail list and the shadow

Final Remarks

- A transaction doesn't need to write shadow pages to disk until it is ready to commit
  - Saves disk writes if a transaction writes a page multiple times or if it aborts
- Main benefit of shadow paging is that doesn't require much code
  - Was used in the Gemstone OO DBMS (1980's)
- But it is not good for TPC benchmarks
  - How many disk updates per transaction?
  - How to do record level locking?
- Most database products use logging.
  - Faster execution time, and more functional, but much more code.

Your Project

- You need not use the exact data structure presented here.
- In particular, you don't necessarily need a page abstraction.
- There are design tradeoffs for you to figure out.

References

- Textbook, Section 7.6.

- P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Chapter 6, Section 7 (pp. 201-204)
  - The book is downloadable from http://research.microsoft.com/pubs/ccontrol/

- Originally proposed by Raymond Lorie in "Physical Integrity in a Large Segmented Database"*ACM Transactions on Database Systems*, March 1977.