Fault-Tolerance I: Atomicity, logging, and recovery

COS 518: *Advanced Computer Systems*
Lecture 3
Kyle Jamieson

What is fault tolerance?

- Building **reliable** systems from **unreliable** components

- Three basic steps:

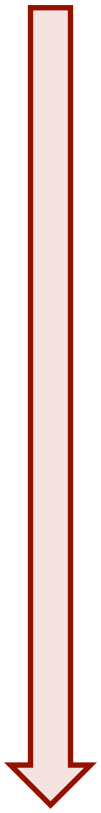
1. ***Error detection:*** Discovering the presence of an error in a data value or control signal

2. ***Error containment:*** Limiting error propagation distance

3. ***Error masking:*** Adding redundancy for correct operation despite the error (possibly correct error)

Why is achieving fault tolerance hard?

Failures Propagate



- Say **one bit** in a DRAM fails:
- …it flips a bit in a memory address the kernel is writing to. Causes big memory error elsewhere, or a **kernel panic**
- …program is running one of many distributed file system storage servers
- …a client **can't read from FS,** so it hangs

So what to do?

1. Do nothing: Silently return the failure

2. Be ***fail-fast:*** Detect the failure and report at interface
   – *e.g.,* Ethernet station jams medium on detecting collision

3. Be ***fail-safe:*** Transform incorrect → acceptable values
   – Failed traffic light controller switches to blinking-red

4. ***Mask*** the failure
   – *e.g.* retry op for transient errors, use error-correcting code for bit flips, replicate data in multiple places

Techniques to cope with failures

- You've already seen some in this and other classes
  - *e.g.*, retransmissions in TCP and RPC

- **Modularity** can isolate failures
  - Prevent error in one component from spreading

- We'll discuss two families of failure-masking techniques:
  - **Atomicity**, **logging**, and **recovery** on one server
  - **Replication** and **consistency** across multiple servers

The fault tolerance design process

1. Identify every fault, **quantify probability** of occurrence

2. Apply **modularity** to contain damage from high-risk errors

3. Design and implement fault tolerance procedures

- Iterate **twice** on this procedure:
  - Once to account for **reduction of faults** from fault tolerance procedures
  - A second time to run the system ***in situ***, improve and revise

Today

- Techniques for coping with failures:

1. **Failures, reliability, and durability**
2. Atomicity
3. Case study: System R DBMS recovery manager

Measuring the availability of a system component

- A component operates correctly for some time, fails, is repaired, then the cycle repeats (***run-fail-repair cycle***)
  - So, ***time to failure*** and ***time to repair*** are quantities of interest. Averaging over multiple run-fail-repair cycles:
    - Mean time to failure (MTTF)
    - Mean time to repair (MTTR)
  - ***Availability:*** MTTF / (MTTF + MTTR) = 1 − ***Down time***
  - ***Mean time between failures:*** MTBF = MTTF + MTTR

- *e.g.:* suppose an OS crashes once per month and takes ten minutes to reboot
  - MTTF = 720 hours = 43,200 min, MTTR = 10 min
  - **Availability** = 43,200 / 43,210 = 0.997 (“two nines”), or two hours **down time** / year

Availability in practice

- Carrier airlines (2002 FAA fact book)
  - 41 accidents, 6.7M departures
  - 99.9993% (five nines) availability
- 911 Phone service (1993 NRIC report)
  - 29 minutes per line per year
  - 99.994% (four nines) availability
- Standard phone service (various sources)
  - 53+ minutes per line per year
  - 99.99+% (> four nines) availability
- End-to-end Internet Availability
  - 95% - 99.6% (one to two nines) availability

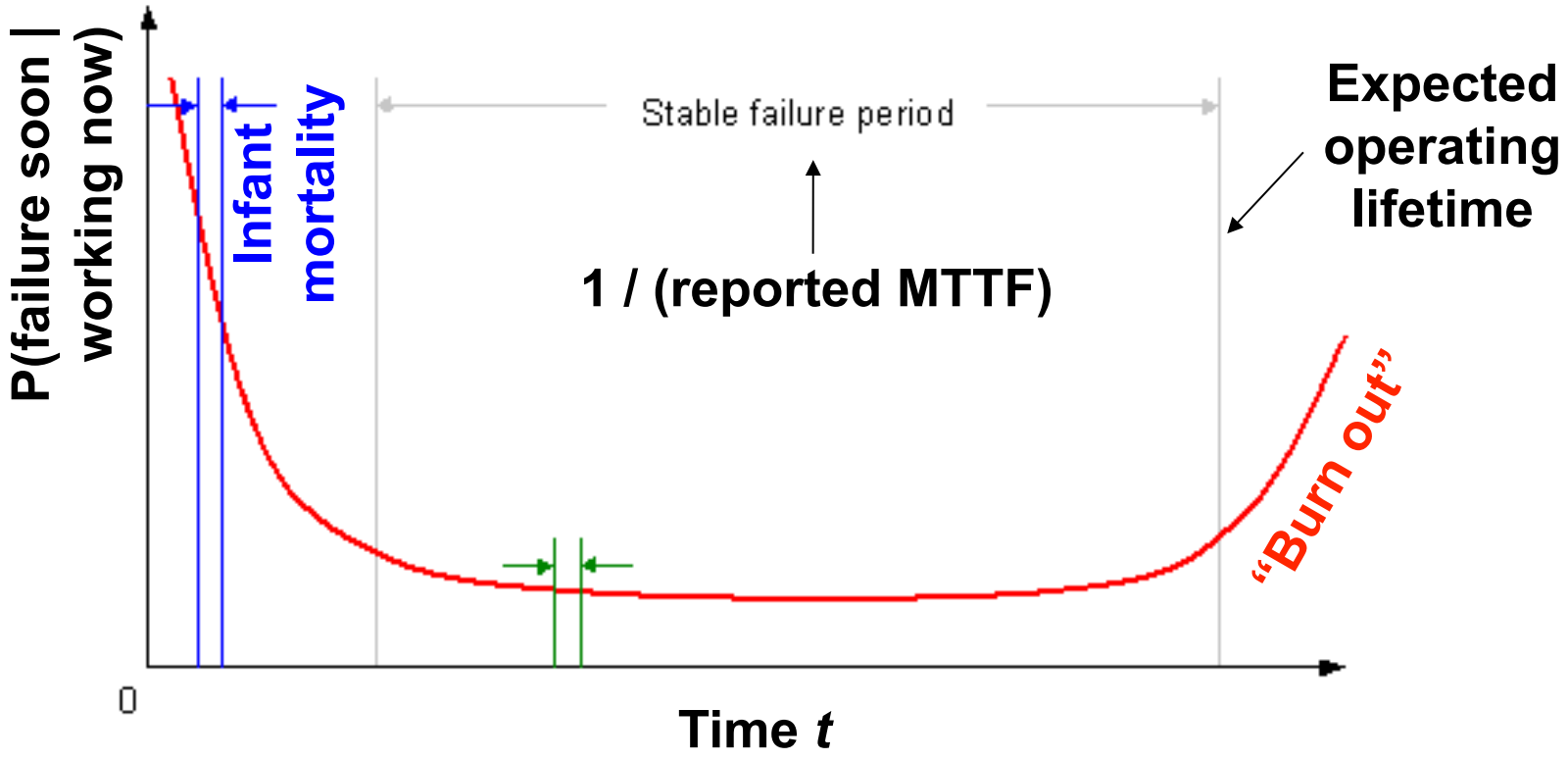
Two cautions

1. Are failures **independent**?
   - Q: If the failure probability of a computer in a rack is ***p***, what is Pr(computer 2 failing | computer 1 failed)?
   - A: Maybe it's *p*... but plugged into same rack power strip, where several racks share same UPS?
     - And servers also share same network switch, which in turn share same border gateway routers?

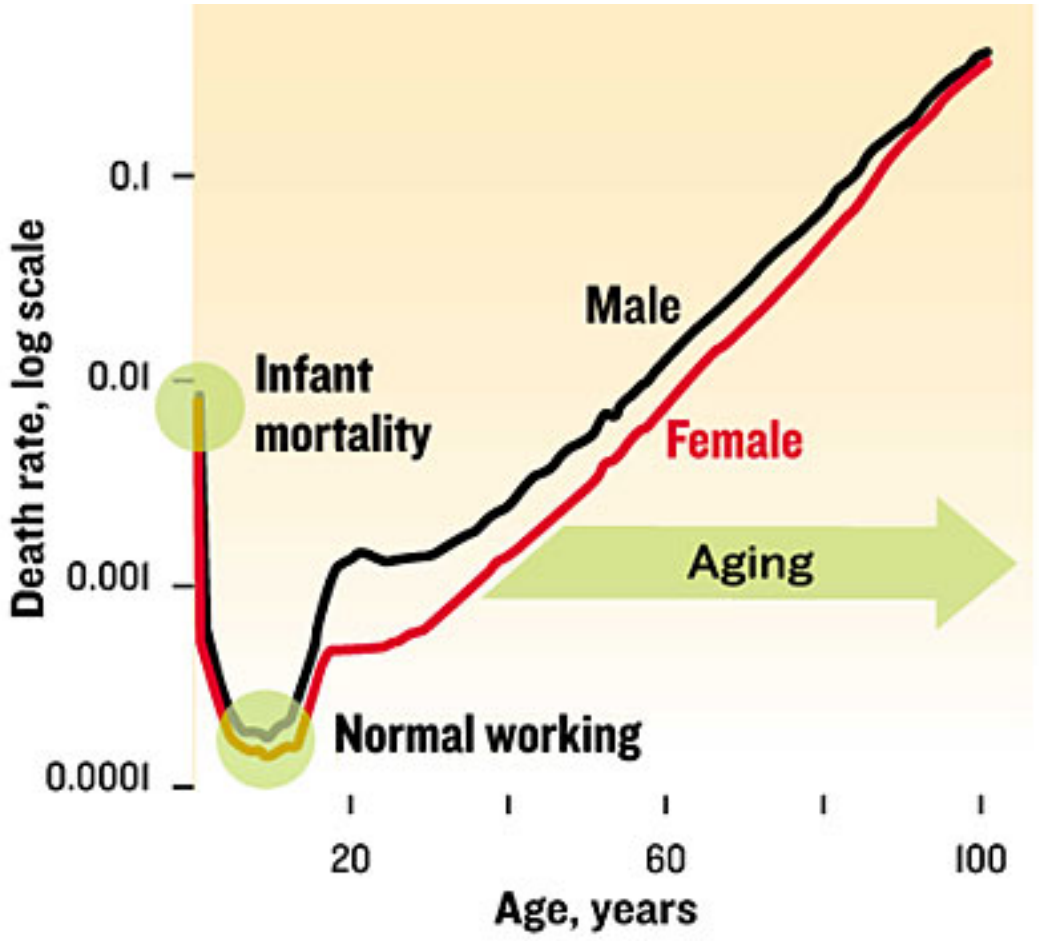
2. Do failures follow a **memory-less** process?
   - Hard disk label advises "expected operational lifetime" of five years…

"Bathtub curve" describes many common component conditional failure rates

- What's the probability the component fails between time *t* and *t* + *dt*, given that it's working at time *t*?



Human mortality rates (USA, 1999)



From: L. Gavrilov & N. Gavrilova, "Why We Fall Apart," IEEE Spectrum, Sep. 2004

Applying redundancy to software

- Key idea: **Separate the state** that may be abandoned in case of failure from state that must be preserved

- The latter is called ***durable storage***
  - Therefore once the action is performed, the result or value of the action persists for some amount of time (***durable action***)

- Primary challenge: Building a software system that **protects** the integrity of **durable storage** despite failures
  - Approach: Build a firewall against failure using the **GET/PUT** interface of **non-volatile storage** devices

Raw disk storage

- The interface that the hard **disk hardware** exposes to the **disk electronics/microcode** above:
  - RAW_SEEK(track) moves disk head into position
  - RAW_PUT(data) writes entire track
  - RAW_GET(data) reads entire track

- Untolerated errors: Dust/ RF noise (soft error), defective sector (hard error), seek error, power failure (causes **partial track write**)

Fail-fast disk storage

- The interface that the **disk electronics/microcode** exposes to the **disk firmware** above:
  - status ← FAIL_FAST_SEEK(track)
  - status ← FAIL_FAST_PUT(data, sector_number)
  - status ← FAIL_FAST_GET(data, sector_number)

- **Error detection code** checks data integrity, ***in situ* sector and track numbers** check seek operation integrity

- Detected errors: Hard/soft/seek errors, power fails during PUT causing partial sector write

- **Untolerated errors:** OS crash during FAIL_FAST_PUT scribbles on data buffer

Careful disk storage

- The interface that the **disk firmware** exposes to the **operating system** above:
  - status ← CAREFUL_SEEK(track)
  - status ← CAREFUL_PUT(data, sector_number)
  - status ← CAREFUL_GET(data, sector_number)

- Checks status of FAIL_FAST_*, **retries** if necessary

- **Masked errors:** Soft errors, seek errors

- Detected errors: Hard errors (can then find someplace else), power failures during CAREFUL_PUT

- **Untolerated error:** OS crash during CAREFUL_PUT scribbles on data buffer

Today

- Techniques for coping with failures

1. Failures, reliability, and durability

2. **Atomicity**

3. Case study: System R DBMS recovery manager

Atomicity

- Beneficial in many different contexts
  - "Purchase" Internet shopping button and power cut
  - You and someone else click "purchase" and one in stock

- ***Atomic*** action: There is no way for a higher layer to discover the internal structure of the action
  - ***All-or-nothing atomic:*** If the action does not complete fully, it leaves no effects
  - ***Before-or-after atomic:*** The action behaves as if it occurred completely before or completely after any other before-or-after atomic action

- An action can be **atomic but not durable**
- An action can be **durable but not atomic**

Logging and crash recovery

- Atomicity and durability via ***transactions***
- Standard ***"crash failure"*** model:
  - Machines are prone to crashes: Disk contents OK (nonvolatile), Memory contents lost (volatile), but machines don't misbehave ("Byzantine")
  - Networks are flaky
    - Drop messages, but handled by retransmissions
    - Corruption detected by checksums

General approach

- ***Transaction durability:*** Once a transaction has committed, effects must be permanent for some amount of time
  - Storing database in memory violates this, as crash will lead to loss of durability

- Failure atomicity: Even when system crashes
  - Must recover so that uncommitted transactions are either aborted or committed

- General scheme: Store enough info on disk to determine global state

Challenges

- High transaction speed requirements
  - If always force writes to disk for each result on transaction, yields terrible performance

- Atomic and durable writes to disk are difficult
  - In manner to handle arbitrary crashes
  - HDDs/SSDs use write buffers on volatile memory

Techniques to overcome challenges

- ***Shadow pages***
  - ***Copy-on-write:*** Keep updated copies of all modified entries on disk, but retain old pages.
  - **Abort** by **reverting back** to shadow page

- ***Write-Ahead Logging*** (WAL)
  - ***Log*** records every operation performed.
  - Update is reliable when log entry carefully-put on disk
  - Keep updated versions of (disk) pages in memory
  - To recover, **replay** log entries to reconstruct correct state

- WAL is more common, as fewer disk operations
  - Transaction committed once logfile entry stored on disk
  - Only need to `fsync` log when encounter COMMIT

Two storage models

1. Database-style
   – Multiple data items (rows, keys)

2. Shared memory in multiprocessor
   – Single register access / key

- More on this later when we talk about consistency models

- **Today:** Database-style
  – Atomicity particularly relevant with multiple keys

Today

- Techniques for coping with failures

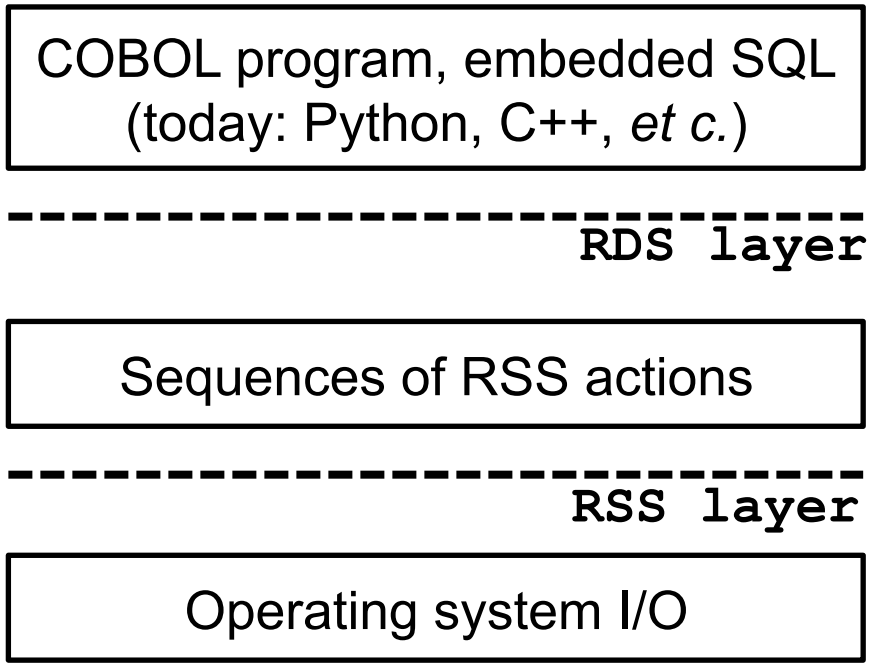
1. Failures, reliability, and durability

2. Atomicity

3. Case study: System R DBMS recovery manager

System R: How do you use it?

- The ***Research Data System*** (RDS)
  - Provides **you** a relational programming model
  - Compiles SQL statements into RSS actions
- The ***Research Storage System*** (RSS)
  - Provides **the RDS** record-based access
  - Issues I/O operations to service RSS actions
  - Provides "transactional" semantics…



RSS "transactions"

- ***RSS transactions:*** sequence of RSS actions framed with **`BEGIN TRANSACTION`**, **`COMMIT TRANSACTION`** RSS actions

- RSS transactions are ***all-or-nothing atomic:*** either do **all** the RSS actions in a transaction, **or none at all**

- ***Before-or-after atomicity:*** two transactions relating to same object appear to execute in a **serial order**
  - **Programmer** must acquire locks to provide this

- RSS actions **themselves** are all-or-nothing, before-or-after

Why is the RSS so complex?

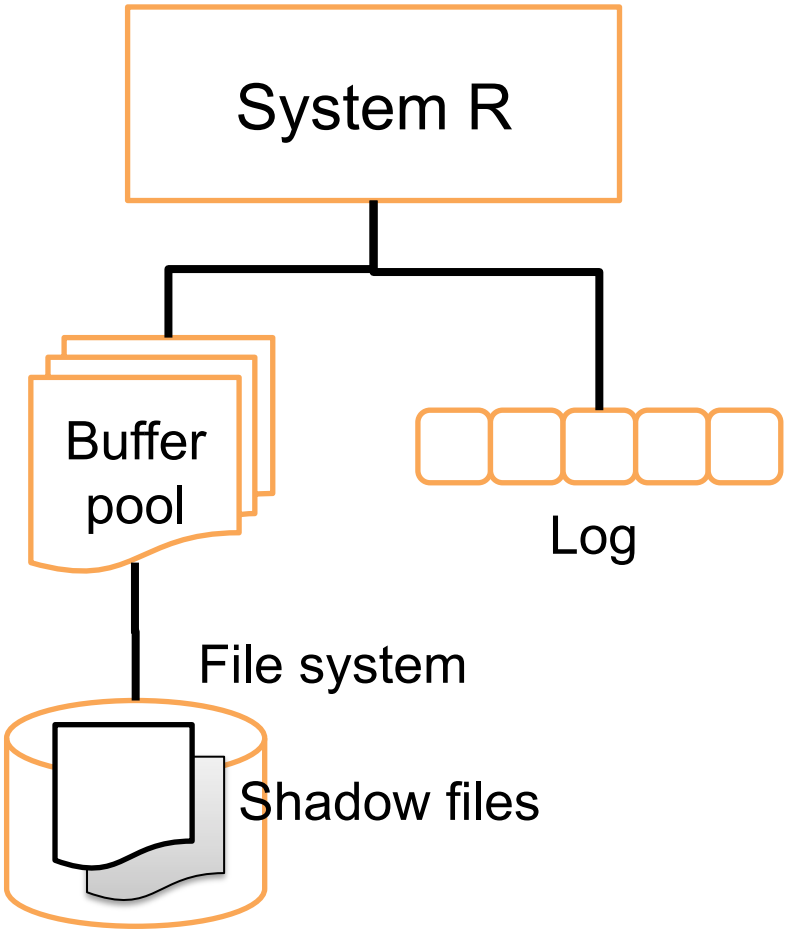
1. Performance
   - System R leverages disk buffering and "lazy write" strategies for speed that interact with recoverability

2. Several simultaneous goals
   - ***Archiving*** storage: Keep old values around
   - ***Durability:*** Always remember committed transactions

3. Change of goals
   - First the system designers focused on surviving crashes (so invented shadow pages)
   - Then, realized they wanted consistent updates to multiple objects (so added log for recoverable transactions)

Why is the RSS so complex? (2)

- Many interacting features:
  - Least-recently-used (LRU) disk buffer pool
  - Shadowed files
  - Log of old/new record values
  - System checkpointing



Failure model; availability requirement

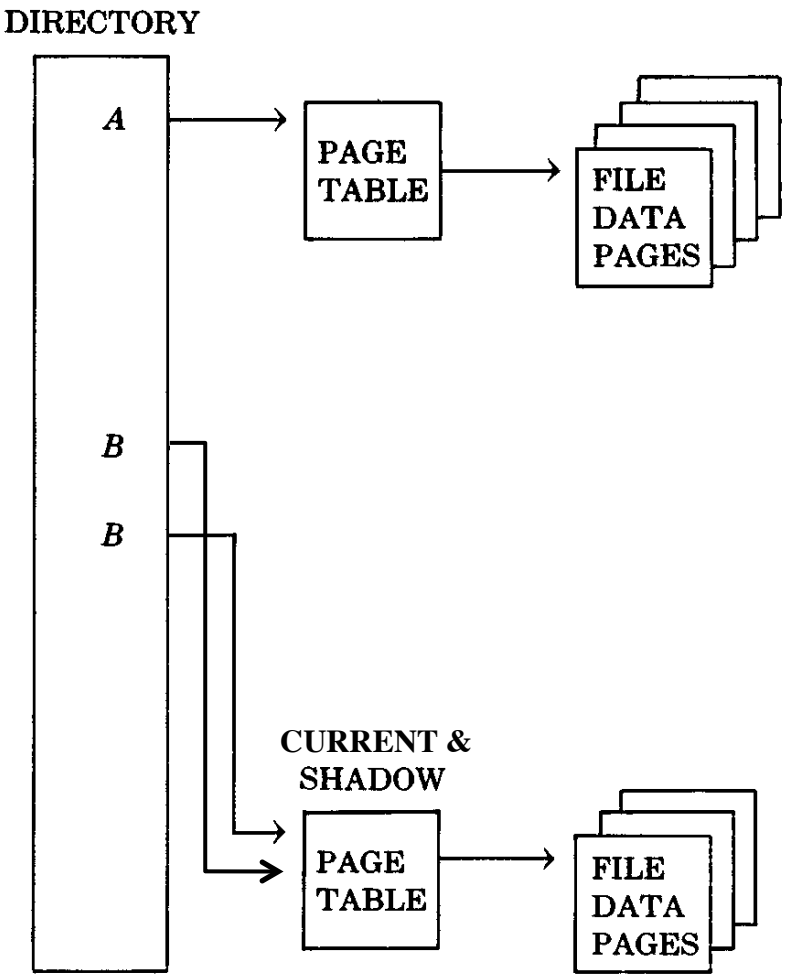
1. Transaction abort
   - Several **per minute:** users cancel or make input errors
   - Recovery time goal: **milliseconds**

2. System crash and restart
   - Several **per month:** H/W or OS failure, or if System R detects a data structure inconsistency
   - Recovery time goal: **seconds**

3. Media failure
   - Several **per year:** disk head crash, S/W failure
   - Recovery time goal: **hours**

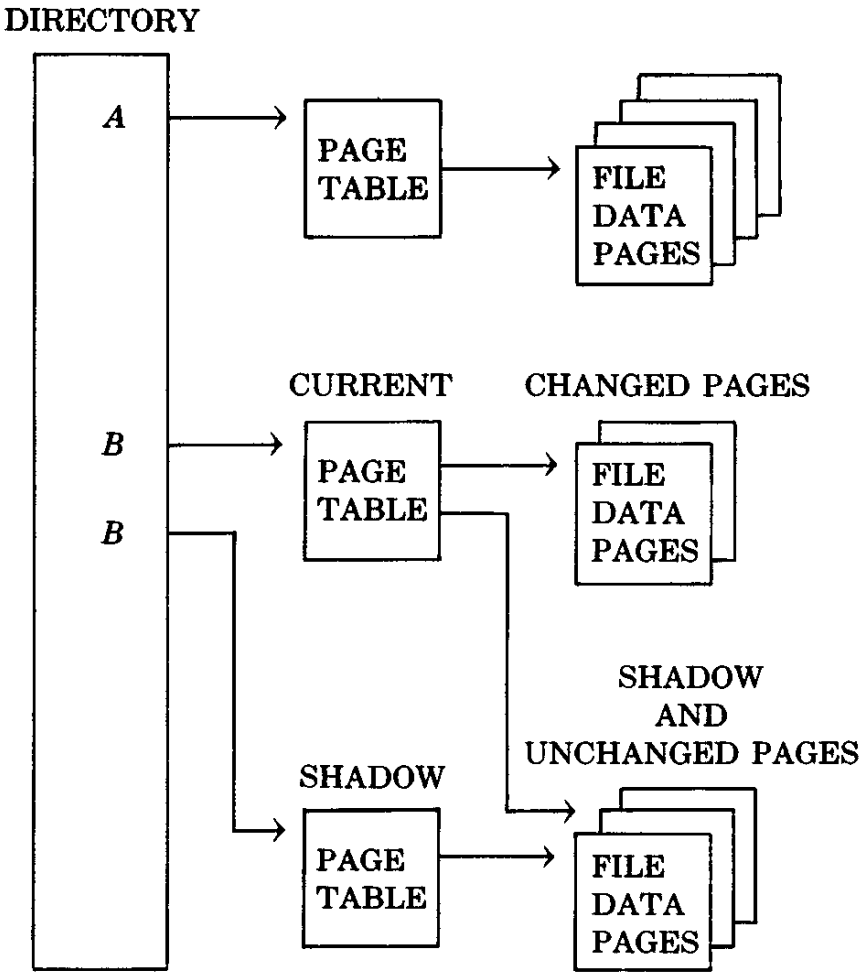
Files and the buffer pool

- Buffer pool is managed with a ***least-recently-used*** (LRU) policy
- File **`A`** is ***non-shadowed:*** System R updates its pages in the buffer pool
- File **`B`** is shadowed:
  - When first opened, ***current*** and ***shadow*** entries point to the same page table



Files and the buffer pool

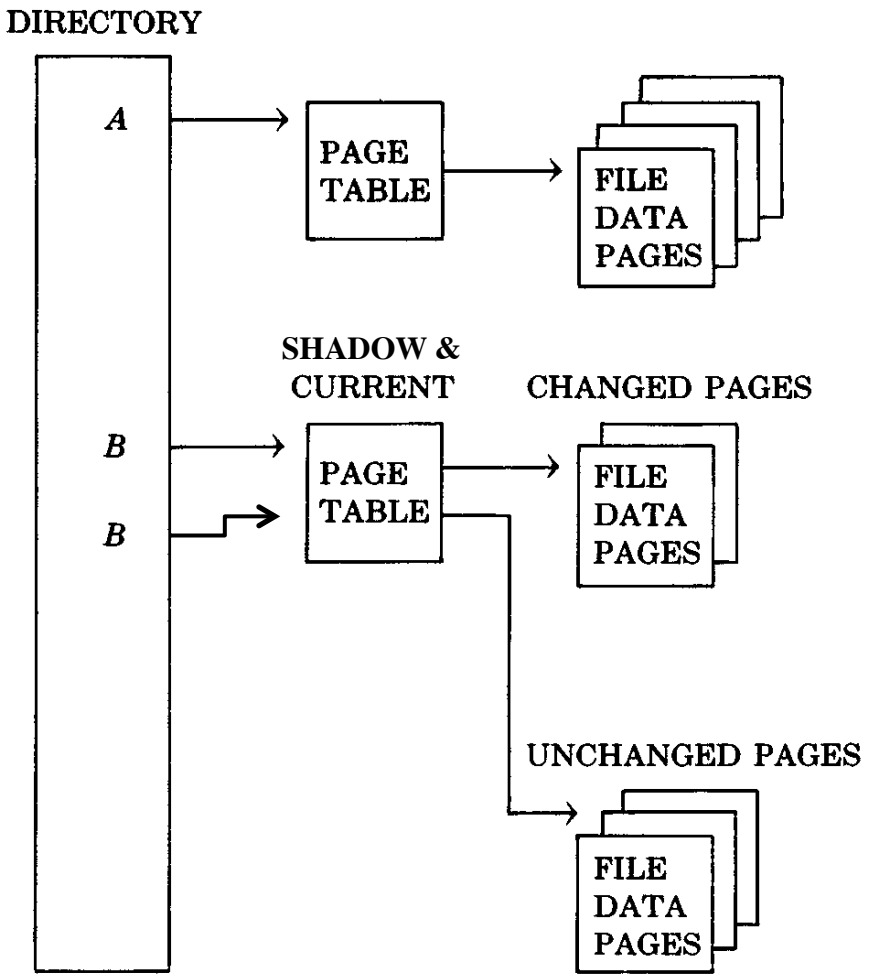
- When the first File **`B`** write occurs:
  - Allocate another page table
  - Point **current** file pointer to the new page table
  - Write data to new page
  - Point to new page in the current page table
  - This is also called ***copy-on-write (COW)***



FILE SAVE

- On FILE SAVE(B):

1. Force pages to disk
2. Force current PT to disk
3. Set **shadow** page table ← **current** page table
4. Force directory to disk
5. Release orphaned (shadow) pages and old (shadow) page table



Properties of shadow files

- Suppose we make changes to a file without `FILE SAVE`, then crash. Do we still have our changes?
  - No! They might not have been flushed

- What if two transactions T1 and T2 are writing data to different parts of the same file: do T1 and T2 commit on `FILE SAVE`?
  - No! `FILE SAVE` does not pertain to transaction, it's only used for checkpoints and crash recovery

- How do we implement `FILE RESTORE`?
  - Set **current** page table ← **shadow** page table
  - Release orphaned pages

The log

- Provides **all-or-nothing atomicity** for RSS xactns
- Consists of a chained list of records:
  - (transaction id, record id, old value, new value)
- Written according to the ***write-ahead log (WAL) protocol:*** force the log to disk **before** `FILE SAVE`
- To force the log to disk: **First** force all transaction's log records to disk, **then** force commit record **last**
  - ***Commit point*** is the instant commit record on disk

Transaction UNDO

- Suppose a transaction is in trouble (*e.g.:* a transaction to book flight and hotel room finds a flight but no hotel room)
  - How does System R `UNDO` the transaction?

- Go to log, follow chain of events for this transaction backward, undo each RSS action
  - Stop when you reach `BEGIN TRANSACTION` record

Thinking about `FILE SAVE`

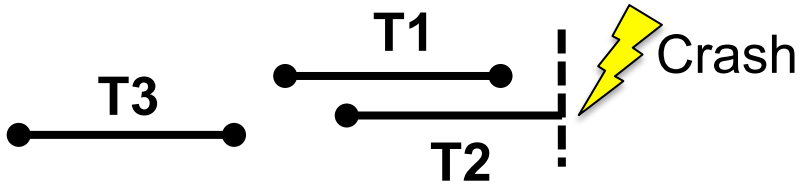
1. Hypothetical: Suppose no one ever calls `FILE SAVE`
   – On crash, all writes lost! But the log contains it all

2. Hypothetical: Suppose System R called `FILE SAVE` only when quiet
   – On crash, only need to REDO xactns after FILE SAVE

Thinking about `FILE SAVE`

1. Hypothetical: Suppose no one ever calls `FILE SAVE`
2. Hypothetical: Suppose System R called `FILE SAVE` only when quiet

3. Hypothetical: Suppose System R called `FILE SAVE` just before anyone commits any transaction
   – On crash, only need to UNDO logged writes of T2 xactns that were pending at the time of the last logged commit T1



Thinking about `FILE SAVE`

1. Hypothetical: Suppose no one ever calls `FILE SAVE`
2. Hypothetical: Suppose System R called `FILE SAVE` only when quiet
3. Hypothetical: Suppose System R called `FILE SAVE` just before anyone commits any transaction

4. **System R:** The only time anyone ever issues `FILE SAVE` is at a periodic ***checkpoint***

System R checkpoint procedure

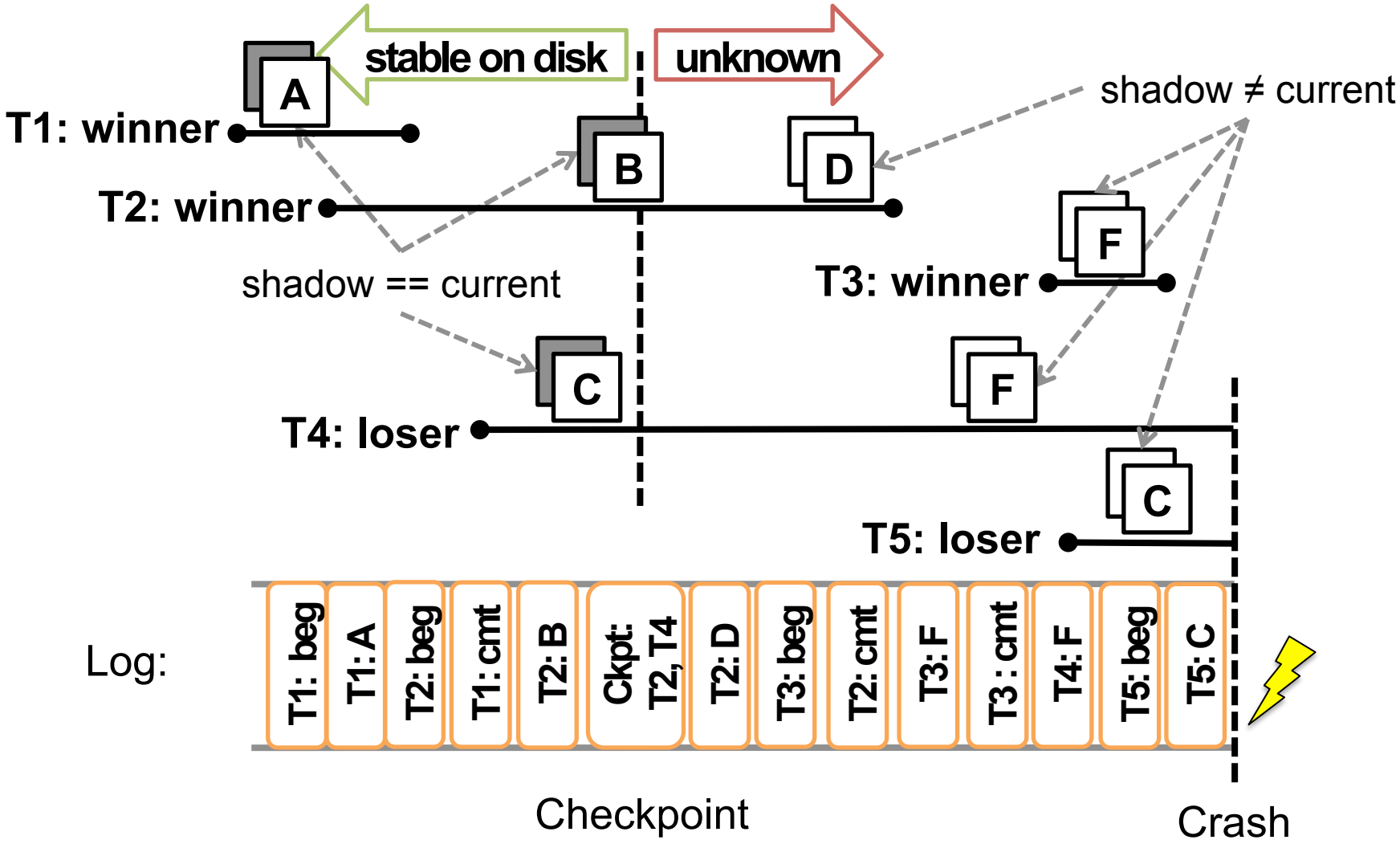
1. First, write ***log checkpoint record***:
   - Contains list of all in-progress xactns and pointers to their most recent log records

2. Force the log to disk

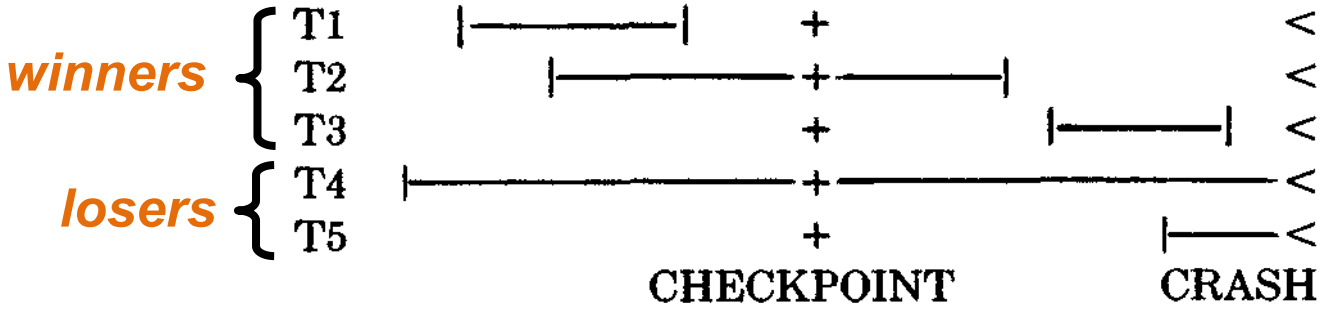
3. Then, `FILE SAVE` **every** open file
   - This forces all shadow page maps to disk

4. Last, remember new checkpoint record
   - Use a careful-put (*cf.* S&K Chp. 8) (*why?*)

System R checkpoint, crash, restart



System R restart procedure



1. File manager restores shadowed files to **shadow** versions

2. Scan **forward** from ckpt; assume active xactns are **T4**
   - If encounter `BEGIN` record, note xactn as **T5**
   - If encounter `COMMIT` record of **T4**, note xactn as **T2**
   - If encounter `COMMIT` record of **T5**, note xactn as **T3**

3. Scan **backward** from ckpt; **undoing** loser ops

4. Scan **forward** from ckpt; **redoing** winner ops

System R restart: Q&A

- **How far back** do we need to scan the log?

- What if System R fails **during the restart procedure?**

- What if a xactn **aborts just before the crash?**
  - It has recorded its writes and an `ABORT` record in the log, but its `UNDO`s are trapped in the buffer cache