Persistent ⟨key, value⟩ storage

- **In programs, often use hash tables**
  - E.g., Buckets are an array of pointers, collision chaining
- **For persistant data, minimize # disk accesses**
  - Traversing linked lists is particularly bad
- **Thus, aggregate ⟨key, value⟩ pairs into blocks**
  - Should find data in expected $O(1)$ block reads

Linear hashing

- **Store $\langle K, V \rangle$ pairs in blocks**
  - When a block is full, split into two blocks
- **Bitmap records which blocks have been split**
  - If block 0 not split, just look in block 0
  - If block 0 not split, consider first bit of hashed key
  - Continue down key until you find an usplit block
  - If n bits considered & K 32 bits, data in block $K >> (32 - n)$
- **Note: Makes use of sparse files**

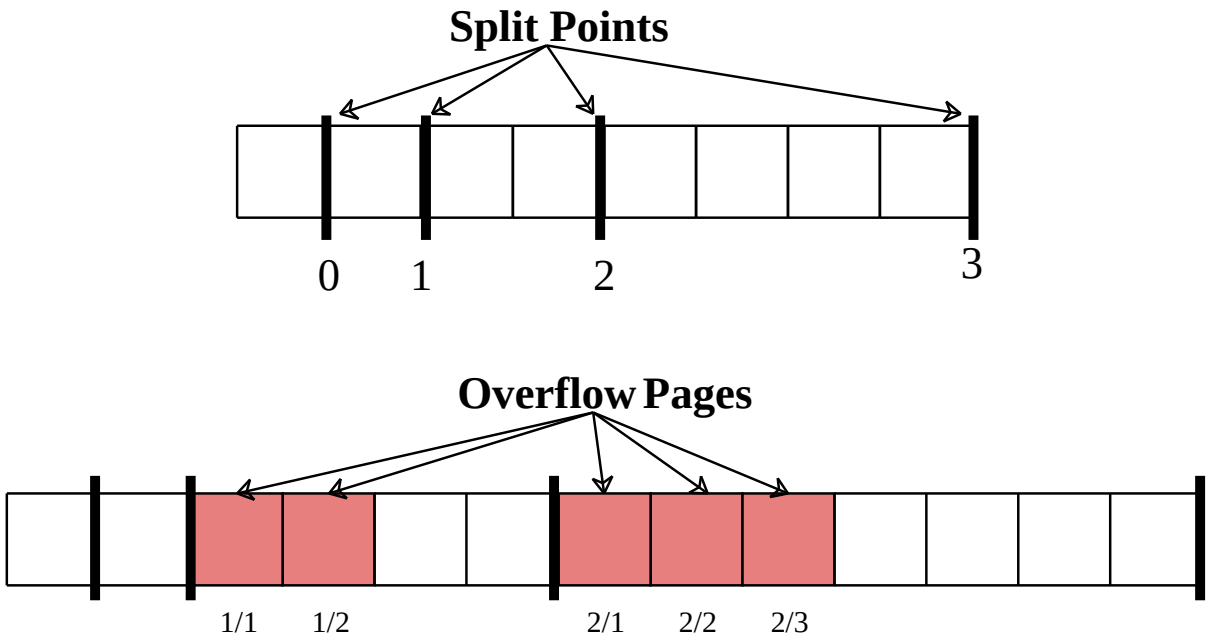
Extendible hashing

- **Idea: Use flat directory to index blocks**
  - E.g., $00 \rightarrow \text{Blk00}$, $01 \rightarrow \text{Blk01}$, $10 \rightarrow \text{Blk1}$, $11 \rightarrow \text{Blk1}$,
  - In example, Keys w. prefix 10 and 11 go to same block
- **Advantages:**
  - Does not use sparse files (so can copy DB efficiently)
  - Reuses space better if one blocks frees up and other fils
  - If directory fits in memory, fast table lookup
- **Disadvantages:**
  - Much bigger directory structure (2^n full block pointers where n is longest split prefix)

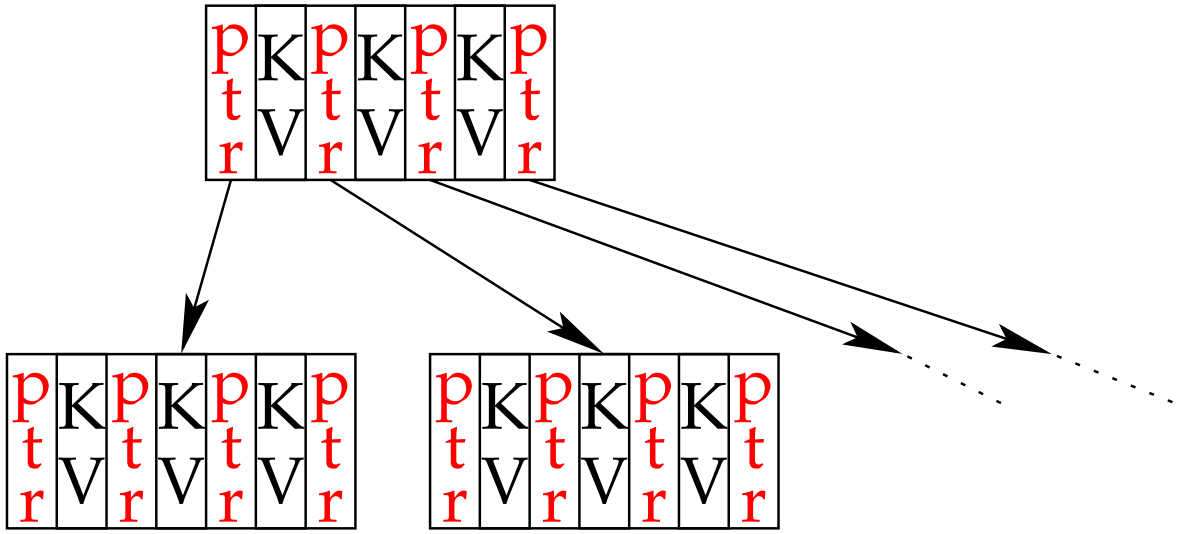
Dealing with large keys

- **What happens if key doesn't fit in block?**
- **Solution: Overflow blocks**
  - Can link from main block to overflow block
  - Might allocate overflow blocks from a different file
- **"Buddy-in-waiting" scheme uses same file**
  - In linear hashing, allocate overflow blocks at end of generation ("split points")
  - Record how many overflow blocks allocated in file header

Buddy-in-waiting



B-trees



- **Indexed data structure stores Keys & Values**
  - Keys must have an ordering defined on them
- **Data is stored in blocks**
  - Each block (except possibly root) has $[t-1, 2t-1]$ keys
  - Non-leaf block with n keys has $n+1$ pointers to *child* nodes
  - All leaf blocks are at the same depth

B-tree insertion

- **Straw man: Search for appropriate leaf & insert**
  - If leaf is full ($2t - 1$ keys), split into two nodes
- **Problem: When splitting, what if parent is full?**
  - Can't insert new divider key in parent
- **Solution: Pre-split nodes as you decend tree**
  - If you traverse internal node w. $2t - 1$ keys, split into two nodes of $t - 1$ keys.
  - Insert median key of split node into parent
  - If root had $2t - 1$ nodes, add new level to the tree
- **Thus, insert requires $O(\log n)$ time for n keys**

B-tree deletion

- **If key is in leaf, delete it**
- **If key in internal node**
  - Suck up last key of previous child or first of next child
  - If both children have $t - 1$ keys join them into one node
- **Problem: May leave node with $< t - 1$ keys**
- **Solution: Pre-join as you traverse**
  - Ensures all parents have at least t keys
- **Thus, delete has $O(\log n)$ time**
- **Note, sometimes people do lazy delete**
  - Sacrifice worst case behavior
  - But better in common case tree doesn't get much smaller

B-tree refinement

- **B+-tree – increase branching factor**
  - All values stored at leaf nodes
  - Linked list at leaf nodes so next constant time
- **B*-tree – like B+ but keep nodes $2/3$ full**
- **Prefix compression**
  - If keys alphabetically ordered, many will share prefix
- **String B-trees – for good worst-case behavior**
  - Can handle humongous keys that don't fit in blocks
  - Patricia trie data structure in nodes allows only necessary bits of keys to be compared

[AFS discussion]

Basic network security threats

- **Packet sniffing**
- **Packet forgery (spoofed from address)**
- **DNS spoofing – wrong IP address for hostname**
- **Assume "bad guy" controls network**
  - Can read all your packets
  - Can tamper with your packets
  - Can inject arbitrary new packets

Old authentication systems

- **Send password**
  - Ethernet sniffer collects everyone's password
- **Use IP address (`.rhosts`, NFS)**
  - Assume traffic from "privileged port" is root on host
  - Attacker can still forge packets
- **Use host name**
  - Worse than IP address (DNS insecurity)
- **One-time passwords**
  - Attacker can hijack TCP connection
  - If OTP derived from password, attacker can guess off-line

Keeping communications secret

- **Encryption guarantees secrecy**
  - Block ciphers (like AES)
  - Stream ciphers – block stream XORed with plaintext
  - Attacker cannot recover plaintext from ciphertext w/o K
- **Problem: Attacker can tamper with messages**
  - Stream ciphers – flip any bit
  - Block ciphers in CBC mode – corrupt a block, flip bit in next

Message authentication codes

- **Message authentication codes (MACs)**
  - Sender & receiver share secret key K
  - On message m, $\mathrm{MAC}(K, m) \rightarrow v$
  - Attacker cannot produce valid $\langle m, v \rangle$ without K
- **To send message securely, append MAC**
  - Send $\{m, \mathrm{MAC}(K, m)\}$, or encrypt $\{m, \mathrm{MAC}(K, m)\}_{K'}$
  - Receiver of $\{m, v\}$ checks $v \stackrel{?}{=} \mathrm{MAC}(K, m)\}$
- **Problem: Replay – don't believe previous** $\{m, v\}$

The Kerberos authentication system

- **Goal: Authentication in "open environment"**
  - Not all hardware under centralized control (e.g., users have "root" on their workstations)
  - Users require services from many different computers (mail, printing, file service, etc.)
- **Model: Central authority manages all resources**
  - Effectivaly manages human-readable names
  - User names: dm, waldman, ...
  - Machine names: class1, class2, ...
  - Must be assigned a name to use the system

Kerberos principals

- ***Principal*: Any entity that can make a statement**
  - Users and servers sending messages on network
  - "Services" that might run on multiple servers
- **Every kerberos principal has a key (password)**
- **Central key distribution server (KDC) knows all keys**
  - Coordinates authentication between other principals

Kerberos protocol

- **Goal: Mutually authenticated communication**
  - Two principals wish to communicate
  - Principals know each other by KDC-assigned name
  - Kerberos establishes shared secret between the two
  - Can use shared secret to encrypt or MAC communication (but most services don't encrypt, none MAC)
- **Approach: Leverage keys shared with KDC**
  - KDC has keys to communicate with any principal

Protocol detail

- **To talk to server s, client c needs key & ticket:**
  - Session key: $K_{s,c}$ (randomly generated key KDC)
  - Ticket: $T = \{s, c, \text{addr}, \text{expire}, K_{s,c}\}_{K_s}$
    (K_S is key s shares with KDC)
  - Only server can decrypt T
- **Given ticket, client creates authenticator:**
  - Authenticator: $T, \{c, \text{addr}, \text{time}\}_{K_{s,c}}$
  - Client must know $K_{s,c}$ to create authenticator
  - T convinces server that $K_{s,c}$ was given to c
- **"Kerberized" protocols begin with authenticator**
  - Replaces passwords, etc.

Getting tickets in Kerberos

- **Upon login, user fetches "ticket-granting ticket"**
  - $c \rightarrow t$: c, t (t is name of TG service)
  - $t \rightarrow c$: $\{K_{c,t}, T_{c,t} = \{s, t, \text{addr}, \text{expire}, K_{s,c}\}_{K_t}\}_{K_c}$
  - Client decrypts with password ($K_c = \text{SHA-1}(\text{pwd})$)
- **To fetch ticket for server** s
  - $c \rightarrow t$: $s, T_{c,t}, \{c, \text{addr}, \text{time}\}_{K_{c,t}}$
  - $t \rightarrow c$: $\{T_{s,c}, K_{s,c}\}_{K_{c,t}}$
- **To achieve mutual authentication with server:**
  - $c \rightarrow s$: $T_{s,c}, \{c, \text{addr}, \text{time}\}_{K_{s,c}}$
  - $s \rightarrow c$: $\{\text{time} + 1\}_{K_{s,c}}$

Authentication in AFS

- **User logs in, fetches kerberos ticket for AFS server**
- **Hands ticket and session key to file system**
- **Requests/replies accompanied by an authenticator**
  - Authenticator includes CRC checksum of packets
  - Note: CRC is not a valid MAC!
- **What about anonymous access to AFS servers?**
  - User w/o account may want universe-readable files

AFS permissions

- **Each directory has ACL for all its files**
  - Precludes cross-directory links
- **ACL lists principals and permissions**
  - Both “positive” and “negative” access lists
- **Principals: Just kerberos names**
  - Extra principles, system:anyuser, system:authuser
- **Permissions: rwlidak**
  - read, write, lookup, insert, delete, administer, lock

Kerberos inconvenience

- **Large (e.g., university-wide) administrative realms**
  - University-wide administrators often on the critical path
  - Departments can't add users or set up new servers
  - Can't develop new services without central admins
  - Can't upgrade software/protocols without central admins
  - Central admins have monopoly servers/services
    (Can't set up your own without a principal)
- **Crossing administrative realms a pain**
- **Ticket expirations**
  - Must renew tickets every 12–23 hours
  - Hard to have long-running backround jobs

Security issues with kerberos

- Spoofing local login
- KDC vulnerability
- Kinit could act as oracle
- Replay attacks
- Off-line password guessing
- Can’t securely change compromised password