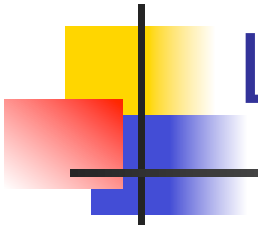
Operating Systems Design (CS 423)

Elsa L Gunter

2112 SC, UIUC

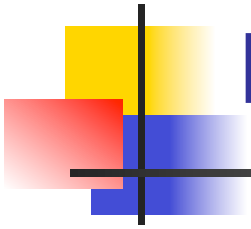
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum



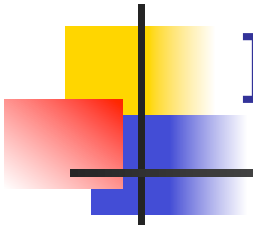
Linked List

- Each block contains pointer to next block of file (along with data)
  - Used by Alto (first personal computer)
- File header contains pointer to first disk block



Linked List

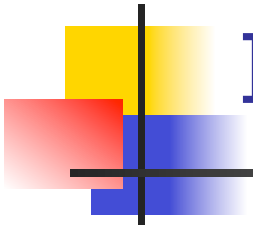
- Pros
  - Grow easily (i.e. append) files
  - No external fragmentation (pick any free block)
- Cons
  - Sequential access quite slow
  - Lots of seeks between blocks
  - Random access is really slow



Indexed Files

- User (or system) declares max # of blocks in file
- System allocates file header with array of pointers big enough to point to that number of blocks
- Extra level of indirection, like a page table

File Block #	Disk Block #
0	18
1	50
2	3
3	22

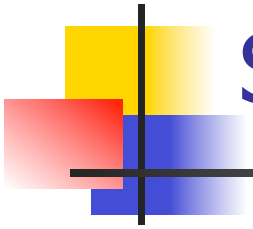


Indexed Files

```
#define FS_BLOCKSIZE 1024
#define FS_MAXFILEBLOCKS 253
#define FS_MAXUSERNAME 7
typedef struct {
 char owner[FS_MAXUSERNAME + 1];
 int size; // size of the file in bytes
 int blocks[FS_MAXFILEBLOCKS]; // array of file blocks
} fs_inode; (note sizeof(fs_inode) == FS_BLOCKSIZE)
disk_readblock(int diskBlockNo, void *buf);
lookup_inode(char *fileName, fs_inode *inode);
```

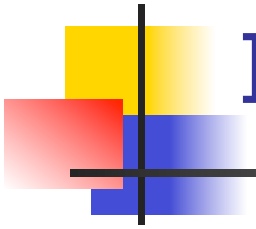
Write code for reading a file block for a given file name

```
fs_readblock(char *fileName, int fileBlockNo, void *buf)
```



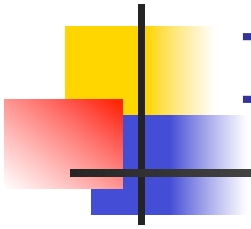
Solution

```
fs_readblock(char *fileName, int fileBlockNo, void *buf) {
   fs_inode inode;
   lookup_inode(fileName, &inode);
   // may involve many disk reads

   // make sure we got an inode back

   // do some error checking to validate

   disk_read_block(inode.blocks[fileBlockNo], buf);
}
```



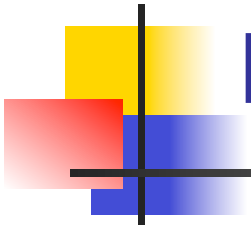
Indexed Files

- Pros
  - Can easily grow (up to # of blocks allocated in header)
  - Easy random access of loc. Calculation
- Cons
  - Lots of seeks for sequential access
    - How can you make this faster without pre-allocation?
  - Can't easily grow beyond # blocks allocation



Indexed Files

- Pros
  - Can easily grow (up to # of blocks allocated in header)
  - Easy random access of loc. Calculation
- Cons
  - Lots of seeks for sequential access
    - How can you make this faster without pre-allocation?
    - Try to keep sequential access in same cylinder on disk
  - Can't easily grow beyond # blocks allocation

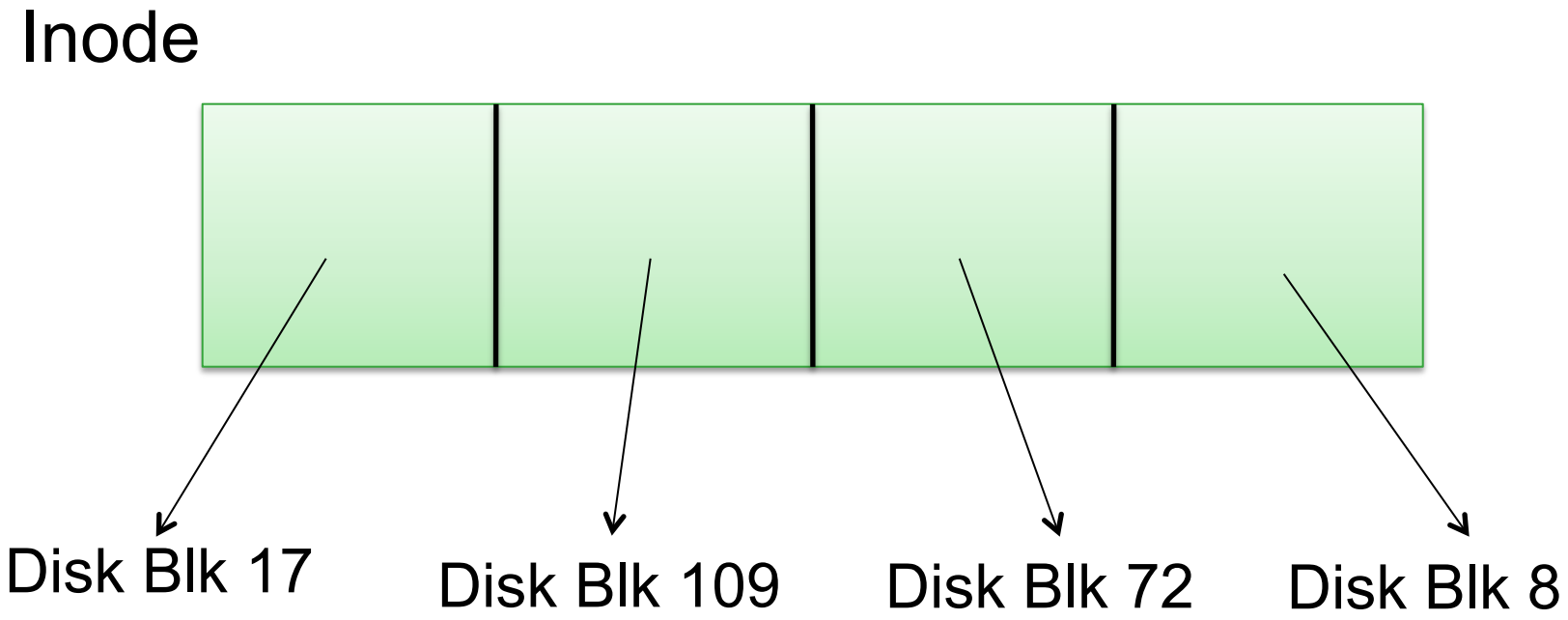


Large Files

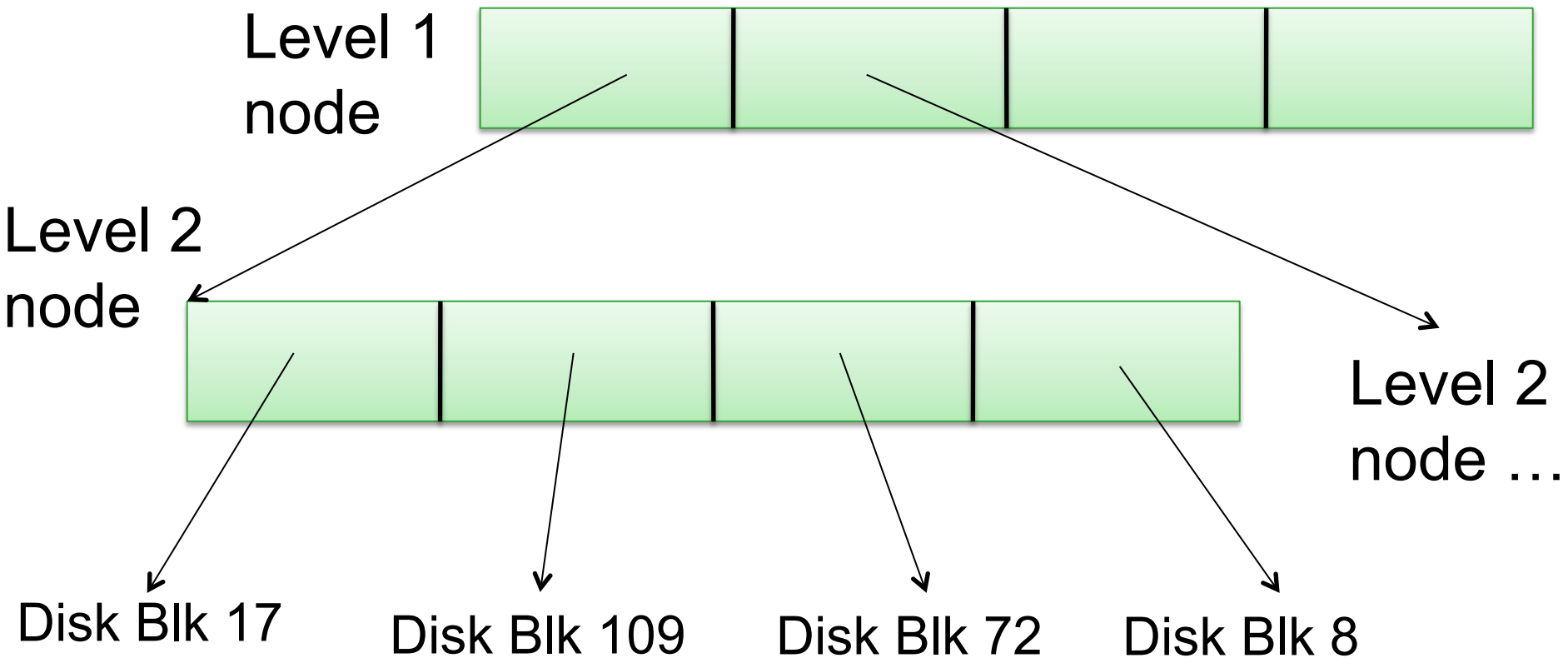
- How to deal with large files?
  - Could you assume file might get really large, allocate lots of space in file header?
  - Could you use larger block size, eg4MB?
- Solution: more sophisticated data structure for file header

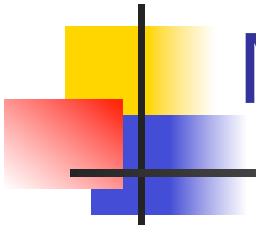
Indexed Files

- Indexed files are like a shallow tree



Muti-level Indexed Files



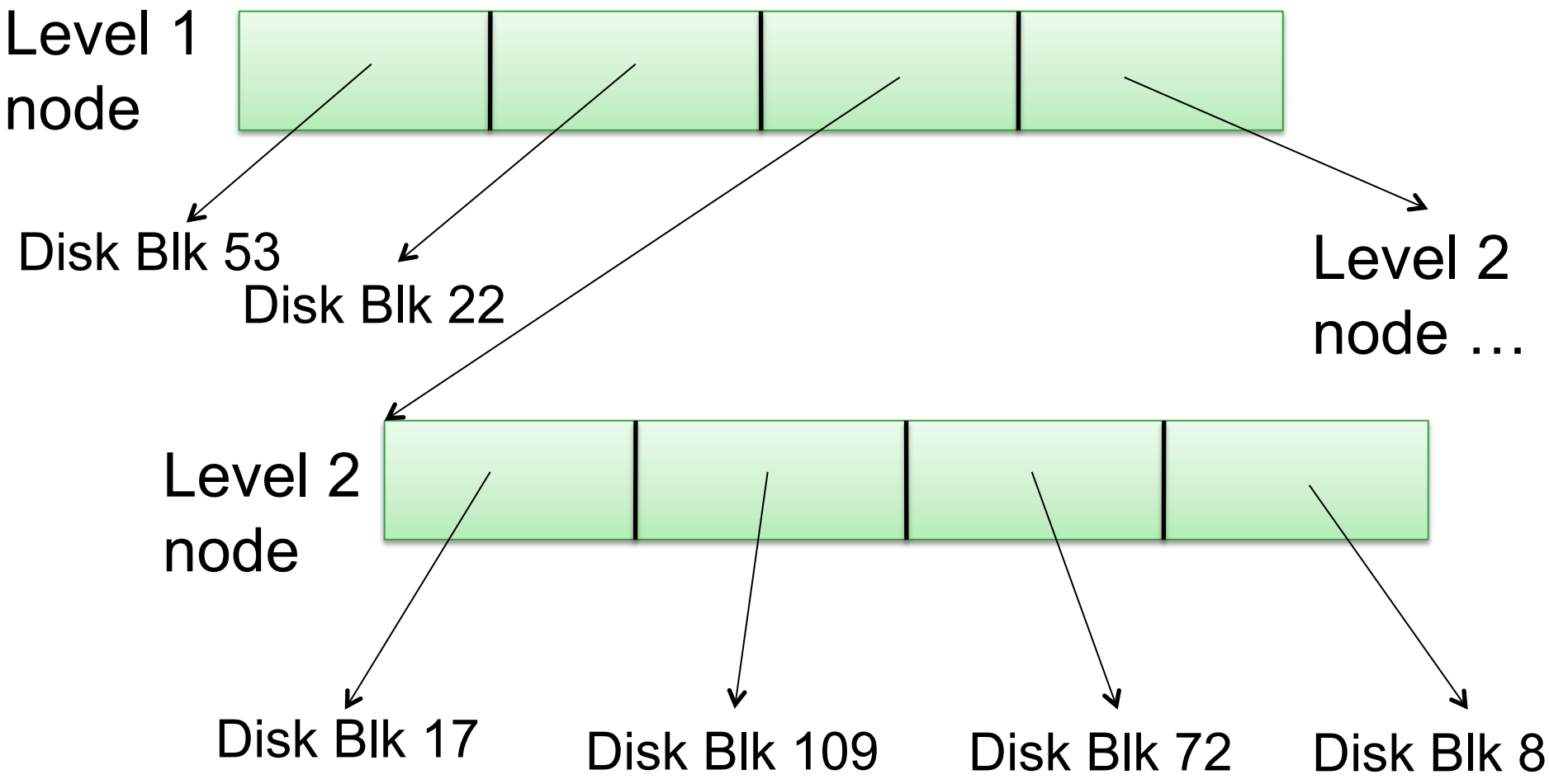


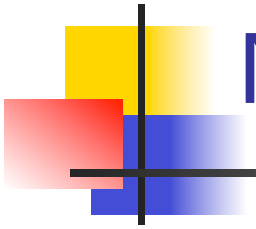
Muti-level Indexed Files

- How many disk accesses to get 1 block of data?

- How do you solve this?

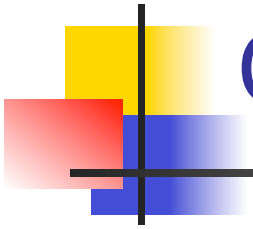
Non-Uniform Muti-level Indexed Files





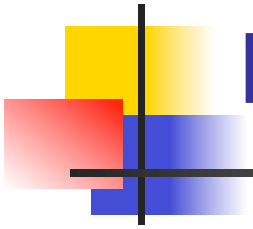
Non-Uniform Muti-level Indexed Files

- Pros
  - Files can expand easily
  - Small files don’t pay full overhead of deep trees
- Cons
  - Lots of indirect blocks for big files
  - Lots of seeks for sequential access



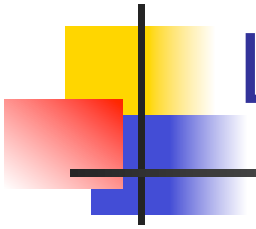
On Disk File Ftructures

- Could have other dynamically allocated data structures for file header
- Key feature: have location of file header on disk NOT change when file grows
  - Why?



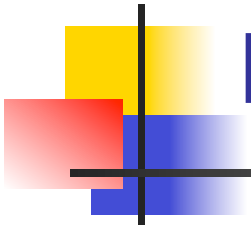
Naming Files

- How do you specify which file you want to access?
  - Eventually OS must find file header you want on disk
  - Need disk block address (number)
- Typically user uses symbolic name
  - OS translates name to numeric file header
  - Possible alternative is to describe contents of file



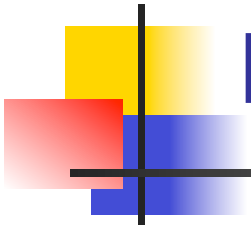
Locating File Header Disk Block

- Could use hash table, expandable array
  - Key is finding disk block number of file inode; then getting contents is easy
- Data structure for mapping file name to inode block number is called a **Directory**



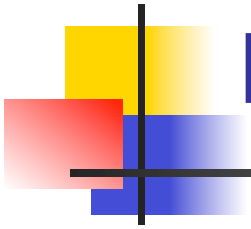
Directories

- Directory – mapping for set of files
  - Name->file header's disk block # for that file
  - Often simple array of(name, file header's disk block #) entries
  - Table is stored in a normal file as normal datd
  - Eg: **`ls`** implemented by reading file and parsing contents



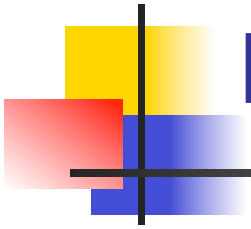
Directories

- Often treat directories and files in same way
  - Same storage structure for data
  - Directory entry points to either "ordinary" file or another directory
- Can we allow user to read/write directories directly, arbitrarily?



Directory Organization

- Directories typically have hierarchical structure
  - Directory **A** has mapping to files and *directories* in directory **A**
- `/home/cs423/index.html`
- `/` is root directory
  - Contains list of root's contents, including `home`
  - For each elt, has mapping form name to file inode disk block #]
    - Including `home`



Directory Organization

- `home` is directory entry within / dir
  - Contains list of files and directories
  - One dir in `home` is `cs423`
- `/home/cs423` names directory within `/home` directory
  - Contains list of files and directories
  - One file is `index.html`
  - How many disk I/Os to access first bytes of `/home/cs423/index.html` assuming no caching