

# **CS414**

# **Minithreads project 3**

# **overview**

Ben Atkin

`batkin@cs.cornell.edu`

# Aim of the project

- Implement a Unix-style file system on an emulated disk
  - Directory organisation
  - Free space management
  - File allocation
  - Extra credit: multithreaded file system

# What you have to do

- Define the superblock, inodes
- Implement directory lookup
- Create and initialise a disk
- Implement
  - free space management
  - path name translation
  - indexed allocation
- Implement the file system interface

# The superblock

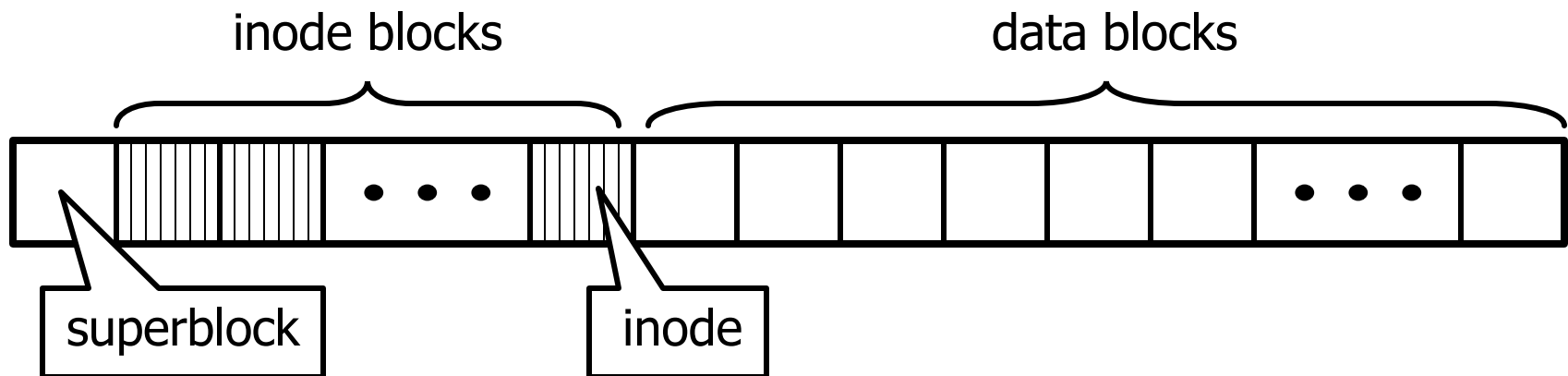
- Stores disk parameters
  - Free space pointers
  - Proportion of inodes
- Read into memory on startup
- Write back on `shutdown`

# Inode contents

- File type (directory, regular file, ...)
- File size in bytes
- Direct block pointers
- Indirect pointer
- Free list pointer?

# Inodes versus data blocks

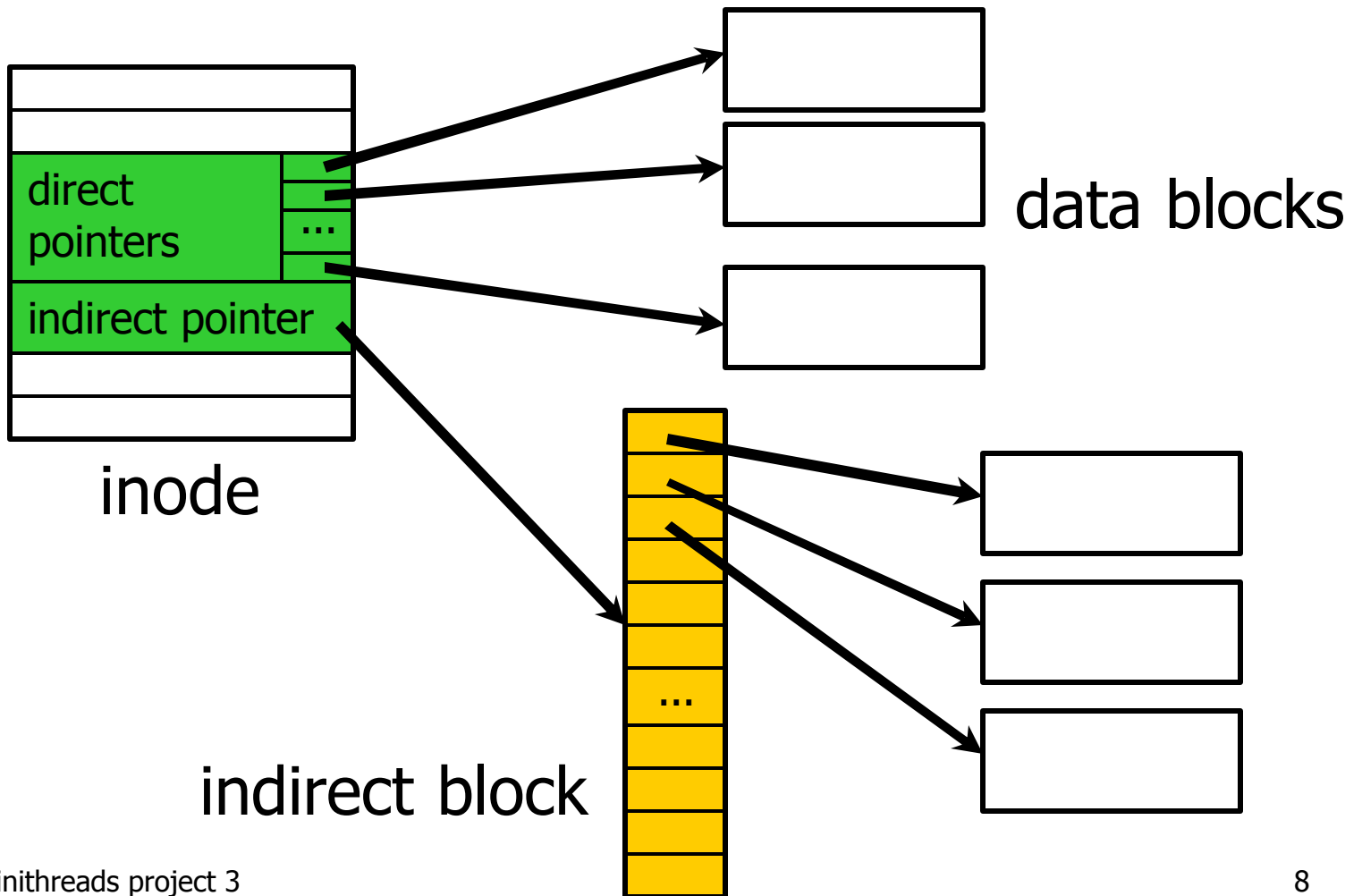
- Multiple inodes can fit in one block
- Finding inodes is easiest if there's an FID-to-block number mapping
  - "inode blocks" at start, then data blocks



# Inodes versus data blocks

- How many inodes are required?
  - Assume each file has at least one data block
  - `INODESPERBLOCK = DISK_BLOCK_SIZE / sizeof(struct inode)`
  - Proportion of blocks devoted to inodes:  
`disk_size / (1 + INODESPERBLOCK)`

# Indexed file structure



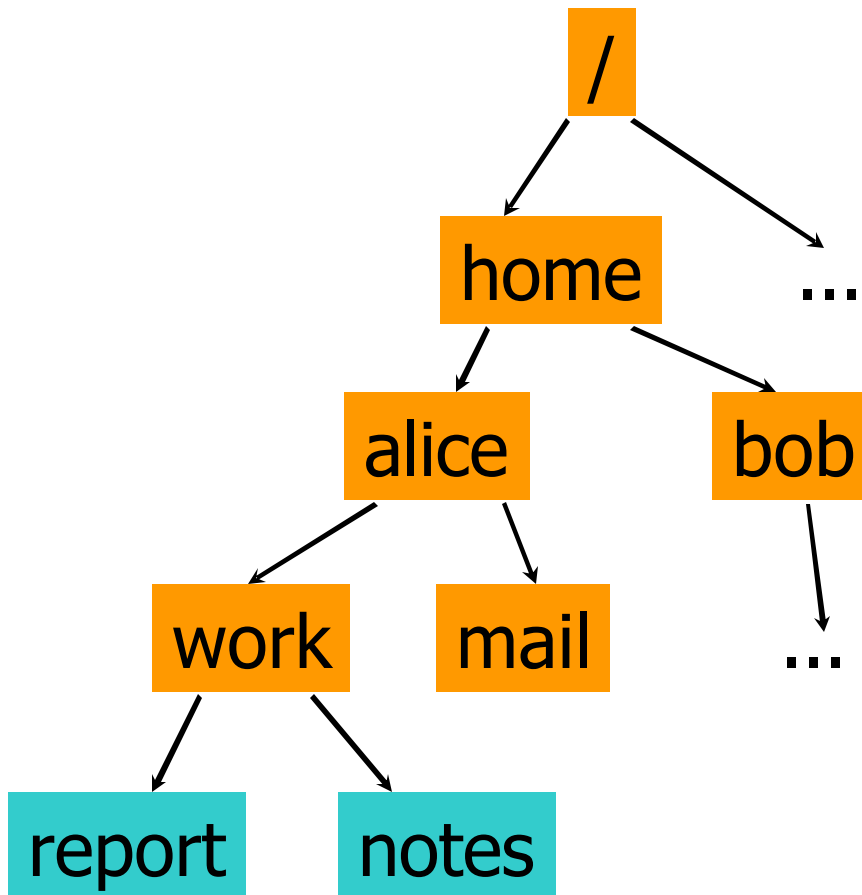
# Free space management

- Free inodes and free data blocks are separate
  - Two pointers in superblock
- No need to worry about crashes
  - Superblock and disk need not be consistent
  - Can write back superblock on **shutdown**

# Directory format

- Just use regular files to hold directories
  - Special `DIRECTORY` file type
  - Internally an array of name, FID pairs
- Fixed-size directory entries are enough
  - Filenames should be up to 256 characters

# Pathname translation



Iteratively read directory from disk and do translate next component of path.

# Disk interface

```
typedef struct {
    struct {
        int size;      /* number of blocks on the disk */
        /* ... */
    } layout;
    /* ... */
} disk_t;

int  disk_create(disk_t *disk, char *name, int size,
                int flags)
int  disk_startup(disk_t *disk, char *name)
int  disk_shutdown(disk_t *disk)

int  disk_read_block(disk_t *disk, int bn, char *buf)
int  disk_write_block(disk_t *disk, int bn, char *buf)
```

# Creating the disk

- Write a `mkfs` application to create and initialise disk
  - `disk_create(&d, "MINIFILESYSTEM", 1024, 0)`
  - Initialise superblock
  - Initialise free inodes and blocks
  - Create the root directory
  - Terminate and `shutdown` disk

# Opening files

- Multiple threads can open the same file concurrently
  - Need "per-thread" open file table and global open file table
  - Define a `struct filedesc` for thread's per-file state (e.g. file pointer)
  - Define a `struct openfile` for global open file state (e.g. inode, etc)

# Closing files and shutdown

- When a file is closed, write back inode
- Before terminating application
  - Close all open files
  - Flush all changes to disk
  - Call `disk_shutdown`

# Synchronous versus asynchronous disks

- There are two versions of the disk
- Synchronous version:
  - `disk.sync.h, disk.sync.c`
  - Only one disk access at a time
  - Easy to program with
- Asynchronous version
  - `disk.async.h, disk.async.c`
  - Use this version for "extra credit"

# Synchronisation with a synchronous disk

- Still require consistent access to file system data structures
  - No parallelism in performing disk operations
  - One mutex is enough

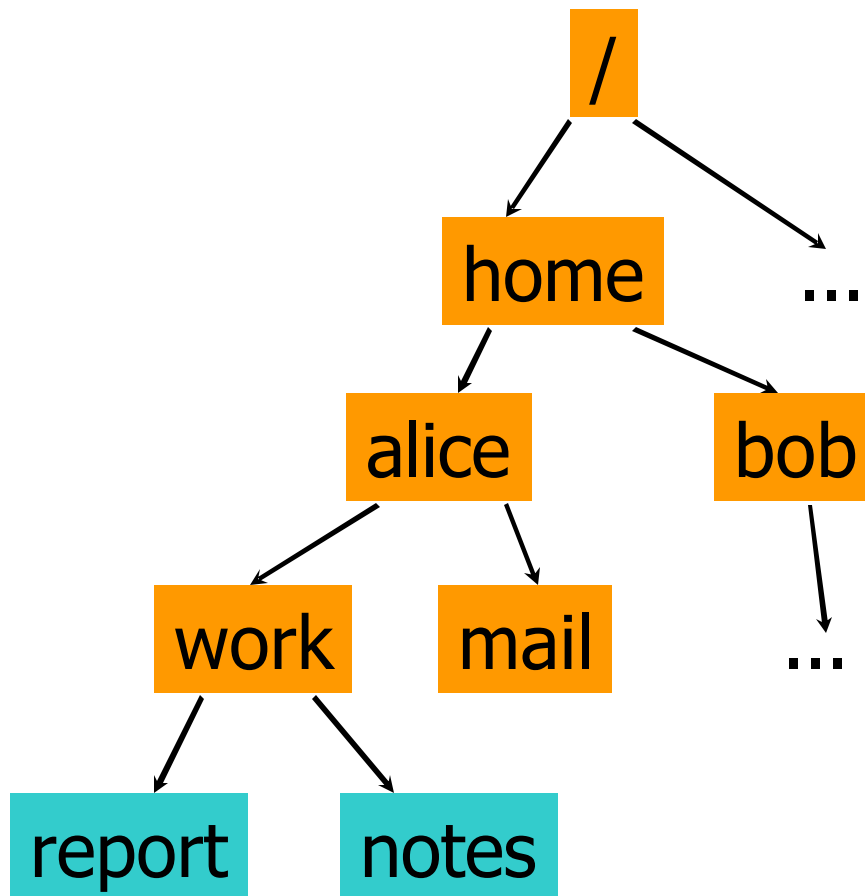
# The asynchronous disk

- Disk requests are nonblocking
  - Interrupt to indicate request completion
  - `disk_handler(disk, request, reply)`
  - Requires separate initialisation
- Multiple requests can be queued on disk simultaneously
  - (potential for disk scheduling)
  - Can make file system multithreaded

# Synchronisation with an asynchronous disk

- File-level locking is mandatory
  - Multiple threads should be able to read/write different files concurrently
  - Must avoid deadlocks between operations modifying multiple files or directories
  - Potential deadlock on path name translations

# Synchronisation examples



e.g. rename  
/home/alice/work/report  
to /home/alice/mail/report

e.g. rename  
/home/alice/work/report  
to /home/alice/report