File System Design: advanced topics

David E. Culler

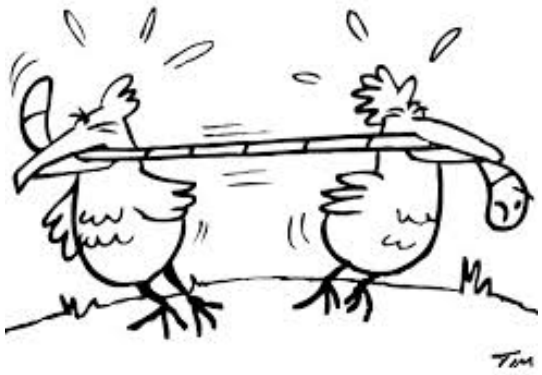
CS162 – Operating Systems and Systems Programming

Lecture 25

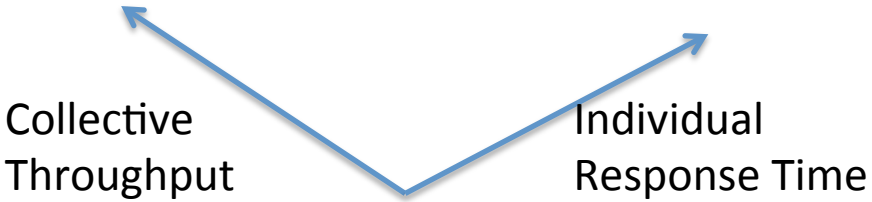
October 27, 2014

Reading: A&D 13.3, 9.6
HW 4 due 10/27
Proj 2 final 11/07

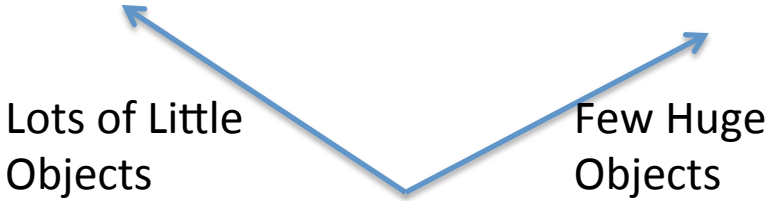
Big Thought-provoking Questions



- The One vs The All

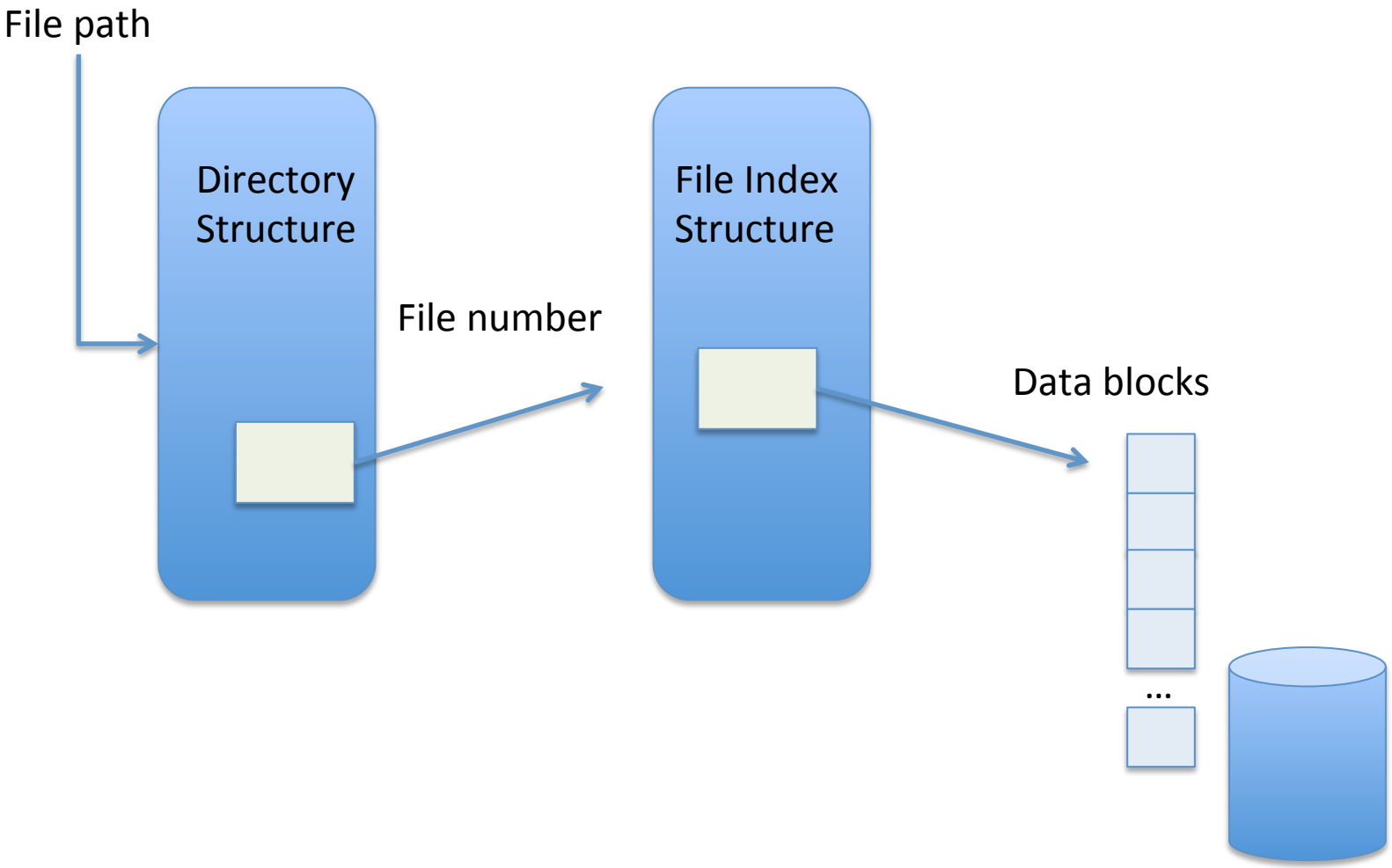


- The Many vs The Few
  - Guided by workload measurements



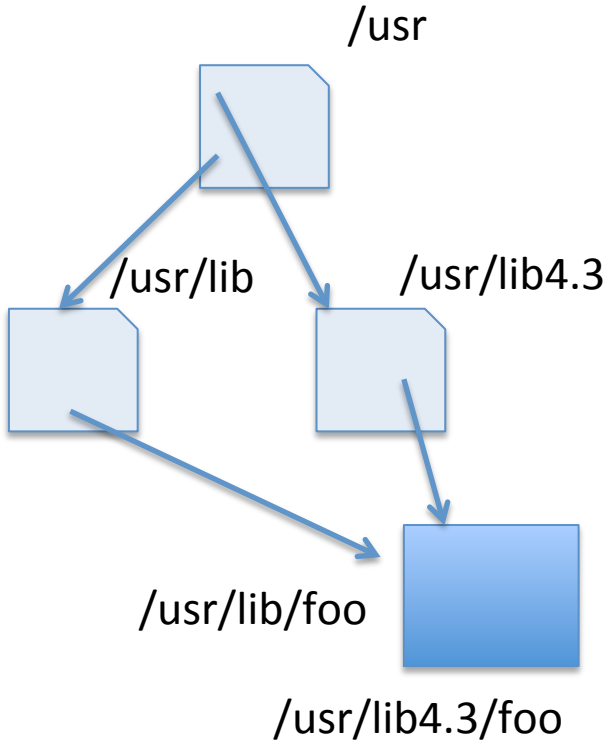
- Simplicity vs Versatility
  - Fixed blocks vs Variable extents
- Reliability vs Performance

Recall: Components of a File System



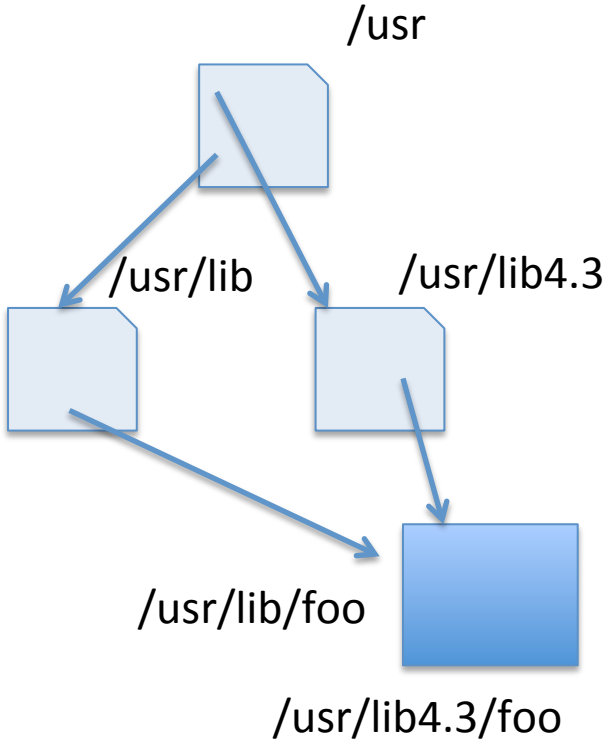
Recall: directories

- Stored in files, can be read, but don't
  - System calls to access directories
  - Open / Creat traverse the structure
  - mkdir /rmdir add/remove entries
  - Link / Unlink
    - Link existing file to a directory
      - Not in FAT !
    - Forms a DAG
- libc support
  - DIR * opendir (const char *dirname)
  - struct dirent * readdir (DIR *dirstream)
  - int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)



When can a file be deleted ?

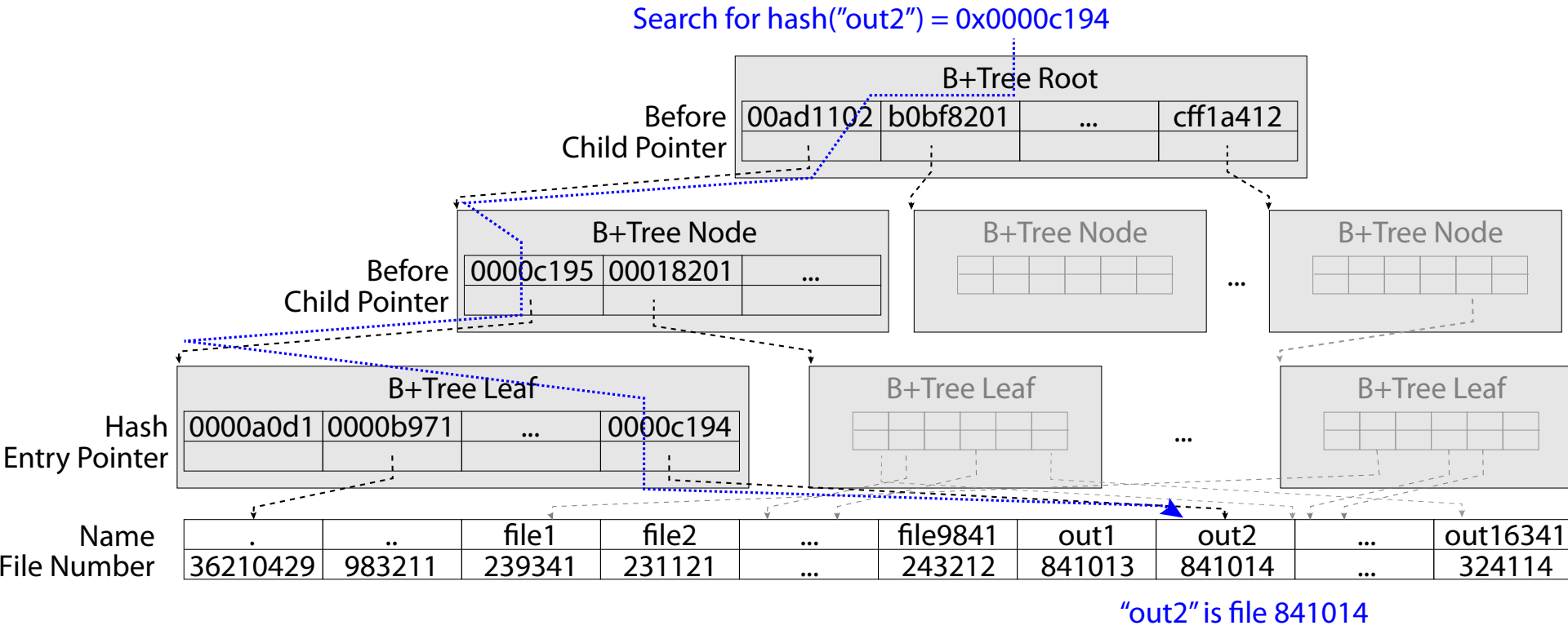
- Maintain reference count of links to the file.
- Delete after the last reference is gone.



Links

- Hard link
  - Sets another directory entry to contain the file number for the file
  - Creates another name (path) for the file
  - Each is “first class”
- Soft link or Symbolic Link
  - Directory entry contains the name of the file
  - Map one name to another name

Large Directories: B-Trees



Data Structure Trade-offs

- Contiguous arrays
  - FAT, inode tables, disk blocks, … , page tables, …
  - Direct index (constant time access), linear search
  - Compact, easy to grow – up to a limit
- Linked lists
  - Simple, Relatively compact
  - Linear time index or search => good for few
- Tree-like structures (tree, b-tree, …, inode, …)
  - Directories, … , Multi-level page tables
  - Complex, Multiple Pointers (but mix in direct)
  - Log time index or search

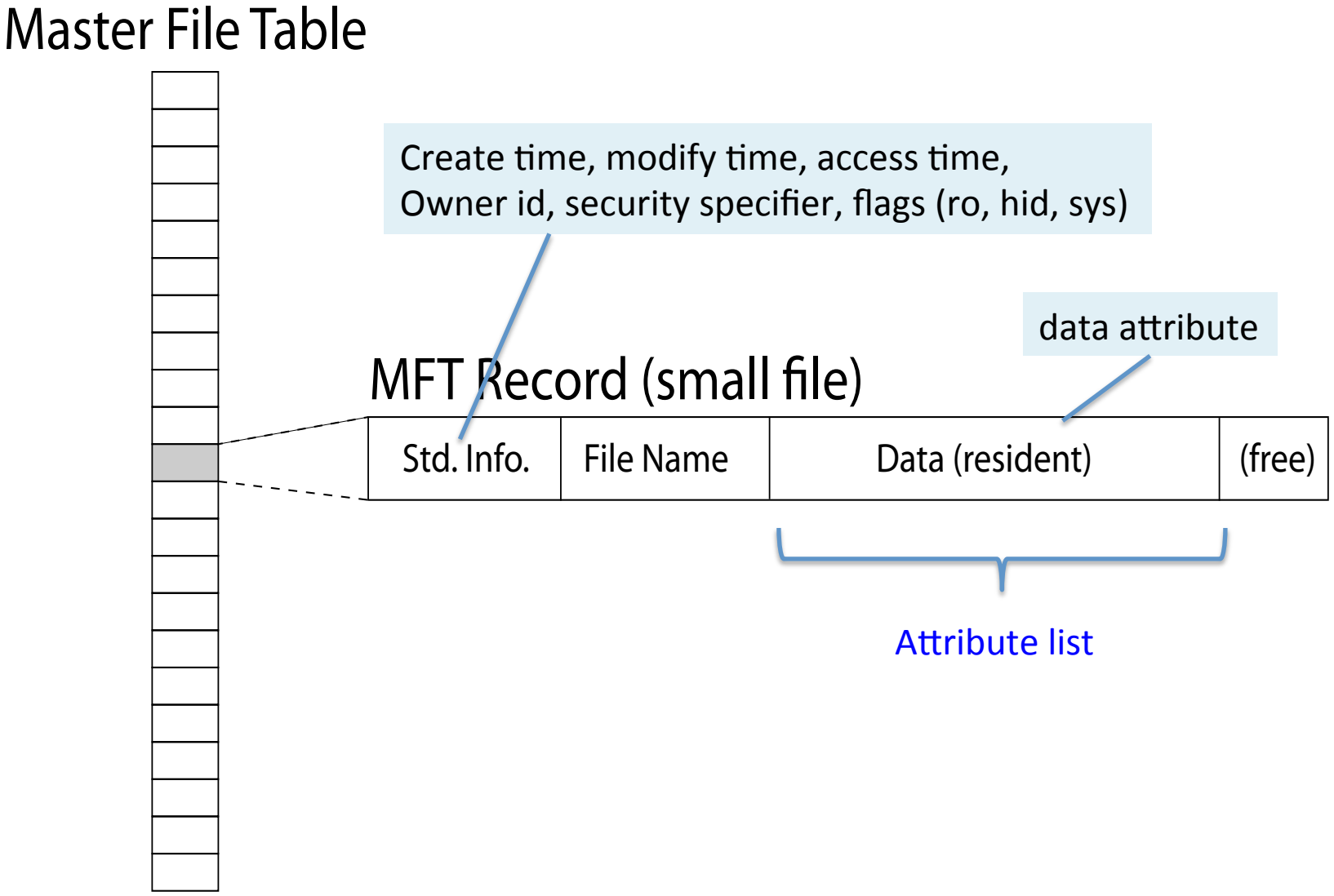
NTFS

- Variable length extents
  - Rather than fixed blocks
- Everything (almost) is a sequence of <attribute:value> pairs
  - Meta-data and data
- Mix direct and indirect freely

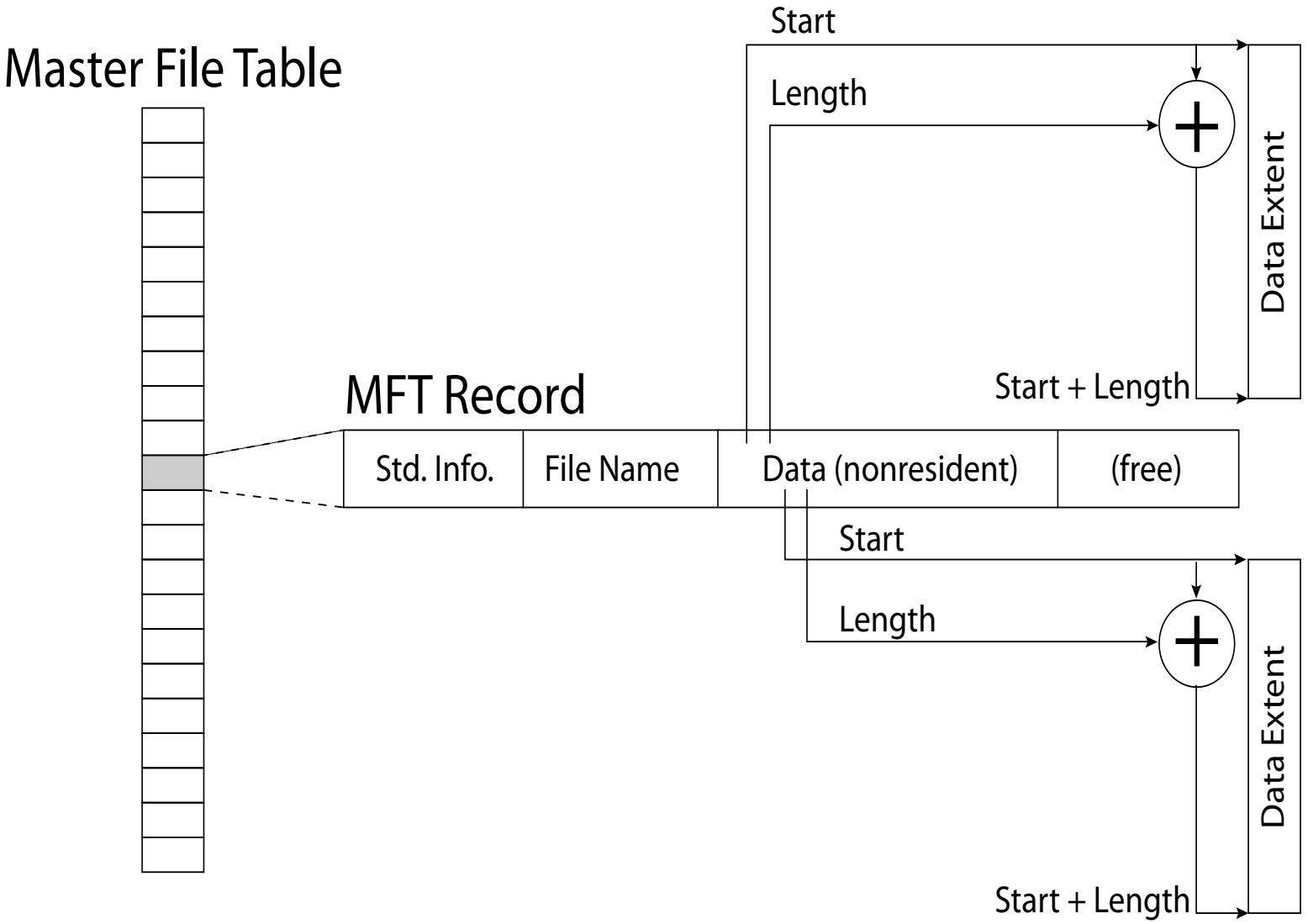
NTFS

- Master File Table
  - Flexible 1KB storage for metadata and data
  - Variable-sized attribute records (data or metadata)
  - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
  - Block pointers cover runs of blocks
  - Similar approach in linux (ext4)
  - File create can provide hint as to size of file
- Journaling for reliability
  - Discussed next lecture

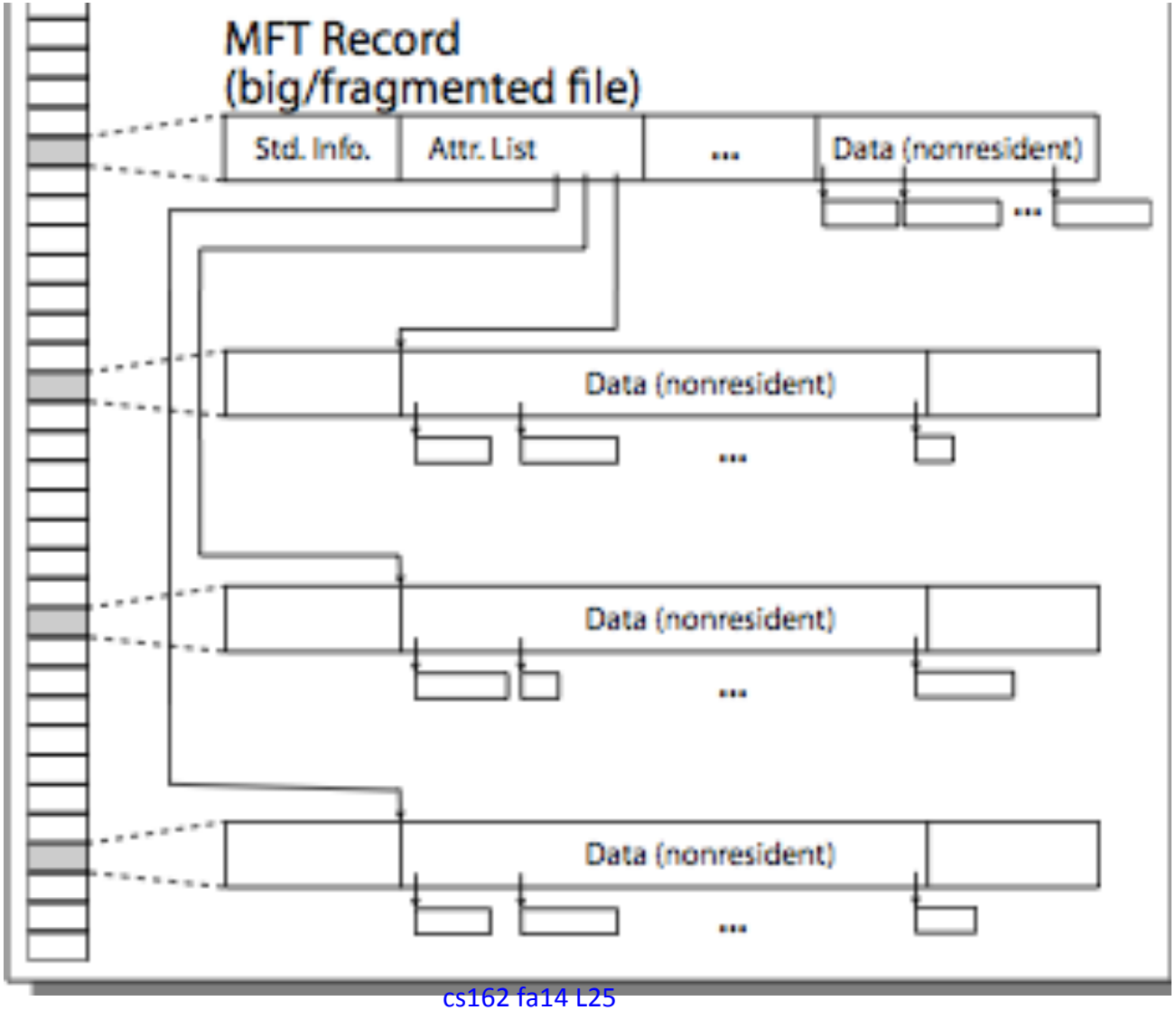
NTFS Small File

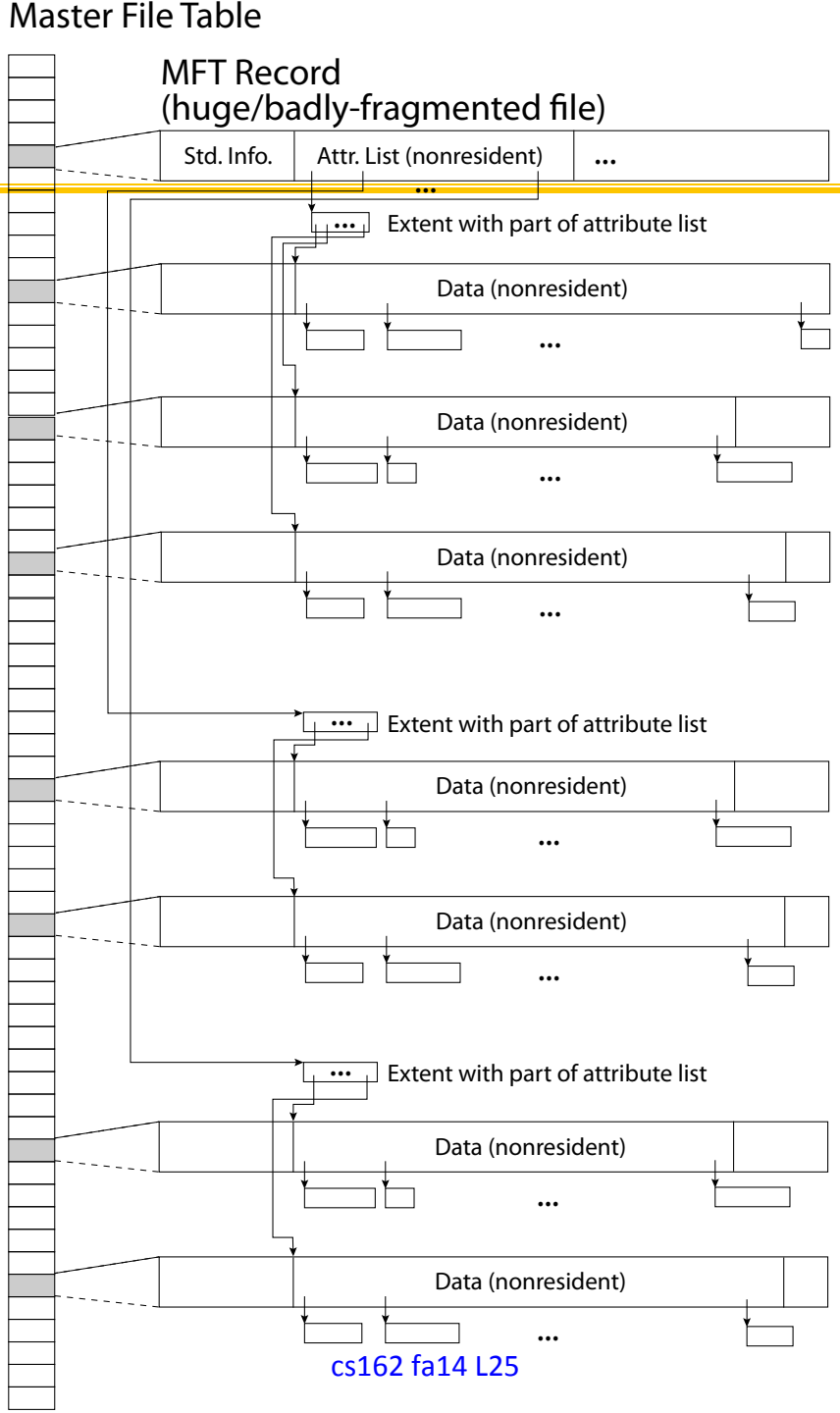


NTFS Medium File



NTFS Multiple Indirect Blocks




Master File Table
MFT Record
(huge/badly-fragmented file)
Std. Info.
Attr. List (nonresident)
...
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)
Extent with part of attribute list
Data (nonresident)
Data (nonresident)

Quizzie: File Systems

- Q1: True _ False _ A hard-link is a pointer to other file
- Q2: True _ False _ inumber is the id of a block
- Q3: True _ False _ Typically, directories are stored as files
- Q4: True _ False _ Storing file headers on the outermost cylinders minimizes the seek time

Quizzie: File Systems

- Q1: True _ False _**x** A hard-link is a pointer to other file
- Q2: True _ False _**x** inumber is the id of a block
- Q3: True **x** False _ Typically, directories are stored as files
- Q4: True _ False _**x** Storing file headers on the outermost cylinders minimizes the seek time

Towards Copy-on-Write

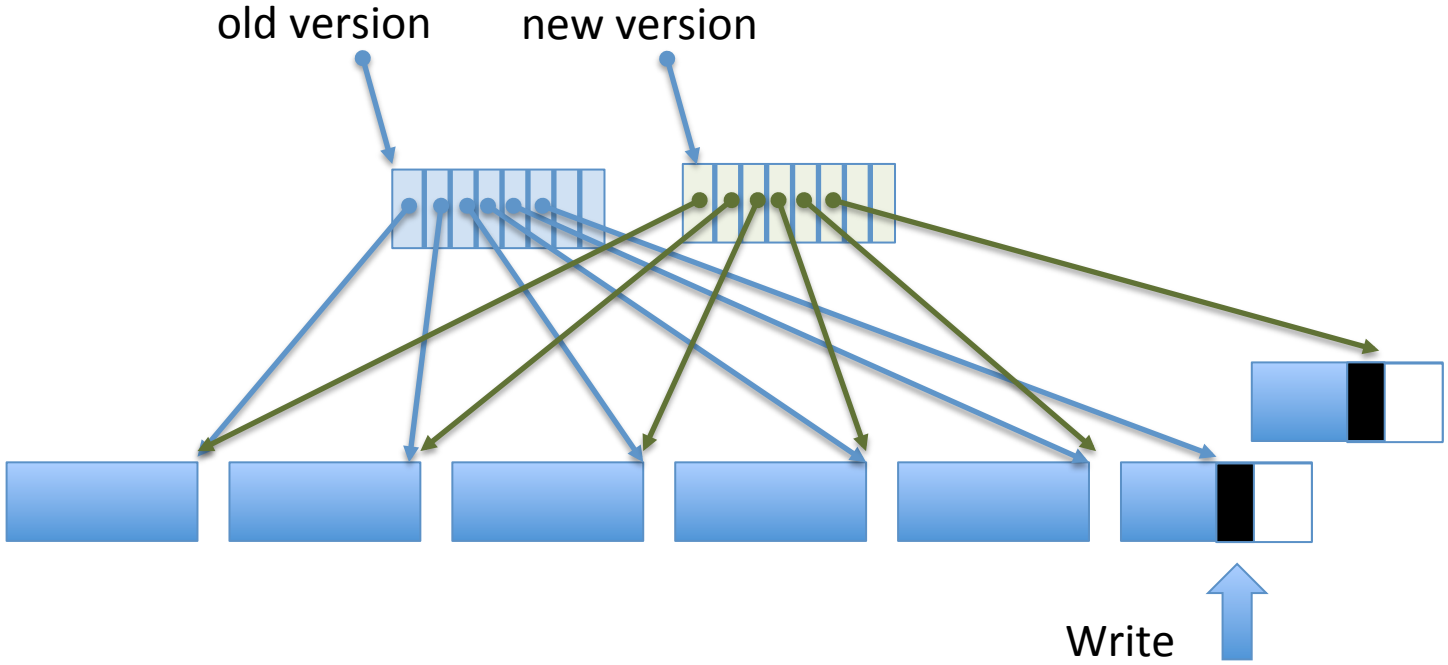
- *Files* are for durable storage AND flexible process-independent, protected namespace
- Files grow incrementally as written
  - Update-in-place file systems start with a basic chunk and append (possibly larger) chunks as file grows
  - Transition from random access to large sequential
- *Disks trends:* huge and cheap, high startup
- *Design / Memory trends:* cache everything
  - Reads satisfied from cache, buffer multiple writes and do them all together
- Application trends: make multiple related changes to a file and commit all or nothing

Emulating COW @ user level

- Transform file **foo** to a new version
- Open/Create a new file **foo.v**
  - where v is the version #
- Do all the updates based on the old **foo**
  - Reading from **foo** and writing to **foo.v**
  - Including copying over any unchanged parts
- Update the link
  - ln –f foo foo.v

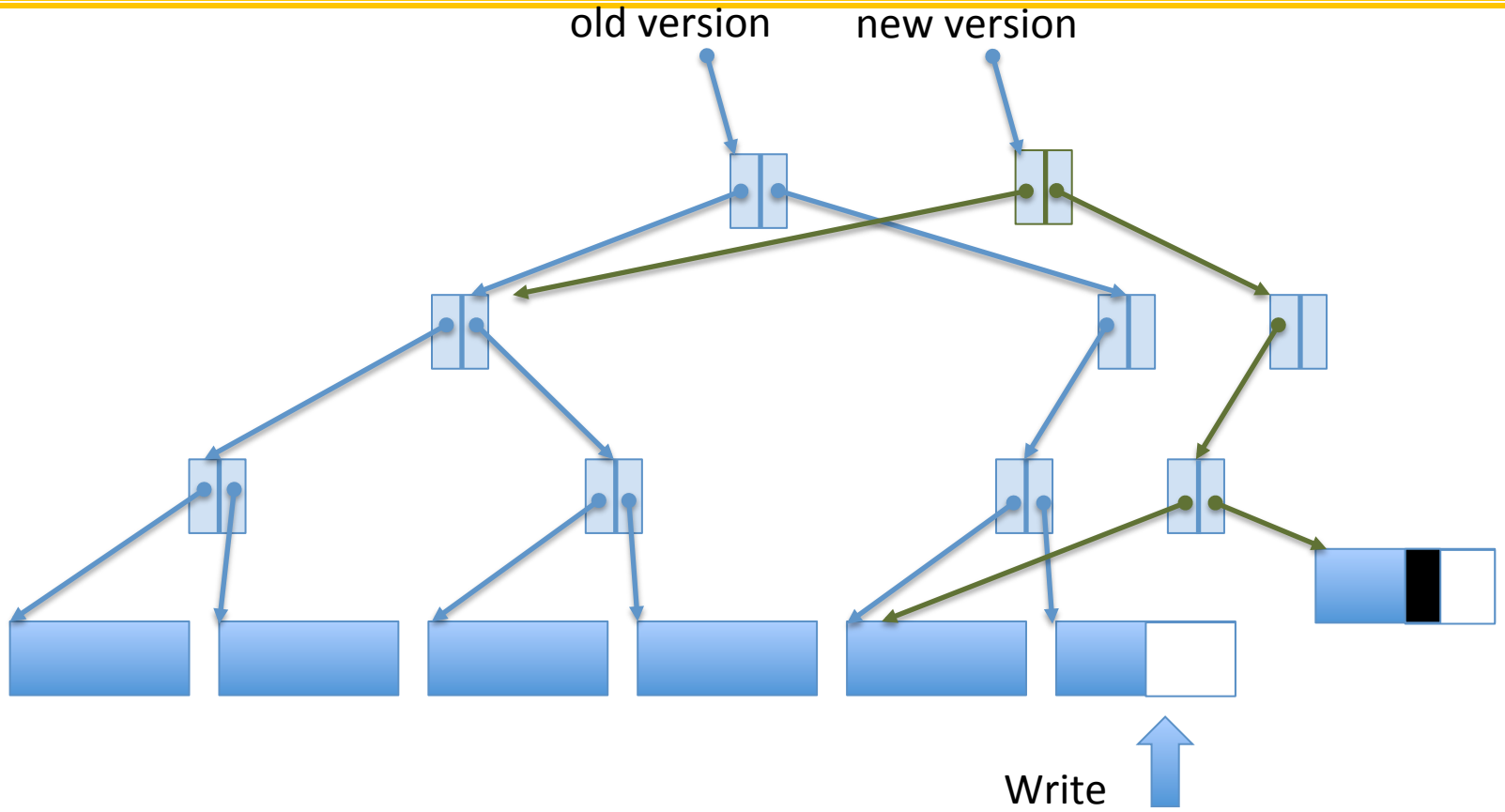
- Does it work?

Creating a New Version



- If file represented as a tree of blocks, just need to update the leading fringe

Creating a New Version

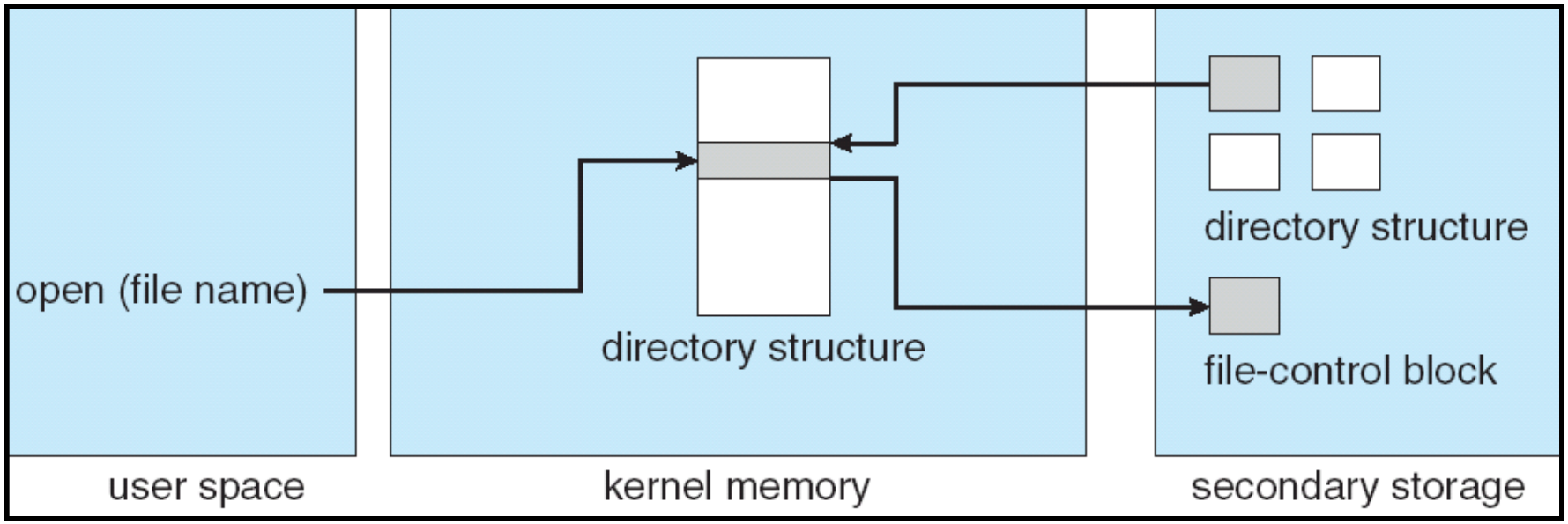


- If file represented as a tree of blocks, just need to update the leading fringe

ZFS

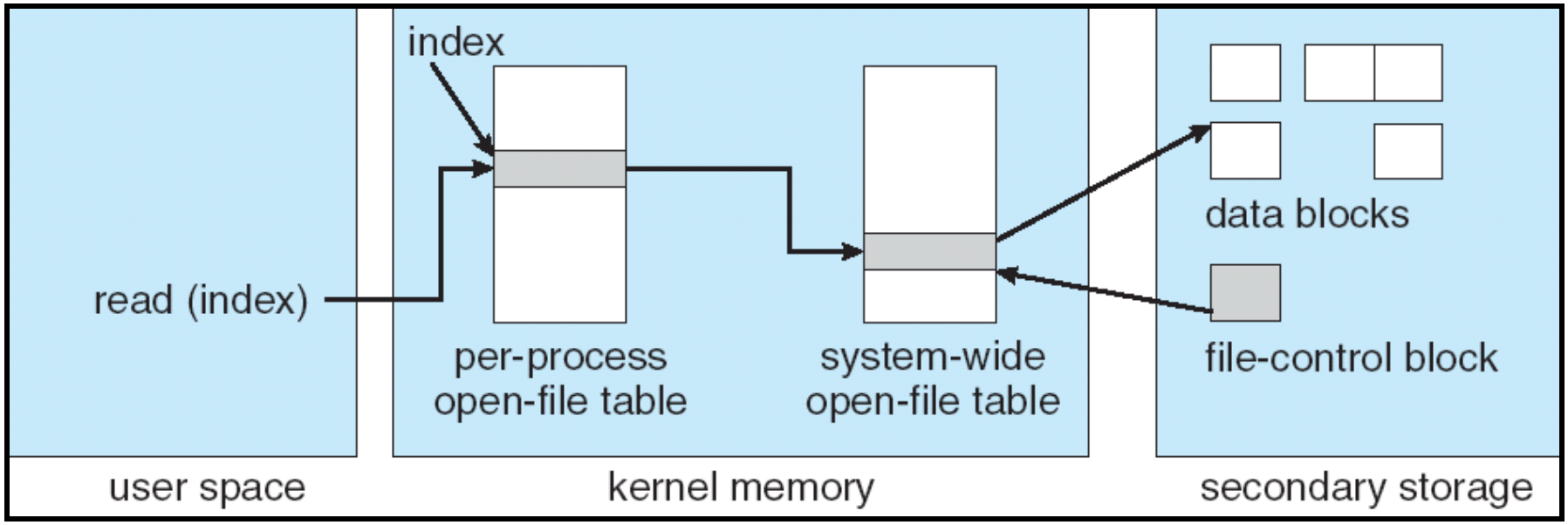
- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table

In-Memory File System Structures

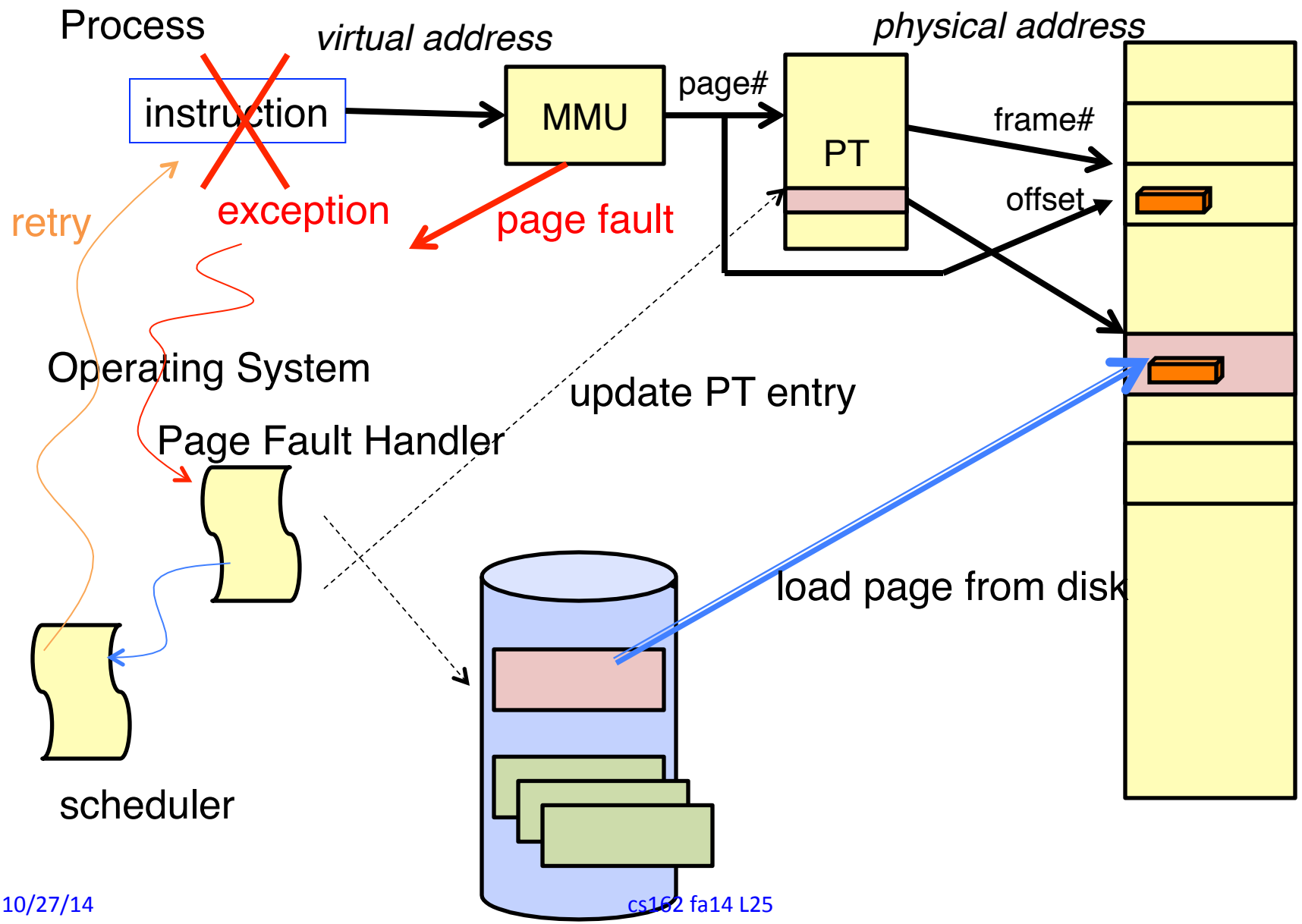


- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

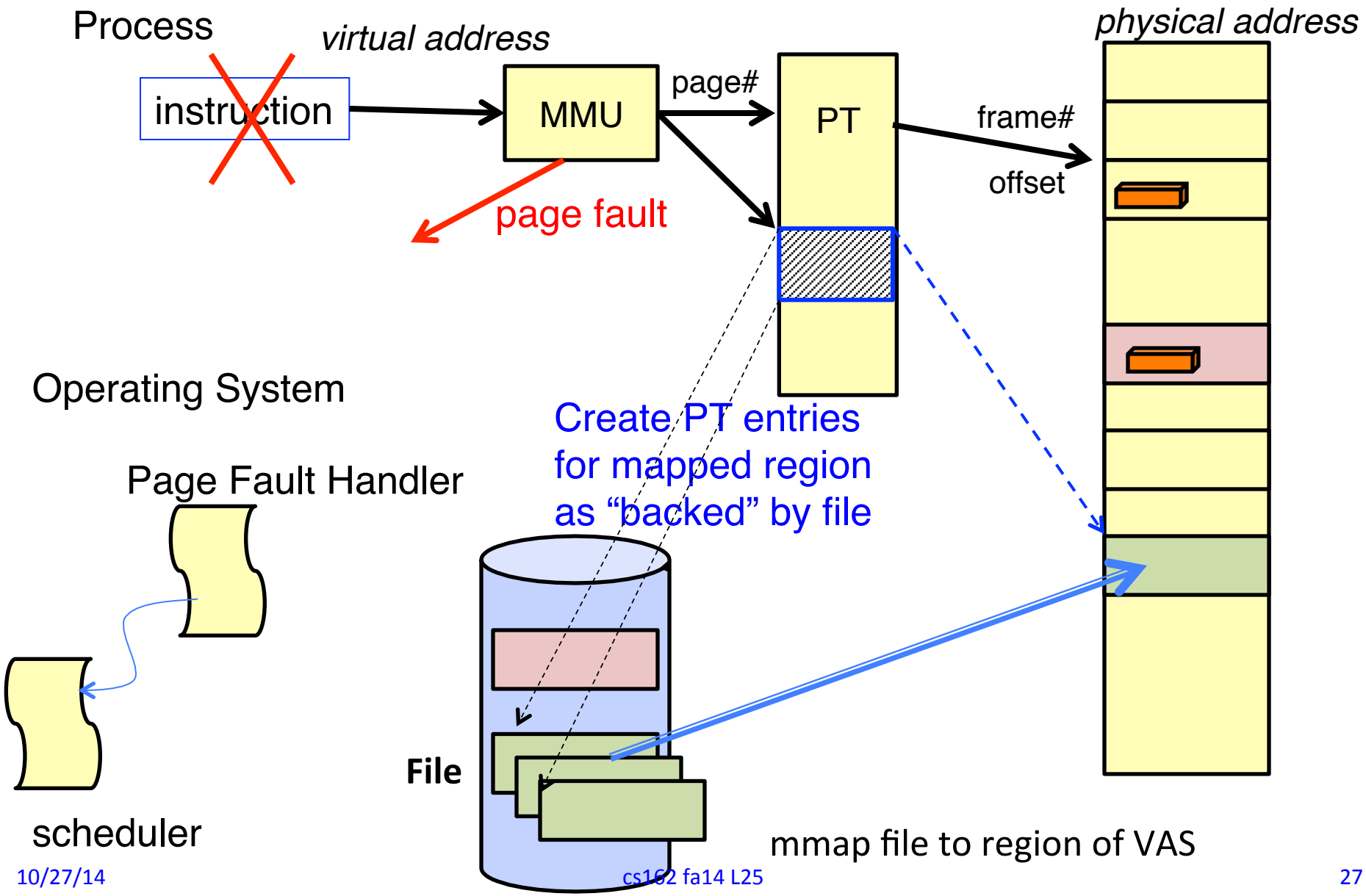
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to regions of a file
  - This involves multiple copies into caches in memory, plus system calls
- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out
- Executable file is treated this way when we exec the process !!

Recall: Who does what when ?



Using Paging to mmap files



mmap system call

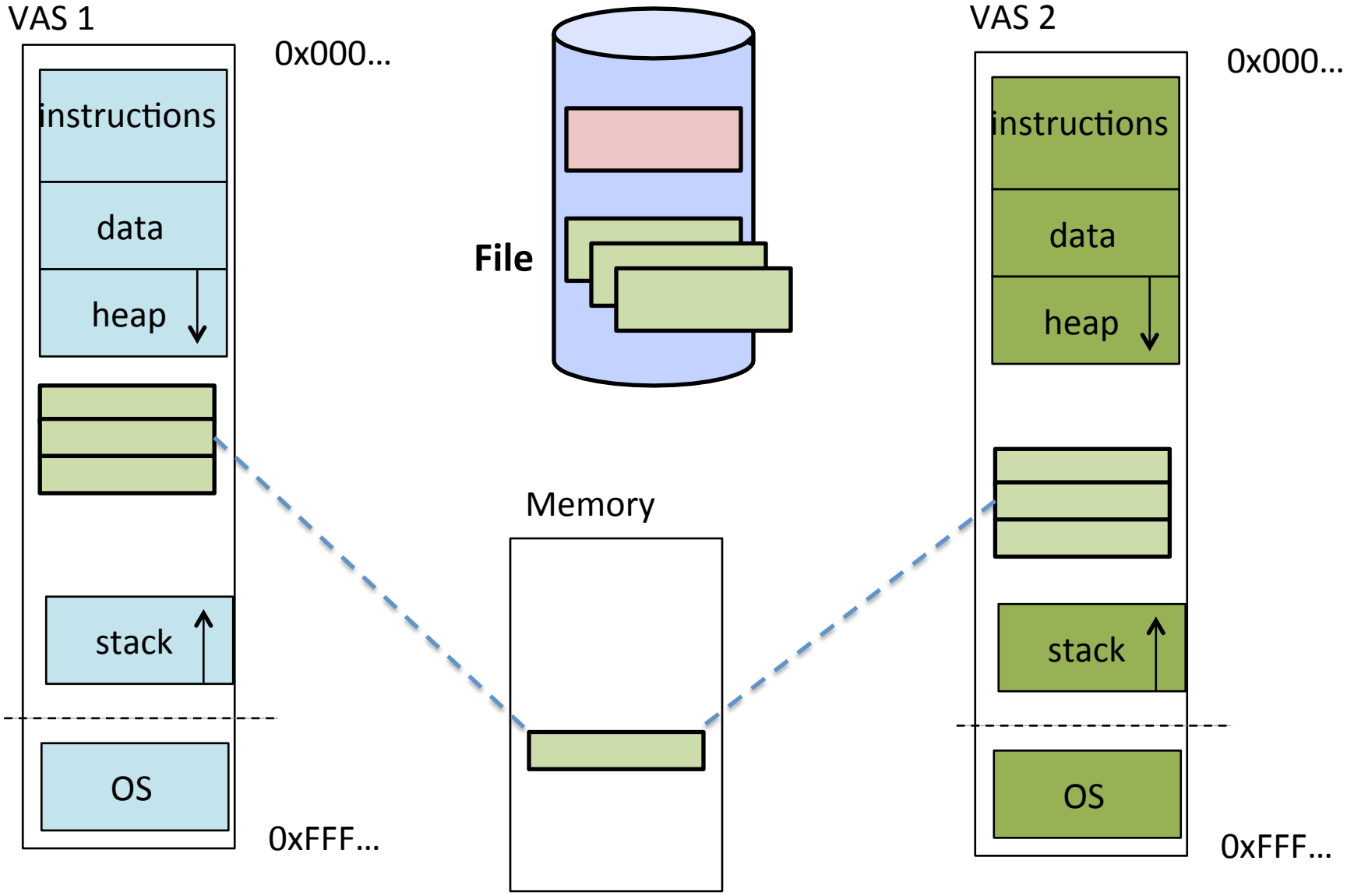
```
MMAP(2)                    BSD System Calls Manual                    MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- May map a specific region or let the system find one for you
  - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

An example

```
#include <sys/mman.h>

int something = 162;

int main (int argc, char *argv[]) {
  int infile;
  char *mfile;
  void *sadddr = 0;
  something++;
  printf("Data  at: %16lx\n", (long unsigned int) &something);
  printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

  mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, infile, 0);
  if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

  printf("mmap at : %16lx\n", (long unsigned int) mfile);

  puts(mfile);
  strcpy(mfile+20,"Let's write over it");
  close(infile);
  return 0;
}
```

Sharing through Mapped Files



File System Summary (1/2)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called “inode”
- Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS uses variable extents, rather than fixed blocks, and tiny files data is in the header
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization

File System Summary (2/2)

- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- File layout driven by freespace management
  - Integrate freespace, inode table, file blocks and directories into block group
- Copy-on-write creates new (better positioned) version of file upon burst of writes
- Deep interactions between memory management, file system, and sharing