Time-traveling File System on JOS

- My Idea
- Implementation Details
- Proof Of Concept

Time-traveling File System on JOS

- My Idea
- Implementation Details
- Proof Of Concept

Ideas

Here's a list of ideas to get you started thinking -- but you should feel free to pursue your own ideas.

- Build a virtual machine monitor that can run multiple guests (for example, multiple instances of JOS), using x86 VM support.
- Do something useful with the x86 Trusted Execution Technology. For example, run applications without having to trust the kernel. Here is a recent paper on this topic.
- Fix xv6 logging to support concurrent transactions, and generally have higher performance, perhaps taking ideas from Linux EXT3.
- Use file system ideas from Soft updates, WAFL, NILFS, ZFS, or another advanced file system.
- Add snapshots to a file system, so that a user can look at the file system as it appeared at various points in the past. You'll probably want to use some kind of copy-on-write for disk storage to keep space consumption down.
- Implement capabilities to provide fine-grained control over what privileges processes have.
- Build a distributed shared memory (DSM) system, so that you can run multi-threaded shared memory parallel programs on a cluster of machines, using paging to give the appearance of real shared memory. When a thread tries to access a page that's on another machine, the page fault will give the DSM system a chance to fetch the page over the network from whatever machine

Analysis of the idea

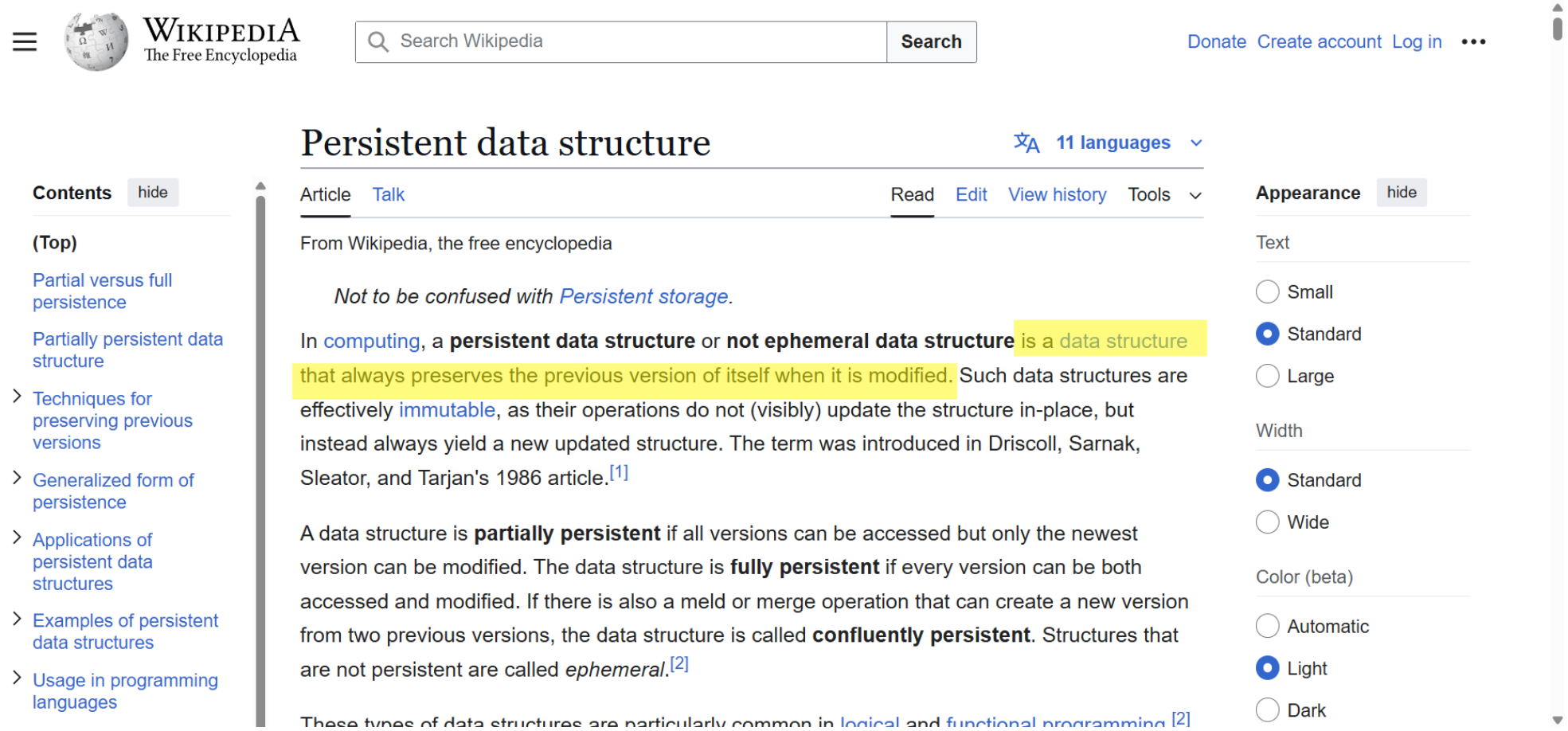
a. When will we take the snapshot? How often?

b. What will go into a snapshot? Is it the entire file system?

Will the user choose?

Or maybe the OS will decide?

Persistent Data Structure



WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Donate | Create account | Log in | •••

Persistent data structure

11 languages

Contents hide

(Top)

- Partial versus full persistence
- Partially persistent data structure
- Techniques for preserving previous versions
- Generalized form of persistence
- Applications of persistent data structures
- Examples of persistent data structures
- Usage in programming languages

Article | Talk | Read | Edit | View history | Tools

From Wikipedia, the free encyclopedia

Not to be confused with Persistent storage.

In computing, a **persistent data structure** or **not ephemeral data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. The term was introduced in Driscoll, Sarnak, Sleator, and Tarjan's 1986 article.[1]

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called *ephemeral*.[2]

These types of data structures are particularly common in logical and functional programming.[2]

Appearance hide

Text
- Small
- Standard
- Large

Width
- Standard
- Wide

Color (beta)
- Automatic
- Light
- Dark

Persistent Data Structure

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Donate | Create account | Log in | •••

Persistent data structure

11 languages

Contents hide

(Top)

Partial versus full persistence

Partially persistent data structure

Techniques for preserving previous versions

Generalized form of persistence

Applications of persistent data structures

Examples of persistent data structures

Usage in programming languages

Article | Talk | Read | Edit | View history | Tools

Appearance hide

Text

Small

Standard

Large

Width

Standard

Wide

Color (beta)

Automatic

Light

Dark

From Wikipedia, the free encyclopedia

Not to be confused with Persistent storage.

In computing, a **persistent data structure** or **not ephemeral data structure** is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. The term was introduced in Driscoll, Sarnak, Sleator, and Tarjan's 1986 article.[1]

A data structure is **partially persistent** if all versions can be accessed but only the newest version can be modified. The data structure is **fully persistent** if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called **confluently persistent**. Structures that are not persistent are called *ephemeral*.[2]

These types of data structures are particularly common in logical and functional programming.[2]

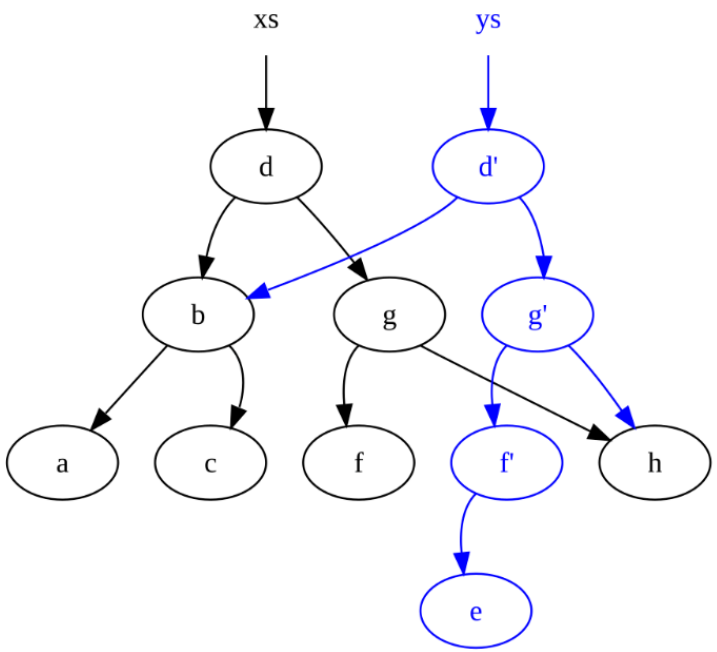
The idea

- File system that behaves like a fully persistent data structure:
  all past versions are preserved and can be modified.
- The user will choose a root and every file under this root will preserve automatically with any change.
- Nothing is truly lost - history is preserved.
- The user can go back to read from any previous state.
- The user can branch off a past version into a new timeline.

How to preserve previous versions?


```
ys = insert ("e", xs)
```

The following configuration is produced:



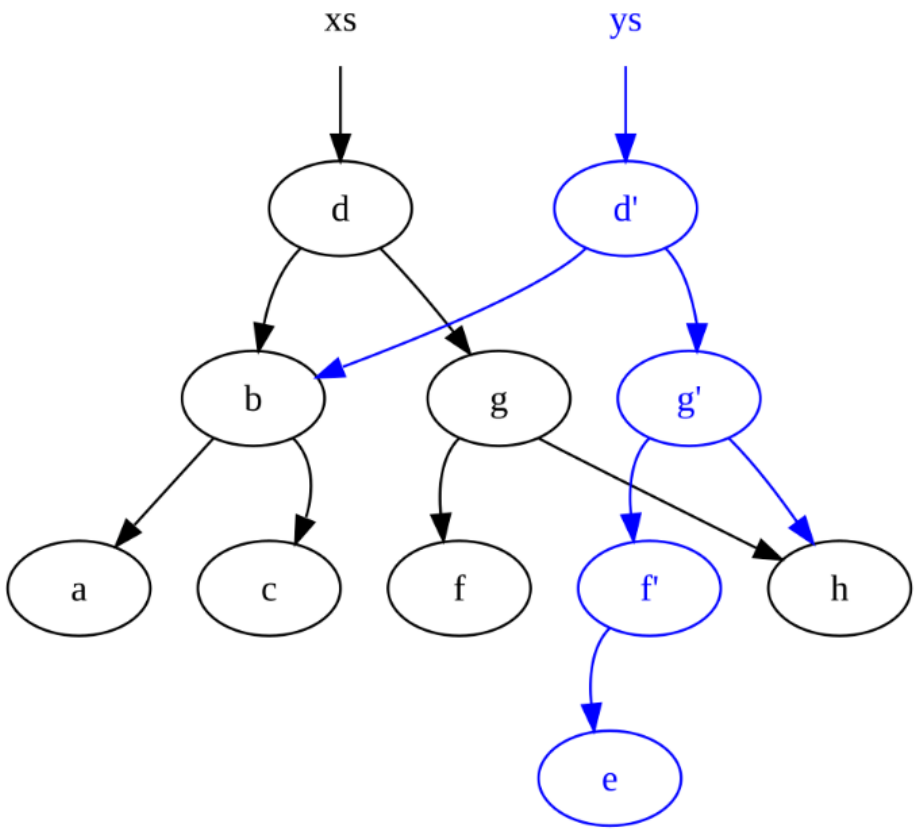
How to preserve previous versions?

space complexity:

the over all space is $O(nlog(n))$

while n is the number of objects across all versions

I implemented the same behavior with $O(n)$



Fat Files

copied for each write, leading to worst case $O(n \cdot m)$ performance characteristics for m modifications of an array of size n. Copy-on-write memory management can reduce the price for an update from $\Theta(n)$ to $O(Bu)$, where B is the memory block size and u the number of pages updated in an operation.[*citation needed*]

Fat node [edit]

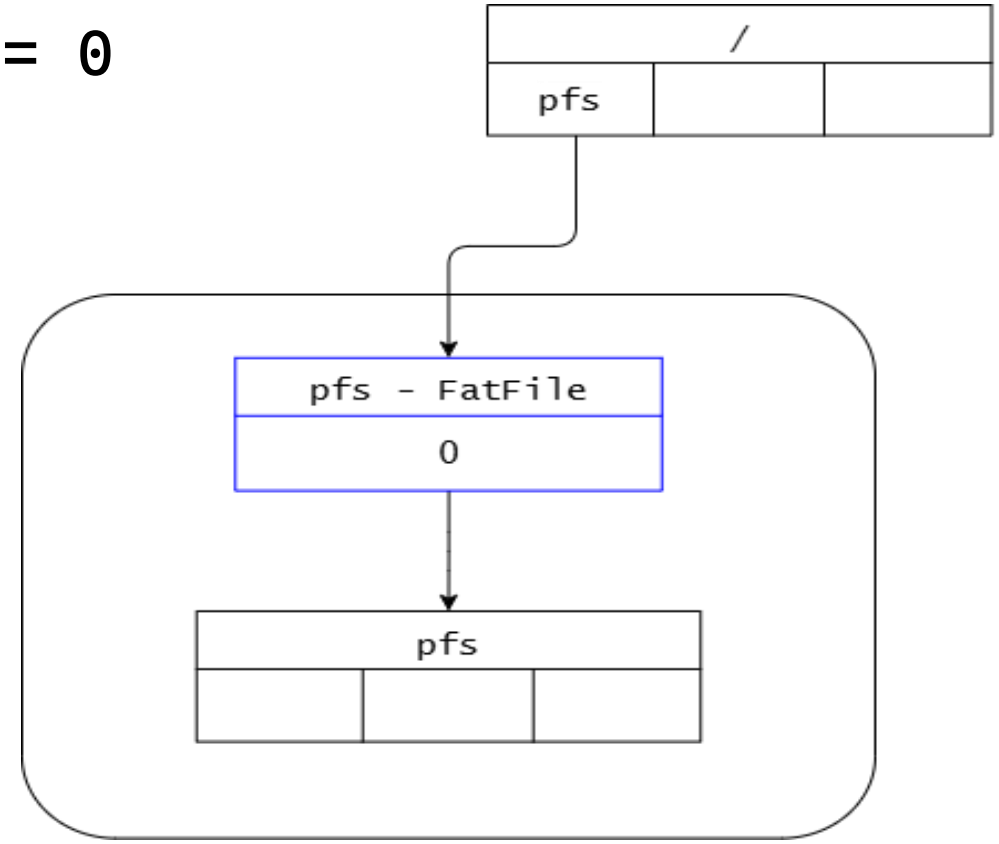
The fat node method is to record all changes made to node fields in the nodes themselves, without erasing old values of the fields. This requires that nodes be allowed to become arbitrarily "fat". In other words, each fat node contains the same information and pointer fields as an ephemeral node, along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp which indicates the version in which the named field was changed to have the specified value. Besides, each fat node has its own version stamp, indicating the version in which the node was created. The only purpose of nodes having version stamps is to make sure that each node only contains one value per field name per version. In order to navigate through the structure, each original field value in a node has a version stamp of zero.

Complexity of fat node [edit]

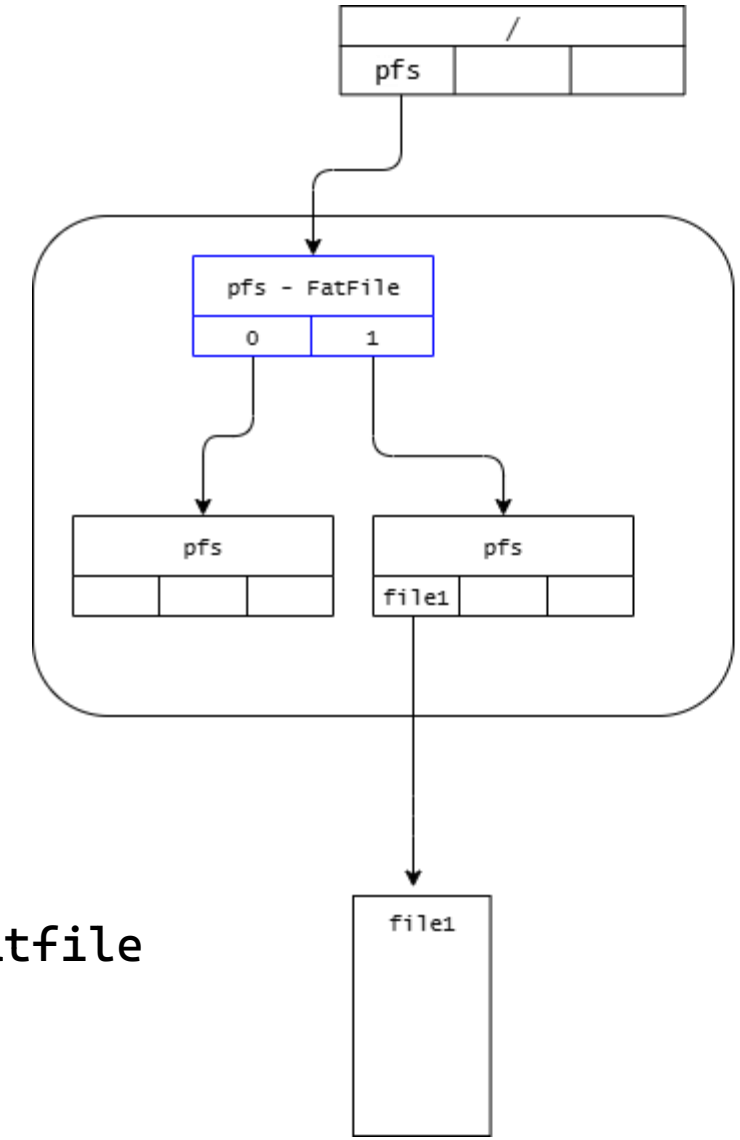
With using fat node method, it requires O(1) space for every modification: just store the new data. Each modification takes O(1) additional time to store the modification at the end of the modification history. This is an amortized time bound, assuming modification history is stored in a

Illustration

timestamp = 0



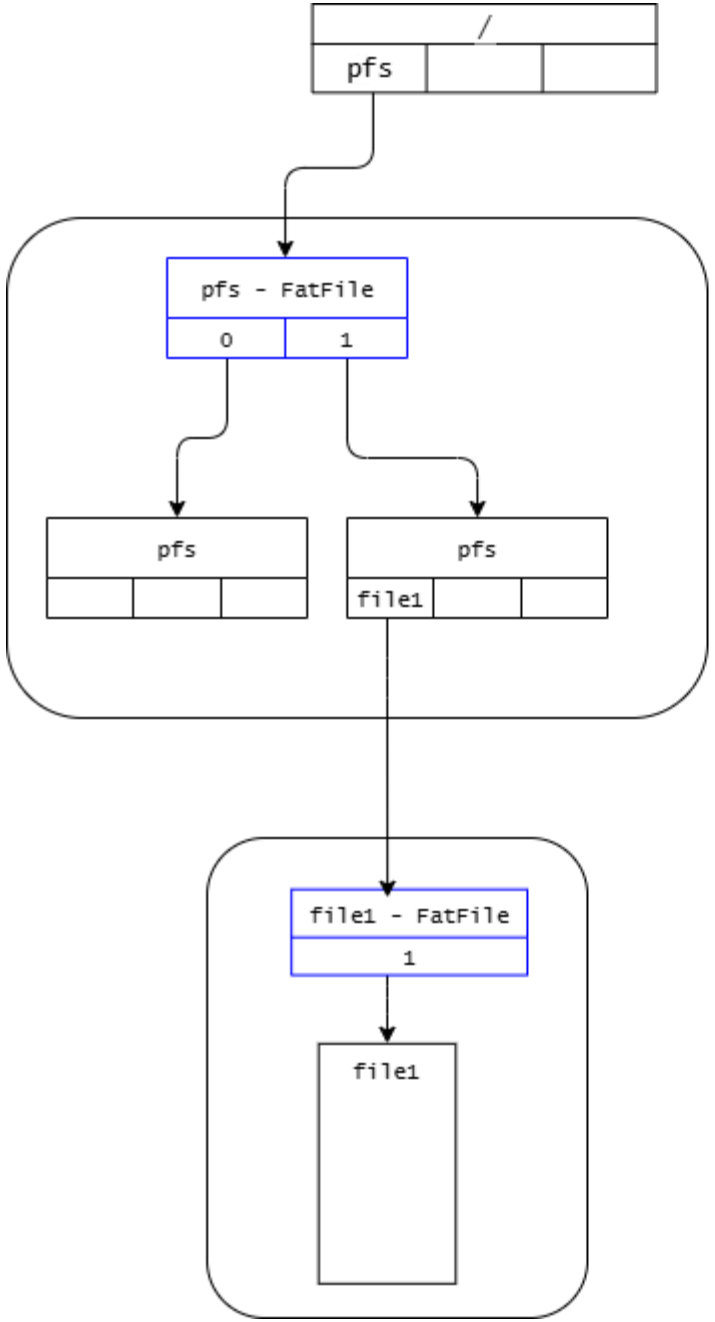
timestamp = 1



ts	command
1	open("/pfs/file1")

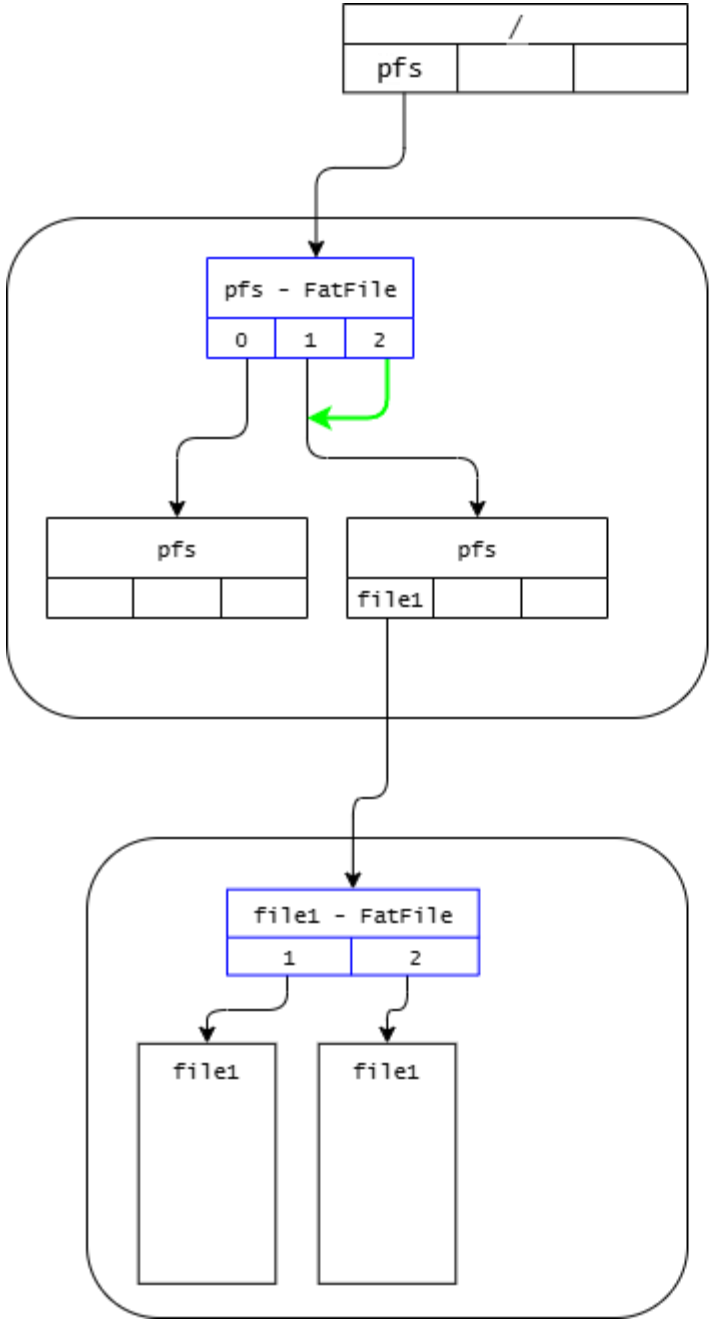
file1 should also be a fatfile

timestamp = 1



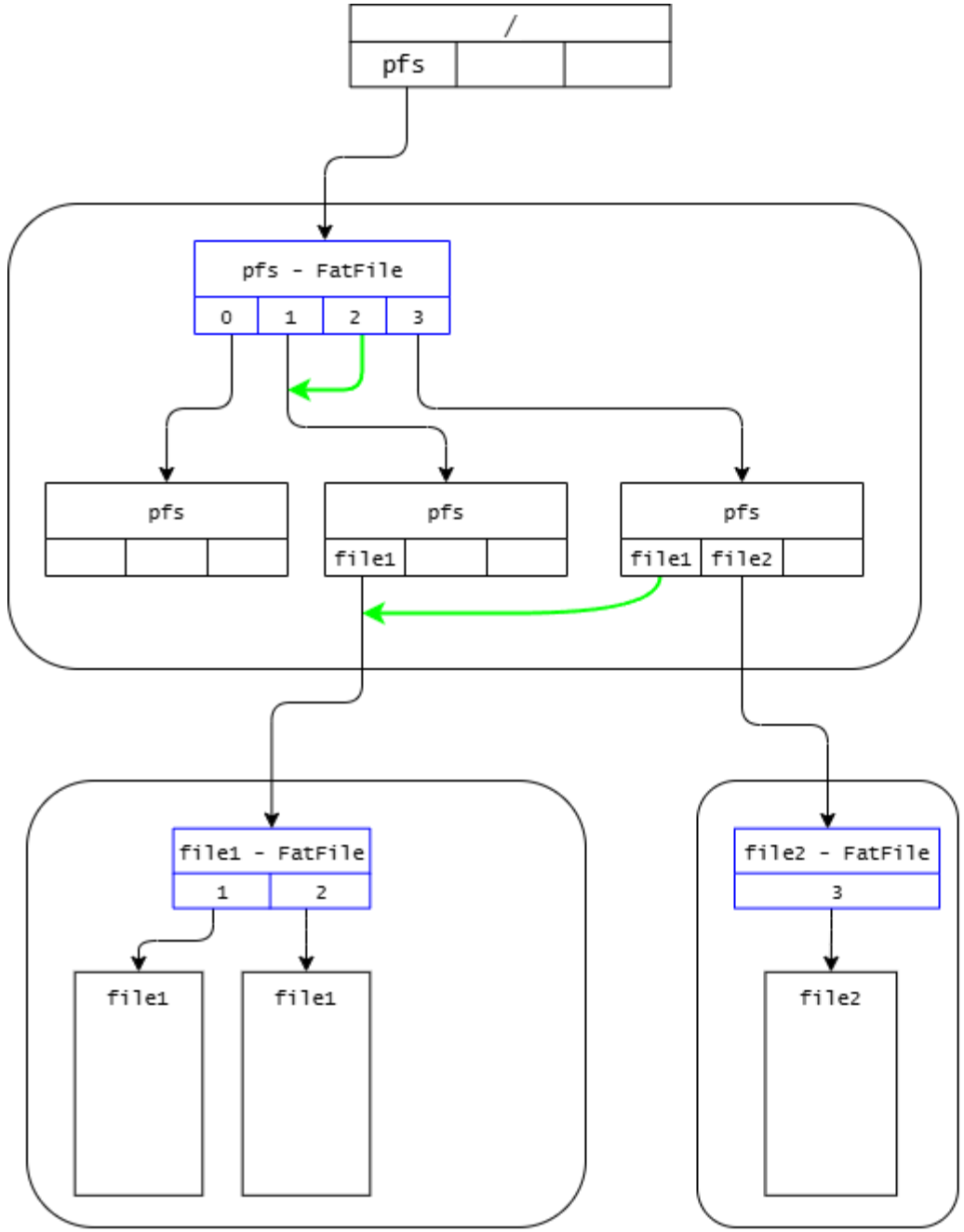
ts	command
1	open("/pfs/file1")

timestamp = 2



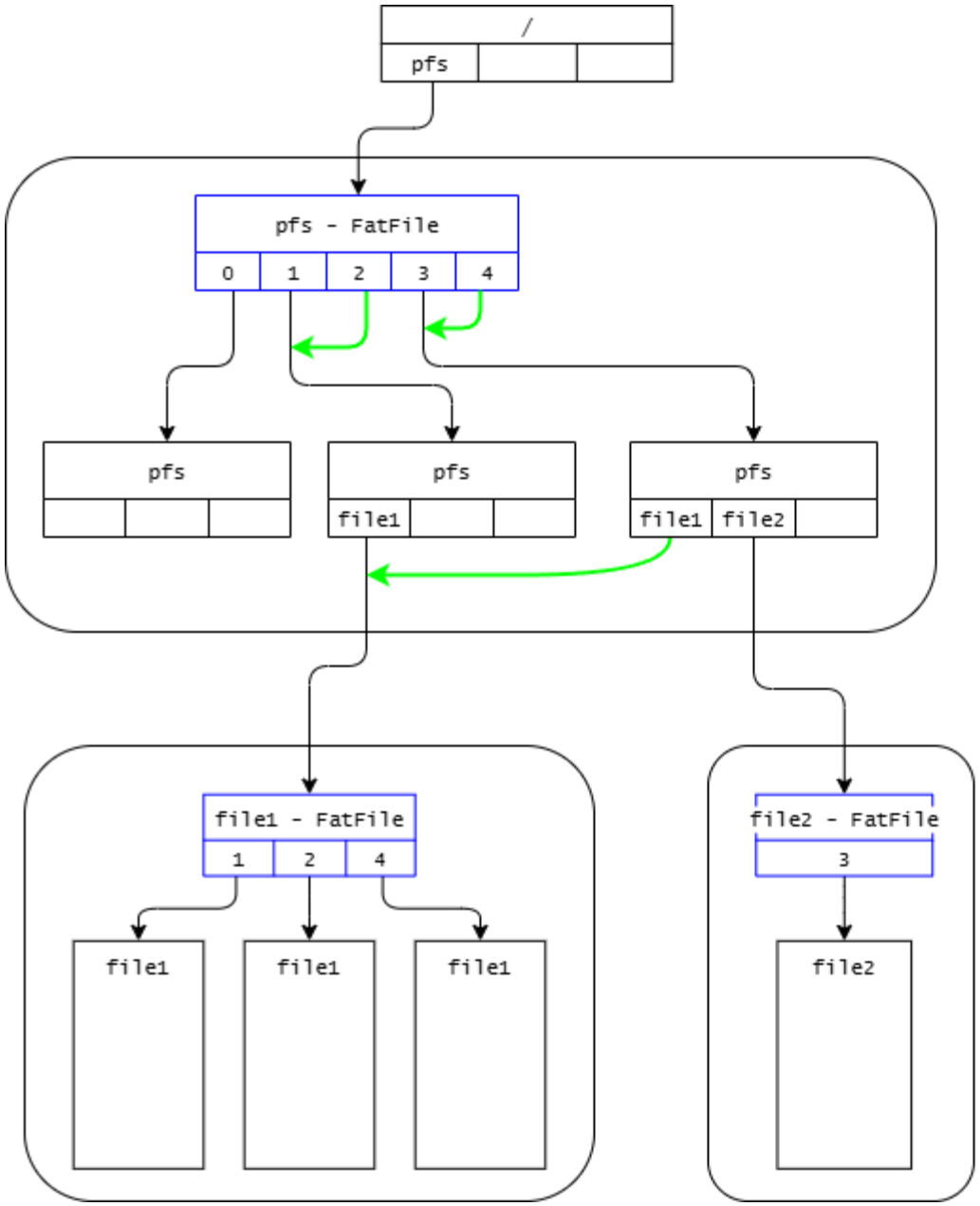
ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")

timestamp = 3



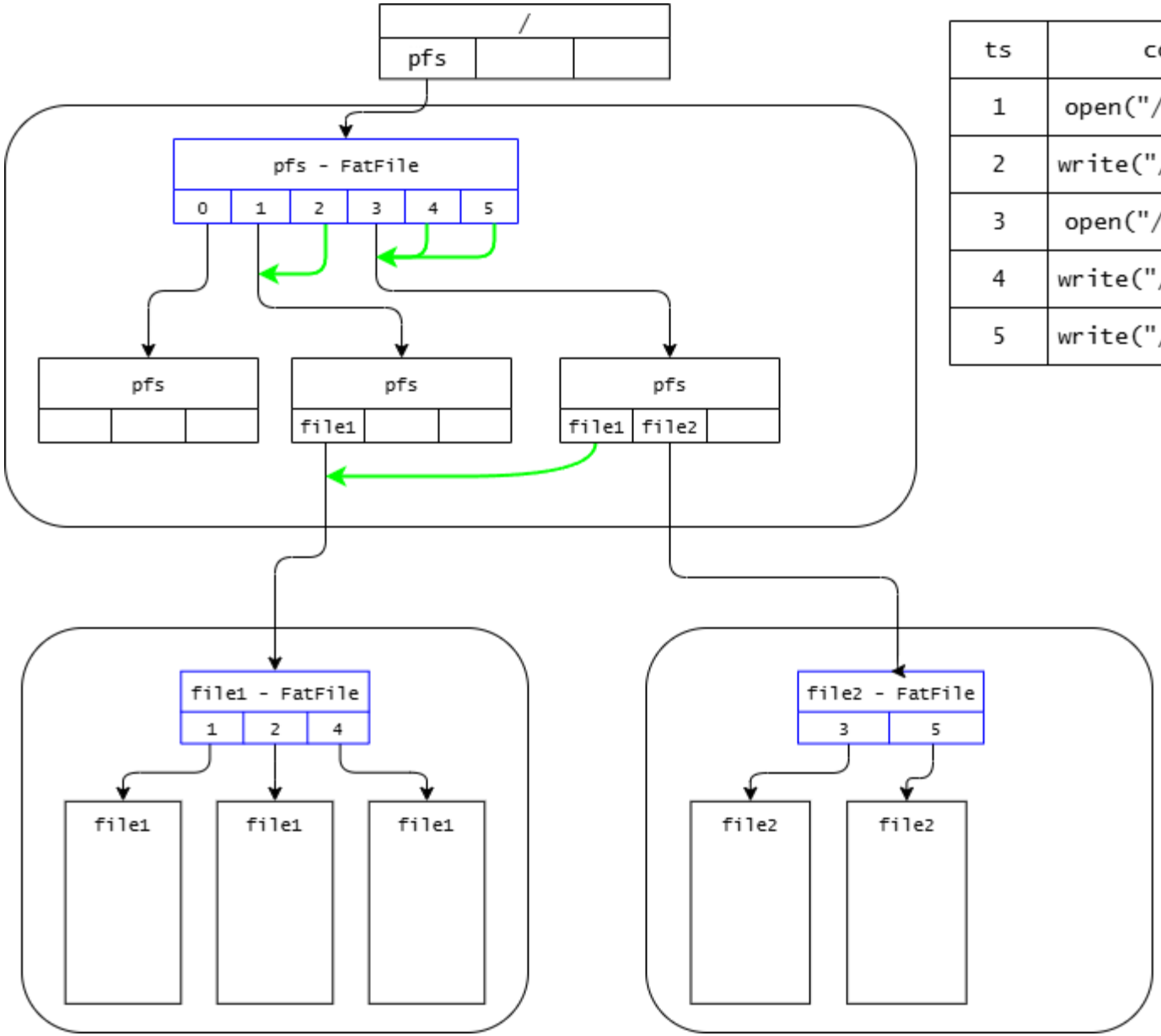
ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")

timestamp = 4



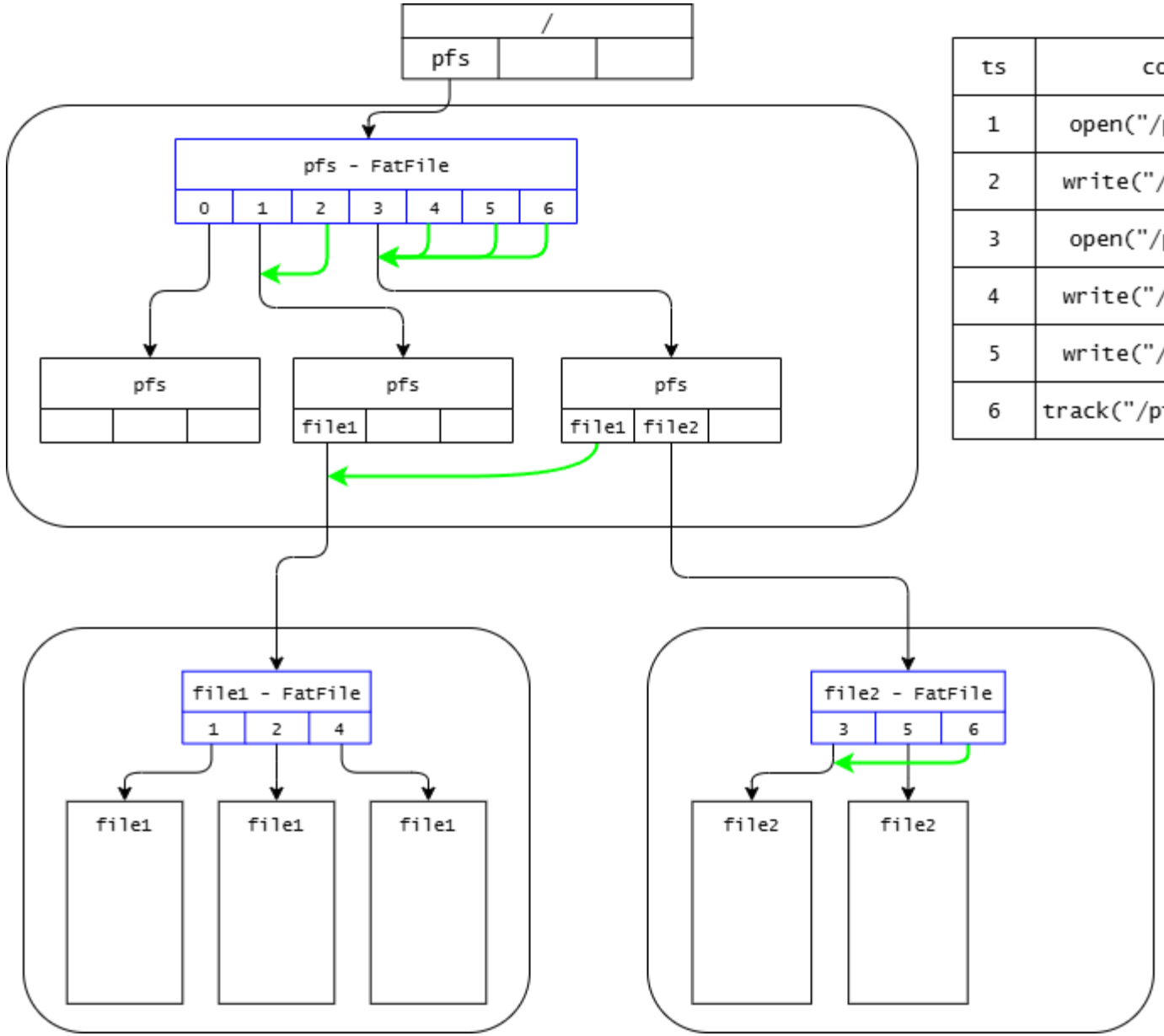
ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")
4	write("/pfs/file1")

timestamp = 5



ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")
4	write("/pfs/file1")
5	write("/pfs/file2")

timestamp = 6

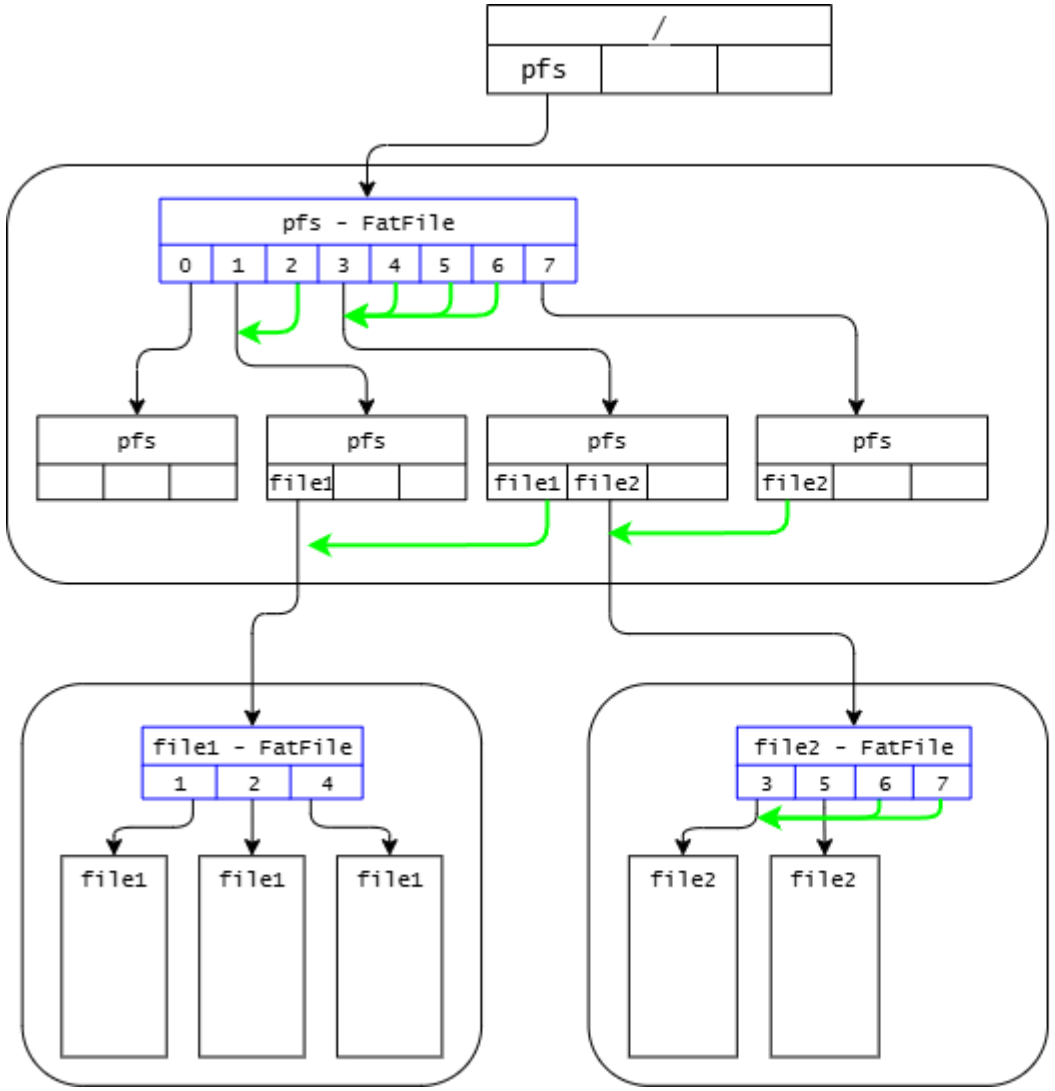


ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")
4	write("/pfs/file1")
5	write("/pfs/file2")
6	track("/pfs/file2", 3)

deleting a file

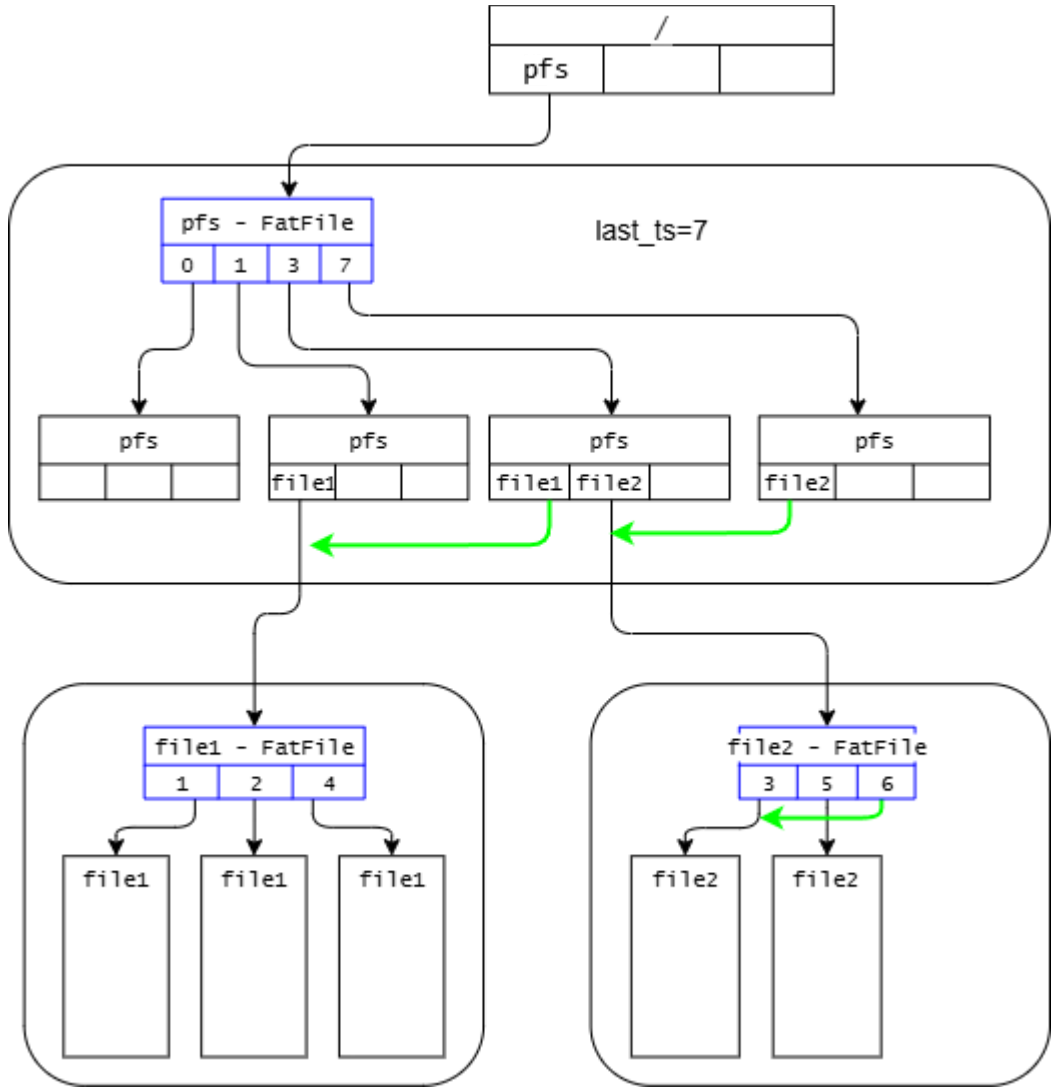
- can we delete a file?
  - of course!
- in our PFS we can remove any file (file/dir) but the file itself. not the FatFile.
- deleting the file is just creating a new timestamp for its ff without the file.

timestamp = 7

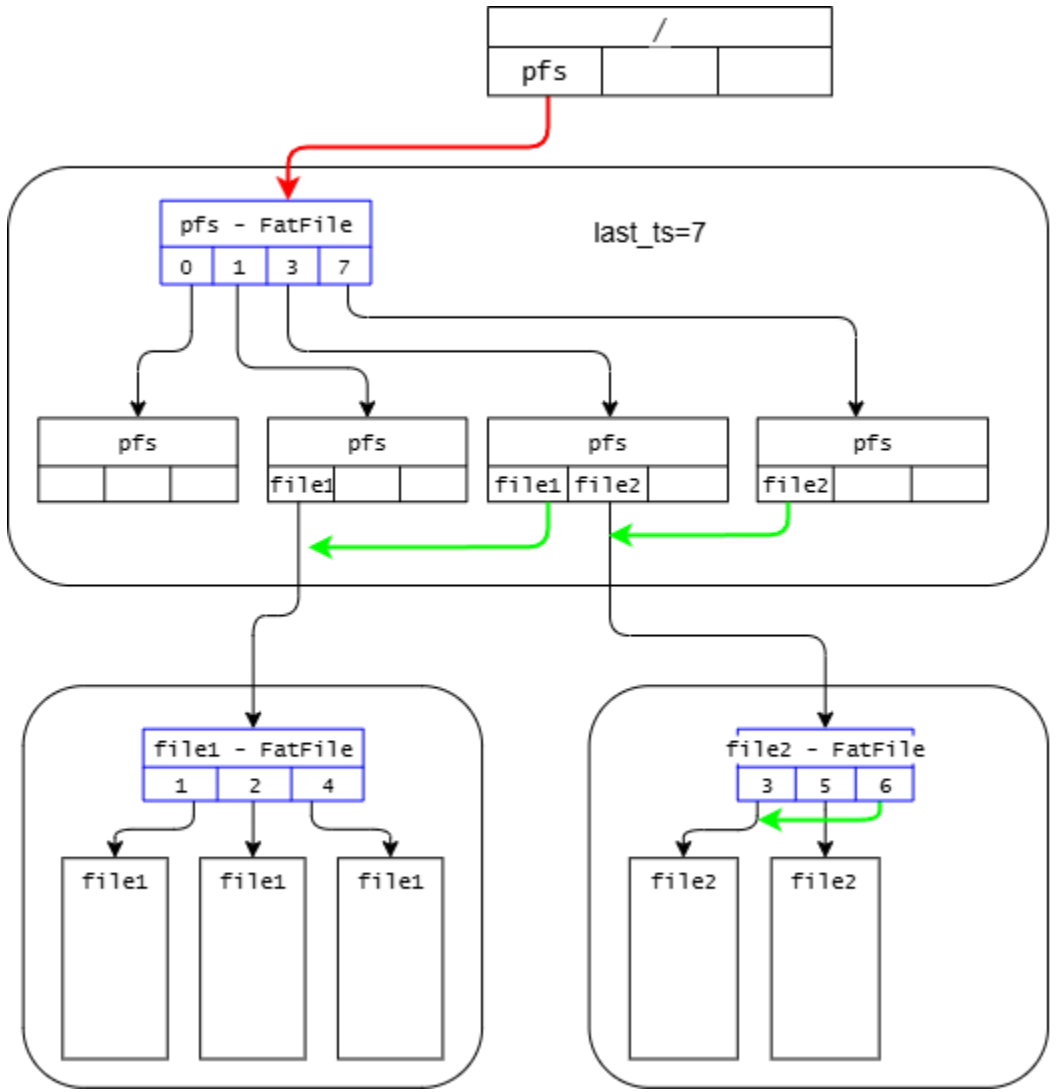


ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")
4	write("/pfs/file1")
5	write("/pfs/file2")
6	track("/pfs/file2", 3)
7	remove("pfs/file1")

Actually,
it looks
like this



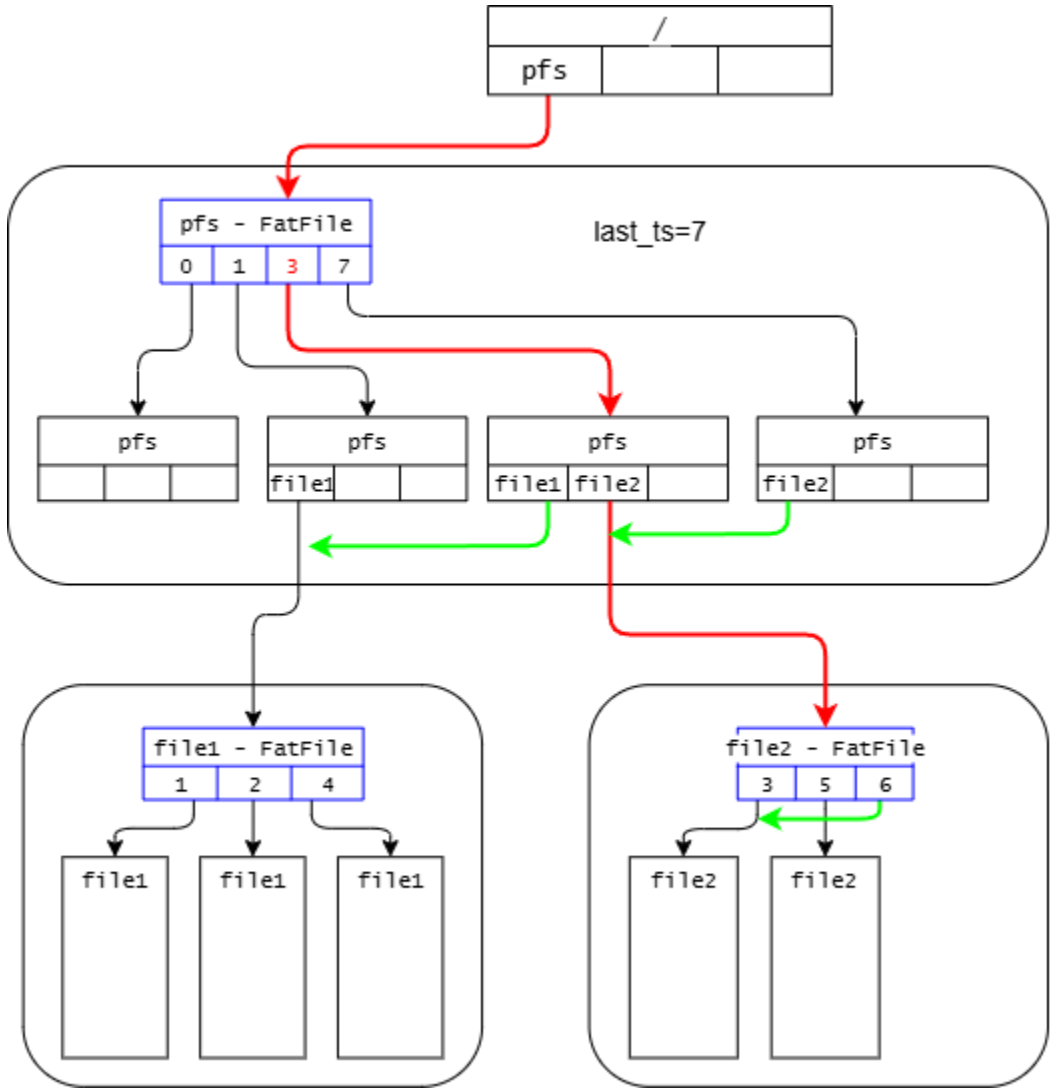
tracking the file system



read("/pfs/file2")

tracking_ts = 4

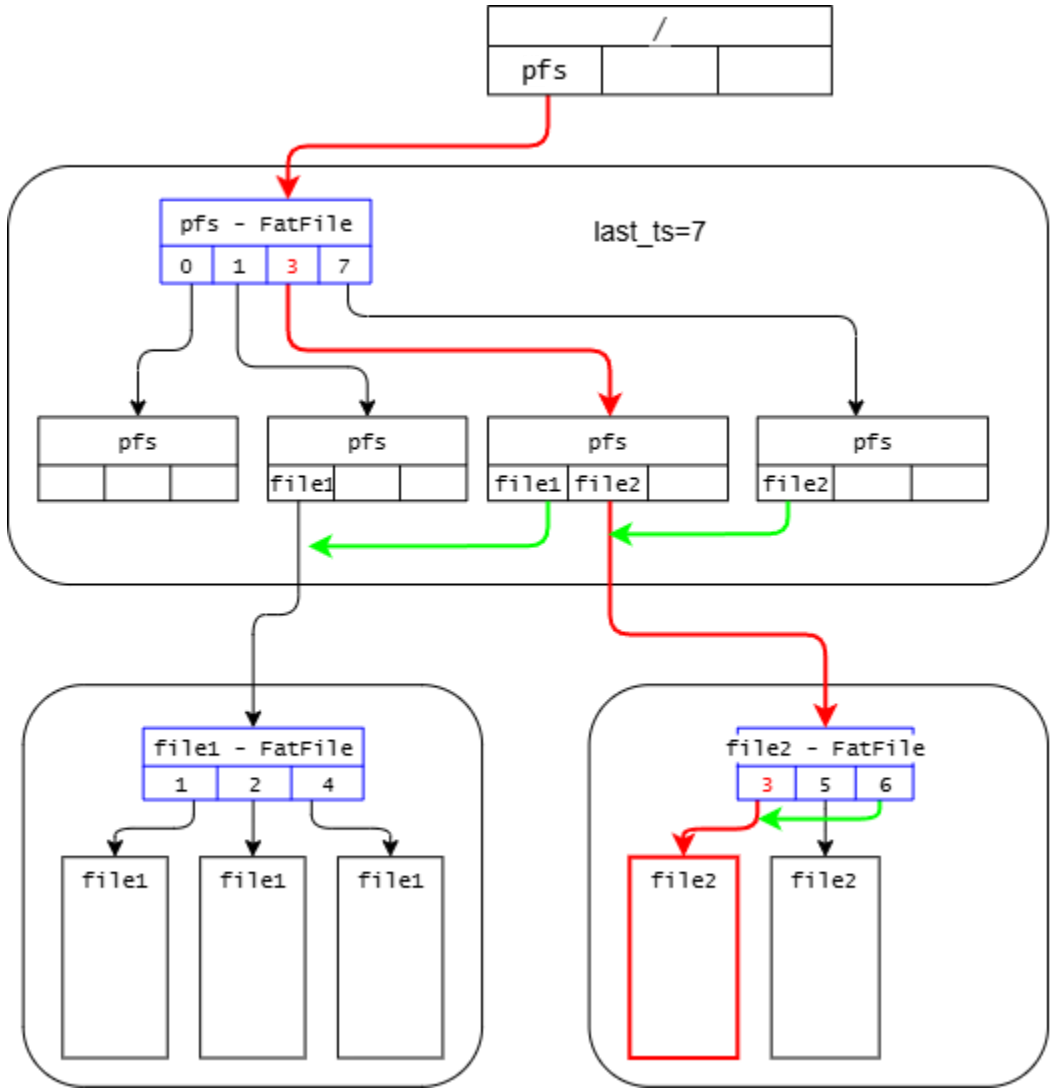
tracking the file system



read("/pfs/file2")

tracking_ts = 4

tracking the file system

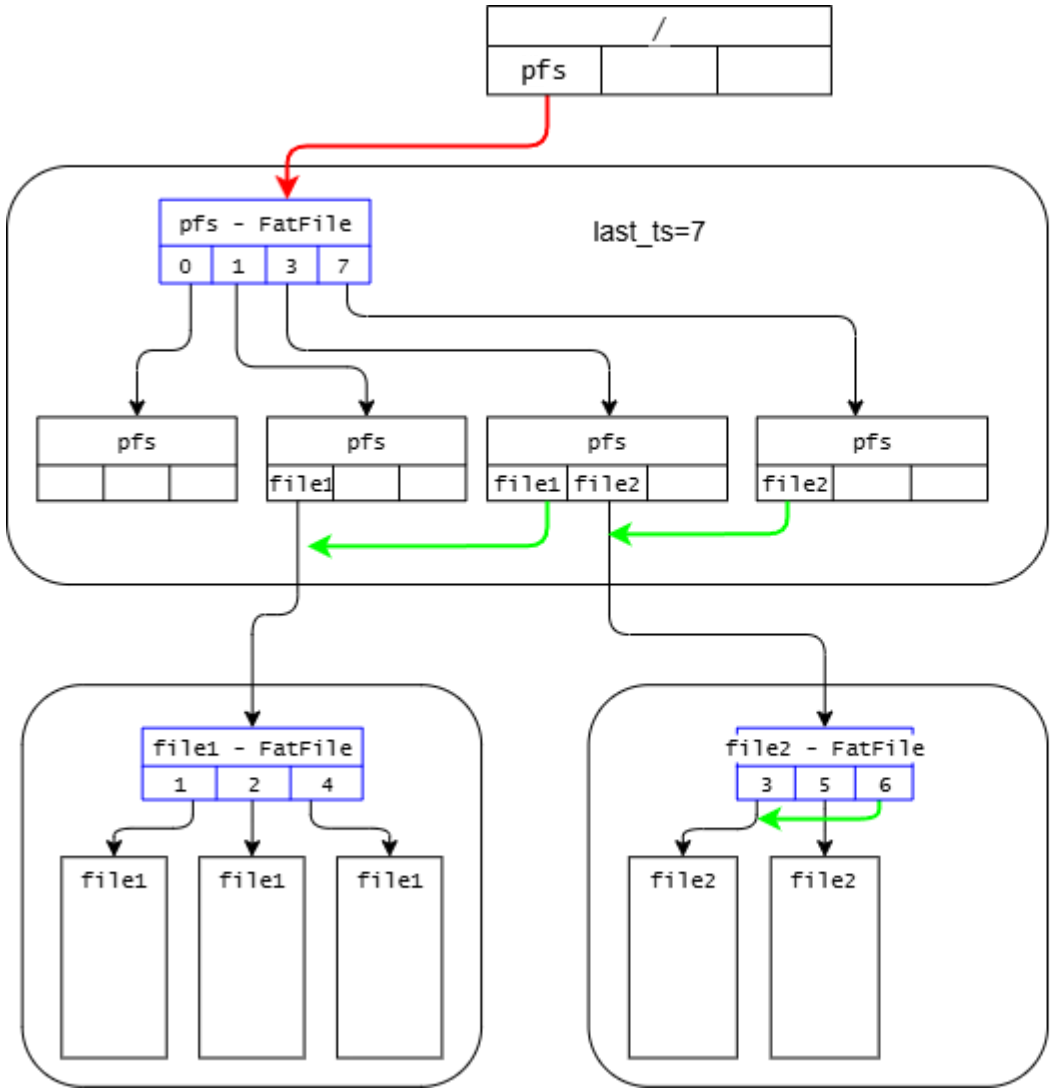


read("/pfs/file2")

tracking_ts = 4

tracking the file system

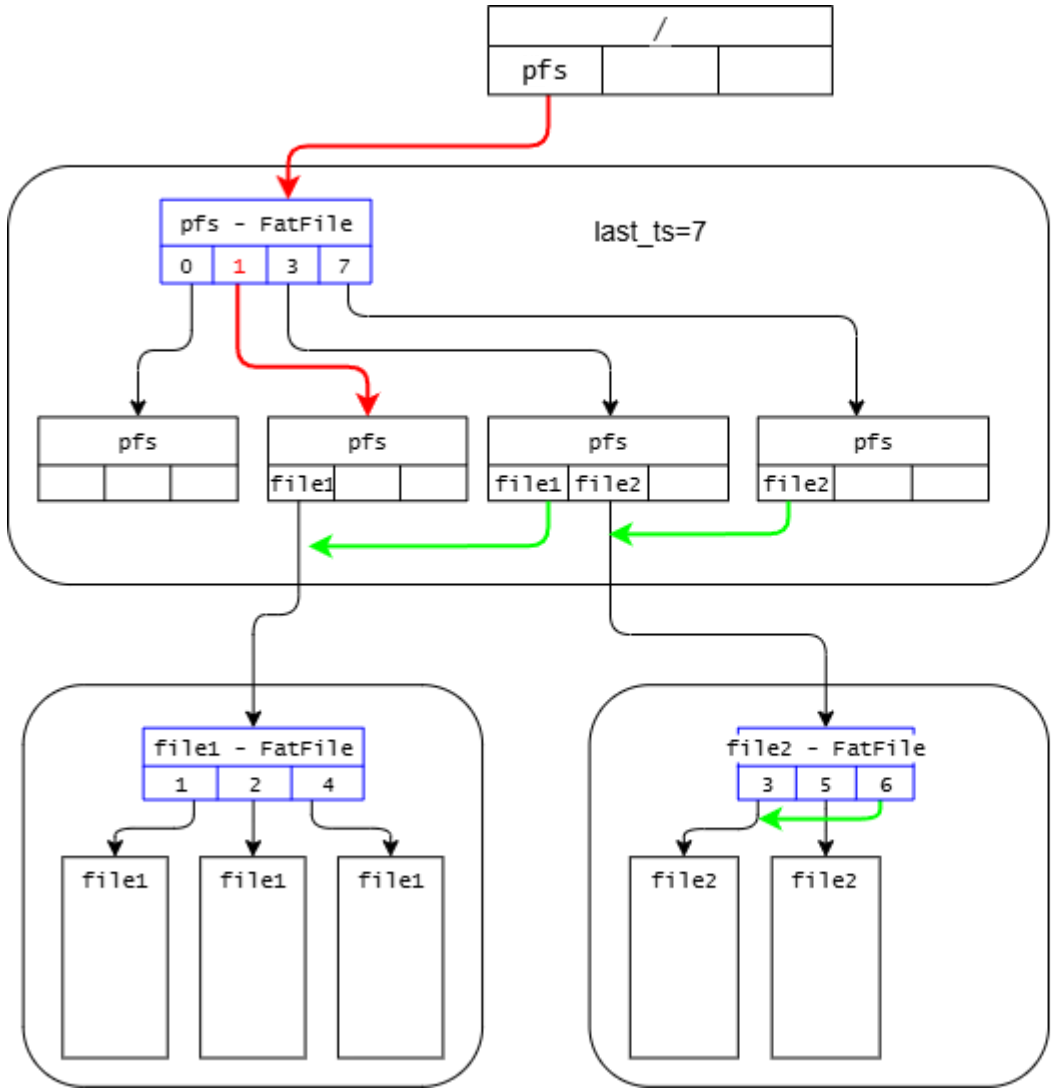
What happens if we try to read a file with a timestamp where the file did not exist?



read("/pfs/file2")
tracking_ts = 2

tracking the file system

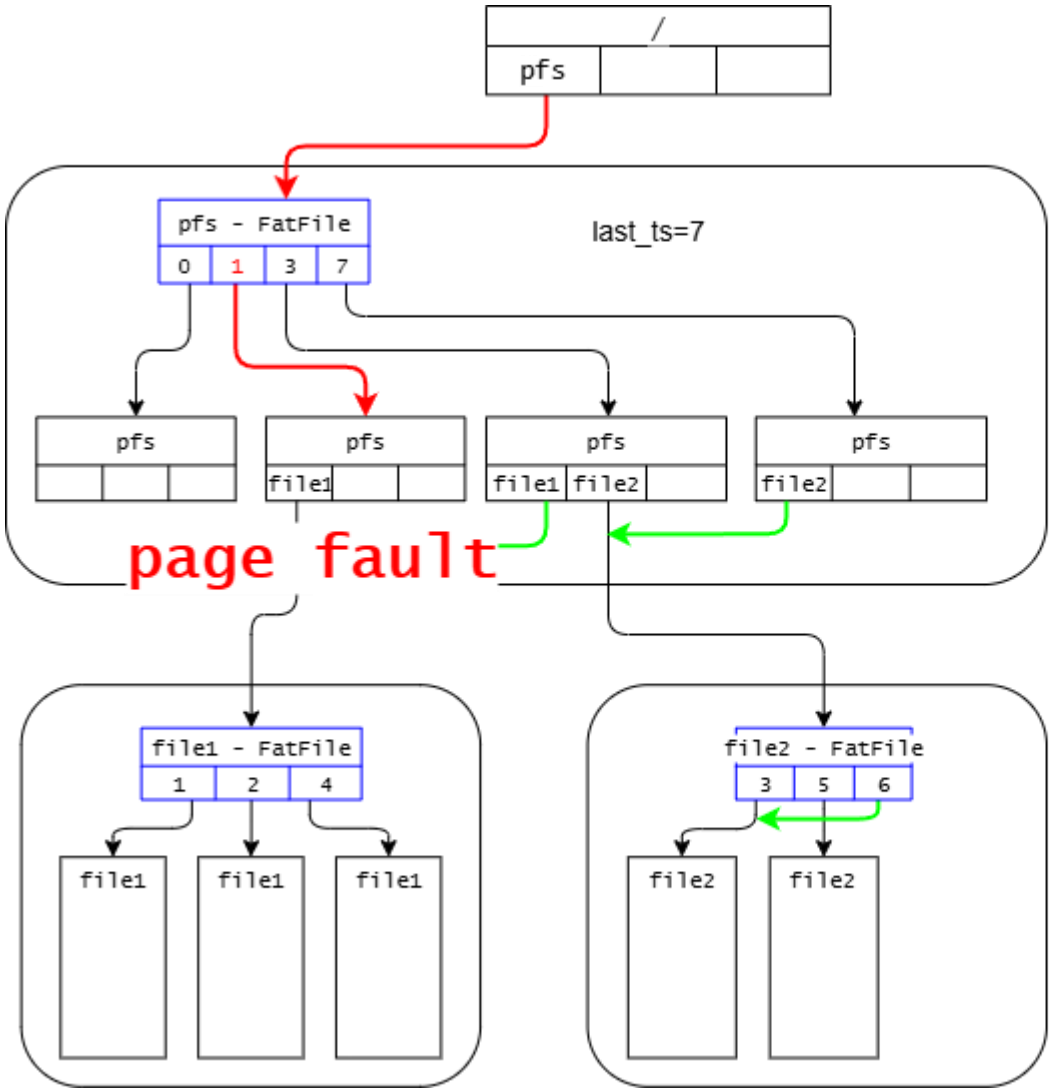
What happens if we try to read a file with a timestamp where the file did not exist?



read("/pfs/file2")
tracking_ts = 2

tracking the file system

What happens if we try to read a file with a timestamp where the file did not exist?

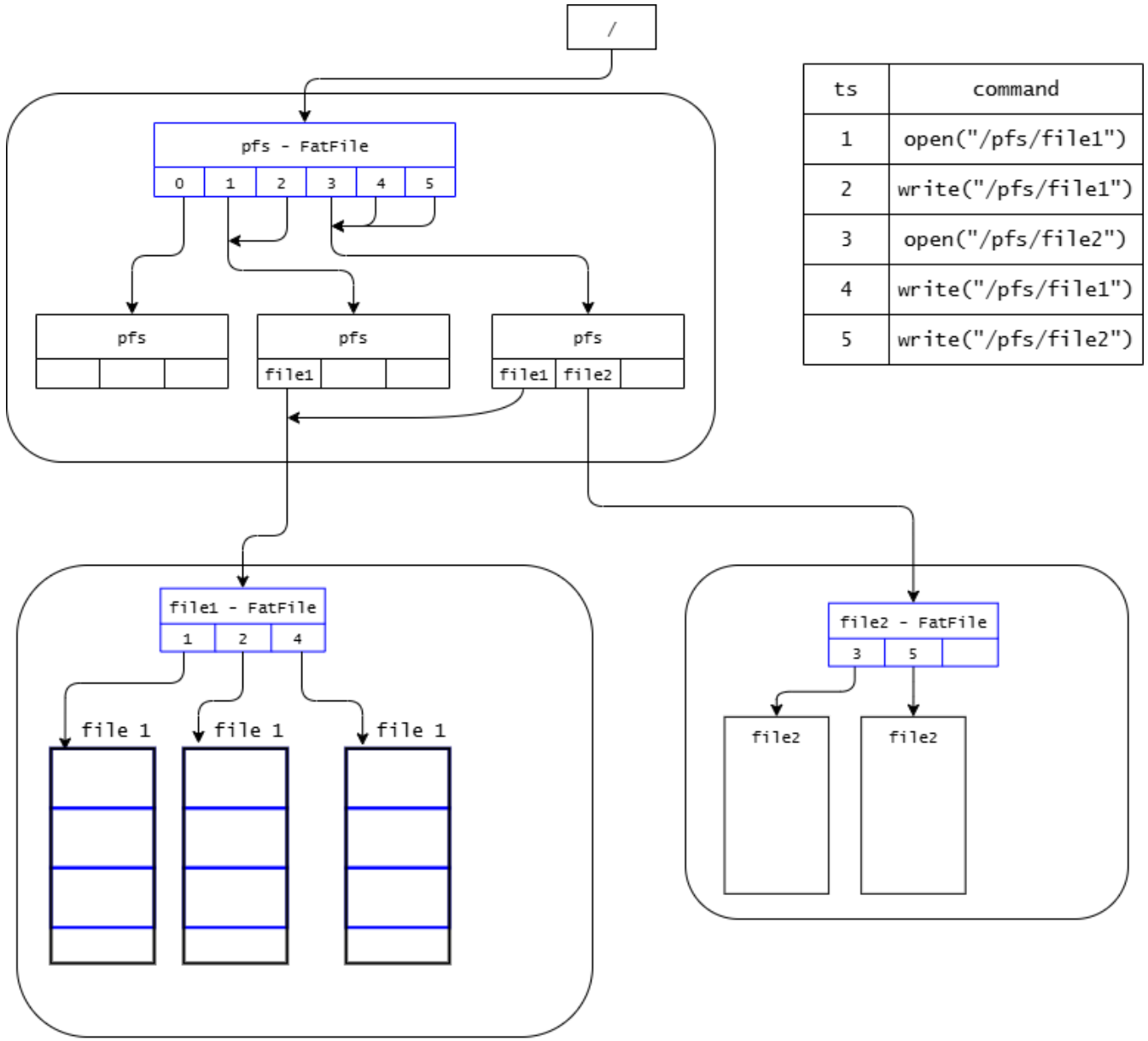


read("/pfs/file2")
tracking_ts = 2

Efficiency

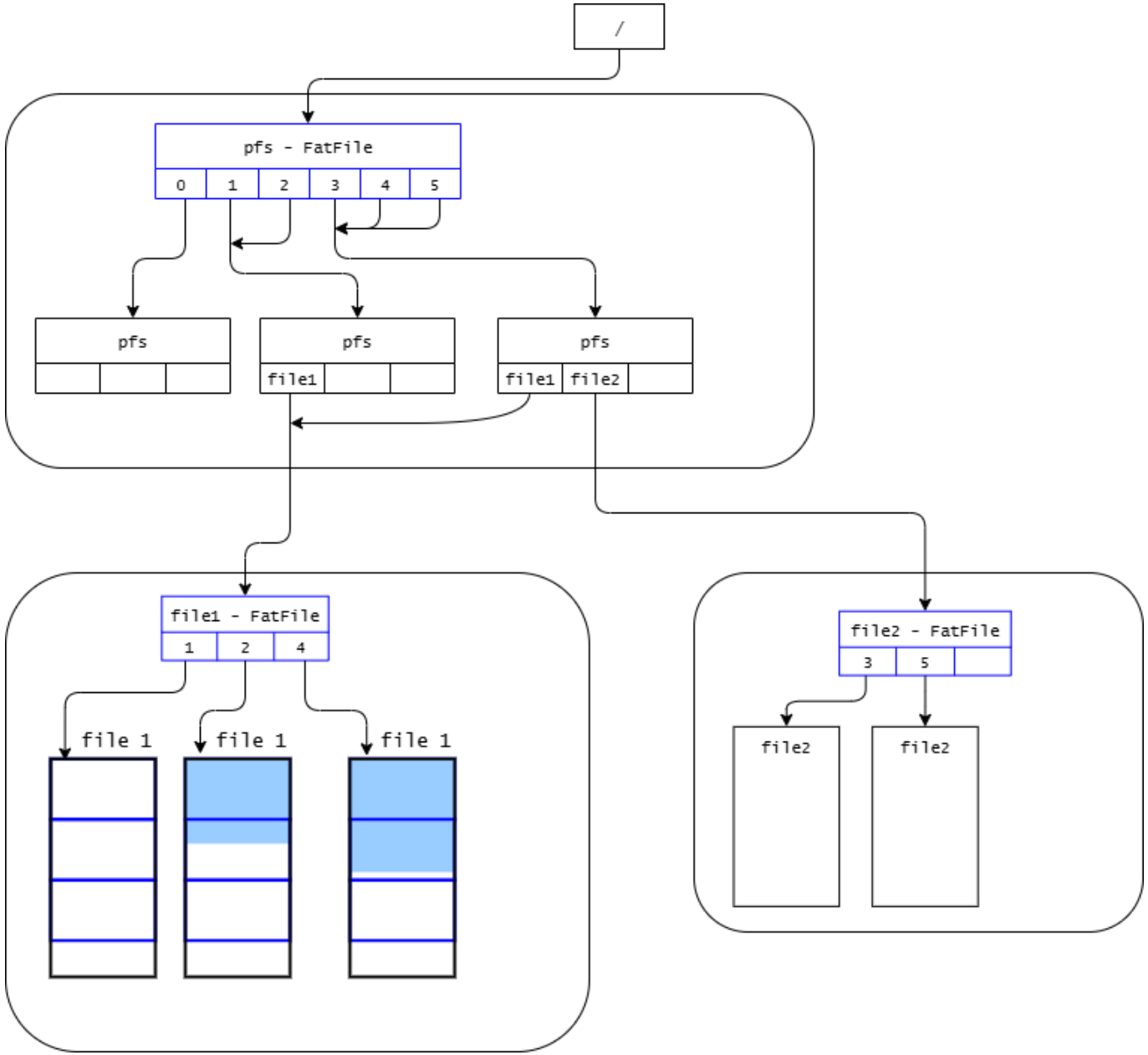
- Every complete file-block saved exactly once.
- Incomplete block – will be copied.
- A regular file and a folder file behave the same.

Deduplication

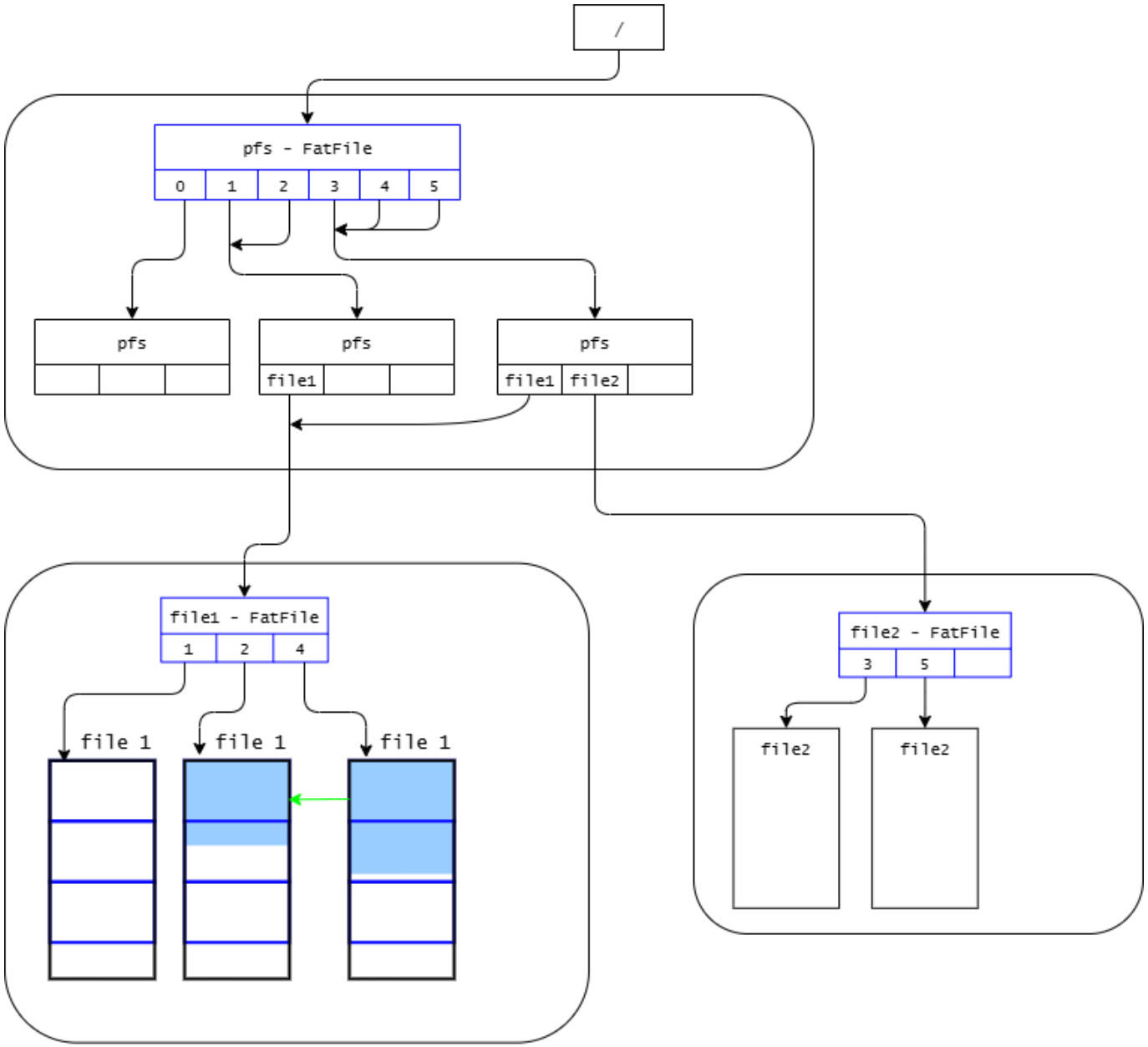


ts	command
1	open("/pfs/file1")
2	write("/pfs/file1")
3	open("/pfs/file2")
4	write("/pfs/file1")
5	write("/pfs/file2")

Deduplication



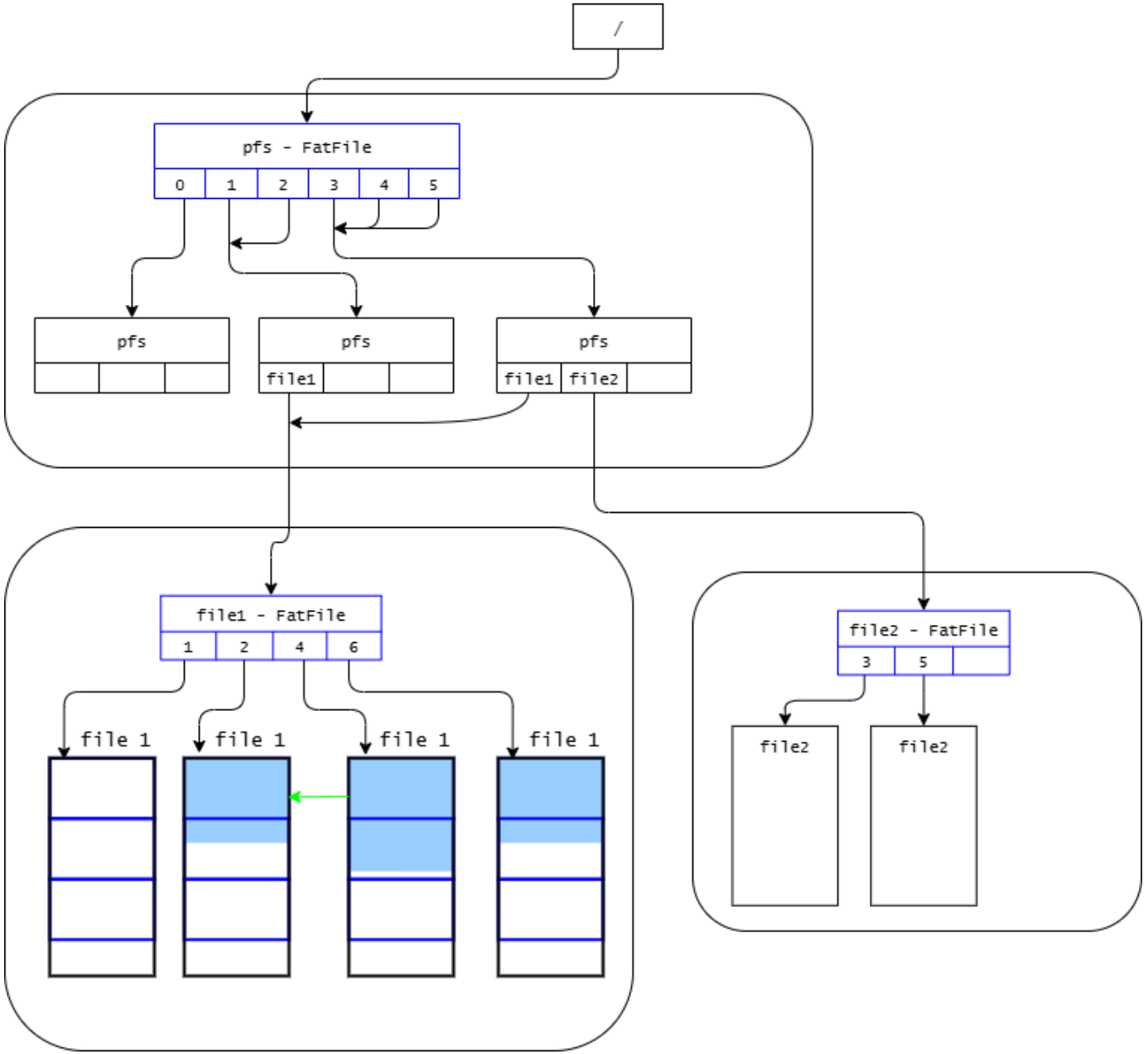
Deduplication



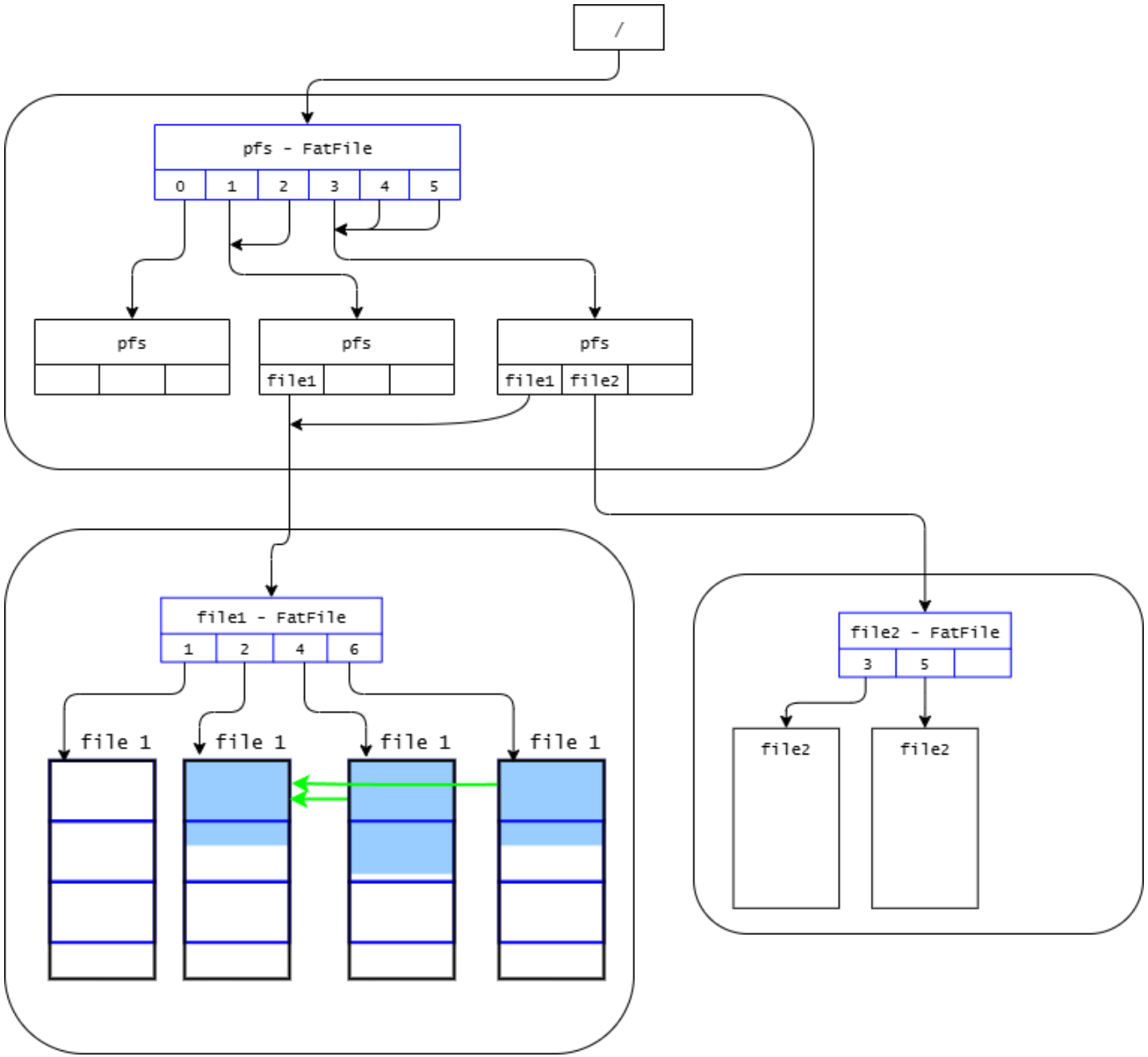
Note

- Unfortunately, the block size is equal to the page size, which means a large amount of data may be copied each time.
- Reducing the block size to match the sector size would require rewriting the file system from scratch
  - and might involve giving up the current IPC-based file system management.

Deduplication



Deduplication



Is that “fully persistent”?

Reminder: A data structure is fully persistent if every version can be both accessed and modified – not just the newest one.

Let’s assume they aren’t read-only.

But the moment you modify one, you must create a new version.

So in practice, they behave as “read-only” anyway.

Time-traveling File System on JOS

- My Idea
- Implementation Details
- Proof Of Concept

Timestamp

- added f_timestamp in struct File
  - will be the latest timestamp of the file

inc/fs.h

```
28
29 typedef ssize_t ts_t;   // PROJECT
30
31 // PROJECT:
32 // When File.f_type is FTYPE_FN it's mean all the blocks of the File
33 // containing versions of the file. Each for every timestamp.
34 struct File {
35         char f_name[MAXNAMELEN];        // filename
36         off_t f_size;                   // file size in bytes
37         uint32_t f_type;                // file type
38         ts_t f_timestamp;               // PROJECT: last timestamp of the file
39
40         // Block pointers.
41         // A block is allocated iff its value is != 0.
42         uint32_t f_direct[NDIRECT];     // direct blocks
43         uint32_t f_indirect;            // indirect block
44
45         // Pad out to 256 bytes; must do arithmetic in case we're compiling
46         // fsformat on a 64-bit machine.
47         uint8_t f_pad[256 - MAXNAMELEN - 12 - 4*NDIRECT - 4];   // PROJECT: Changed from -8 to -12
48 } __attribute__((packed));      // required only on some 64-bit machines
49
```

Timestamp

- The current timestamp is preserved on the disk!
- It is stored in the superblock.
  - It is read from the disk in fs_init() and updated whenever it increases.
  - Initially set to 0 in fsformat.c:opendisk()

inc/fs.h

```
64 struct Super {
65         uint32_t s_magic;               // Magic number: FS_MAGIC
66         uint32_t s_nblocks;             // Total number of blocks on disk
67         ts_t last_ts;                   // PROJECT: save global timestamp on-disk!
68         struct File s_root;             // Root directory node
69 };
```

FatFile Type

- A new file type

inc/fs.h

```
53
54 // File types
55 #define FTYPE_REG       0x00            // Regular file
56 #define FTYPE_DIR       0x01            // Directory
57 #define FTYPE_FF        0x10            // Fat File     // PROJECT
58
```

- FatFile type is
  - FTYPE_FF | FTYPE_REG for file
  - FTYPE_FF | FTYPE_DIR for folder

FatFile – Changes required

- Basically, almost nothing needed to be changed, thanks to the fact that FatFile behaves like a folder.
- The main thing that needed to be maintained was for walking the path from the root.
  - I wrote a function that takes FatFile and returns the correct TS. and that's it..
  - More on that later.

- One small thing: Since a fatfile's timestamp file has FTYPE_REG only, I also need to store the parent FatFile in the OpenFile struct.

fs/serv.c

```
31
32 struct OpenFile {
33         uint32_t o_fileid;      // file id
34         struct File *o_file;    // mapped descriptor for open file
35         struct File *o_fatfile; // PROJECT: the fatfile that contain the file. NULL if there is no such one.
36         int o_mode;             // open mode
37         struct Fd *o_fd;        // Fd page
```

JOS FS Build

- fsformat.c’s main() build the JOS FS
  - It creates the root directory and write to it all the user files.

- I added to root a new folder there, called pfs.
- It’s the first fatfile and the root for the Persistent FS.
- It’s created automatically when the file system is built.

JOS FS Build

- There is a clean-fs.img backup file in obj directory.
- clean-fs.img preserves the disk contents between JOS sessions.
- When a file in obj/fs changes, fs.img is rebuilt.
- In that case, all disk files are lost, so last_ts will reset to zero.

fs/Makefrag

```
62 # How to build the file system image
63 $(OBJDIR)/fs/fsformat: fs/fsformat.c
64         @echo + mk $(OBJDIR)/fs/fsformat
65         $(V)mkdir -p $(@D)
66         $(V)$(NCC) $(NATIVE_CFLAGS) -o $(OBJDIR)/fs/fsformat fs/fsformat.c
67
68 $(OBJDIR)/fs/clean-fs.img: $(OBJDIR)/fs/fsformat $(FSIMGFILES)
69         @echo + mk $(OBJDIR)/fs/clean-fs.img
70         $(V)mkdir -p $(@D)
71         $(V)$(OBJDIR)/fs/fsformat $(OBJDIR)/fs/clean-fs.img 1024 $(FSIMGFILES)
72
73 $(OBJDIR)/fs/fs.img: $(OBJDIR)/fs/clean-fs.img
74         @echo + cp $(OBJDIR)/fs/clean-fs.img $@
75         $(V)cp $(OBJDIR)/fs/clean-fs.img $@
76
77 all: $(OBJDIR)/fs/fs.img
```

Support for chdir

- The native JOS FS is very poor in terms of file navigation.
- There is no support for moving between different directories.
- So first I had to add the cd command.

Shared PATH

- cd is switching the PATH for all shell commands
- Therefore, the new PATH needs to be saved somewhere after the cd process exits.
- So where and how to store it?
- My first attempt was very naive..

Shared PATH – First try

- Keep PATH as a global variable in fs.h
- Write two functions in fs.c that read and write it.
- Include them in inc/lib.h

It doesn’t work!

- Changes are not synchronized.
- User environments cannot access a variable that lives in the memory space of the FS environment.
- cd change its private memory only.

Shared PATH – Second try

- Save PATH in the kernel space.
  - Wrote a new file: kern/path.c
- PATH will be a kernel variable only, and it will copy for the user with strcpy(user_path, PATH)
- Read and update through syscalls
  - sys_chdir – for cd
  - sys_get_shell_path – for any shell command that wants to read PATH
- In the meantime, I saw that xv6 also serves cd with sys_chdir and I was sure I had solved it.

Shared PATH – Second try

That didn't work either.

- And I’m not sure why..

  Kern/syscall.c:

```
int sys_get_shell_path(char *user_buf) {
     strcpy(user_buf, PATH);  // page fault from kernel stack here
     return 0;
}
```

- Eventually I solved the page fault
- But then I got the same behavior again as in the first try

Shared PATH

- I had a few more ideas
  - Create shared pipe like FIFO
  - Create a new IPC request for sharing PATH between the FS env and users envs
  - Write it to a file
- But then I came up with the simplest solution I could think of – a shared page allocated by the shell!

Shared PATH

Added a macro that preserve a free user address for the PATH.

inc/lib.h

```
28
29 // SHELL-PATH address
30 #define PATH_VA         ((void*) 0xB0000000)    // PROJECT
31
```

And shell’s main() allocate a new page there with PTE_SHARE permission.

user/sh.c

```
335
336         // Alloc a page that will be shared with every shell child
337         // This page will keep the current PATH and will update by cd command
338         if((r = sys_page_alloc(thisenv->env_id, PATH_VA, PTE_U | PTE_P | PTE_W | PTE_SHARE)) < 0) // PROJECT
339                 panic("init_PATH: sys_page_alloc return %e\n", r);
340
```

And that's it!

Shared PATH

- Now, every user who wants a PATH access all they need to do is declare this global variable.

user/cd.c, user/ls.c, user/touch.c , user/track.c, user/undo.c

```
1 #include <inc/lib.h>
2
3 char* PATH = (char*)PATH_VA;
4
```

- Since the shell creates all users with fork, the PATH will always be in sync.

lib/fork.c

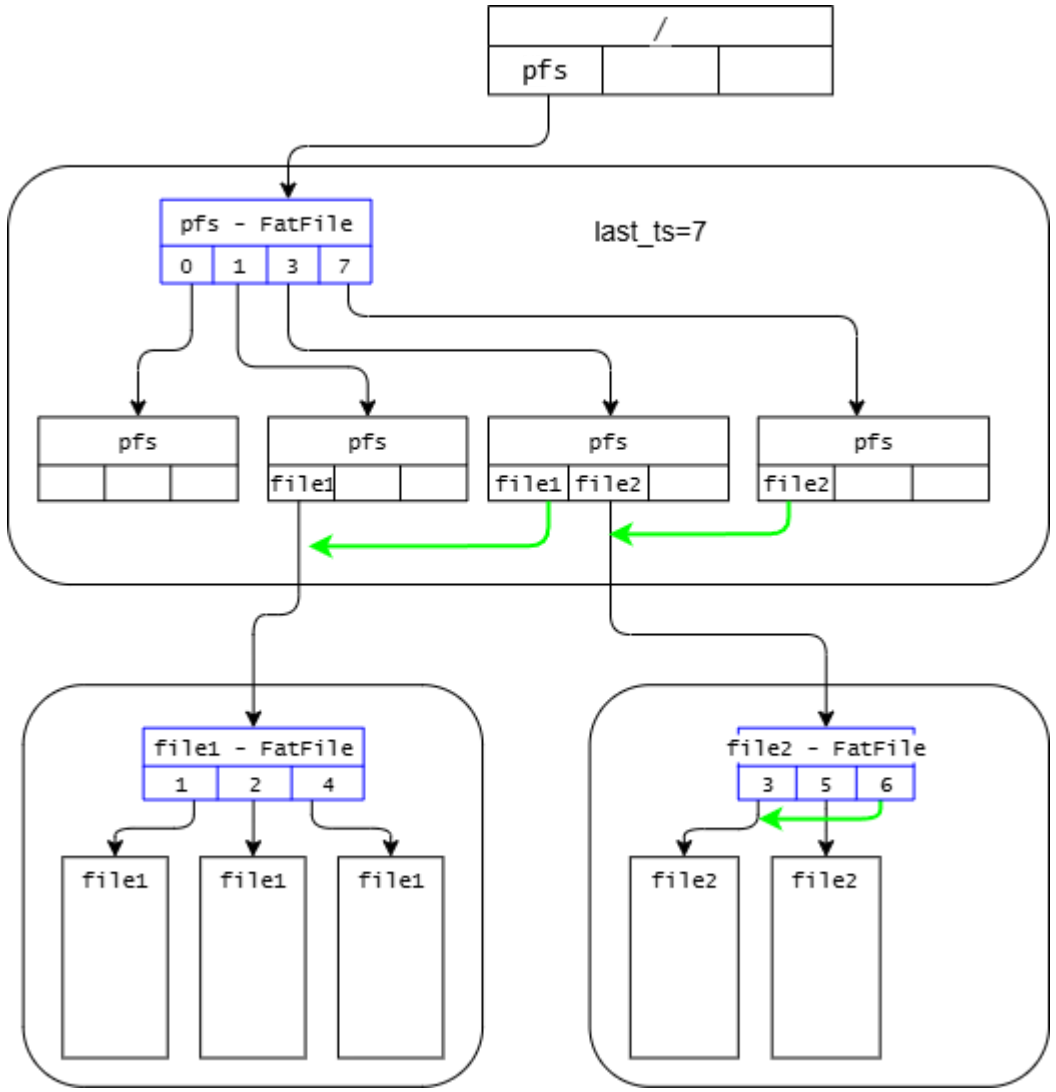
```
75
76          if(uvpt[pn] & PTE_SHARE){
77
78                  if((r = sys_page_map(parent_envid, pgaddr, envid, pgaddr, uvpt[pn] & PTE_SYSCALL)) < 0)
79                          return r;
80          }
81
```

user/cd.c

```
2  int
3  chdir(char* new_path)
4  {
5          int i, cur_len;
6
7          if(!new_path)
8                  return -E_BAD_PATH;
9
10         if(new_path[0] == '/'){
11                 // user gave absolute path
12                 strcpy(PATH, new_path);
13                 goto chdir_end;
14         }
15
16         // support for "cd .." and  "cd ../../<dir>"
17         while(strlen(new_path) > 1 && new_path[0] == '.' && new_path[1] == '.'){
18
19                 new_path += 2;
20                 path_pop();
21
22                 if(*new_path == '\0')
23                         goto chdir_end;
24
25                 if(new_path[0] != '/')
26                         return -E_BAD_PATH;
27
28                 new_path++;
29         }
30
31         cur_len = strlen(PATH);
32
33         if(cur_len + strlen(new_path) + 1 > MAXPATHLEN)
34                 panic("PROJECT: chdir: strlen(%s) + strlen(%s) + 1 > MAXPATHLEN\n", PATH, new_path);
35
36         if(cur_len > 0 && PATH[cur_len - 1] != '/')
37                 strcat(PATH, "/");
38
39         strcat(PATH, new_path);
40
41 chdir_end:
42         PATH[strlen(PATH)] = '\0';
43         return 0;
44 }
```

cd command

Writing to a FatFile – Implementation

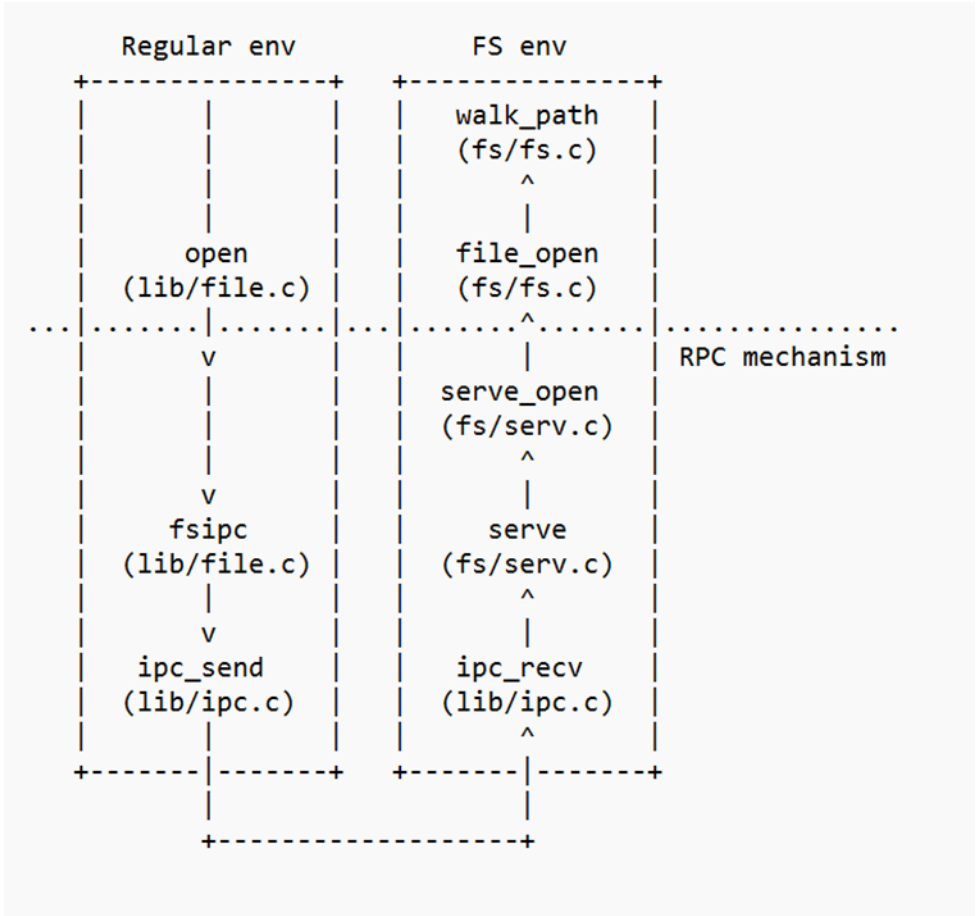


Some Diagnoses and Conclusions

1. Every change creates a new timestamp
   - → Every file is written exactly once
     - → Writing to a file always starts from an empty file
     - → Blocks always start aligned
2. Files are never removed
   - → We don’t need to handle shared pointers

Writing to a FatFile

- Everything happens in open()
  - File Descriptor contains both the file itself and its fatfile
- The manipulation and management of fatfile is done (almost) only from walk_path()



Writing to a FatFile – walk_mode

- fs/serv.c: serve_open()

```
120
121         if((req->req_omode & O_CREAT) || (req->req_omode & O_WRONLY))   // PROJECT
122                 walk_mode = WALK_CREATE;
123         else
124                 walk_mode = WALK_RDONLY;
125
```

- fs/fs.c: walk_path()

```
416
417             if(dir->f_type & FTYPE_FF){     // PROJECT
418
419                     *ff = dir;
420
421                     if(walk_mode == WALK_CREATE && dir->f_timestamp < super->last_ts){
422
423                             if((dir = ff_lookup(dir)) == 0)
424                                     panic("PROJECT: walk_path: ff_lookup return NULL for dir.f_name=%s while super->last_ts=%d\n", (*f
425
426                             dir = file_shalldup(*ff, dir);
427                             (*ff)->f_timestamp = super->last_ts;
428                     }
429                     else if((dir = ff_lookup(dir)) == 0)
430                             panic("PROJECT: walk_path: ff_lookup return NULL for dir.f_name=%s while super->last_ts=%d\n", (*ff)->f_na
431             }
432
```

Writing to a FatFile – ff_lookup

fs/fs.c

```
244 // PROJECT: New function.
245 // Return the file/dir from fatfile according to requested ts
246 static struct File*
247 ff_lookup(struct File* ff)      // PROJECT
248 {
249         int r;
250         uint32_t i, j, nblock;
251         char* blk;
252         struct File *f, *ret_f;
253
254         if((ff->f_type & FTYPE_FF) == 0)
255                 return ff;
256
257         ret_f = 0;
258
259         // We maintain the invariant that the size of a fatfile
260         // is always a multiple of the file system's block size (like a directory-file).
261         assert((ff->f_size % BLKSIZE) == 0);
262         nblock = ff->f_size / BLKSIZE;
263
264         // TODO: Performance can be improved by replacing this loop with a binary search.
265         for(i = 0; i < nblock; ++i){
266
267                 if((r = file_get_block(ff, i, &blk)) < 0)
268                         panic("ff_lookup: file_get_block return %d for file %s with track_ts %d\n", r, ff->f_name, track_ts);
269
270                 f = (struct File*)blk;
271
272                 for(j = 0; j < BLKFILES; ++j){
273
274                         if(f[j].f_name[0] == '\0')
275                                 continue;
276
277                         if(f[j].f_timestamp <= track_ts)
278                                 ret_f = &f[j];
279                 }
280         }
281         return ret_f;
282 }
283
```

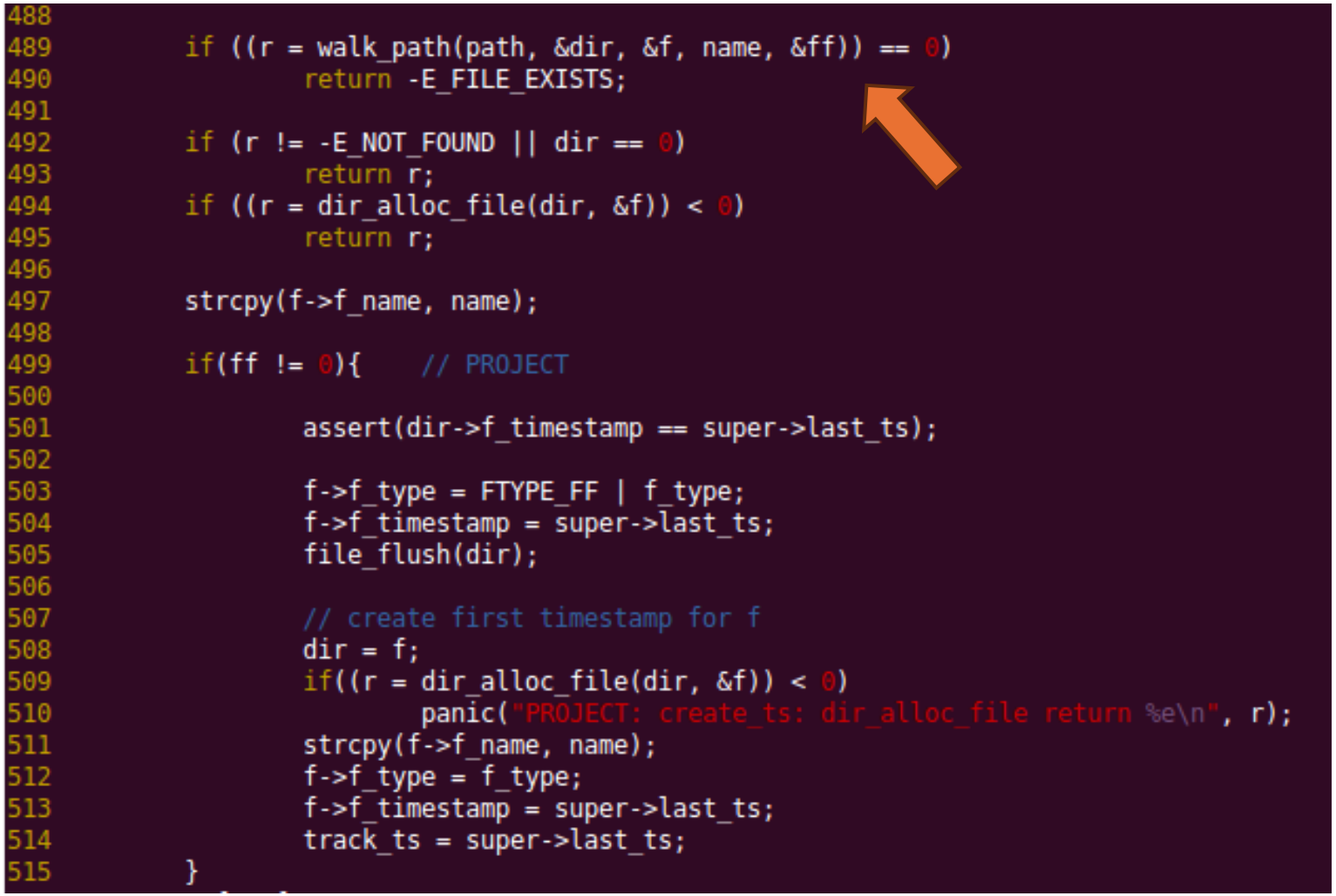
```
323 // copy blocks numbers form fromfile to a new file (dir/reg).
324 // if fromfile is reg, last block will deep copy to support appending to it without page fault.
325 struct File*
326 file_shalldup(struct File *ff, struct File *fromfile)   // PROJECT
327 {
328         int r;
329         uint32_t i, last_bn;
330         struct File *f, *newfile;
331         void *buf;
332         size_t count;
333         off_t offset;
334
335         if((r = dir_alloc_file(ff, &newfile)) < 0)
336                 panic("PROJECT: file_shalldup: dir_alloc_file return %e\n", r);
337
338         strcpy(newfile->f_name, fromfile->f_name);
339         newfile->f_type = fromfile->f_type;
340         newfile->f_timestamp = super->last_ts;
341
342         if(fromfile->f_type & FTYPE_DIR)
343                 last_bn = (fromfile->f_size + BLKSIZE - 1) / BLKSIZE;
344         else
345                 last_bn = fromfile->f_size / BLKSIZE;
346
347         for(i = 0; i < MIN(NDIRECT, last_bn); ++i)
348                 newfile->f_direct[i] = fromfile->f_direct[i];
349
350         if(last_bn >= NDIRECT){
351
352                 if((newfile->f_indirect = alloc_block()) < 0);
353                         panic("PROJECT: file_shalldup: we are out of blocks\n");
354
355                 memmove(diskaddr(newfile->f_indirect), diskaddr(fromfile->f_indirect), BLKSIZE);
356         }
357
358         if(fromfile->f_type & FTYPE_DIR || fromfile->f_size == last_bn * BLKSIZE){
359                 newfile->f_size = fromfile->f_size;
360                 return newfile;
361         }
362
363         newfile->f_size = last_bn * BLKSIZE;    // Only whole blocks
364
365         // deep copy for last block
366         if(last_bn < NDIRECT)
367                 buf = diskaddr(fromfile->f_direct[last_bn]);
368         else
369                 buf = diskaddr(((uint32_t*)diskaddr(fromfile->f_indirect))[last_bn - NDIRECT]);
370         count = fromfile->f_size % BLKSIZE;
371         offset = last_bn * BLKSIZE;
372
373         if((r = file_write(newfile, buf, count, offset)) < 0)
374                 panic("PROJECT: file_shalldup: file_write return %e\n", r);
375         if(r != count)
376                 panic("PROJECT: file_shalldup: file_write wrote only %d bytes\n", r);
377
378         assert(newfile->f_size == fromfile->f_size);
379         return newfile;
380 }
```

fs/fs.c

Writing to a FatFile – file_shalldup

Writing to a FatFile – New FatFile

fs/fs.c: file_create()



```
488
489         if ((r = walk_path(path, &dir, &f, name, &ff)) == 0)
490                 return -E_FILE_EXISTS;
491
492         if (r != -E_NOT_FOUND || dir == 0)
493                 return r;
494         if ((r = dir_alloc_file(dir, &f)) < 0)
495                 return r;
496
497         strcpy(f->f_name, name);
498
499         if(ff != 0){    // PROJECT
500
501                 assert(dir->f_timestamp == super->last_ts);
502
503                 f->f_type = FTYPE_FF | f_type;
504                 f->f_timestamp = super->last_ts;
505                 file_flush(dir);
506
507                 // create first timestamp for f
508                 dir = f;
509                 if((r = dir_alloc_file(dir, &f)) < 0)
510                         panic("PROJECT: create_ts: dir_alloc_file return %e\n", r);
511                 strcpy(f->f_name, name);
512                 f->f_type = f_type;
513                 f->f_timestamp = super->last_ts;
514                 track_ts = super->last_ts;
515         }
```

touch command

user/touch.c

```
 8
 9 void
10 touch(int argc, char** argv){
11
12         int i, fd;
13         char file_path[MAXPATHLEN];
14
15         if(argc == 1)
16                 return;
17
18         for(i = 1; i < argc; ++i){
19
20                 if(argv[i][0] == '/')
21
22                         strcpy(file_path, argv[i]);
23                 else{
24                         strcpy(file_path, PATH);
25
26                         if(strlen(file_path) > 1)
27                                 strcat(file_path, "/");
28
29                         strcat(file_path, argv[i]);
30                 }
31
32                 if((fd = open(file_path, O_CREAT)) < 0){
33
34                         printf("can't open %s: %e\n", file_path, fd);
35                         return;
36                 }
37
38                 close(fd);
39         }
40 }
41
```

mkdir command

user/mkdir.c

```
 4
 5 void
 6 mkdir(char* path)
 7 {
 8         int fd;
 9         struct Stat st;
10
11         if((fd = open(path, O_CREAT | O_MKDIR)) < 0){
12                 printf("can't open %s: %e\n", path, fd);
13                 return;
14         }
15         close(fd);
16 }
17
```

Read from a FatFile

Added support for opening a file by timestamp without affecting existing behavior

inc/lib.h

```
31
32 #define TS_UNSPECIFIED  MAX_SSIZE       // PROJECT
33
```

lib/fd.c

```
315 int
316 stat_ts(const char *path, struct Stat *stat, ts_t req_ts)
317 {
318         int fd, r;
319
320         if ((fd = open_ts(path, O_RDONLY, req_ts)) < 0)
321                 return fd;
322         r = fstat(fd, stat);
323         close(fd);
324         return r;
325 }
326
327 int
328 stat(const char *path, struct Stat *stat)       // PROJECT
329 {
330         return stat_ts(path, stat, TS_UNSPECIFIED);
331 }
332
```

lib/file.c

```
 2 int
 3 open(const char *path, int mode)
 4 {
 5         return open_ts(path, mode, TS_UNSPECIFIED);
 6 }
 7
 8 int
 9 open_ts(const char *path, int mode, ts_t req_ts)
10 {
11         int r;
12         struct Fd *fd;
13
14         if (strlen(path) >= MAXPATHLEN)
15                 return -E_BAD_PATH;
16
17         if ((r = fd_alloc(&fd)) < 0)
18                 return r;
19
20         strcpy(fsipcbuf.open.req_path, path);
21         fsipcbuf.open.req_omode = mode;
22         fsipcbuf.open.req_ts = req_ts;
23
24         if ((r = fsipc(FSREQ_OPEN, fd)) < 0) {
25                 fd_close(fd, 0);
26                 return r;
27         }
28         return fd2num(fd);
29 }
```

track command

user/track.c

```
 9 void
10 track(char* path, ts_t req_ts)
11 {
12         int fd, i, num_blk;
13         struct Stat st;
14
15         if(req_ts == TS_UNSPECIFIED){
16
17                 for(; req_ts >= 0 && stat_ts(path, &st, req_ts) == 0; req_ts = st.st_ts - 1){
18
19                         printf("%-6d %-7dB   %-20s\tblk_num: [", st.st_ts, st.st_size, st.st_name);
20
21                         num_blk = (st.st_size + BLKSIZE - 1) / BLKSIZE;
22                         for(i = 0; i < MIN(num_blk, NDIRECT); ++i){
23                                 printf(" %d ", st.st_blkn[i]);
24                         }
25                         printf("]\n");
26                 }
27                 return;
28         }
29
30         if((fd = open_ts(path, O_CREAT | O_WRONLY, req_ts)) < 0){
31                 printf("can't open %s: %e\n", path, fd);
32                 return;
33         }
34         write(fd, NULL, 0);
35         close(fd);
36 }
37
```

Writing to a FatFile – serve_write

fs/serv.c

```
258 int
259 serve_write(envid_t envid, struct Fsreq_write *req)
260 {
261         if (debug)
262                 cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);
263
264         // LAB 5: Your code here.
265         struct OpenFile* o;
266         size_t count;
267         int r, i;
268
269         if((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
270                 return r;
271
272         if(o->o_fatfile != 0){  // PROJECT
273
274                 o->o_file = file_shalldup(o->o_fatfile, o->o_file);
275                 o->o_fatfile->f_timestamp = super->last_ts;
276
277                 o->o_fd->fd_offset = o->o_file->f_size;
278         }
279
280         if((r = file_write(o->o_file, req->req_buf, req->req_n, o->o_fd->fd_offset)) < 0)
281                 return r;
282         count = r;
283
284         o->o_fd->fd_offset += count;
285
286         return count;
287 }
```

Appending support

- Added support for the >> operator to the shell
- Using the operator adds the flag O_APPEND to open()
- Which set the fd_offset to be f_size

lib/serv.c: serve_open()

```
185         if(req->req_omode & O_APPEND)   // PROJECT
186                 o->o_fd->fd_offset = o->o_file->f_size;
```

undo command

user/undo.c

```
1  #include <inc/lib.h>
2
3  char* PATH = (char*)PATH_VA;
4
5  void
6  umain(int argc, char** argv)
7  {
8          char path[MAXPATHLEN];
9          int fd;
10
11         if(argc != 2){
12                 printf("usage: undo <file>\n");
13                 return;
14         }
15         if(argv[1][0] == '/')
16                 strcpy(path, argv[1]);
17         else{
18                 strcpy(path, PATH);
19                 if(strlen(path) > 1)
20                         strcat(path, "/");
21                 strcat(path, argv[1]);
22         }
23         if((fd = open_ts(path, O_CREAT | O_WRONLY, -1)) < 0){
24                 printf("can't open %s: %e\n", path, fd);
25                 return;
26         }
27         write(fd, NULL, 0);
28         close(fd);
29 }
30
```

- It is held that:

  undo <file> ≡ track –t –1 <file>

- It would have made more sense to use ‘track’.

Time-traveling File System on JOS

- My Idea
- Implementation Details
- Proof Of Concept

Things I didn't get to do

- rm command
  - deleting a file from folder
- Creating a new PFS directory
  - using the -f flag for touch/mkdir
- Reading older timestamps without duplicate