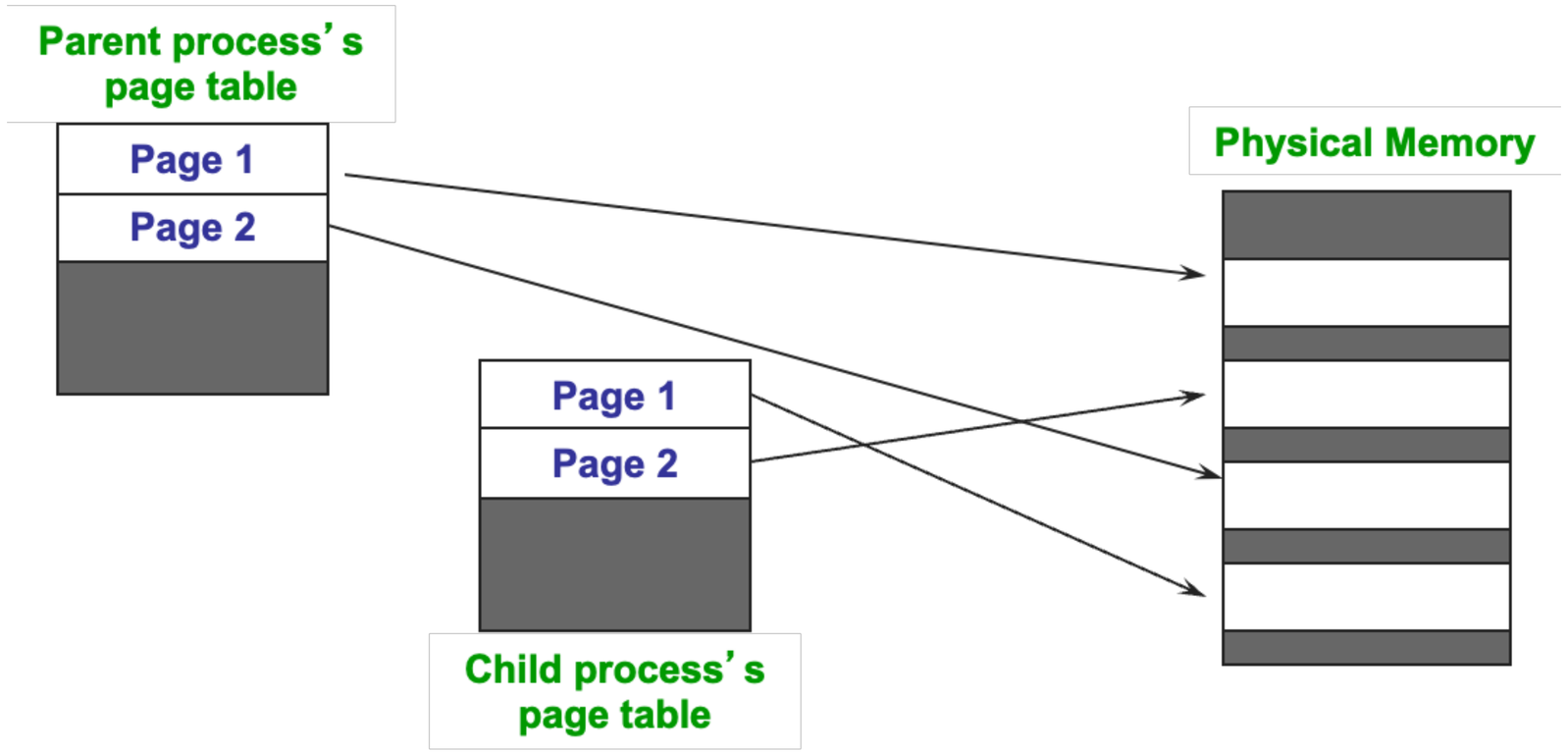
CS179F: Projects in Operating System

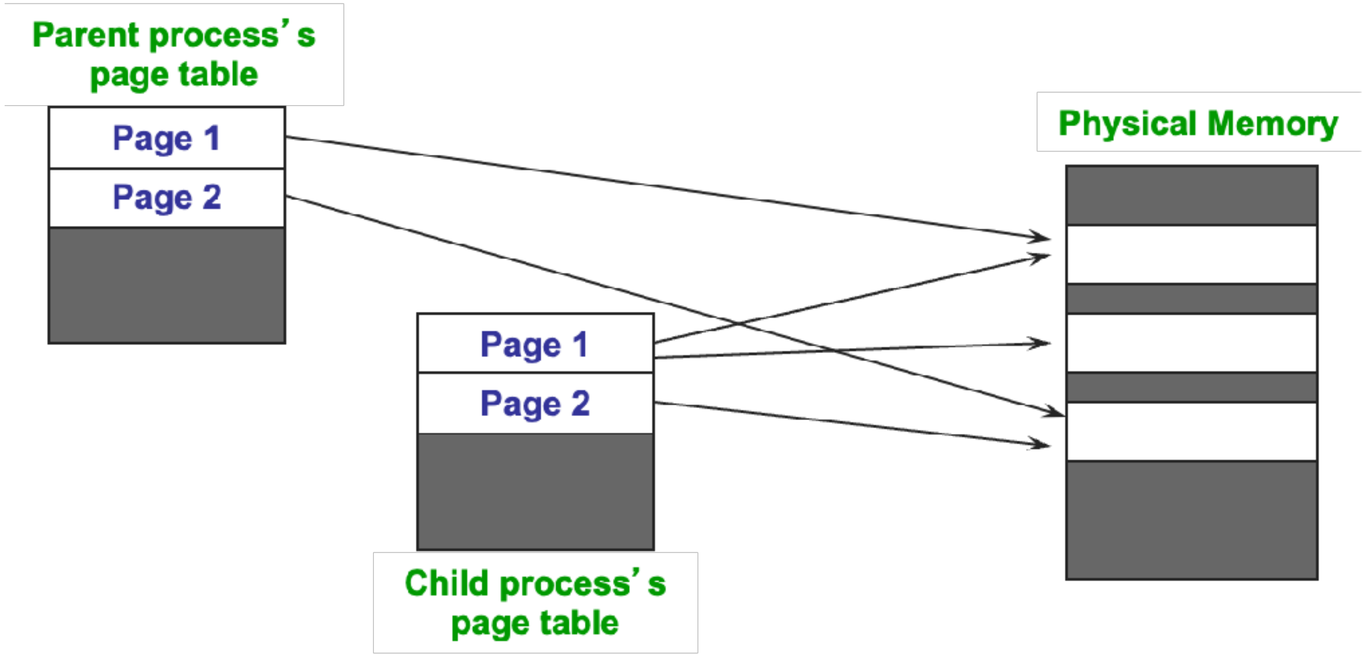
Lab 4: File System

Emiliano De Cristofaro and Lian Gao

fork() without CoW



fork() with CoW



Things to pay attention to

- What is shared?
  - Physical pages, address translations, page tables?
- Page fault handling, how do you know it’s caused by CoW?
- When allocating a new physical page, map it to the parent or child’s address space?

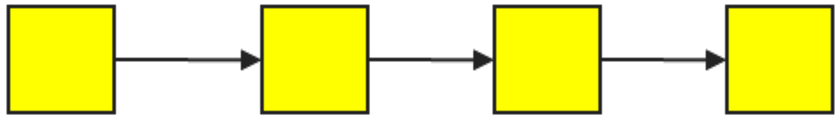
File system design

- How to allocate and keep track of files and directories?
- Does it matter? What is the difference?
  - Performance, reliability, limitations on files, overhead, …
- Many different file systems have been proposed and continue to be proposed

Disk layout strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?
  - Continues allocation

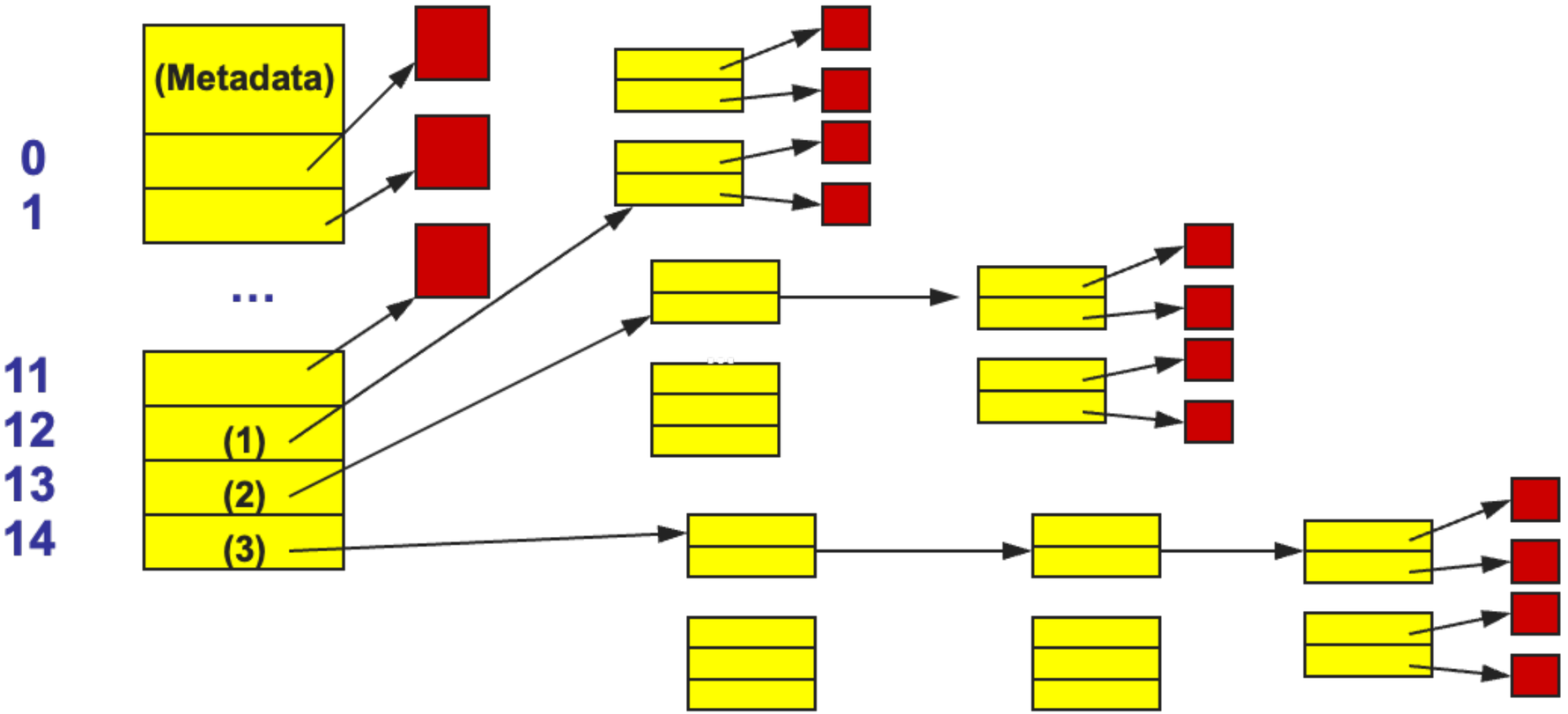
  - Linked structure



  - Indexed structure (indirection, hierarchy)

UNIX inode

An indexed structure for files



Links

- What is a file?
  - From the OS perspective: the file object (a collection of disk blocks and the corresponding metadata)
  - From users perspective?
- **A link is a pointer to a file**
  - Hard link
  - Symbolic link

Hard Links

Or directory entries

- Hard link is a reference to a file object (e.g., an inode)
  - All named files are hard links!
  - More than one name can refer to the same file object
- Example
  - `ln myfile mylink`
  - `ls -il`

```
38753 -rw-rw-r-- 2 uli  uli    29 Oct 29 08:47 myfile
38753 -rw-rw-r-- 2 uli  uli    29 Oct 29 08:47 mylink
 ^                ^
 |-- inode #      |--link counter (2 links)
```

Hard Links

Operations and Limitations

- How to create/remove a hard link?
  - `link/unlink` <- system calls
- Hard links can only refer to data that exists on the **same** file system
  - Why?
- You **cannot** create hard link to a directory
  - Why?

Symbolic Links

- A symbolic link is an indirect pointer to a file – a file whose content is a "pointer/reference" to another "file"
  - "file" actually means a hard link / directory entry
  - so "pointer/reference" means a path
- Example
  - `ln -s myfile symlink`
  - `ls -li myfile symlink`

```
44418 -rw-rw-r-- 1 uli uli  49 Oct 29 14:33 myfile
44410 lrwxrwxrwx 1 uli uli   6 Oct 29 14:33 symlink -> myfile
 ^    ^File type ^
 |-- inode #     |--link counter (2 links)
```

Symbolic Links

- How does the OS handle symbolic link? (lab4)
  - A symbolic link can be a relative path or a absolute path
- A symbolic link can point to a file on a different file system
  - Why?
- A symbolic link can point to a non-existent file (referred to as a “broken link”)
  - Why?

Symbolic Links

- You can create a symbolic link to a directory
  - Why?
- Can we create loops?
  - YES!
  - How to avoid being trapped in a loop?

Lab 4

- xv6 files are limited to 268 blocks, or 268*BSIZE bytes (BSIZE is 1024 in xv6)
  - This because an xv6 inode contains 12 "direct" block numbers and one "singly-indirect" block number, which refers to a block that holds up to 256 more block numbers, for a total of 12+256=268 blocks.
- In this life, we will learn how to increase the maximum size of an xv6 file
- The bigfile command creates the longest file it can, and reports that size:

Bigfile

- The **bigfile** command creates the longest file it can, and reports that size:

- The test fails because bigfile expects to be able to create a file with 65803 blocks, but the unmodified xv6 limits files to 268 blocks.
- Change xv6 to support a "doubly-indirect" block in each in-ode, containing 256 addresses of singly-indirect blocks, each of which can contain up to 256 addresses of data blocks.
- Will create file will be able to consist of up to 65803 blocks, or 256*256+256+11 blocks
  - 11 instead of 12 because we will sacrifice one of the direct block numbers for the double-indirect block

mkfs

- Creates the xv6 file system disk image and determines how many total blocks the file system has
- Size controlled by FSSIZE in kernel/param.h (set to 200k blocks in repo)
- You should see the following output from mkfs/mkfs in the make output:

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks
   25) blocks 199930 total 200000
```

- This line describes the file system that mkfs/mkfs built: it has 70 meta-data blocks (blocks used to describe the file system) and 199,930 data blocks, totaling 200,000 blocks.
- If you need to rebuild the file system from scratch, you can run make cleanwhich forces make to rebuild fs.img

What to look at (1/2)

- Format of on-disk inode is defined by struct dinode in fs.h
  - Look at NDIRECT, NINDIRECT, MAXFILE, and the addrs[] element of struct dinode
  - Look at Figure 8.3 in the xv6 text for a diagram of the standard xv6 inode

- The code that finds a file's data on disk is in bmap() in fs.c.
  - Look at it and understand it. bmap() is called both when reading and writing a file
  - When writing, bmap() allocates new blocks as needed to hold file content, as well as allocating an indirect block if needed to hold block addresses

What to look at (2/2)

- Note: bmap() deals with two kinds of block numbers.
  - The bn argument is a "logical block number — a block number within the file, relative to the start of the file
  - The block numbers in ip->addrs[], and the argument to bread(), are disk block numbers
  - You can view bmap() as mapping a file's logical block numbers into disk block numbers

Your job

- Modify bmap() so that it implements a doubly-indirect block, in addition to direct blocks and a singly-indirect block
- You'll have to have only 11 direct blocks, rather than 12, to make room for your new doubly-indirect block; you're not allowed to change the size of an on-disk inode
- The first 11 elements of ip->addrs[] should be direct blocks; the 12th should be a singly-indirect block (just like the current one); the 13th should be your new doubly-indirect block

Your job

- You are done with this exercise when bigfile writes 65803 blocks and usertests runs successfully:

```
$ bigfile
..........................................................
wrote 65803 blocks
done; ok
$ usertests
...
ALL TESTS PASSED
$
```

- Hints: see https://github.com/emidec/cs179f-fall23/blob/xv6-riscv-fall23/doc/lab4.md

Symbolic Links

- Next, you will add symbolic links to xv6
  - Symbolic links (or soft links) refer to a linked file by pathname; when a symbolic link is opened, the kernel follows the link to the referred file
  - Symbolic links resembles hard links, but hard links are restricted to pointing to file on the same disk, while symbolic links can cross disk devices
- Although xv6 doesn't support multiple devices, implementing this system call is a good exercise to understand how pathname lookup works

Your job

- You will implement the symlink(char *target, char *path) system call, which creates a new symbolic link at path that refers to file named by target.
  - For further information, see the man page symlink.
  - To test, add symlinktest to the Makefile and run it.
- Your solution is complete when the tests produce the following output (including usertests succeeding):

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
...
ALL TESTS PASSED
$
```

Hints: see git repo