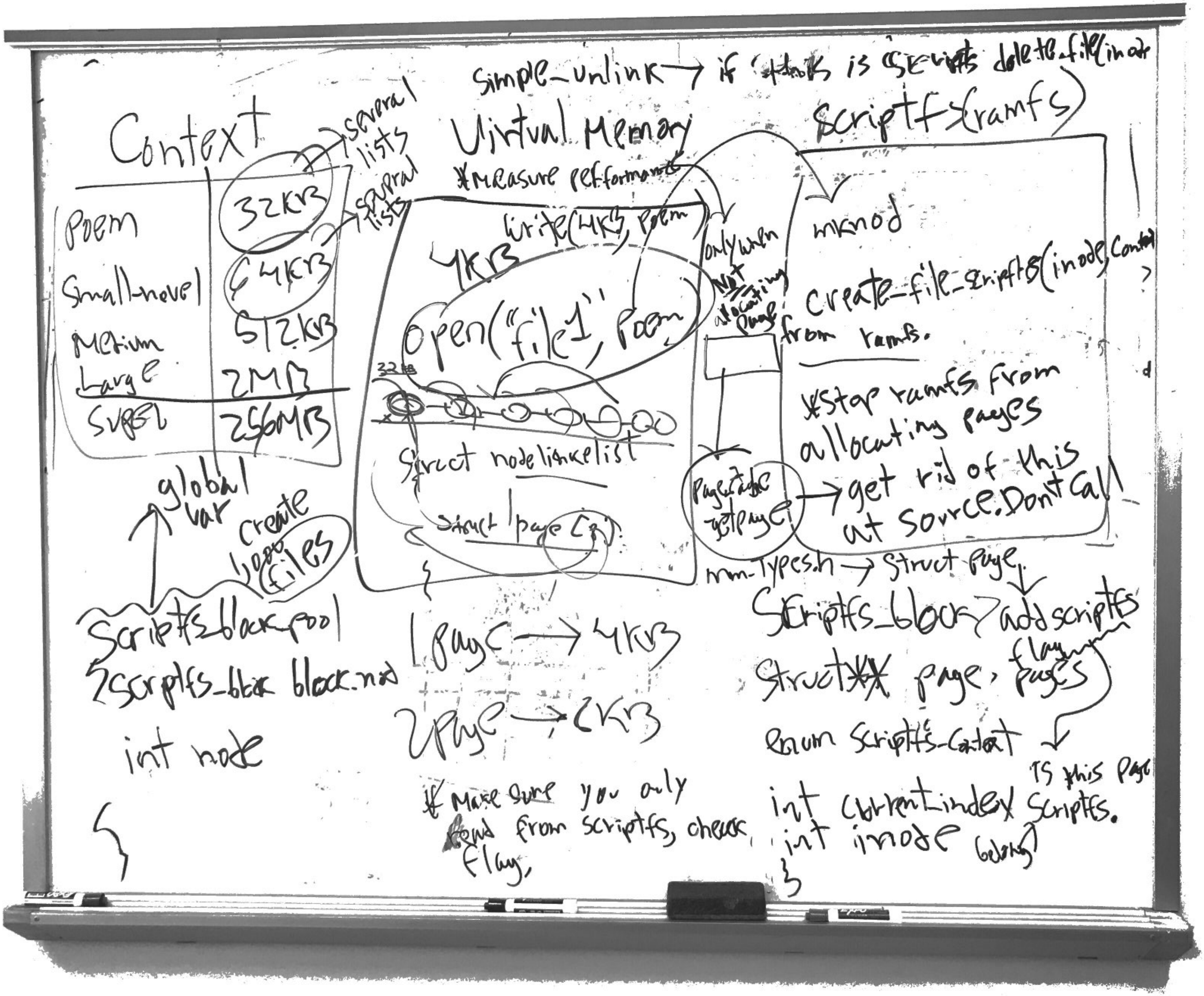


Simple_unlink → if ~~stacks~~ is ~~scripts~~ delete_file(inode

Context

Poem	32KB
Small-novel	64KB
Metium	512KB
Large	2MB
Super	256MB

→ several lists

→ several lists

global var

create 1,000 files

Scriptfs_block_pool

{ scriptfs_blk block_nod

int node

}

Virtual Memory

*measure performance

4KB

write(4KB, Poem)

open("file1", Poem)

32KB

Struct nodelinkelist

Struct page [3]

}

1 page → 4KB

2 page → 8KB

* Make sure you only read from scriptfs, check flag,

Scriptfs (ramfs)

only when Not allocating page

mknod

create_file_scriptfs(inode, Context)

from ramfs.

*Stop ramfs from allocating pages

Pagetable getpage → get rid of this at source. Dont call

mm_types.h → Struct page

↓ add scriptfs flag

Scriptfs_block {

Struct ~~**~~ page, pages

enum Scriptfs_Context

int currentindex

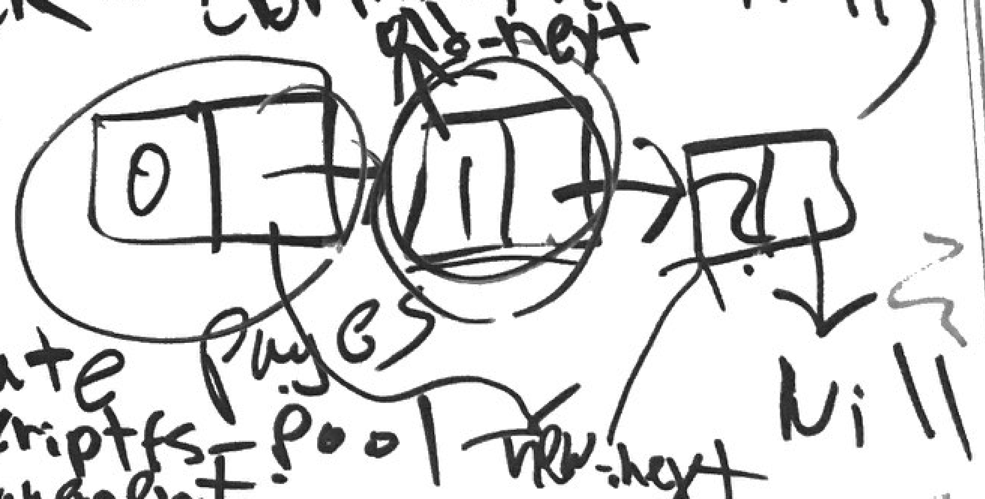
int inode

}

→ Is this Page Scriptfs.

belong

Struct scriptfs-fs_block * current_file=Null;



mknod:

pre-allocate pages

add to scriptfs_pool

write_management:

fetch nex_page(inode)

struct Scriptfs_block add_to_pool(page)

{ Struct page** pages;

int current_index;

enum scriptfs_context context;

}

after

80 MB before

global

local_ptr

write function trace for 1KB:

1. mknod → creat file ant inode

2. Simple_write_begin

3. Simple_write_begin len arg = 1024

4. simple_write_begin pos arg = 0

5. Simple_write_begin calls grab_cache_page_write

6. It test if the page returned from grab_cache_page is valid.

7. It calls pageupdate

If pageupdate ~~succeeds~~ fails, It writes a zero_segment to that page

It looks like it allocates a page.

THEN After the write ~~a~~ is over write_end is called to release the pages (give back) ~~to memory~~

~~1986~~ → Pid

write function trace for 2KB:

For the most part the same as 1KB call, but len is different.

len = 2048

4k alloc → write_begin, write_end

ramfs basically allocates 4K chunks on demmand to the same inode host

$\frac{28672}{4096}$

1st Type of block → 32KB

1879 → Pid

1882

1885

1889

File writing:

1. mknod → create inode

2. ~~wrap~~ wrapper-generic_write

3. Real generie_generic_write

performs checks such as update notifications

4. generic_perform_write():

Calls write_begin

↓

address space operations for ramfs

→ translates/points to simple_write_begin

Simple-write_begin()
preparation

→ grap page
write them

it does Not

returns the status of a page!

(whether it is dirty or not

allocates memory pages

ext4 → does Not do this.

write_end()

does the actual writing and metadata

updatate time

actually write data to file

- Every time a file [illegible]

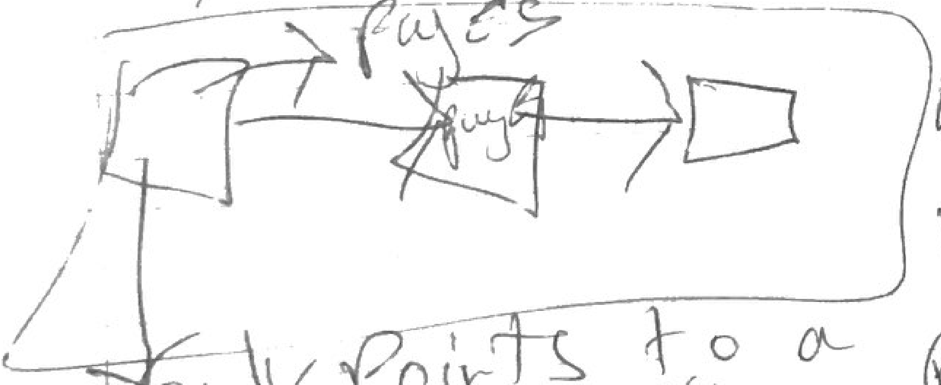
allocate additional pages/blocks

Pre-allocate some extra pages

keep these pages in a list

16KB, 32KB

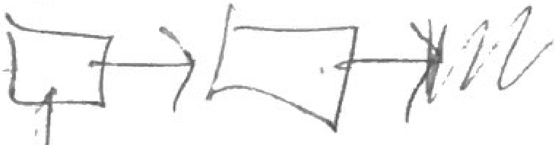
- multiple linked_list



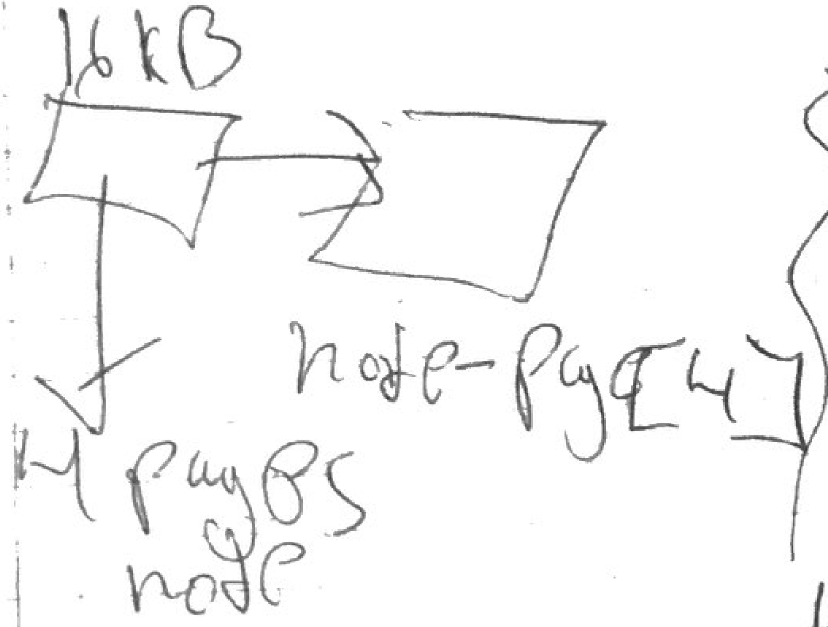
4K linked list

only points to a page

8K Linked-list



- points to two pages



mount

pre-allocate
1,000 pages
when mounting
filesystem

give me 2 blocks → get
reg-page
instead of calling grab-page-cache
→
call your own
function that
accesses to
the preallocated
pages from the
beginning

When you write,
you have

write(4K ↓ 4K)

write8(16K)

↓

for 16K-block linked-lists

when delete files
get rid of pages

→ important

Page Allocation trace

1. generic_write:

1a. write_begin } → allocates page

1b. grab_cache_page_write_begin

↓

~~this function checks if~~

If page is cached:
return page with increased refcount

else:
Allocate a new page
goto no_page:

__page_cache_alloc

this allocates a new page.

→ calls cpuset_do_page_mem_spread.

what does this do??

→ then it calls read_mems_allowed_begin()

If there is a page allocated
Don't call this function.

cpuset_* functions look like hardware directives that are specific to archetecture/hardware.

*Then we call __alloc_pages_node

this ~~try~~ tries ~~its~~ its best to allocate a page with the passed node id.

→ then, it calls __alloc_pages

this calls __alloc_pages_nodemask

the "heart" of the zone buddy allocator, according to the linux docs.

*uses the alloc_context struct

→ calls prepare_alloc_pages

↓

fills in alloc_context struct

If prepare_alloc_pages fails
then return NULL

else:

Call finalise_ac

alloc_context struct

Then, call get_page_from_freelist.

iterates through a zones list

zone struct:

*we can use __page_cache_alloc to ~~allocate~~ pages.

*we can use alloc_pages function.

~~* we're~~

* we're missing the mapping from our custom page's address space to the file's address space. *

→ __add_to_page_cache_locked

this function maps the page's address space to the file's address space.

→ this function is ~~called~~ called by add_to_page_cache_lru.

ratix_tree_lookup_slot just checks if the page is cached in the address space.

→ shall ignore this when pre-allocating

Testing new kernel.

Syscalls test ✓

Scriptfs header file ✓

ramfs/inode.c ✓

mm/filemap.c — Compilation Failed

→ block of code which was allocating a page inside fill_super_block function was preventing kernel, from booting.

#9, #10, #14, #21

Trace for new file:
lock page does NOT run ✓

*when appending data to file
fgp-flags and FGP LOCK is true:
FGP-Wait is NOT true

* pagecache_get_page is getting called by someone before write_begin
↓
who is it?

*check ~~fo~~ find_get_entry * for Null page.

~~ffli: Cf~~ (filemap.c, buffer.c)

Page Allocation trace:

pagecache_get_page() gets ~~called a couple of times (2 times)~~ gets called ~~4~~ once.

~~on the se~~ Then, getblk_slow() gets called.

~~then~~ pagecache_get_page() gets called again

* By the time it returns, the reference count for the page is -1. *

this is a problem: Someone is messing with our reference counts!!!

__find_get_block does this!!

the out_unlock label does this!!!

* There's a lot of uncertainty surrounding page allocation/filesystem

mapping; this is why the kernel needs so many locks, page locks and sometimes it tries to guess whether a page is there or not. My gut tells me that because entities like the DMA aren't 100% accurate so ~~&~~ we need come up with workarounds/hacks that can potentially introduce bugs into the system and create memory leaks.

let's find out what create-page flags are?

No flag assighment happens

however

Charge log :-

1. *Ignore page allocation when scriptfs is mounted*

This approach works

Simple usually works

Now let's find out how ramfs allocates a second page

ramfs just calls alloc-pages again

~~A~~ What are the offset values when allocating pages??

when you allocate a new page, this is incremented by 1.

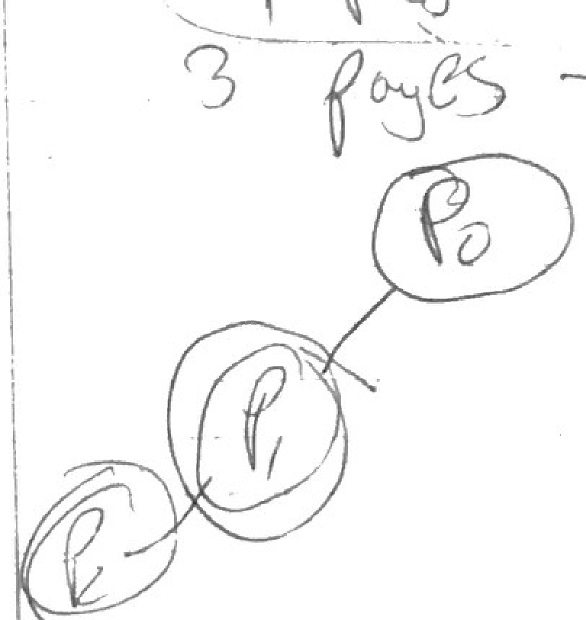
* the offset is used on pagecache-get-page to pass it to find-get-entry to do a radix-tree loop up and find out if the page is already ~~ch~~ cached or Not!!

→ I don't think we'll need to really use this...

→ 1 page → offset = 0

2 pages → offset = 1

3 pages → offset = 2



Tracing New poems allocation

pre-allocate poem pages ✓ works

writing to poem_page

this works, but sometimes linux appears to want to allocate more pages than needed

Scriptfs needs to ~~keep~~ keep track of this!

Check if page-alloc fails

I think the python script is removing the pages when it trims the file!

print panic message on put page.

scriptfs panic!

Improve panic for scriptfs!

0KB → 1 page
4KB → 2 pages
8KB → 3 pages
12KB → 4 pages
16KB → 5 pages
20KB → 6 pages
24KB → 7 pages
28KB → 8 pages

Poem

Who gets called when ramfs is unmounted?

* modified mempolicy → alloc_pages_current

* page_to_nid → looks interesting