Welcome $\text{back(back)}^{\text{back(back)}}\text{back(back)}$ to CS439H!

No quiz everybody meow!

Stress

- 439H is **not an easy class**
  - Lots of new material
  - Unfamiliar programming environments
  - Fast, often relentless pace
- Struggling in this course is normal
  - There will be times you won't know the answer or solution
  - This is expected - we want everyone to succeed, but the only way we can help is if you ask for it
- If you find yourself overwhelmed or spending more time on this class than you think you should be, **please *reach out*** to Dr. Gheith or the TAs
  - We can help out as far as the class goes
  - We can provide other resources if we are not able to help

Mental health resources available at UT

```
execl(
   "/sbin/feedback",
   "p5",
   NULL
);
```

How is p5 going?

A. that's a thing?
B. Cloned the project.
C. Looked through the starter code.
D. Started planning/writing code
E. Done with at least one part of the project
F. Done with the whole project but still failing a couple test cases
G. Fully syscalling

Question 1:

Base Superblock Fields

These fields are present in all versions of Ext2

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Total number of inodes in file system
4	7	4	Total number of blocks in file system
8	11	4	Number of blocks reserved for superuser (see offset 80)
12	15	4	Total number of unallocated blocks
16	19	4	Total number of unallocated inodes
20	23	4	Block number of the block containing the superblock (also the starting block number, NOT always zero.)
24	27	4	log_2 (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)
28	31	4	log_2 (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)
32	35	4	Number of blocks in each block group
36	39	4	Number of fragments in each block group
40	43	4	Number of inodes in each block group
44	47	4	Last mount time (in POSIX time)

Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Block address of block usage bitmap
4	7	4	Block address of inode usage bitmap
8	11	4	Starting block address of inode table
12	13	2	Number of unallocated blocks in group
14	15	2	Number of unallocated inodes in group
16	17	2	Number of directories in group
18	31	X	(Unused)

Question 1:

If block size is wrong, we can’t even access the BGDT

Base Superblock Fields

These fields are present in all versions of Ext2

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Total number of inodes in file system
4	7	4	Total number of blocks in file system
8	11	4	Number of blocks reserved for superuser (see offset 80)
12	15	4	Total number of unallocated blocks
16	19	4	Total number of unallocated inodes
20	23	4	Block number of the block containing the superblock (also the starting block number, NOT always zero.)
24	27	4	log_2 (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)
28	31	4	log_2 (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)
32	35	4	Number of blocks in each block group
36	39	4	Number of fragments in each block group
40	43	4	Number of inodes in each block group
44	47	4	Last mount time (in POSIX time)

Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Block address of block usage bitmap
4	7	4	Block address of inode usage bitmap
8	11	4	Starting block address of inode table
12	13	2	Number of unallocated blocks in group
14	15	2	Number of unallocated inodes in group
16	17	2	Number of directories in group
18	31	X	(Unused)

Question 1:

If block size is wrong, we can't even access the BGDT

If starting block address is wrong, those blocks are corrupted but others are fine

Base Superblock Fields

These fields are present in all versions of Ext2

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Total number of inodes in file system
4	7	4	Total number of blocks in file system
8	11	4	Number of blocks reserved for superuser (see offset 80)
12	15	4	Total number of unallocated blocks
16	19	4	Total number of unallocated inodes
20	23	4	Block number of the block containing the superblock (also the starting block number, NOT always zero.)
24	27	4	log_2 (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)
28	31	4	log_2 (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)
32	35	4	Number of blocks in each block group
36	39	4	Number of fragments in each block group
40	43	4	Number of inodes in each block group
44	47	4	Last mount time (in POSIX time)

Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Block address of block usage bitmap
4	7	4	Block address of inode usage bitmap
8	11	4	Starting block address of inode table
12	13	2	Number of unallocated blocks in group
14	15	2	Number of unallocated inodes in group
16	17	2	Number of directories in group
18	31	X	(Unused)

Question 1:

If block size is wrong, we can't even access the BGDT

If starting block address is wrong, those blocks are corrupted but others are fine

If last mount time is wrong, the filesystem is still fine

Base Superblock Fields

These fields are present in all versions of Ext2

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Total number of inodes in file system
4	7	4	Total number of blocks in file system
8	11	4	Number of blocks reserved for superuser (see offset 80)
12	15	4	Total number of unallocated blocks
16	19	4	Total number of unallocated inodes
20	23	4	Block number of the block containing the superblock (also the starting block number, NOT always zero.)
24	27	4	log_2 (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)
28	31	4	log_2 (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)
32	35	4	Number of blocks in each block group
36	39	4	Number of fragments in each block group
40	43	4	Number of inodes in each block group
44	47	4	Last mount time (in POSIX time)

Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

Starting Byte	Ending Byte	Size in Bytes	Field Description
0	3	4	Block address of block usage bitmap
4	7	4	Block address of inode usage bitmap
8	11	4	Starting block address of inode table
12	13	2	Number of unallocated blocks in group
14	15	2	Number of unallocated inodes in group
16	17	2	Number of directories in group
18	31	X	(Unused)

Question 2:

Given: Block #567858

Block Size: 1024

1024/4 = 256 Pointers per Block

Direct: 12

Singly Indirect: 256

Doubly Indirect: 65536

567858 - 12 = 567846

567846 - 256 = 567590

567590 - 256*256 = 502054

40	43	4	Direct Block Pointer 0
44	47	4	Direct Block Pointer 1
48	51	4	Direct Block Pointer 2
52	55	4	Direct Block Pointer 3
56	59	4	Direct Block Pointer 4
60	63	4	Direct Block Pointer 5
64	67	4	Direct Block Pointer 6
68	71	4	Direct Block Pointer 7
72	75	4	Direct Block Pointer 8
76	79	4	Direct Block Pointer 9
80	83	4	Direct Block Pointer 10
84	87	4	Direct Block Pointer 11
88	91	4	Singly Indirect Block Pointer (Points to a block that is a list of block pointers to data)
92	95	4	Doubly Indirect Block Pointer (Points to a block that is a list of block pointers to Singly Indirect Blocks)
96	99	4	Triply Indirect Block Pointer (Points to a block that is a list of block pointers to Doubly Indirect Blocks)

Question 2:

502054 / 65536 = 7 (First Level if Indirection)

502054 % 65536 = 43302

43302 / 256 = 169 (Second Level of Indirection)

43302 % 256 = 38 (Third Level of Indirection)

Answer: inode_array[14]->first_array[7]->second_array[169]->third_array[38]->data

Question 3:

Large block size:

- Less block pointers required, potentially less usage of indirection
- Smaller inodes
- Disk can read one large block faster than several small blocks due to the data being physically close

Small block size:

- Less wasted space with small files and files that aren't multiples of the block size

Question 3:

Many direct pointers:

- Less pointer dereferencing results in faster lookup times

Few direct pointers:

- Store bigger files in smaller inodes

Question 3:

Ext2 block pointers:

- Allows faster reading of data near the middle/end of files

Linked list:

- Would allow performing certain write operations such as prepending to a file or inserting something in the middle of a file faster
- Potentially less space used due to not having to store indirect pointer tables
- Simpler to implement
- Smaller inodes

Question 4:

- Why don’t we store multiple files’ contents inside a single block?
  - Extra metadata to track exactly which bytes belong to which files
  - Accessing/modifying files is more complicated/requires more work
    - Can split reads/writes across sector/block boundaries much more
    - Indexing into a file becomes linear
  - Fragmentation tradeoffs
    - Sounds like something from architecture?

Virtual Memory

Why Virtual Memory?

1. What stops a p5 test case from doing `*(uint32_t*) 0x100000 = 0xAAAA`?

Why Virtual Memory?

1. What stops a p5 test case from doing `*(uint32_t*) 0x100000 = 0xAAAA`?
    a. (the test case will be invalid)
    b. There is nothing that our kernel can currently enforce that stops malicious programs like this!

Why Virtual Memory?

1. What stops a p5 test case from doing `*(uint32_t*) 0x100000 = 0xAAAA`?
    a. (the test case will be invalid)
    b. There is nothing that our kernel can currently enforce that stops malicious programs like this!
2. How can we run multiple processes at once?
    a. Problem: What if two processes want to be loaded at the same address?

Why Virtual Memory?

1. What stops a p5 test case from doing `*(uint32_t*) 0x100000 = 0xAAAA`?
    a. (the test case will be invalid)
    b. There is nothing that our kernel can currently enforce that stops malicious programs like this!
2. How can we run multiple processes at once?
    a. Problem: What if two processes want to be loaded at the same address?
    b. Possible solutions
        i. Don’t let processes run at the same address (single address space)
        ii. Our approach (the vastly common approach): lie about it

Why Virtual Memory?

1. What stops a p5 test case from doing `*(uint32_t*) 0x100000 = 0xAAAA`?
    a. (the test case will be invalid)
    b. There is nothing that our kernel can currently enforce that stops malicious programs like this!
2. How can we run multiple processes at once?
    a. Problem: What if two processes want to be loaded at the same address?
    b. Possible solutions
        i. Don't let processes run at the same address (single address space)
        ii. Our approach (the vastly common approach): lie about it
    c. Problem: What if one process is evil and tries to mess with another process's memory?
        i. Once again, nothing that we currently have can catch this

Virtual Memory

- Virtual memory lets us have two address types
  - Physical addresses: what we have been using so far
    - The pointer directly maps onto some spot in physical memory, and no other fancy tricks are applied
  - Virtual addresses: the lying part
    - The pointer goes through address translation and converted into a physical address, which is then used to map into physical memory like before

Virtual Memory Implementation, approach 0

- Approach 0: ban the user program from ever interacting with memory directly
  - Every memory address is manually translated by the kernel and loaded/stored by the kernel.
  - Extremely inefficient!

Virtual Memory Implementation, approach 1

- Approach 1: map virtual address by segments
  - A process's virtual addresses map onto a contiguous range of physical addresses
  - (This is why we call memory addressing errors segmentation faults)
- Translation
  - Take the virtual address and add a base offset - very simple
  - If the virtual address is too big, we can detect an out-of-bounds access
- Drawbacks
  - Like malloc - if processes are dynamically being created and destroyed, can lead to fragmentation of physical memory
  - Growing a process's memory allocation is not always feasible without a very large copy to a new spot in physical memory
- Having more segments per process has a growing cost in hardware required to support multiple segments

Virtual Memory Implementation, approach 2

- Approach 2: Divide physical memory into units of pages
  - Pages are our finest unit of memory control - can't enforce anything at chunks of bytes smaller than the page size
  - Size is traditionally 4 KB, but can be bigger on certain architectures
  - Not limited by the allocation issues that segmentation has
- Translation
  - Divide the virtual address into two parts - the **Virtual Page Number (VPN)** and the remaining sub-page part
  - Use the VPN as a lookup into some data structure that maps it to a **Physical Page Number (PPN)**
  - Combine the PPN and the sub-page part to form the physical address
- Drawbacks
  - There is some smallest granularity that we can manage memory at - similar tradeoffs as with block size in file systems.
  - How do we implement the lookup in translation?

Virtual Memory Implementation, continued

- How can we implement this translation?
  - Approach 1: Use the VPN as a direct index into an array that gives output PPNs
    - Array needs to be as large as the number of possible VPNs - on a 32 bit system, this is 2^{20} entries!
    - For an extremely dense address space (where most entries are populated), this is efficient
    - Common case: address spaces are sparse
      - The code, data and heap may live in low addresses, while stacks live at high addresses
      - Lots and lots of unused addresses in the middle that we don't need to reserve space for
  - Approach 2: Use indirection, and a multi-level address lookup
    - Use a tree - avoid allocation space for these large regions of memory that we don't use

Virtual Memory Implementation, continued

- Page directories/page tables
  - Our lookup mechanism to map VPNs to PPNs
  - The page directory contains a list of PPNs corresponding to page tables + extra attributes
    - Special register tells us the PPN of the page directory itself
  - Each page table contains a list of PPN outputs + extra attributes
  - On our 32-bit system, virtual addresses are divided into 3 parts
    - The highest 10 bits are used for a first level lookup
    - The next 10 bits are used for a second level lookup
    - The lowest 12 bits are passed directly to the physical address (page offset)
- Translation
  - Use the first 10 bits of the virtual address to index into our page directory and get the address of a page table
  - Use the next 10 bits to index into the page table and get the output PPN
  - Combine this PPN with the remaining page offset to form the output physical address

Virtual Memory Implementation, continued

```
0xaaaa aaaa aabb bbbb bbbb cccc cccc cccc
  |----------| = index into page directory
              |----------| = index into page table
            page offset =  |------------|
```

TLBs

- Is this practical?
  - Clearly, needs hardware support to be fast enough
  - Each memory address turns into three lookups!
  - Memory is already slow (relative to cache/registers) - this would be even more unacceptably slow.
  - Solution: caches

TLBs

- Is this practical?
  - Clearly, needs hardware support to be fast enough
  - Each memory address turns into three lookups!
  - Memory is already slow (relative to registers) - this would be even more unacceptably slow.
  - Solution: caches
- **Translation Lookaside Buffer**: caches our VPN to PPN conversions
  - Automatically populated for us whenever an entry for a VPN isn't cached
  - What happens if we change or remove an entry from our page tables?

TLBs

- Is this practical?
  - Clearly, needs hardware support to be fast enough
  - Each memory address turns into three lookups!
  - Memory is already slow (relative to registers) - this would be even more unacceptably slow.
  - Solution: caches
- **Translation Lookaside Buffer**: caches our VPN to PPN conversions
  - Automatically populated for us whenever an entry for a VPN isn't cached
  - What happens if we change or remove an entry from our page tables?
  - We need to **manually invalidate TLB entries** if we change entries like this
    - Semi-software managed cache

Other virtual memory details

- Now that we've encapsulated accessing memory, we can add more things!
  - Memory protections: restrict access to some page by user/kernel, read/write
    - Do not allow the user to access kernel memory
    - Do not allow writes to memory marked as read only
    - In modern VM, also can prohibit execution of certain pages
  - Page usage: whether or not a page has been accessed or written to
    - Useful for certain caching strategies
  - More attributes depending on the specific VM architecture

Questions?