Project 2

Hybrid Cloud Storage System

Project due on May 2nd (11.59 EST)

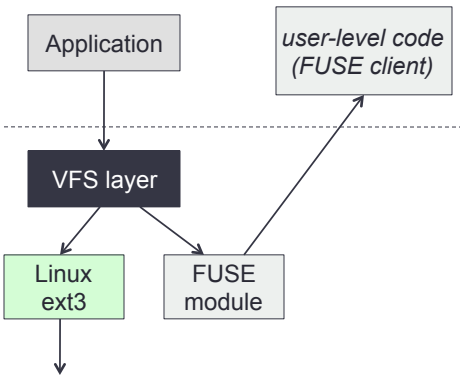
- Start early ☺: We have three graded milestones
  - Milestone 1: demo part 1 by April 3rd
  - Milestone 2: demo part 2 by April 17th
  - Milestone 3: demo part 3 by May 1st
  - Final Report Due: May 2nd
- "Oh well, I will write the report on May2nd"
  - Bad idea ☺
  - 25% of the project grade is associated with the final report
  - Project report consists of a design overview and performance analysis of your results on all three parts.
  - Project report must answer questions on page 21 of handout.

Summary of the project

- Write a user-level file system called CloudFS that manages hybrid storage devices (SSD and cloud storage)
  - File system in user-space (FUSE) toolkit
  - Code to be written in C/C++
- Testing and evaluation setup
  - All development and testing done in Linux using the VirtualBox virtual machine setup
  - Your code must compile in the Linux images on the virtual machine
- Do not obtain & read source code from a prior 746 class
  - This is cheating and should lead to failing the class
  - Besides, each year's 746 is different

Part 0: Primer on the FUSE toolkit

- Interposition layer to redirect VFS calls to your user-level code
- FUSE client code is programmable
  - Talk to remote nodes
  - Interact with local FSs
- FUSE clients need to implement a minimal set of protocols



FUSE API

- Supports most VFS calls
  - This API is the "high-level" interface using path names
- You don't need to implement all the calls
  - Simple systems, such as CloudFS, can get way with basic calls

```
int(*  getattr )(const char *, struct stat *)
int(*  readlink )(const char *, char *, size_t)
int(*  mknod )(const char *, mode_t, dev_t)
int(*  mkdir )(const char *, mode_t)
int(*  unlink )(const char *)
int(*  rmdir )(const char *)
int(*  symlink )(const char *, const char *)
int(*  rename )(const char *, const char *)
int(*  link )(const char *, const char *)
int(*  chmod )(const char *, mode_t)
int(*  chown )(const char *, uid_t, gid_t)
int(*  truncate )(const char *, off_t)
int(*  utime )(const char *, struct utimbuf *)
int(*  open )(const char *, struct fuse_file_info *)
int(*  read )(const char *, char *, size_t, off_t, struct fuse_file_info *)
int(*  write )(const char *, const char *, size_t, off_t, struct fuse_file_info *)
int(*  statfs )(const char *, struct statvfs *)
int(*  flush )(const char *, struct fuse_file_info *)
int(*  release )(const char *, struct fuse_file_info *)
int(*  fsync )(const char *, int, struct fuse_file_info *)
int(*  setxattr )(const char *, const char *, const char *, size_t, int)
int(*  getxattr )(const char *, const char *, char *, size_t)
int(*  listxattr )(const char *, char *, size_t)
int(*  removexattr )(const char *, const char *)
int(*  opendir )(const char *, struct fuse_file_info *)
int(*  readdir )(const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *)
int(*  releasedir )(const char *, struct fuse_file_info *)
int(*  fsyncdir )(const char *, int, struct fuse_file_info *)
void *(*  init )(struct fuse_conn_info *conn)
void(*  destroy )(void *)
int(*  access )(const char *, int)
int(*  create )(const char *, mode_t, struct fuse_file_info *)
int(*  ftruncate )(const char *, off_t, struct fuse_file_info *)
int(*  fgetattr )(const char *, struct stat *, struct fuse_file_info *)
int(*  lock )(const char *, struct fuse_file_info *, int cmd, struct flock *)
int(*  utimens )(const char *, const struct timespec tv[2])
int(*  bmap )(const char *, size_t blocksize, uint64_t *idx)
unsigned int  flag_nullpath_ok: 1
unsigned int  flag_reserved: 31
int(*  ioctl )(const char *, int cmd, void *arg, struct fuse_file_info *, unsigned int flags, void
*data)
int(*  poll )(const char *, struct fuse_file_info *, struct fuse_pollhandle *ph, unsigned
*reventsp)
```

Part 1: Hybrid Flash-Cloud Systems

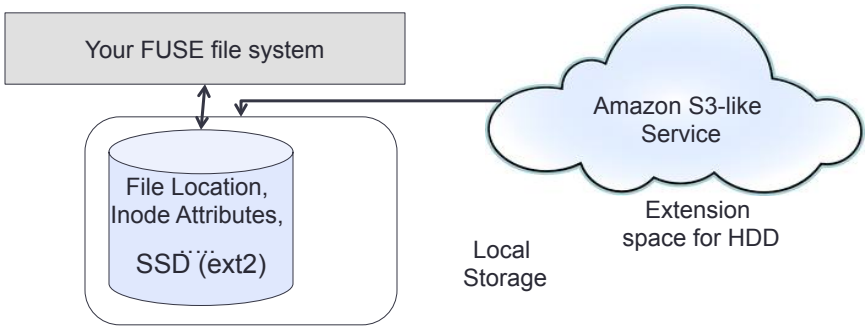
- Hybrid storage systems: Best of both worlds
  - Unlimited capacity (in cloud storage)
  - Fast random access speed (in local flash)
- CloudFS is a layered file system
  - Higher-level file system (CloudFS) splits data between SSD and cloud storage.
  - SSD is running its own local file system (Ext2) which you will *not* modify
  - Cloud storage is running as a separate object store which you will access through Amazon S3-like interface without modification
- Key idea
  - All small objects in flash, all large objects in cloud storage
  - All small IO (metadata) accesses should be satisfied in flash

Amazon S3 storage model

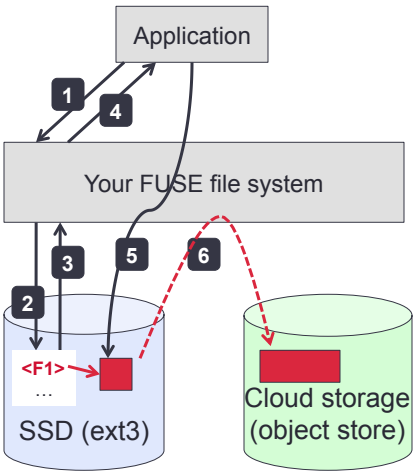
- Object storage in flat namespace
  - Structure: S3://BucketName/ObjectName
  - (Bucket is like a non-hierarchical directory, object could be a file)
  - List operations: look up the buckets or look up objects in a bucket
  - Put: write an entire object into S3
  - Get: read an entire object from S3
- Pricing（scale up to fit our tests):
  - Capacity pricing: $0.095 per MB (max capacity during one test)
  - Request pricing: $0.01 per request
  - Data Transfer Pricing: $0.120 per MB (out of S3 only; that is, reads)
  - Note: cost will probably NOT be dominated by capacity

System Overview

- SSD manages metadata information:
  - File location: in SSD or Cloud?
  - Namespace and inode attributes
- Cloud extends the capacity of local SSD
  - SSD is not a strictly a cache, because some data never goes to cloud



Size-based data-placement



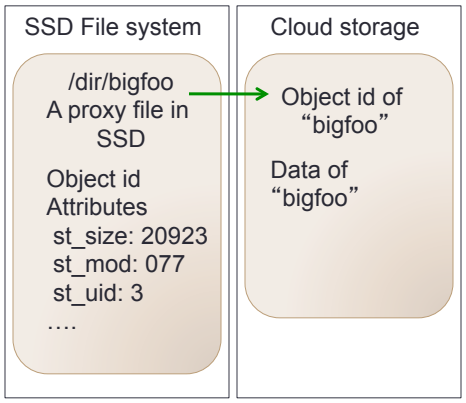
Skeleton Code for CloudFS is provided to you in the distribution

1) App does create("f1.txt")
2) CloudFS creates "f1.txt" in SSD
3) Ext3 on SSD returns a handle for "f1.txt" to FUSE
4) FUSE "translates" that handle into another handle which is returned to the app
5) App uses the returned handle to tell you to write to "f1.txt" on the SSD
6) When "f1.txt" closes and is big, CloudFS moves it to cloud storage and "f1.txt" on the SSD becomes a proxy file that contains mapping to the object on cloud storage

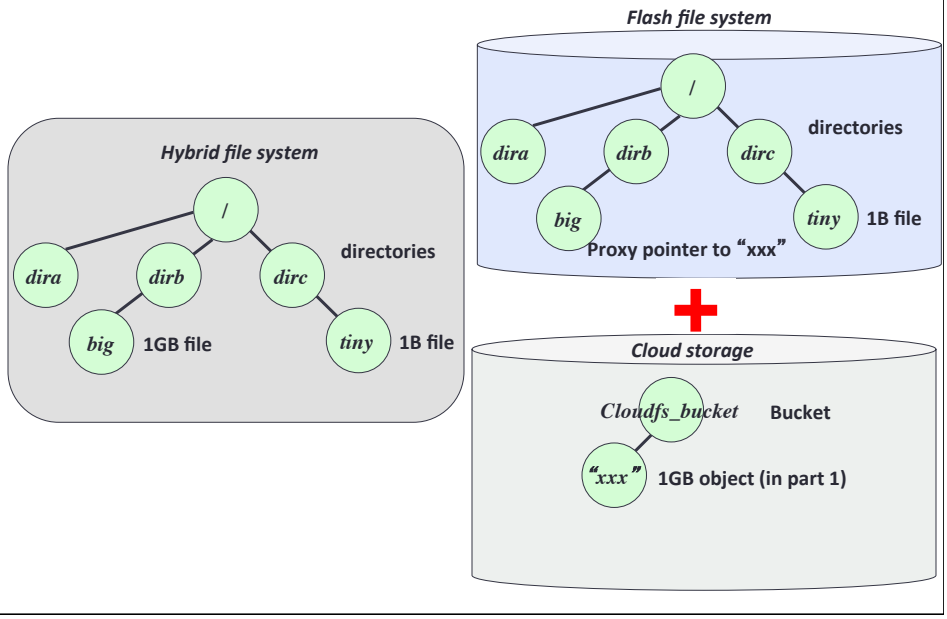
For design simplicity, this migration to cloud storage is done only when the file is closed.

Keep mapping and attributes in SSD

- After migration from SDD to cloud storage
  - Keep mapping between a proxy file in SSD and an object in cloud storage
  - Keep attributes of the file in SSD to avoid accessing (or even storing) in cloud storage
- Mapping information is stored in data block of proxy file or customized list inside a file in SSD (your design)
- Inode attributes are stored as (you choose): (1) extended attributes, or (2) customized list inside a file in SSD (again your design)



Logical and Physical View of CloudFS



Testing and Evaluation

- Test scripts (test_part1.sh) that perform three steps
  - Extract an .tar.gz file in the CloudFS mount point
  - Compute checksum on all the files and compare
  - Perform a directory scan of all files ("ls –alR")
- Script allows you to test with different dataset sizes
- To facilitate measurements, each test …
  - … will empty caches before runs by remounting CloudFS
  - … will measure number of block I/O to SSD and cloud storage using vmstat and cloud stat info
  - *(virtualized setup makes time-based measurement hard)*
- More details in the README file in the "src/scripts/"

Expected output for correctness

- Test scripts returns the number of blocks read/written during each of the step (by parsing "vmstat –d", cloud log)

Testing step	Expected Correct Output
Extracting TAR file in CloudFS	Big files (>threshold) should go to cloud storage. Small files (<threshold) should be in the SSD.
Performing a checksum on the all the files	MD5 of files in CloudFS should match with MD5 of original files.
Scanning whole directory of the TAR file using "ls –laR"	Only the SSD should have block reads (Cloud storage should see no object GETs)

- Another useful tool is btrace/blktrace

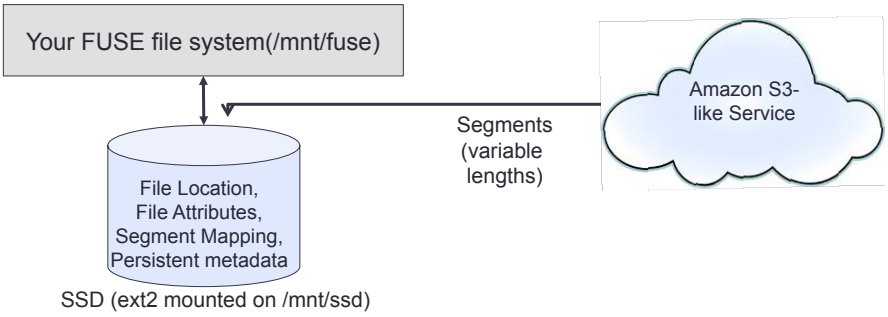
Part 2: Deduplicate & Compress

- Deduplicate: store only unique content in the cloud
  - divide content into segments using Rabin Fingerprinting
  - compute a content hash of each segment
  - lookup each segment's hash in a catalog
  - store segments with new/unique hashes as objects in the cloud
  - add new hashes to the catalog
  - keep track of segment metadata (hashes, object key) on SSD
- Use loseless compression as well – it significantly reduces size

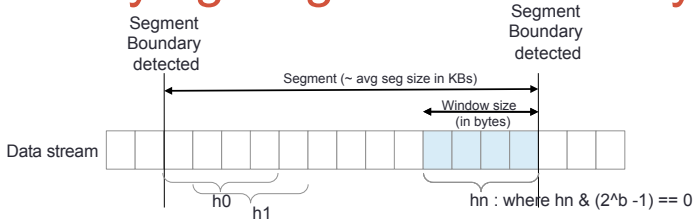
- Goal: Reduce cloud costs
  - Max capacity consumed should be reduced
  - Data transfer costs should be reduced
  - Number of operations (PUTs) will be increased

System overview

- Segments defined by Rabin Segmentation API
- All segments stored in Cloud, compress before or after segmentation
- File attributes (size, permissions etc) stored on SSD
- File-to-Segment mapping on SSD
- Persistent Hash lookup table on SSD



Identifying Segment Boundary



- Define segments by data contents, not external alignment
- “Rolling” hash computed for each byte across last 'window size' bytes
- Declare a new segment boundary if we find a specific hash value
  - E.G. if last b bits of hash value hn are 0
  - With “good” hash function, the average segment size will be 2^b
  - Details: identical windows leads to tiny segments: set a min size
- All of this is implemented in Rabin Segmentation provided
- Use the code structure in src/cloudfs/rabin-example.c

Identify duplicated segment

- Decide on a naming scheme for the segments
  - E.g. counter of segments created
  - E.g. Md5sum of content
- Identifying duplicate segment – lookup
  - Maybe a hash table of segments in the cloud
  - Maybe a database, maybe even just a bloom filter
  - But make it fast, persistent, small and above all **simple**
  - Can recreate hash data structure on startup using LIST, if you choose to

Mapping files to segment

- File segments may be required to be brought to SSD for read/write operations.
- So need to maintain file-segments mapping
  - Can store in same file that was used to store file attributes in part1 (if you used one)
  - Can design a segment store that keeps key->values mapping for all files in the cloud
  - Think about the naming scheme for segments, some are better than others
  - Again, make it fast and simple

Read and Write operations

- (1) File can be entirely reconstructed on SSD when open
  - read() & write() operate on local SSD copy of segment. May be costly if not all segments are needed.
- Or (2) only the segments required for read() and write() operations could be “faulted” in from Cloud to SSD
  - read() and write() still operate on local SSD copy of segment, but not all segments have to be in SSD together. Cheaper, as not all segments might be fetched
  - Allows users to access files larger than SSD as operations only on specific segment not entire file.
- It is acceptable to assume that all files are written sequentially without any seeks (which makes (2) easier)

Cleaning up on file deletion

- Recovering space of unneeded segments in cloud
  - Probably best to do on some delete() operations
- Come up with a reference counting scheme for segments
  - Can store it as part of hash table
  - Can manage it separately, but this information has to be persistent across remounts of file system so segments don’t get removed too early

Testing and Evaluation

- Part2 test scripts will test :
  - Correctness
    - Copy the same file multiple times, read back each copy and compare
    - Prepend, append some data and read back
  - Performance: cloud usage charges
    - For all tests compare the cloud usage with and without dedup (–no-dedup) and compression (-no-compress)
    - Cost reduction/increase – be wary of paying more for dedup'd cloud!

Part 3: Caching

- Could storage charge for operations

Type	Price used in our tests	Real price of Amazon S3
Capacity	$ 0.095 per MB (max usage during one test)	$ 0.095 per GB per month for the first 1 TB
Operation pricing	$ 0.01 per request	PUT, COPY, POST, LIST: $ 0.01 per 1,000 requests. GET: $ 0.01 per 10,000 requests.
Data transfer pricing	$ 0.12 per MB (out from S3 only)	$ 0.12 per GB per month for up to 10 TB (out from S3 only).

- Goal: reduce cloud cost by segment-level caching (and make laptop faster)
- You are going to define a segment replacement policy and explore tradeoffs in the project report

Designs (hints)

- A simple example of caching
  - LRU: Recently written segments should remain in local file system. When local file system is nearly full, least recently accessed segments are moved to cloud
  - Write-through: synchronize updates to segment in cache and cloud whenever modified
  - Write-back: delay, possibly never, write to cloud

  We expect better segment replacement policy than LRU that considers price model of cloud storage.
- All segments of file may be cached too, but costly.

Designs (hints)

- Persistency of cache
  - "Warm start" is better than "cold start"
  - For warm start, need a way of identifying segments in cache on mount
  - Full scanning of cache region on mount time or a persistent data structure could be one design option

Testing and Evaluation

- Part3 test script performs two subtests
  - Stress test will generate random sized files and check correctness
  - Cache test will run a set of workloads and measure performance metrics.
- In part 3, cost savings is the most important evaluation criteria
- Your implementation has to pass stress test as well to show correctness

Summary: data structures for CloudFS

- A list of data structures to maintain:
  - File Locations: URL identifies the location of a file
  - Namespace (directory entries)
  - Inode attributes (timestamp, access mode …)
  - LRU list: a list of objects in local storage sorted by last closed time
  - Hash value of file segments for de-duplication
  - Reference count for each object
- DO's and DON'T's:
  - No in-memory only solutions
  - Out-of-core structures should use space on SSDs

Assumptions for simplification

- No need to store all file metadata in Cloud (e.g. attributes)
- All metadata will fit into SSD (i.e. SSD is big enough)
- Single threaded FUSE only
- No sharing in the cloud – cloud is dedicated to one SSD in one client

Testing and Evaluation

- Correctness:
  - Basic functionality:
    - Read/Write files from cloud storage
  - Persistency: No data loss after normal umount/remount
  - Cache policy: LRU or any other advanced policy you invent
  - De-duplication: remove redundant contents
  - Lossless compression
- Performance
  - Cloud storage usage costs
  - Local ssd I/O traffics
  - CPU and memory usage

How to write the report?

- Key design ideas and data-structures
  - Pictures are useful; but good one need thought and time (start early ☺)
- Reason about the performance
  - Don't just copy-paste the output in the report
  - Show us that you know why it is happening
- Answer the three questions in the handout

workload	SSD Blk IO	Operations on cloud storage	cloud cost
w1	# of Blk R/W	Amount of data transfer # of ops	Cost to complete a workload
w2	# of Blk R/W	Amount of data transfer # of ops	Cost to complete a workload

Tools provided

- Amazon S3 client library:
  - Libs3: A C Library API for Amazon S3, a complete support of Amazon S3 based on HTTP protocols
  - Provide a wrapper of libs3 in CloudFS skeleton code, simplified synchronous call
- Amazon S3 server simulation:
  - A Python simulation run in VirtualBox
  - Implement simple APIs: list, put, get, delete
  - Store data in default local file system inside virtual box
  - Provide simple statistics about usage cost

How to submit?

- Use Autolab for submission
  - Test compilation and correctness for milestones
  - Performance tests for grading are manually run with virtual box outside Autolab
- Deliverables:
  - Source code:
    - Good documentation in codes
    - Correct format for Makefile and input parameters
    - Follow instructions in handout to organize the code
  - Project reports
    - Key design ideas, data structures and reasons
    - Evaluation: local SSD I/Os, cloud storage costs, etc.
    - No more than 5 pages, single column with 10 pts.

Once again – start early ☺

- Project due on May 2nd
  - Milestone 1: demo part 1 by April 3rd
  - Milestone 2: demo part 2 by April 17th
  - Milestone 3: demo part 3 by May 1st
  - Final Report Due: May 2nd

Some Test Cases

- Functionality tests:
  - copy, untar, delete, calculate md5sum
  - Build simple projects
- Large file tests:
- Cache policy tests:
  - LRU and Write-Back Cache Policy
  - Generate LRU friendly access pattern
- De-duplication tests:
  - Generate several large files with the same contents
- Persistency tests:
  - umount / mount
  - Repeat the above tests to test performance difference

Monitoring

- End-to-end running time
- SSD and HDD traffic:
  - Total number of read/write requests
  - Total number of read/write sectors
- Cloud storage:
  - Capacity usage
  - Total number of requests
  - Total bandwidth consumption
- CPU and memory usage