

# File Systems Using Cloud Storage

This file system leverages the FUSE (Filesystem in Userspace) interface to allow users to interact with cloud storage with the same ease and familiarity as a local file system. Central to this functionality is the GCSFS class, which inherits from fuse.Operations. Upon initialization, this class sets up a connection to GCS using credentials from a service account file (nikhil.json) and targets a specific GCS bucket for operations.

The system's core lies in translating traditional file system operations into actions on GCS blobs, where each blob in the bucket is treated as a file or directory. For instance, methods like opendir, readdir, and getattr handle the opening, reading, and attribute retrieval of directories and files, respectively. Files and directories are distinctly managed: files as GCS blobs and directories as blobs with names ending in a slash ('/'). This distinction allows for operations like mkdir and rmdir to simulate directory creation and deletion in the cloud environment, where GCS inherently does not have a traditional directory structure.

File operations are sophisticatedly handled: the create method facilitates new file creation by establishing new blobs, while read and write manage data transfer to and from these blobs. This involves downloading the blob content, performing the operation, and re-uploading it to GCS if there are modifications. We had to download the file to modify it as GCS doesn't allow appending data to the file.

I tried to implement a comprehensive error handling strategy, which employs the FuseOSError to signal various file system errors, enhancing the robustness and reliability of the interface. Additionally, integrated logging captures essential operations and errors, offering valuable insights for maintenance and debugging which helped a lot during the development of this system.

In the main function, the script mounts the GCSFS filesystem at a defined mount point. This crucial step makes the GCS bucket appear as a local filesystem, allowing users to perform standard file operations like navigating directories, editing files, and managing file attributes directly on the cloud storage, just as they would on their local machine.

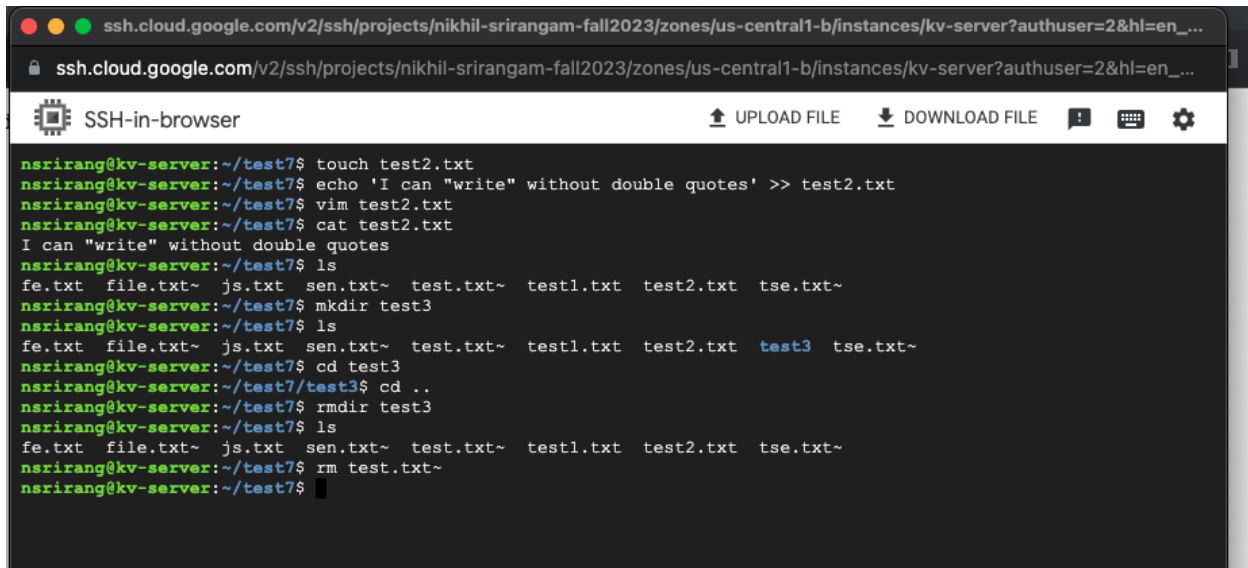
Program - FS.py

Credentials- nikhil.json

Service-account - [fuse-527@nikhil-srirangam-fall2023.iam.gserviceaccount.com](mailto:fuse-527@nikhil-srirangam-fall2023.iam.gserviceaccount.com)

Testing:

This filesystem is able to implement all the basic file and directory operations which were mentioned in the assignments, and it can be seen in the below screenshot



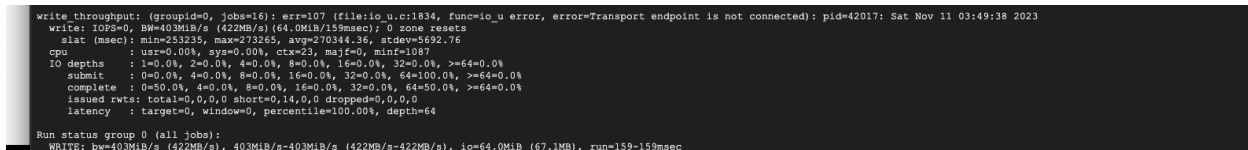
```
ssh.cloud.google.com/v2/ssh/projects/nikhil-srirangam-fall2023/zones/us-central1-b/instances/kv-server?authuser=2&hl=en_...
SSH-in-browser
nsrirang@kv-server:~/test7$ touch test2.txt
nsrirang@kv-server:~/test7$ echo 'I can "write" without double quotes' >> test2.txt
nsrirang@kv-server:~/test7$ vim test2.txt
nsrirang@kv-server:~/test7$ cat test2.txt
I can "write" without double quotes
nsrirang@kv-server:~/test7$ ls
fe.txt  file.txt~  js.txt  sen.txt~  test.txt~  test1.txt  test2.txt  tse.txt~
nsrirang@kv-server:~/test7$ mkdir test3
nsrirang@kv-server:~/test7$ ls
fe.txt  file.txt~  js.txt  sen.txt~  test.txt~  test1.txt  test2.txt  test3  tse.txt~
nsrirang@kv-server:~/test7$ cd test3
nsrirang@kv-server:~/test7/test3$ cd ..
nsrirang@kv-server:~/test7$ rmdir test3
nsrirang@kv-server:~/test7$ ls
fe.txt  file.txt~  js.txt  sen.txt~  test.txt~  test1.txt  test2.txt  tse.txt~
nsrirang@kv-server:~/test7$ rm test.txt~
nsrirang@kv-server:~/test7$
```

## Testing using benchmarking tools:

1. Test write throughput by performing sequential writes with multiple parallel streams (16+), using an I/O block size of 1 MB and an I/O depth of at least 64 and these are the results command:

```
fio --name=write_throughput --directory=$TEST_DIR --numjobs=16 --size=1M --time_based
--runtime=90s --ramp_time=2s --ioengine=libaio --direct=1 --verify=0 --bs=1M --iodepth=64
--rw=write --group_reporting=1 --iodepth_batch_submit=64 --iodepth_batch_complete_max=64
```

The above commands was not stopping automatically so had to terminate it manually to see the results



```
write throughput: (groupid=0, jobs=16): err=107 (file:io.u.c:1834, func=io_u_error, error=Transport endpoint is not connected): pid=42017: Sat Nov 11 03:49:38 2023
write: IOPS=0, BW=403MiB/s (422MB/s) (64.0MiB/159msec): 0 zone resets
slat (msec): min=253225, max=273265, avg=270344.36, stdev=5692.76
cpu
: usr=0.00%, sys=0.00%, ctx=23, majf=0, minf=1087
IO depths : 1=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit   : 0=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=100.0%, >=64=0.0%
complete : 0=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=50.0%, >=64=0.0%
issued rwts: total=0,0,0 short=0,14,0,0 dropped=0,0,0
latency   : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
WRITE: bw=403MiB/s (422MB/s), 403MiB/s-403MiB/s (422MB/s-422MB/s), io=64.0MiB (67.1MB), run=159-159msec
```

2. Test read IOPS by performing random reads, using an I/O block size of 4 KB and an I/O depth of at least 256:

command:

```
sudo fio --name=read_throughput --directory=$TEST_DIR --numjobs=16 \
--size=1M --time_based --runtime=60s --ramp_time=2s --ioengine=libaio \
--direct=1 --verify=0 --bs=1M --iodepth=64 --rw=read \
--group_reporting=1 \
--iodepth_batch_submit=64 --iodepth_batch_complete_max=64
```

```

herirang@kv-server:~$ sudo fio --name=read_throughput --directory=$TEST1_DIR --numjobs=16 \
--size=1M --time-based --runtime=60s --ramp_time=2s --ioengine=libaio \
--direct=1 --verify=0 --bs=1M --iodepth=64 --rw=read \
--group_reporting=1 \
--iodepth_batch_submit=64 --iodepth_batch_complete_max=64
read_throughput: (g=0): rw=read, bs=(R) 1024KiB-1024KiB, (W) 1024KiB-1024KiB, (T) 1024KiB-1024KiB, ioengine=libaio, iodepth=64
...
fio-3.25
Starting 16 processes
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
read_throughput: Laying out IO file (1 file / 1MiB)
Jobs: 3 (f=2): [R(1), (8), f(1), (5), R(1)] [100.0%][r=446MiB/s][r=445 IOPS][eta 00m:00s]
read_throughput: (groupid=0, jobs=16): err= 0: pid=42109: Sat Nov 11 03:56:19 2023
  read: IOPS=102, BW=117MiB/s (122MB/s) (8004MiB/68565msec)
    slat (usec): min=12, max=139, avg=40.32, stdev= 9.45
    clat (msec): min=1856, max=21061, avg=8790.54, stdev=2857.26
    lat (msec): min=2018, max=21061, avg=8804.86, stdev=2839.20
    clat percentiles (msec):
      | 1.00th=[ 2140],  5.00th=[ 3373], 10.00th=[ 5403], 20.00th=[ 7886],
      | 30.00th=[ 8020], 40.00th=[ 8490], 50.00th=[ 8658], 60.00th=[ 8792],
      | 70.00th=[ 9060], 80.00th=[ 9194], 90.00th=[13355], 95.00th=[14832],
      | 99.00th=[16308], 99.50th=[16979], 99.90th=[17113], 99.95th=[17113],
      | 99.99th=[17113]
    bw ( KiB/s): min=32763, max=1218769, per=100.00%, avg=508921.70, stdev=20642.70, samples=475
    iops       : min= 31, max= 1190, avg=496.48, stdev=20.17, samples=475
    lat (msec) : 2000=0.21%, >2000=114.19%
    cpu        : usr=0.01%, sys=0.06%, ctx=7962, majf=0, minf=946
    IO depths  : 0=0.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=97.9%
    submit     : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
    complete   : 0=0.0%, 4=99.8%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.2%, >=64=0.0%
    issued rwts: total=6996,0,0,0 short=0,0,0,0 dropped=0,0,0,0
    latency    : target=0, window=0, percentile=100.00%, depth=64

Run status group 0 (all jobs):
  READ: bw=117MiB/s (122MB/s), 117MiB/s-117MiB/s (122MB/s-122MB/s), io=8004MiB (8393MB), run=68565-68565msec

Disk stats (read/write):
  sda: ios=32962/9, merge=0/0, ticks=273233381/85668, in_queue=273320217, util=99.95%

```

We can see that the test indicates that the storage system is capable of a high read bandwidth (117 MiB/s), but it experiences significant completion latencies (average of over 8 seconds). The high latency values suggest potential bottlenecks or inefficiencies in handling read requests. The IO depth distribution shows that the system is handling many I/O requests simultaneously, which could be contributing to the high latency. The disk utilization (util-99.95%) also indicates that the disk was almost fully utilized during the test, which might be a factor in the high latency observed.

#### Drawbacks:

1. The major drawback of this system is I was not able to implement reading files in the nested directory, this will definitely be a part of future improvements to this system.

#### References:

1. <https://cloud.google.com/compute/docs/disks/benchmarking-pd-performance>
2. <https://thepythoncorner.com/posts/2017-02-27-writing-a-fuse-filesystem-in-python/>