memfs – A FUSE Memory File System

Softwarepraktikum für Fortgeschrittene

Michael Kuhn

Parallele und Verteilte Systeme
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

2008-10-28

FUSE

- Goal was to measure the overhead of the FUSE
- `ctfs` indicated that FUSE introduces significant overhead when a large number of files is processed
- FUSE file systems run in user space
  - They use the special device `/dev/fuse` to communicate with the kernel part of FUSE
- More expensive context switches have to be performed

memfs

- What?
  - A FUSE memory file system
  - Like `tmpfs`
- Why?
  - Measure FUSE overhead
    - Eliminate the influence of the relatively slow hard disk
  - `tmpfs` for normal users

Overview

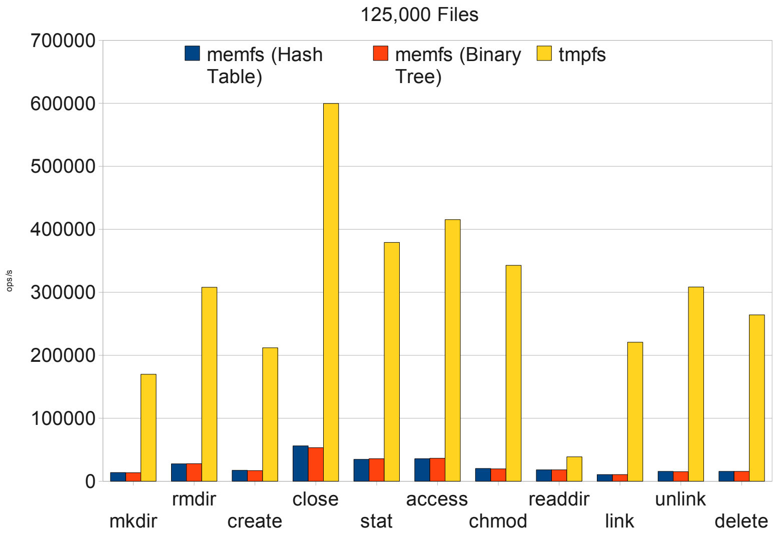
- Works like any other file system
- Selectable backends for directory entries
  - Currently hash tables and balanced binary trees are supported
- `chmod`, `chown`, `open` and `utimens` are merely empty stubs
  - `fileop` will not run without those
- Idea: Use empty operations to measure FUSE overhead

/opts directory

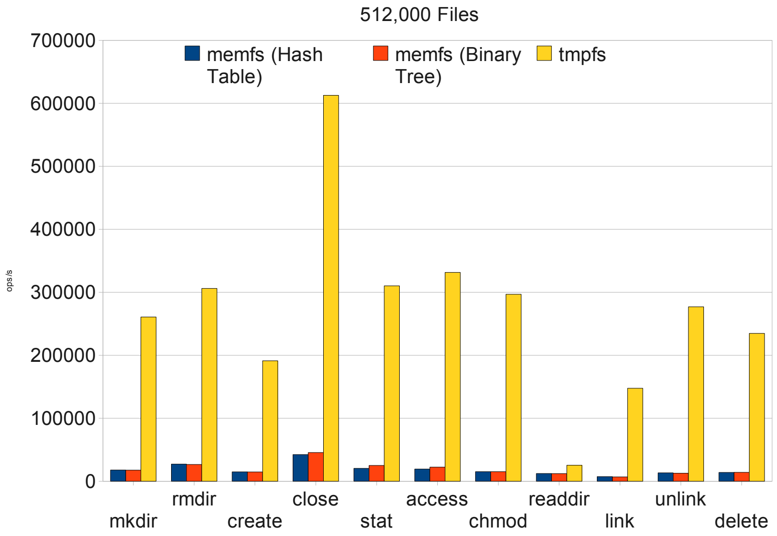
- Like `/proc`, just for `memfs`
- Can configure options at runtime
- Currently only `no_data` is supported
  - Discards any data written to a file
  - Returns bogus data
  - File size is updated correctly
- For example:
  - `$ echo 1 > $HOME/memfs/opts/no_data`
  - `$ cat $HOME/memfs/opts/no_data`

- Some FUSE file system operations are complex
    - They are internally made up of several file system operations
- `setattr()`
    - After `chmod()`, `chown()`, `truncate()` and `utimens()` an implicit `getattr()` is performed
- `lookup()`
    - After `create()`, `mknod()`, `mkdir()`, `symlink()`, and `link()` an implicit `getattr()` is performed
- `close()` does not do (too much) implicit work
    - Let's use that one

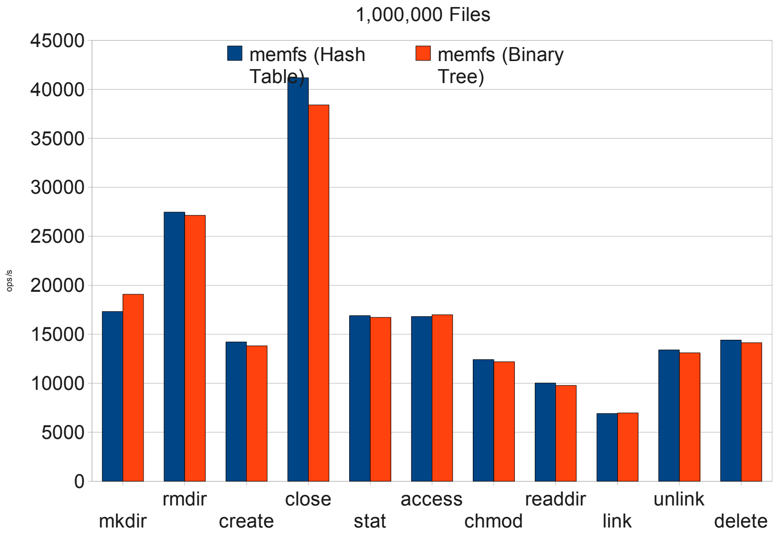
Evaluation


125,000 Files
700000
600000
500000
400000
300000
200000
100000
0
ops/s
memfs (Hash Table)
memfs (Binary Tree)
tmpfs
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

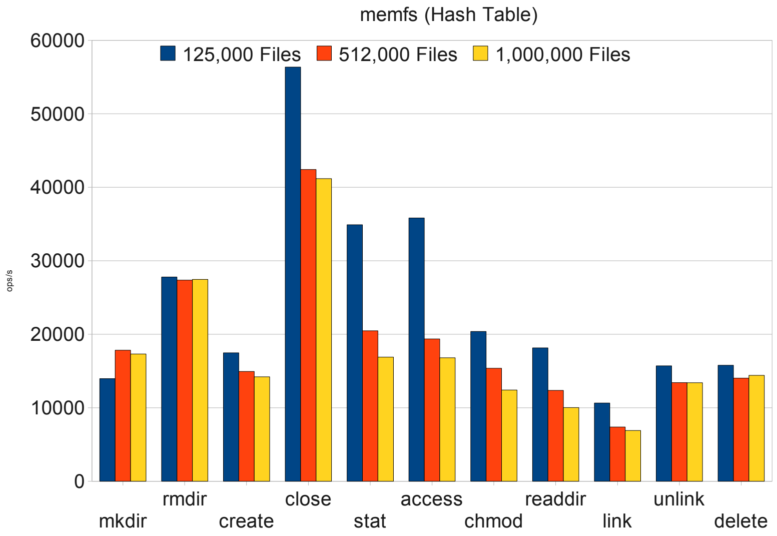
Evaluation


512,000 Files
memfs (Hash Table)
memfs (Binary Tree)
tmpfs
ops/s
0
100000
200000
300000
400000
500000
600000
700000
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

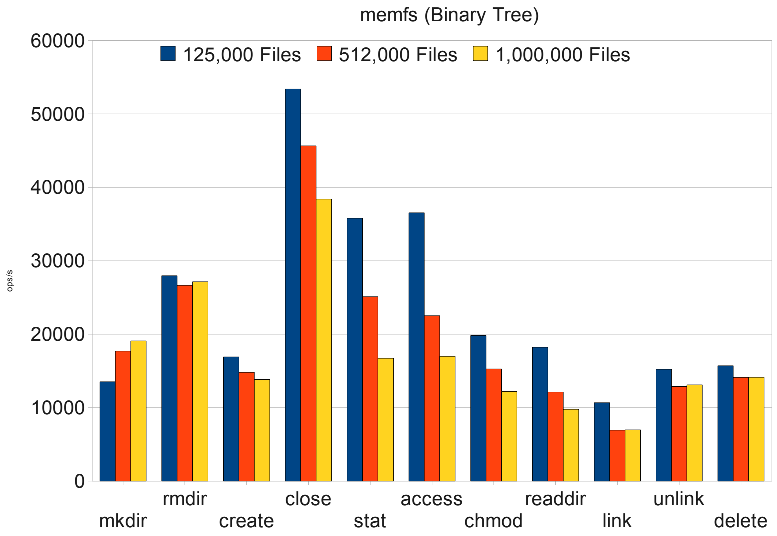
Evaluation


1,000,000 Files
memfs (Hash Table)
memfs (Binary Tree)
ops/s
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

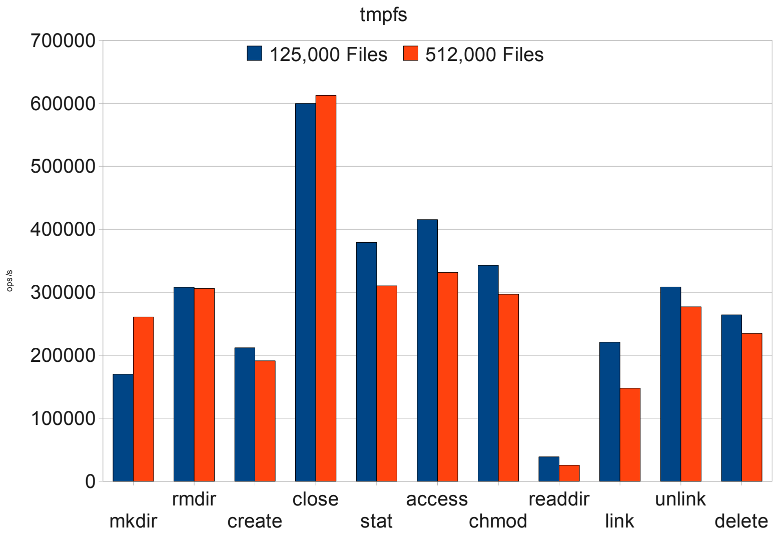
Evaluation


memfs (Hash Table)
125,000 Files
512,000 Files
1,000,000 Files
ops/s
60000
50000
40000
30000
20000
10000
0
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

Evaluation


memfs (Binary Tree)
125,000 Files
512,000 Files
1,000,000 Files
ops/s
60000
50000
40000
30000
20000
10000
0
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

Evaluation


tmpfs
125,000 Files
512,000 Files
ops/s
0
100000
200000
300000
400000
500000
600000
700000
mkdir
rmdir
create
close
stat
access
chmod
readdir
link
unlink
delete

Costs

- `tmpfs`
    - Mode switch into the kernel
    - Mode switch out of the kernel
- `memfs`
    - Mode switch into the kernel
    - Context switch into `memfs`
    - Context switch out of `memfs`
    - Mode switch out of the kernel

Conclusion and Future Work

- `memfs` is a memory file system that is configurable at runtime
  - Can be easily extended to use arbitrary data structures as backends
  - Basis for benchmarking and – hopefully – tuning of FUSE with large amounts of files
- It is hard to measure the overhead with empty stub operations
  - FUSE performs implicit `getattr()` calls for most of them
  - `release()` is one of the few operations that can be used
  - Should give a good estimate of the possible maximum that FUSE is capable of
  - Modify the FUSE user-space library to make the implicit `getattr()` calls conditional