Simplifying the Userland-Kernel API: Channel Programs

Chris Siden
Background: ZFS Administrative Operations

```
zfs snapshot rpool/fs@snap
```

- **Kernel**
  - Userland starts snapshot ioctl
  - Operations assigned a txg, Wait for that txg to Sync
  - Actual work happens during syncing context

- **Sync**
  - ioctl returns with results of operation
Background: Dependent Operations

zfs snapshot rpool/fs@snap &&
zfs set zfs:p=test rpool/fs@snap

snapshot ioctl

set prop ioctl
Background: Syncpass Time

- Syncpasses take longer when pool processing lots of writes
- Each one can take seconds
- Userland sees massive delay for each operation

![Diagram showing the process of syncpasses](image)
Background: Syncpass Time

```
zfs snapshot rpool/fs@snap &&
zfs set zfs:p=test rpool/fs@snap
```

For 2 dependent operations: 10 seconds
Background: Atomicity

```
zfs snapshot rpool/fs@snap &&
zfs set zfs:p=test rpool/fs@snap
```

Snapshot exist without property set
Background: Atomicity

zfs promote rpool/clone &&
zfs destroy rpool/fs

What if someone creates a new clone?
How an ioctl Evolves: Snapshots

1. Start simple:
   - snapshot("rpool/fs@snap")

2. Need atomicity/speed for multiple snapshots:
   - snapshot("rpool/fs@snap", "rpool/fs2@snap", …)
   - All or nothing: if any snapshot fails none are created

3. ‘zfs snapshot -r’ doesn’t work with “all or nothing”:
   - If any snapshot fails with something other than ENOENT none are created

4. Want to set properties while creating snapshots:
   - snapshot("rpool/fs@snap", "rpool/fs2@snap", …, props= {map})

Why not just have an ioctl for ‘zfs snapshot -r’?
How an ioctl Evolves: Destroy

1. Start simple:
   - destroy("rpool/fs")
   - destroy("rpool/fs@snap")

2. Need speed for multiple snapshots (but not filesystems):
   - destroy("rpool/fs")
   - destroy("rpool/fs@snap", rpool/fs@snap2", …)

Would like:
- Mix snapshot/filesystem destroys (zfs destroy -R takes forever)
- ‘zfs destroy -r @snap’ with in-kernel iteration
Simplify the ioctl APIs: Channel Programs

- Core operations are not changing frequently:
  - `snapshot("rpool/fs@onesnap")`
  - `create("rpool/onefs")`
  - `destroy("rpool/onefs", defer=true/false)`

- Stop creating a new ioctl for every possible combination of core operations

- Have syncing context interpret “channel programs” that describe what combination of operations to perform, how to do iteration, and how to deal with errors
Channel Programs: An Example

- `zfs promote <?> && zfs destroy <fsname>`

- `lastsnap = zfs.list.snapshots(input.fsname)`
  `clone = zfs.list.clones(lastsnap)`
  `err = zfs.sync.promote(clone)`
  `if err =~ 0 then`
    `return err`
  `end`

- Returns `zfs.sync.destroy(input.fsname)`

- Picks one clone of the latest snapshot and promotes it before doing the destroy
Channel Programs: An Example

```
zfs promote rpool/clone &&
zfs destroy rpool/fs
```

What if someone creates a new clone?
Channel Programs: An Example

```
zfs channel myscript.zfs
```

channel program ioctl

Both promote and destroy happen in same syncpass
Channel Programs: Another Example

- `zfs snapshot -r <snapname>`

- `rootfs = split(input.snapname, "@")[0]`
  `snap = split(input.snapname, "@")[1]`
  `result = {}`
  `for fs in zfs.list.snapshots(rootfs) do`
    `s = fs .. "@" .. snap`
    `result[s] = zfs.sync.snapshot(s)`
  `done`
  `return result`

- Does recursive snapshot with iteration in the kernel, not userland like it is today
Channel Programs: Another Example

- `zfs clone <fname> <clonename>`

- `snap = input.fname .. "@tmp"`
  
  ```
  err = zfs.sync.snapshot(snap)
  if err ~= 0 then return err done
  err = zfs.sync.clone(snap, input.clonename)
  zfs.sync.destroy(snap, defer=true)
  return err
  ```

- Clones the current state of a filesystem, creating a new snapshot that is deferred-destroyed in the same transaction
Channel Programs: Version 1.0

- All the listing and synctasks from the examples
- Must be privileged user to run arbitrary programs:
  - No per-synctask permissions checking (yet)
  - Not great memory limiting
  - No protections against infinite loops
- Works best for programmatic consumers
- “Built-in” channel programs (compiled into the kernel) used to implement as many existing ioctls as possible
- Not apply to every ZFS operation fits into this model, e.g. adding devices
THANK YOU
ANY QUESTIONS?