A few limitations in the available fs-related system calls...

Nick Kossifidis <mick@ics.forth.gr>
Integrated all-European Hardware and Open-Source Software for Cloud Services and Applications

RISER (RISC-V for Cloud Services) – CompContinuum Workshop – HiPEAC’24 (Jan. 19, 2024)

Contact: Dr. Manolis Marazakis
Organization: FORTH (Greece)
Email: maraz@ics.forth.gr

RISC-V for Cloud Services

Source: EPI and EUPilot projects (chips)
* Currently operating on system boards designed for dev/test purposes

Server Boards (PCB + firmware)
Standard form factors (PCIe accelerator card, Microserver)
* Following industry standards to utilize server I/O peripherals

Boot Firmware
Initialization of execution platform, including high-speed I/O peripherals (storage, networking)

OS, drivers, runtime
Configured/adapted for cloud services: workload acceleration, networked storage, containerized execution
*Integration in IaaS environment

RISER: RISC-V based Linux server for cloud services, built on open interfaces

RISC-V Processors

DRAM Memory

NVM-Express Storage

100 Gbps Ethernet

Contact:
https://riser-project.eu
https://twitter.com/RiserProject

https://riser-project.eu
https://twitter.com/RiserProject
Copying a file...

- Preserve file data
  - Time efficiency
  - Space efficiency

- Preserve file metadata
  - Permission bits
  - Ownership (user/group)
  - Timestamps
  - Old school attributes
  - Extended attributes
System calls for copying data...

- The naive approach: `open()`, `read()`, `write()`, `close()`
  - The most generic/portable way but very inefficient
  - Datapath goes through userspace, kernel copies to user on read, from user on write

- Using `sendfile()`
  - Linux, FreeBSD (thank you Netflix!)
  - Copying is done in-kernel, without going through userspace
  - Uses a temporary buffer: source -> buffer (pipe) -> dest
  - Probably the most common technique used today

- Using `copy_file_range()`
  - Linux-only
  - Takes advantage of fs features (e.g. COW, REFLINK, NFS server-side copy etc), and in the future will also take advantage of hw features (e.g. NVme simple copy)
  - This is meant to be the new/default API for this

- Preserve holes on sparse files: `lseek(SEEK_DATA/SEEK_END)`, `ftruncate()`
System calls for preserving metadata...

- Permission bits using `{f}chmod{at}`()
- Ownership using `{f,l}chown{at}`()
- atime/mtime using `utimens{at}`()
- Preserve old-style 32bit attributes mask using `ioctl(FS_IOC_{G,S}ETFLAGS)`
System calls for preserving metadata...

- Extended attributes (key:value pairs), using \{list, set, get\}xattr()
  - "POSIX" ACLs (acl(7)): system.posix_acl_access/default
  - NFSv4 ACLs (honored by the nfs client): system.nfs4acl/nfs4_acl
  - Inline-data (ext4(5)): system.data
  - Per-file capabilities (capabilities(7)): security.capability
  - SELinux file contexts: security.selinux/security.sehash
  - AppArmor labels (apparmor_xattrs(7)): e.g. security.apparmor
  - SMACK attributes: security.SMACK64*
  - Integrity measurement: security.evm/security.ima
  - Privileged userspace stuff: trusted.*
  - Unprivileged userspace stuff: user.*
  - and more...
  - Honor /etc/xattrs.conf, that includes xattr patterns to skip
Issues so far...

- `copy_file_range()` may expand holes on sparse files
- No io_uring op for `sendfile()` / `copy_file_range()`
- The `{at}` system call variants (using O_PATH descriptors) are very useful!
  - But there are no `{list,set,get}xattrat()` syscalls!
  - `fchmodat()` doesn’t support the AT_EMPTY_PATH flag -> Fixed on 6.6 with `fchmodat2()`
  - `utimensat()` does support AT_EMPTY_PATH but the man page doesn’t mention it
- IMHO There should be a single API for file attributes, having to use ioctl() doesn’t look nice.
- No registry of xattrs used by the kernel, more documentation is needed! Multiple xattrs cannot be set through xattr API.
Capabilities required for backup...

- For read access to files we don’t own: CAP_DAC_READ_SEARCH
- For preserving special files (devices/sockets etc): CAP_MKNOD
- For preserving ownership: CAP_CHOWN
- For chmod/utimens, attrs, most xattrs, using O_NOATIME etc: CAP_FOWNER
  - If we have CAP_CHOWN we can skip this, we can preserve all we can and then change owner
- For the APPEND/IMMUTABLE attr: CAP_LINUX_IMMUTABLE
- For security.capabilities: CAP_SETFCAP
- For security/trusted xattrs: CAP_SYS_ADMIN -> That’s overkill!
- This is confusing and inconsistent!
When to backup a file...

- We can track data changes through mtime/size and compare between src/dst
  - But this is insecure/unreliable.
  - Rsync does crc32 which is still insecure, we could do e.g. SHA on both src/dst but that also has a serious overhead.
  - We could use IMA (security.ima) but that’s not available over NFS.
  - We could compare ctime to make sure that mtime wasn’t modified since our last backup but we can’t preserve ctime on dst to do the comparison!
- We can’t track metadata changes without reading them all (including all xattrs)!
  - Also because ctime cannot be preserved on dst, so we can’t compare it with src.
On preserving ctime for comparison...

- Why are we able to preserve atime/mtime and not ctime?
  - There is a chicken-and-egg issue, since changing ctime should also update ctime
  - It’s the most reliable way to determine if a file’s data/metadata changed, better let the kernel handle it
- But there are ways around this for privileged users
  - One can set the system time and force a ctime update by performing a modification on data/metadata
  - It’s possible to modify the data on-disk, like I did for example with ext4backup ([https://github.com/mickflemm/ext4backup](https://github.com/mickflemm/ext4backup))
  - It could even be done without unmounting the partition, using fsfreeze.
- And in some cases it’s not maintained in a consistent way e.g. for networked file systems (look for S_NOCTIME).
- So why not have a privileged API (e.g. a flag on utimens{at}) or something new, with a proper capability e.g. CAP_CTIME)?
What about btime/crtime?

- It’s probably more useful as it is, no need to preserve it.
  - There are cases where a file will be re-created on edit (e.g. vi does that) so btime/crtime says nothing about when the file’s contents were created.
  - We could however have a standard xattr for file content creation (in case it’s not supported by the file format).

- BTW NFS server exports btime/crtime but NFS client doesn’t use it.
Backing up encrypted files...

- With eCryptfs -> just copy the encrypted files (and ~/.ecryptfs etc)
- With fscrypt -> Not possible!
  - We can use statx to see if a file/dir/symlink is encrypted (STATX_ATTR_ENCRYPTED)
  - We can determine if the required key is present (so that we can copy them unencrypted)
    - For regular files we can try to open() them and fail with ENOKEY
    - For dirs we can do an ioctl()
    - For symlinks -> Not possible!
  - No way to copy data in encrypted form!
Summary...

- Add {list, set, get}xattrat() syscalls.
- Wrap old attrs as xattrs so that we don’t use ioctl(FS_IOC_{G,S}ETFLAGS) and have a common API for all attributes.
- Add a flag to copy_file_range() to preserve holes on sparse files, and also make it an io_uring op.
- Document all special xattrs / those used/set by the kernel, and the required capabilities to get/set them. Maybe also a new capability to set security/trusted xattrs without requiring CAP_SYS_ADMIN.
- Come up with a way to get a file’s measurement (or even just a hash of its data/metadata, as long as it’s only the kernel that can set it) without having to read the whole thing in userspace, that works over NFS.
- Come up with a privileged API to preserve ctime.
- Come up with an API for backing up fscrypt files in encrypted form.
Questions ?
Thank you !