Tracking filesystem modifications

Jan Kára <jack@suse.cz>

SUSE Labs, Novell

Introduction

- Lots of application need to watch for modification of files or changes of directory hierarchy
  - Backup / home directory synchronization
  - Desktop search / caching pre-parsed configuration files
  - Virus scanning
- Overview of possibilities for tracking changes in Linux

Outline

- Simple directory scanning
- Dnotify
- Inotify
- Fanotify
- What Btrfs has for us?
- Recursive modification timestamps

Directory scanning

- Read all directory entries using `readdir(3)` and `stat(2)`, compare modification time

Pluses

- Works everywhere

Minuses

- Need to stat all the files
  - Polluting caches
  - Slow

- Possible improvements
  - Sort statted files by inode number
  - Use st_nlink for subdirectory detection

Dnotify

- Linux's first attempt for improvement over plain dir scanning
- Process can register for events about modification
- Mostly of historical interest these days
- Interest in events expressed by calling `fcntl(2)` on directory file descriptor
- Events are delivered using signals (`siginfo`)
- Issues:
  - Dirs have to be open while receiving events
  - No way to watch single file
  - Signals are a poor interface

Inotify

Dnotify done right

Inotify interface

- Process can register for events about file / dir modification
- Setup:

```
fd = inotify_init1(flags)
wd = inotify_add_watch(fd, “path”, events)
...
```

events – a mask of event types we are interested on path – open, close, read, write, create, delete, move to / from

- Possible to have one shot / repeating event notification

Inotify interface (2)

- Receiving events:

```
read(fd, buf, bufsize)
```

receives events of the form

```
struct inotify_event {
  int wd;
  uint32_t mask;
  uint32_t cookie;
  uint32_t len;
  char name[];
}
```

- `fd` is pollable, may be non-blocking

Inotify troubles

- Event queue can overflow and events get lost
- Impossible to reliably access changed object
  - Tough to implement correct watching of a whole subtree
- Watches pin inodes in memory
  - Number of watches limited to 65536 by default unless root
- Time to setup all watches limiting for scarce tasks / when start time matters

Fanotify

Fanotify basics

- Motivated by needs of antivirus scanners
  - Verify writes, possibly block reads
- Doesn't supersede inotify
  - Limited to superuser
  - Does not support directory change notification
- Added in 2.6.36 kernel

Fanotify features

- Intent to see events for a given object called *mark*
- 4 types of events:
  - Open
  - Close
  - Read
  - Write
- Marks can be attached to files, directories (can receive events for all objects in a directory), mount points
- Ignore marks
  - Cleared on modification unless flagged

Fanotify features (2)

- Marks for mediating open / read of a file
  - Operation is suspended until the process which placed the mark allows or denies access
- Events return with open file descriptor to the object where an event happened

Fanotify interface

- Similar to inotify
- Setup:

```
fd = fanotify_init(descflags, markflags)
fanotify_mark(fd, flags, events, dfd, “path”)
```

- *`flags`* specify action to happen
  - create inode / mountpoint mark, remove mark, watch children, create ignore mark, create permanent ignore mark
- *`events`* specify type of event
  - Open, close, read, write, mediate-open, mediate-read

Fanotify interface (2)

- Receiving events by reading of `fd`

```
struct fanotify_event_metadata {
  uint32_t event_len;
  uint32_t vers;
  int32_t fd;
  uint64_t mask;
  int64_t pid;
}
```

Fanotify interface (3)

- When one of mediate events happen, decision is communicated by writing to `fd`

```
struct fanotify_response {
  int32_t fd;
  uint32_t response;
}
```

Fanotify shortcomings

- Unbounded event queues
  - Reason for restriction to superuser
  - Necessary for AV scanners
  - Event merging
- Misses directory events
- Mount point marks either need to process lots of events or we have to add lots of ignore marks

Persistent change tracking

What's that?

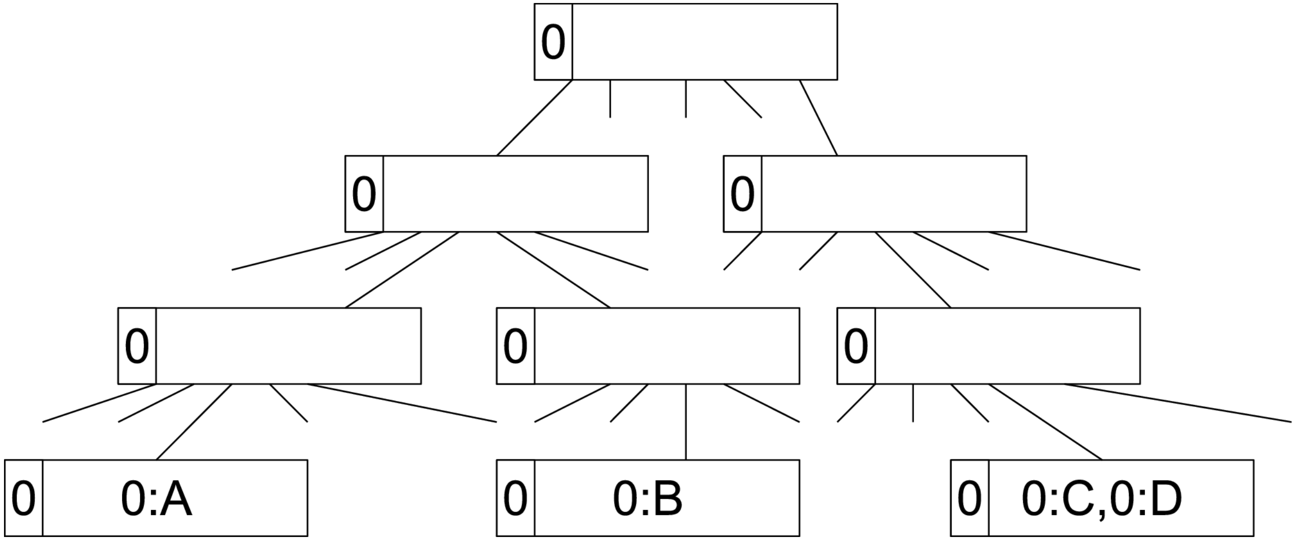
- Ability to track down modifications even after reboot
  - Possibly even after a crash
- Directory scanning using modification time works
- Inotify / fanotify hard to use
- Needs some filesystem support

Btrfs change tracking

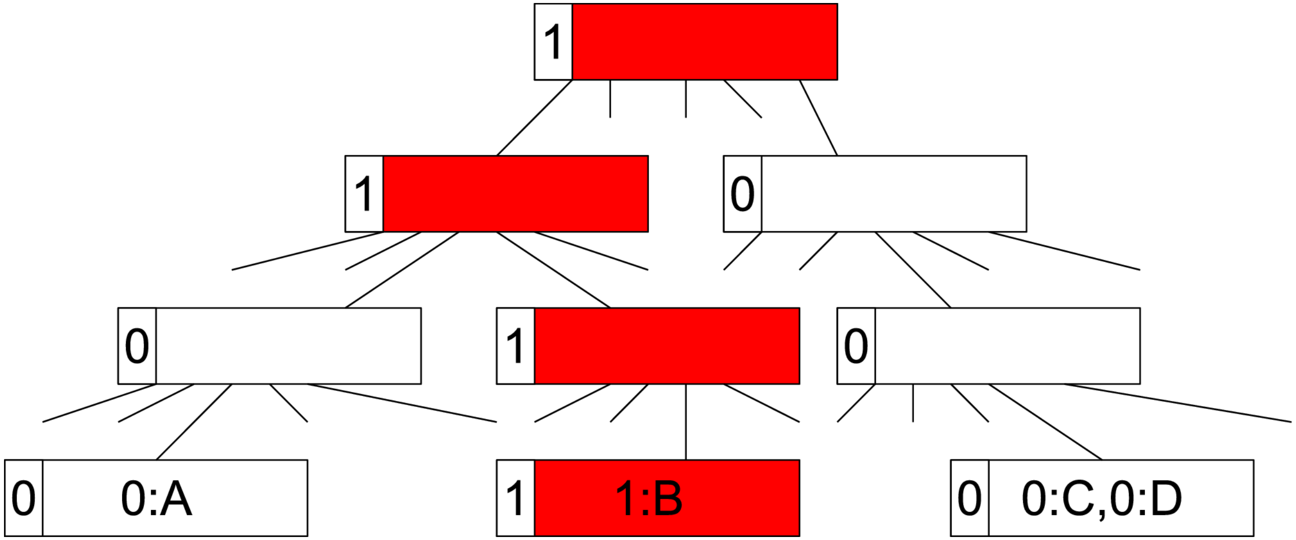
Btrfs design (parts)

- Filesystem items kept in a big B-tree
- Copy-on-write
  - Changes accumulated into transactions (30s)
- Each item and tree node has transaction ID when it was written
- Allows for fast (O(m log n)) search for items with given transaction ID or newer

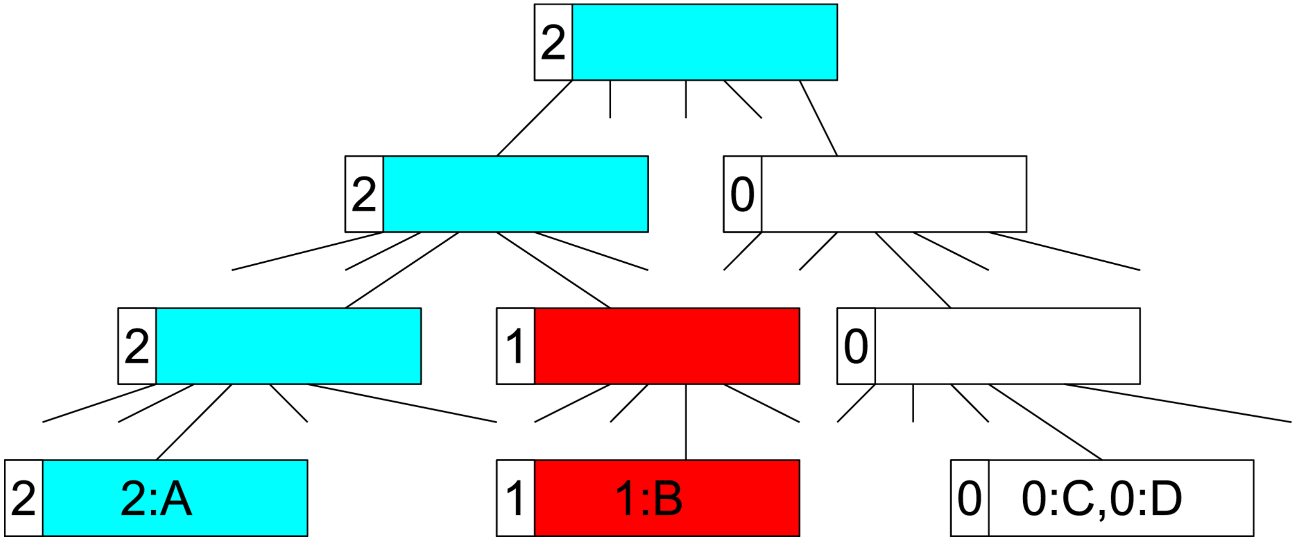
Copy-on-write and transaction IDs



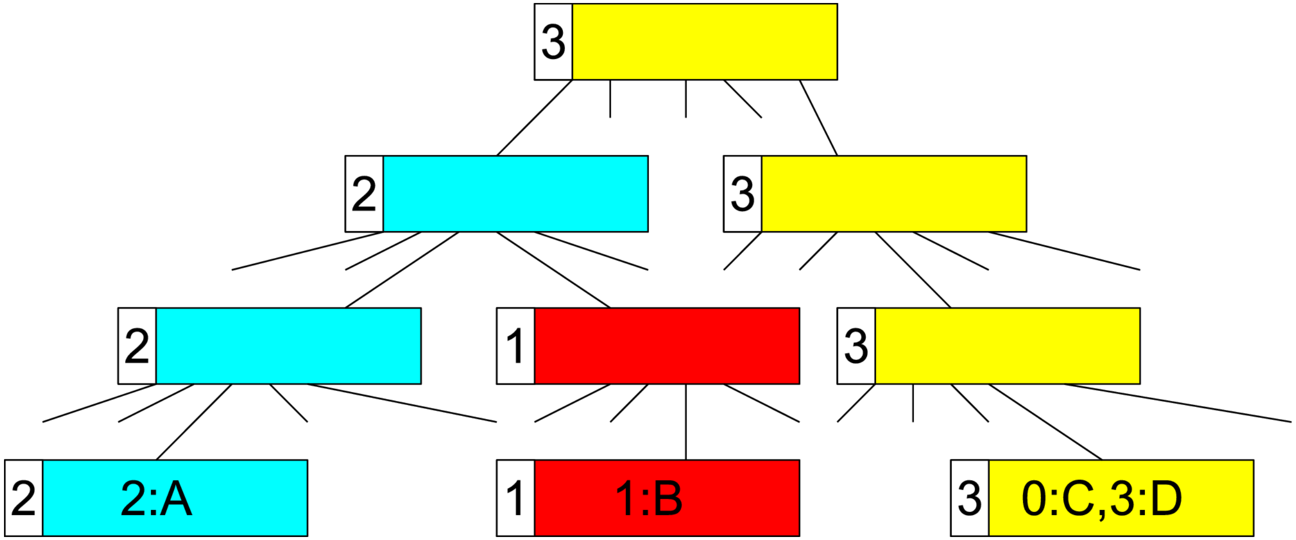
Copy-on-write and transaction IDs



Copy-on-write and transaction IDs



Copy-on-write and transaction IDs



Interface

- `BTRFS_IOC_SEARCH_TREE`
  - Interval searches in a tree
  - Rather complex with lots of fs details
- `btrfs` subvol find-new <mntpoint> <transid>
- Superuser only

Advantages and disadvantages

- Plus:
  - No initialization needed
  - All changes persistently tracked
  - No extra cost
  - Fast scan, selection mechanism for tree intervals
- Minus:
  - Specific to btrfs
  - 30 second granularity, changes in last 30 seconds not seen
  - Superuser only

Recursive modification time

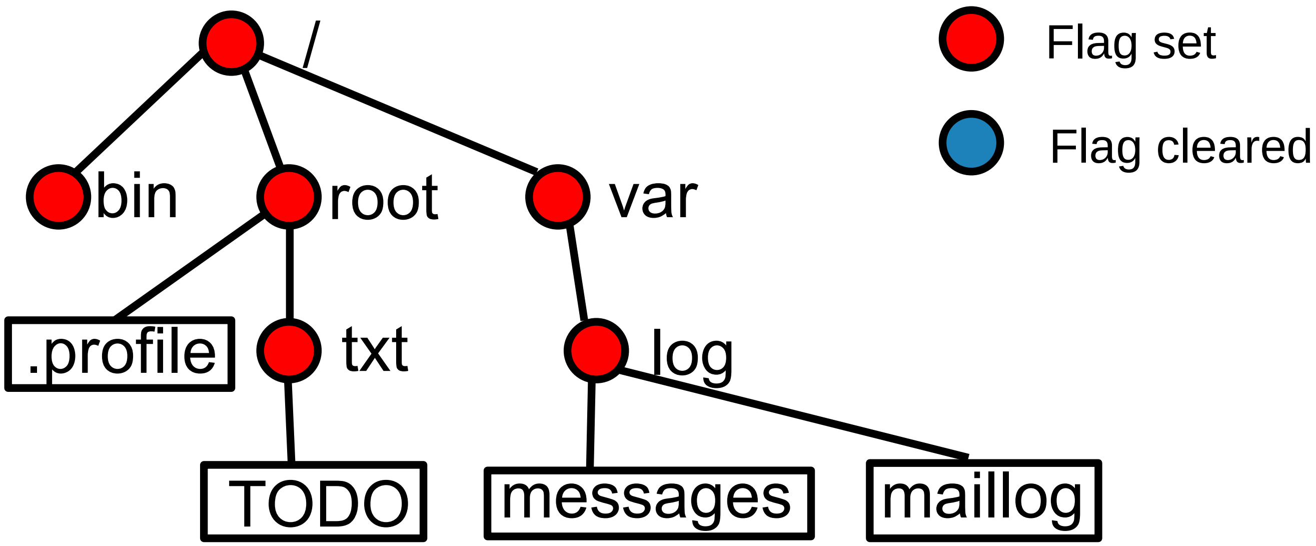
Recursive modification time basics

- Not in a mainline kernel
- Filesystem keeps with each directory a flag and a timestamp
- When a file in a directory is changed, it updates flags and timestamps starting by that directory as follows:
  - while current directory has the flag set
  - clear the flag
  - set timestamp to current time
  - go to the parent directory

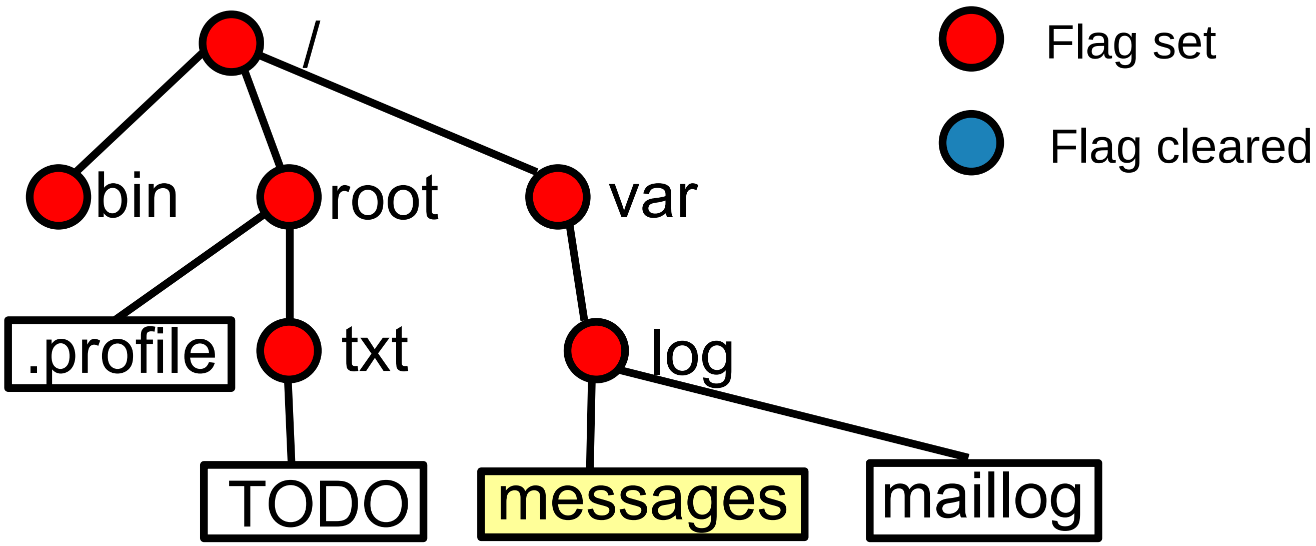
Recursive modification time usage

- Initialization: Set flag on directory application is interested in and its subdirectories
  - Needed just once per existence of each directory
- When it wants to check for changes, it can skip subdirectories whose timestamp is smaller than the time of the previous scan.
- Works for arbitrary number of applications watching the same directory
  - Only scans of this directory are going to happen more often and thus the cost of keeping flags and timestamps rises

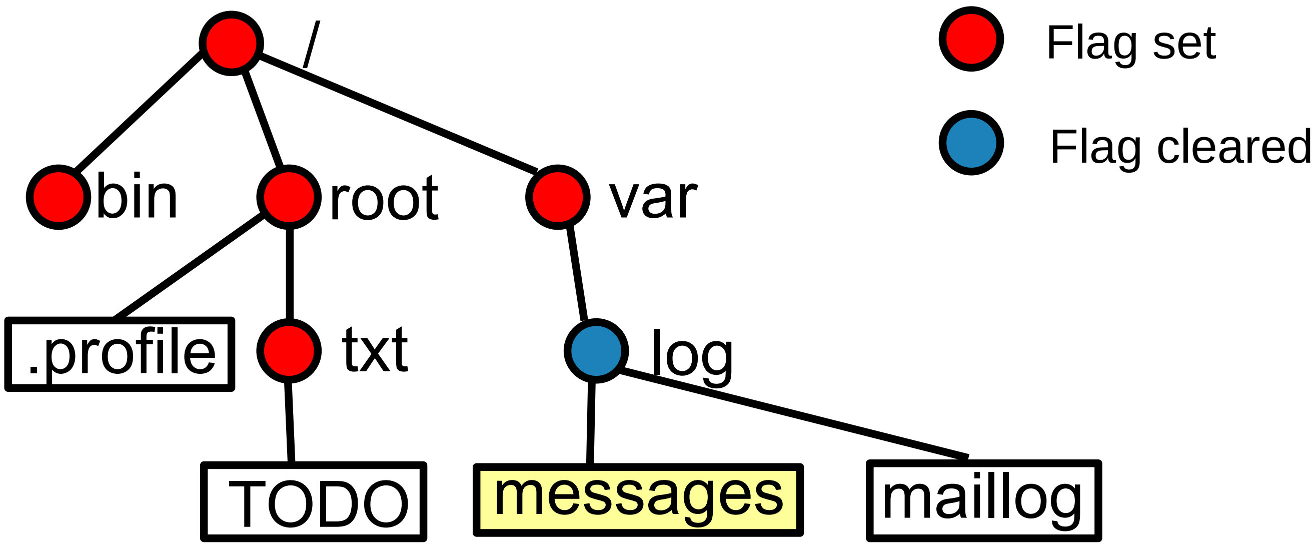
Example


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

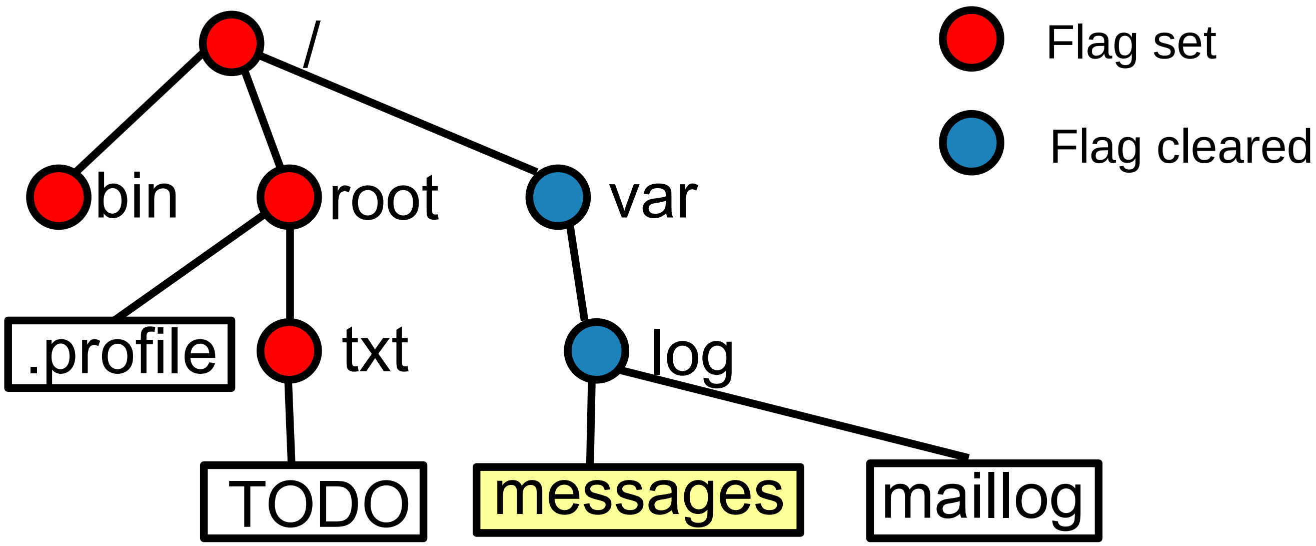
Example


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Example

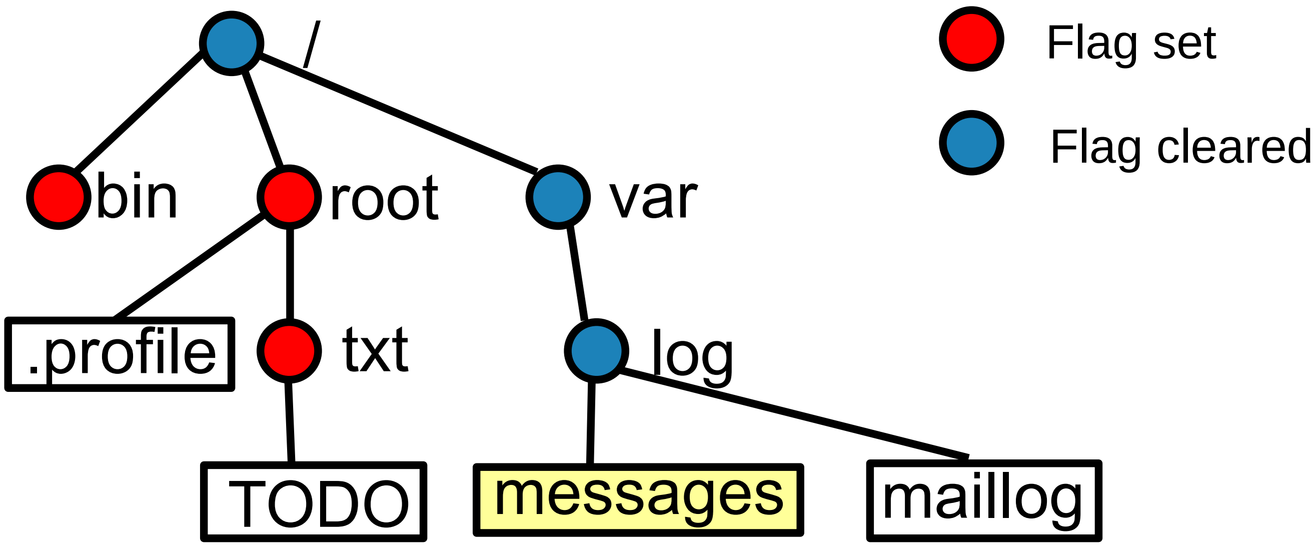

/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Example

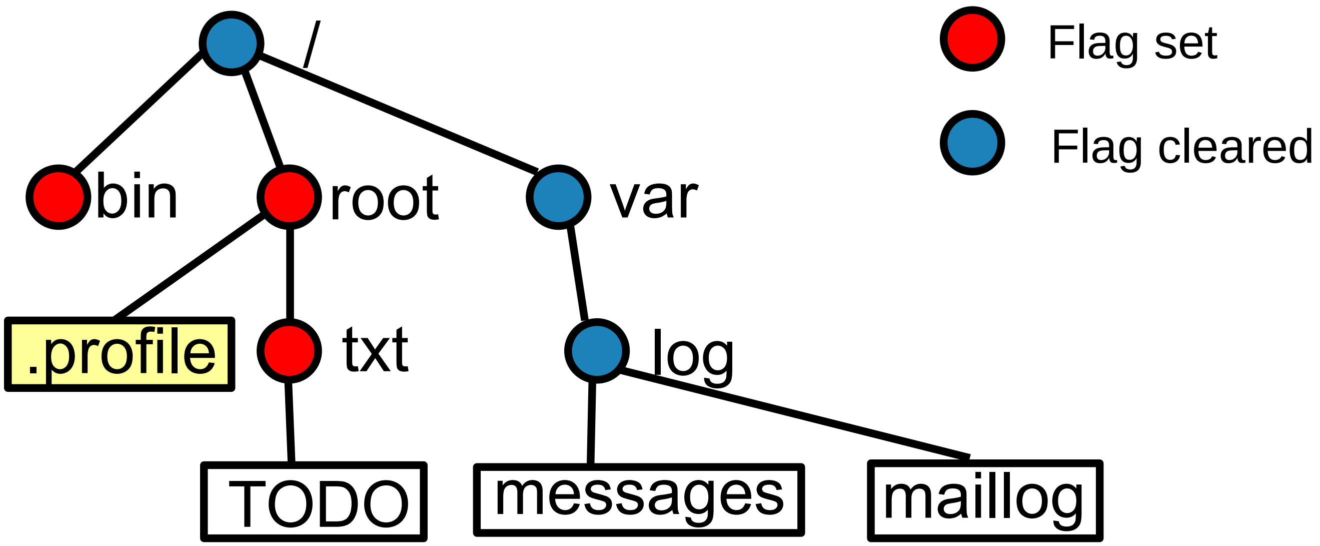


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

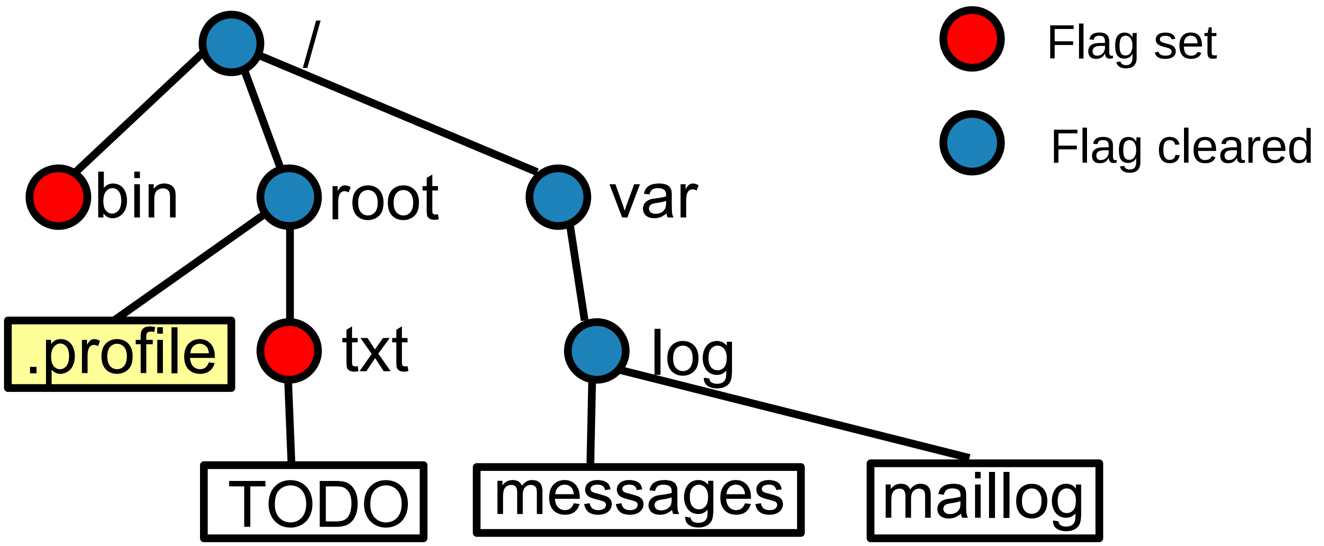
Example


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

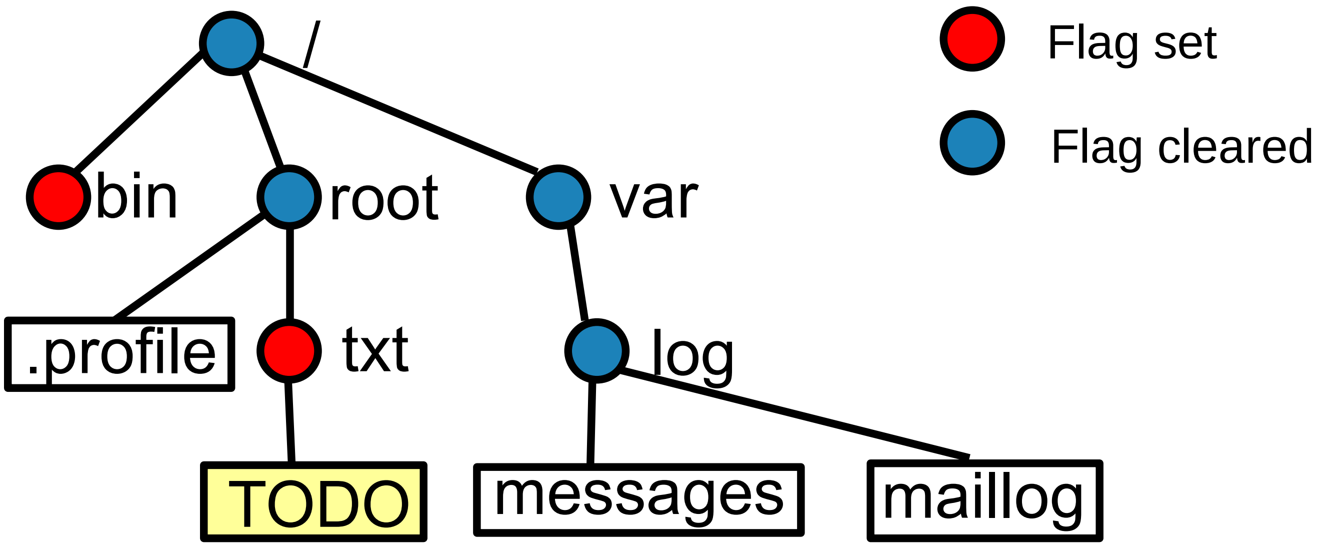
Example


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

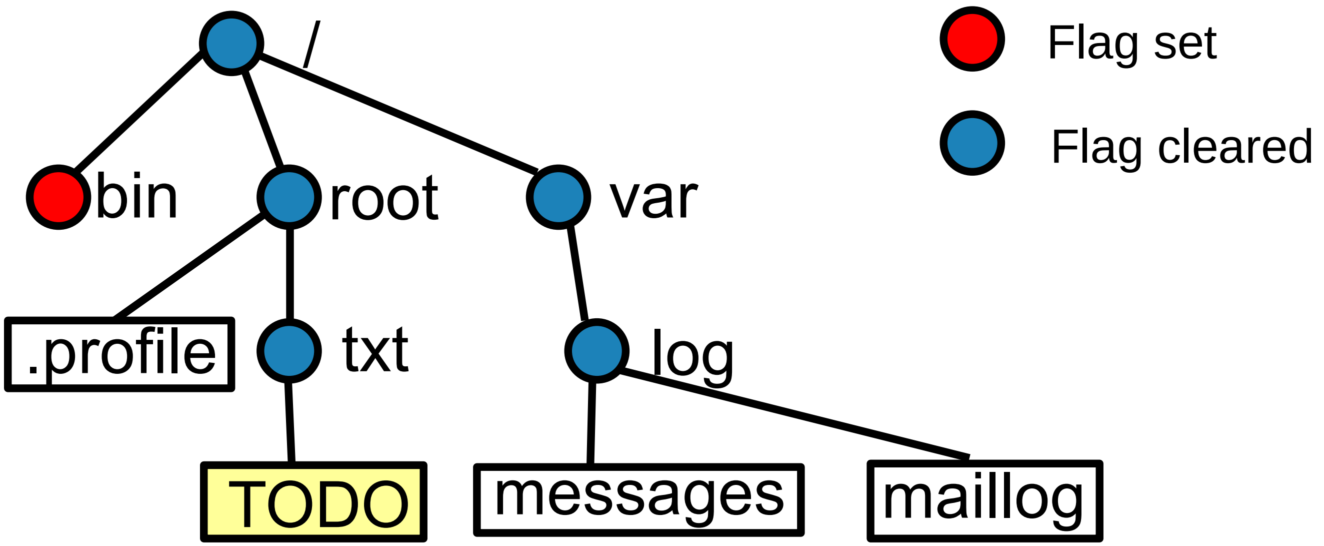
Example


/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Example

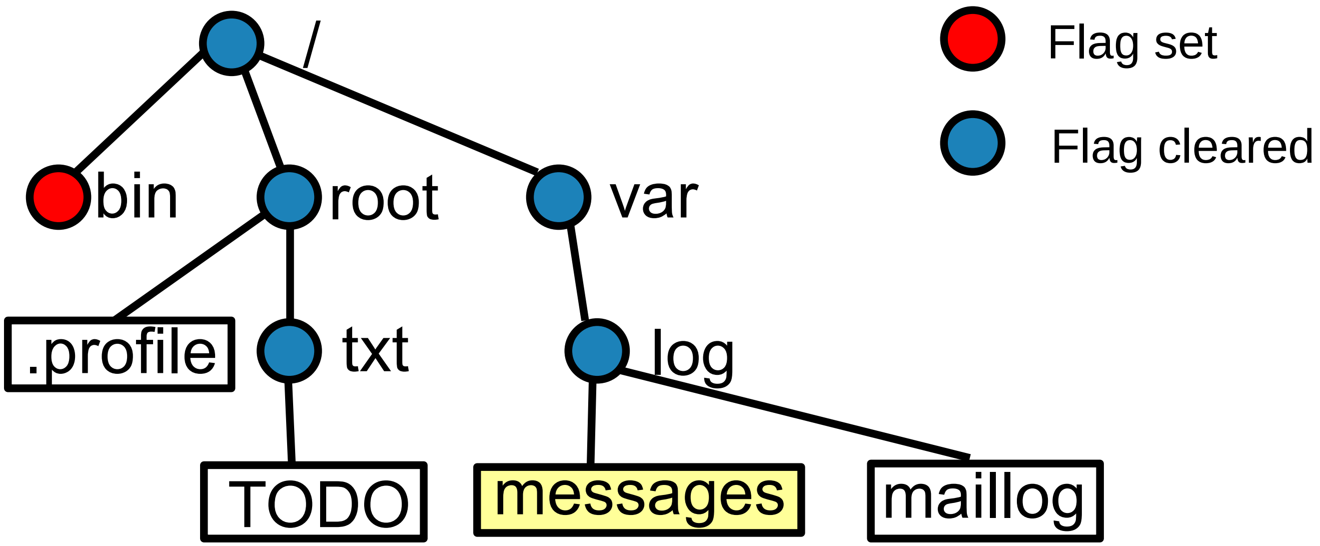

/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Example



/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Example



/
bin
root
var
.profile
txt
log
TODO
messages
maillog
Flag set
Flag cleared

Interface

- Flag is kept in an inode – `IOC_GETFLAGS`, `IOC_SETFLAGS`
- Nanosecond timestamp as `system.rtime` xattr

Advantages

- Requires just once-per-life initialization of each watched directory
- Scan for changes does not require entering unmodified directories
- Between two scans, timestamp and flag is changed at most once (good for frequently modified files)
- Scales well (easily to the whole filesystem)

Disadvantages

- Application still has to find which files were modified in a directory
- Userspace must handle hardlinks and propagation of information across mountpoints

Measurements

Setup

- 300 GB partition on a 1T SATA drive
- 1GB of RAM
- 2.6.36-rc4 kernel

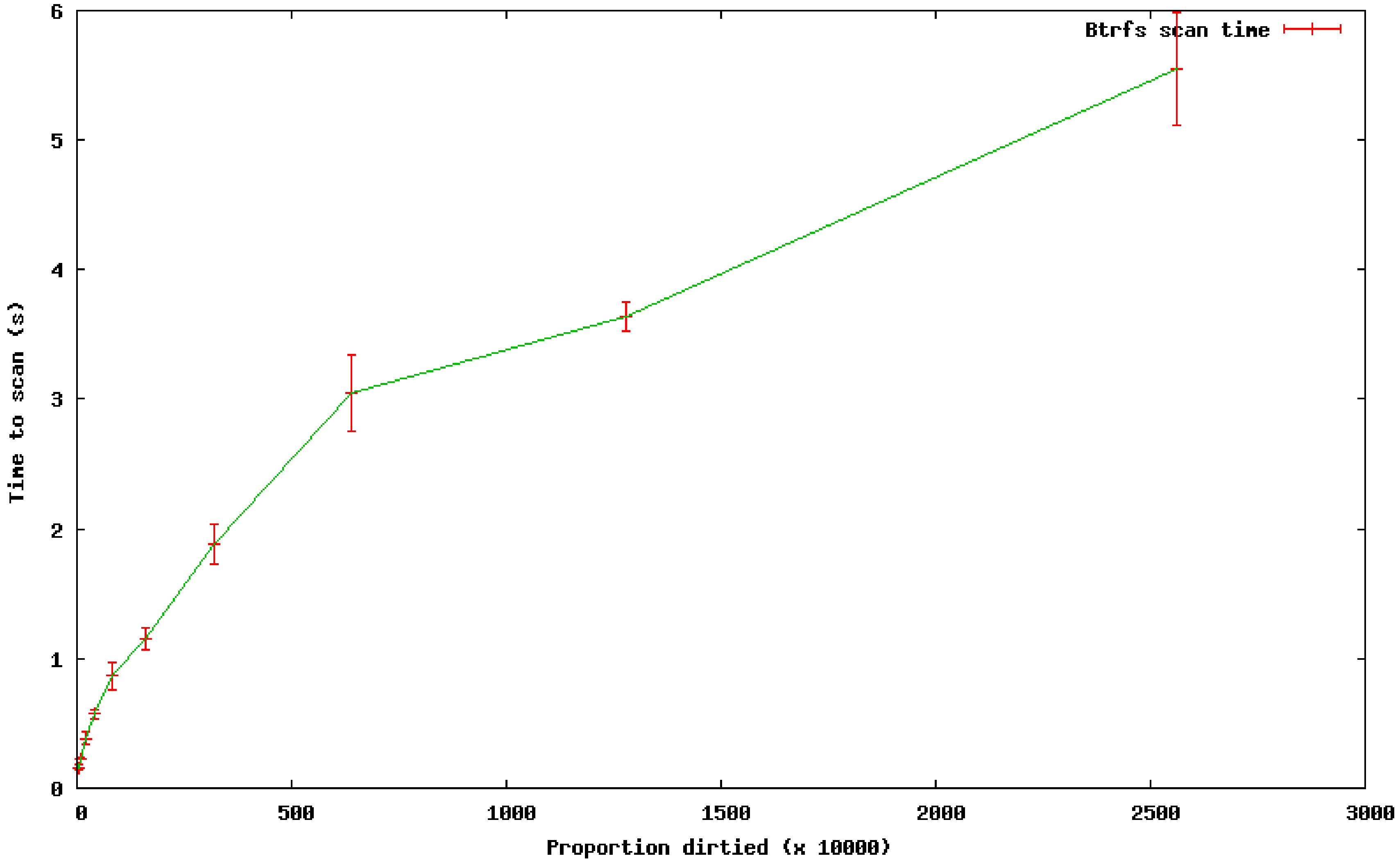
Plain directory scan

- Took a compiled kernel tree
  - 46878 files in 4255 directories
- Cleaned caches before each run

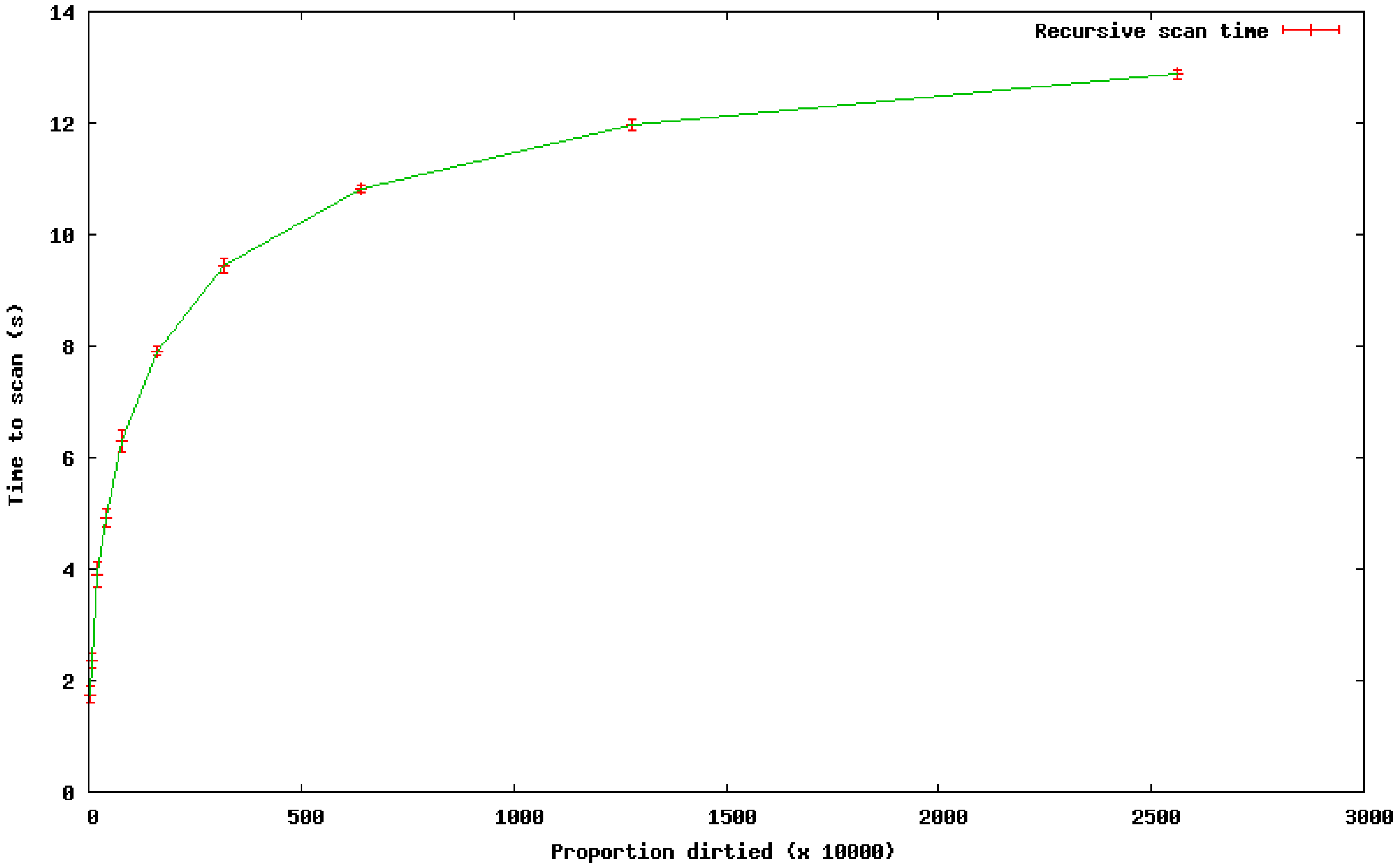
	Average	Std dev
Ext3	15.936	0.065
Ext3 + sort	13.569	0.066
Ext3 + nlink	6.062	0.145
Btrfs	17.349	0.754

Btrfs started scanning at 12.5 seconds

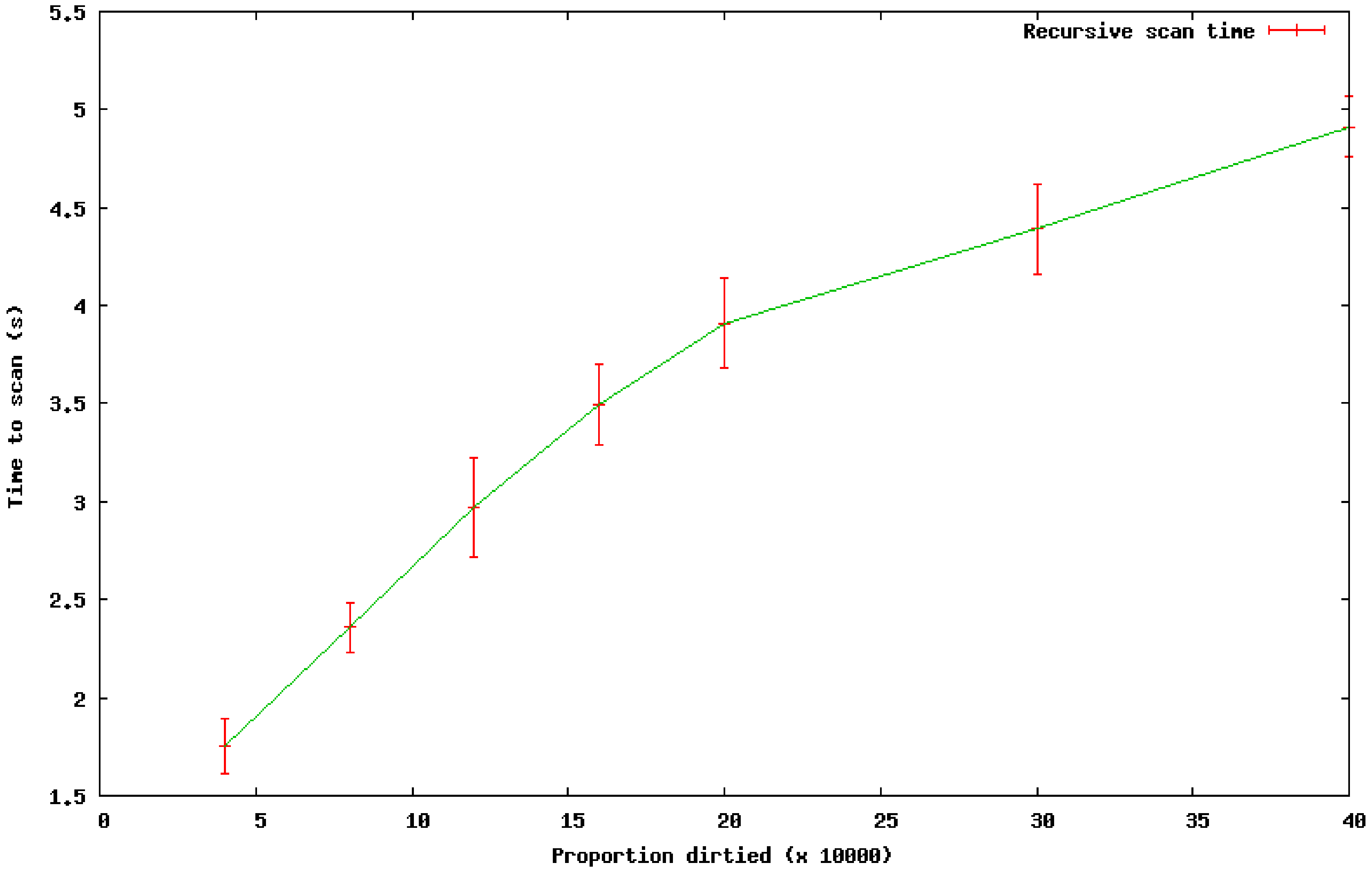
Btrfs modification tracking



Recursive modification time scan



Recursive mtime scan detail



Conclusion

- Seen several frameworks for event notification
  - Dnotify
  - Inotify – good general purpose
  - Fanotify – good for special cases
- Three methods of persistent modification tracking
  - Scanning using modification time – works everywhere
  - Btrfs modification tracking – fastest
  - Recursive modification time – possible to implement for a wide range of filesystems

Thank you