Blog

Fast Unix Commands

The world's fastest rm command and one of the fastest cp commands

Published Mar 24, 2023 • Last updated Mar 28, 2023 • 3 min read

Fast Unix Commands (FUC) is a project that aims to create the world's fastest Unix commands. Currently, this means `rm` and `cp` replacements named `rmz` and `cpz` (the 'z' stands for "zippy"). When better performance cannot be achieved, the next highest priority is efficiency. In practice, `rmz` appears to be the fastest file deleter available while `cpz` wins in most cases, only losing in flat directory hierarchies.

Myth busting

Many Stack Overflow answers will tell you to use this or that as a faster alternative to `rm` or `cp`. Let's look at the data!

Rsync

Using `rsync` for copying is always slower than `cp` as far as I can tell. This should not come as a surprise given that it performs data integrity checks. Interestingly enough, `rsync` deletes very large directories faster than `rm`, but is slower in all other cases.

Find

`find` and `rm` are approximately equivalent in terms of performance.

Tar

Shockingly, collecting a directory into a tarball and then extracting it into a new directory to copy it is often faster than `cp`.

Technical overview

Both tools are built using the same scheduling algorithm, following similar principles to FTZZ's scheduler. The key insight is that file operations in separate directories don't (for the most part) interfere with each other, enabling parallel execution. The intuition here is that directories are a shared resource for their direct children and must therefore serialize concurrent directory-modifying operations, causing contention. In brief, file creation or deletion cannot occur at the same time within one directory. Thus, the goal is to schedule one task per directory and execute each task in parallel.

Doing this for copies is relatively easy: iterate through every directory, spawn a new task when a directory is encountered and copy files in place. File removal is far more interesting because you cannot remove a directory until all of its children (including subdirectories) have been fully removed. As a consequence, file removal tasks must wait until their children have completed before finally removing the current directory. Unfortunately, this approach is slow: memory and time must be spent keeping track of child tasks, and children must somehow notify their parents of completion.

Flipping the problem on its head reveals a beautiful solution: what if children were in charge of deleting their parents? With a little bit of atomic reference counting, this solution is straightforward to implement and comes at almost no additional cost. While traversing directories, each spawned child directory task includes a parent (smart) pointer, implicitly creating a dynamic tree structure that models the directory hierarchy. These parent pointers are reference counted and trigger the directory deletion when fully freed. Additionally, each task decrements its reference count upon completion. That's it! Now, regardless of whether a parent finishes after all of its children or vice versa, the last "user" of a directory will delete its directory chain.

Pseudocode might make this clearer:

```
def delete_dir(node @ Node { dir, parent, ref_count }, task_queue):
    for file in dir:
        if file is dir:
            ref_count++
            task_queue.spawn(new Node { dir: file, parent: node, ref_count:
        else:
            file.delete()

    ref_count--
    while node.ref_count == 0:
        node.dir.delete()
        node = node.parent
        node.ref_count--
```

Enjoy blazing fast copies and deletions! 🚀

Appendix—directory contention benchmark

```
/*
This benchmark creates and then deletes ~16K files in N directories with tw
zip=every thread creates/deletes its own directory
chain=every thread creates/deletes a sub-set of the directory, one director

$ cargo b --release
$ cp target/release/test test
$ hyperfine --warmup 3 -N "./test /var/tmp 8 zip" "./test /var/tmp 8 chain"
Benchmark 1: ./test /var/tmp 8 zip
  Time (mean ± σ):     528.5 ms ±  11.5 ms    [User: 131.0 ms, System: 3460
  Range (min … max):   518.6 ms … 553.2 ms    10 runs

Benchmark 2: ./test /var/tmp 8 chain
  Time (mean ± σ):      1.488 s ±  0.140 s    [User: 0.143 s, System: 7.991
  Range (min … max):    1.297 s …  1.659 s    10 runs

Summary
  './test /var/tmp 8 zip' ran
    2.82 ± 0.27 times faster than './test /var/tmp 8 chain'
```

```rust
*/

use std::{
    env,
    fs::{create_dir, remove_dir, remove_file, File},
    io,
    path::PathBuf,
    thread,
};

const FILES: usize = 1 << 14;

fn main() {
    let arg = |n| env::args().nth(n).unwrap();
    let root = PathBuf::from(arg(1));
    let n: usize = arg(2).parse().unwrap();
    let method = arg(3);

    let dirs = (0..n)
        .map(|i| {
            let dir = root.join(format!("test{i}"));
            create_dir(&dir).unwrap();
            dir
        })
        .collect::<Vec<_>>();

    let run = {
        assert_eq!(FILES % n, 0);
        let batch_size = FILES / n;

        let dirs = dirs.clone();
        move |op: fn(PathBuf) -> io::Result<()>, id: usize| match &*method
            "zip" => {
                for i in 0..FILES {
                    op(dirs[id].join(format!("f{i}"))).unwrap();
                }
            }
            "chain" => {
                for dir in dirs {
                    let start = batch_size * id;
                    for i in start..(start + batch_size) {
                        op(dir.join(format!("f{i}"))).unwrap();
                    }
                }
```

```
            }
            _ => {
                unreachable!()
            }
        }
    };
    let bench = |op| {
        (0..n)
            .map(|id| {
                thread::spawn({
                    let run = run.clone();
                    move || run(op, id)
                })
            })
            .collect::<Vec<_>>()
            .into_iter()
            .map(|task| task.join().unwrap())
            .for_each(drop);
    };

    bench(|arg| File::create(arg).map(drop));
    bench(remove_file);

    for dir in dirs {
        remove_dir(dir).unwrap();
    }
}
```