Quash Report

EECS 678 Project I

Sandy Urazayev, Jacob McNamee*

286; 12021 H.E.

Contents

1 Quash

Quash is the best shell in the entirety of our existence. Let's walk through how it's built!

2 Forking and Executing

Quash is implemented in Go, which itself is a garbage-collected language that runs threads to maintain the language runtime. When we want to fork from a Go application, the forking will only spawn a copy of the thread that initiated the forking. Therefore this new subprocess that just got forked lacks all the supporting threads that Go applications **absolutely must have** for adequate runtime performance. Therefore, Go does allow us to call `fork`, but we **have** to run `exec` immediately, such that the call and execution stack is immediately replaced by the newly loaded program.

*University of Kansas (ctu@ku.edu)

This is achieved by `syscall.ForkExec` library call that only spawns a subprocess with a loaded program and returns the new process's `pid`.

```
pid, err := syscall.ForkExec(
        paths, args, &syscall.ProcAttr{
                Dir:   string,
                Env:   []string,
                Files: []uintptr,
                Sys:   &syscall.SysProcAttr{},
        })
```

Notice that we have to pass in a couple of parameters, where `Dir` is the current active directory where we are located, `Env` is a slice of strings, which contains our environmental variables, `Files` is a slice of unsigned file descriptor pointer values, and `Sys` is a struct to pass additional options.

3 PATH

In order to run executables, we have to have a list of directors where we would look for one. For this, we have our `PATH` environmental variable. Quash solves this problem rather simply by going through all the directories in `PATH` and searching for an exact executable name match in their globs. The binary finding code is below

```
// lookPath tries to find an absolute path to an executable
// name by searching directories on the PATH
// If the name is an absolute path or a shortened path (./)
// then this path is returned
func lookPath(name string) (string, error) {
        if filepath.IsAbs(name) { //if the user has absolute path then we good
                return name, nil
        }

        absPath := filepath.Join(currDir, name)
        _, err := os.Stat(absPath)
        if !os.IsNotExist(err) {
                return absPath, nil
        }
        path := getenv("PATH")
        if path == "" {
                err := errors.New("executable not found")
```

```
                return "", err
        }
        directories := strings.Split(path, ":")
        for _, directory := range directories {
                dirInfo, err := os.ReadDir(directory)
                if err != nil {
                        //quashError("%s : %s", errors.Unwrap(err), directory)
                        continue
                }
                for _, file := range dirInfo {
                        if file.Name() == name && !file.IsDir() {
                                return directory + "/" + name, nil
                        }
                }
        }
        err = errors.New("executable not found")
        return "", err

}
```

Notice that the function would return the full path for a binary (example if `PATH = /usr/bin` and executable is `echo`, `lookPath` would return `/usr/bin/echo`). `getenv` and `setenv` are our user-defined functions that access the global variable `myEnv`, which holds all of our active environmental variables.

4 Background processes