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

*University of Kansas (ctu@ku.edu)

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.

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 Pipes

Quash allows the user to sequentially run multiple programs while passing the output data from one program to the input data of the next program in the sequence. This is accomplished with the use of pipes. When Quash receives a command, it separates the command into the programs the command wants us to run and creates pipes to connect the processes to be created.

```
// split input into different commands to be executed
commands := strings.Split(input, "|")
```

```
for index, command := range commands {
        commands[index] = strings.TrimSpace(command)
        args := strings.Split(commands[index], " ")
        args[0] = strings.TrimSpace(args[0])
        if builtinFunc, ok := builtins[args[0]]; ok && len(commands) == 1 {
                builtinFunc(args)
                addToHistory(input)
                return
        } else if ok {
                quashError("built-in command inside pipe chain")
                return
        }
}

pipeRead, pipeWrite := createPipes(len(commands) - 1)
```

While the processes are being created (see Forking and Executing), the processes are assigned a custom file descriptor table created using the `fileDescriptor()` function. If there are pipes present in the command, then `fileDescriptor()` will use the created pipes as files in the descriptor table, overwriting the default behavior that uses the operating systems standard input (`stdin`) and standard output (`stdout`).

```
// fileDescriptor returns a custom file descriptor for a call to ForkExec
// if there is only one command with no pipes, Stdin Stdout and Stderr are used
// pipes overwrite read, write, or both for processes inside of a pipe chain.
func fileDescriptor(
        index int,
        readPipe []*os.File,
        writePipe []*os.File,
        in *os.File,
        out *os.File,
        err *os.File,
) []uintptr {
        // One command, so no pipes
        if len(readPipe) == 0 {
                return []uintptr{
                        in.Fd(),
                        out.Fd(),
                        err.Fd(),
                }
        }
        // first in a chain
```

```
        if index == 0 {
                return []uintptr{
                        in.Fd(),
                        writePipe[0].Fd(),
                        err.Fd(),
                }
        }
        // last in a chain
        if index == len(readPipe) { ... }
        // middle of a chain
        return []uintptr{ ... }
}
```

Finally, we must close the pipes within the quash process in order to properly transmit EOF when a child process finishes execution. This is done using the `closePipe()` function, which closes the pipe ends that we distributed to the child process using the `fileDescriptor()` function.

```
// closePipe closes used pipe ends based on where they are in a chain of piped
// commands if only one command exists, there are no pipes and this function
// does nothing.
func closePipe(index int, readPipe []*os.File, writePipe []*os.File) {
        // One command, so no pipes
        if len(readPipe) == 0 {
        } else if index == 0 {
                // first in a chain
                writePipe[0].Close()
        } else if index == len(readPipe) {
                // last in a chain
                readPipe[index-1].Close()
        } else {
                // middle of a chain
                readPipe[index-1].Close()
                writePipe[index].Close()
        }
}
```

Note that in C you would have to also close excess pipes between the fork and execute function calls in the child process, but in Go we only assigned the child process the necessary pipes, so no additional pipes need to be closed.

5 Background Processes

6 Builtins

7 Arrow Keys