Quash Report

EECS 678 Project I

Sandy Urazayev, Jacob McNamee*

286; 12021 H.E.

Contents

*University of Kansas (ctu@ku.edu)

1 Building and running

You have to have a proper go 1.17 installation to build it. Simply run

```
go get -u -v ./...
```

to get all the dependencies and then run

```
go build -v
```

to build it. You will get an executable `quash` in your directory. Simply run it. We also provide a makefile, running `make` will do it all for you.

To execute quash, type in `./quash` or run

```
go install ./...
```

to run it globally by simply invoking `quash` anywhere on your system. The binary is fully independent.

2 Quash

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

3 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.

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.

4 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.

5 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.

6 Background Processes

Like many other shell programs, Quash has the ability to execute programs in either the foreground or the background. A program or group of programs running in the background is called a job. A program is designated to run in the background as a job by adding the & character to the end of the command. A set of programs linked by pipes can also be run in the background the same way, using a single & at the very end. For example, `ls &` and `ls | wc &` both create jobs that will execute in the background.

```
// job is the struct that holds info about background processes
type job struct {
        // pid associated with currently running process in the job
```

```
        pid int
        // jid associated with this job
        jid int
        // command that created this job
        command string
        // reference to the current process
        process *os.Process
}
```

Each job in an instance of Quash will be assigned a unique job identifier (`jid`). Jobs are referenced using these identifiers when using built in commands such as jobs or kill (see **Builtins**). Additionally, each job will print a message when they are first created and when they terminate. If one process within a pipe chain terminates with an error, the job will terminate.

7 Builtins

Quash has a handful of pre-defined keywords that perform special functionality for the user. These commands are: `exit`, `quit`, `set`, `cd`, `kill`, `jobs`, and `history`. These built in functions cannot be executed as part of a chain of processes, as they are not themselves process. Instead they are functions that manipulate aspects of the shell, such as changing the environment.

7.1 `quit` / `exit`

`quit` and `exit` are aliases for the same function within Quash. This function terminates Quash.

```
Usage: quit or exit
```

7.2 `set`

`set` allows the user to change environment variables, such as the current working PATH. The initial variables and values are set by the OS. `set` can also add a new variable to Quashs environment (but not the OSs environment).

```
Usage: set variable
```

where `variable` is the name of the variable to add or update, and value is the value to set variable as.

7.3 `cd`

`cd` stands for change directory. cd changes the current directory that Quash is working within.

```
Usage: cd directory
```

where `directory` is an absolute or relative path to change to. If no directory is specified, then cd will change the directory to the `$HOME` directory specified in Quashs environment.

7.4 `kill`

`kill` allows the user to manually send signals to a currently executing job. This is especially useful for sending signals to forcefully end the job, hence the name `kill`.

```
Usage: kill signal jid
```

where `signal` is the number of the signal you wish to send (check your OS to see what number each signal corresponds to) and `jid` is the job identification number corresponding to the job you wish to signal.

7.5 `jobs`

`jobs` prints all currently executing background jobs.

```
Usage: jobs
```

Output: `[jid] pid running in background` where `jid` is the job identification number for the job and `pid` is the process identification number for the currently executing process within the job. This line is printed for each currently running job, sorted by `jid`.

7.6 `history`

`history` prints a list of all previous valid commands used within the current execution of Quash. If the command failed, such as misspelling an executable name, the command will not be added to the history.

```
Usage: history
```

Output: `number cmd` where `number` is the index of the command starting at 1 and `cmd` is the entire text of the previous command. This line is printed for every previous valid command, sorted by number.

8 Arrow Keys

We support arrow key movements! We do this by manually catching keyboard interrupts from `/dev/tty` with `keyboard` interface and then depending on each key pressed, we decide on what to do. This actually changes the input logistics completely, as in when the user presses a key, it doesn't get flushed onto the screen, we swallow it and must decide what to do with it. We catch all the special keys and then print all printable characters we caught. The subroutine for it looks like the following

```
// takeInput reads a newline-terminated input from a bufio reader
func takeInput(reader *bufio.Reader) string {
        if err := keyboard.Open(); err != nil {
                panic(err)
        }
        defer func() {
                _ = keyboard.Close()
        }()

        cmdNum := len(goodHistory)
        var readCharacter rune
        input := ""
        curPosition := 0

        for {
                char, key, err := keyboard.GetKey()
                if err != nil {
                        quashError("bad input: %s", err.Error())
                }
                readCharacter = char

                // See what key we actually pressed, I tried doing switch
                // but it works kinda wonky. If statements forever <3
                // ------------------------------------------------
```

```
// On enter, flush a newline and return whatever we have
if key == keyboard.KeyEnter {
        fmt.Fprint(os.Stdout, NEWLINE)
        return input + string(char)
}
// On Ctrl-D or Escape just close the shell altogether
if key == keyboard.KeyEsc {
        if isTerminal {
                fmt.Fprint(os.Stdout, NEWLINE)
        }
        exit(nil)
}
// Only exit on Ctrl-D if input is empty
if key == keyboard.KeyCtrlD {
        if curPosition != 0 || len(input) != 0 {
                continue
        }
        if isTerminal {
                fmt.Fprint(os.Stdout, NEWLINE)
        }
        exit(nil)
}
// On a space just set readCharacter to a space run
if key == keyboard.KeySpace {
        readCharacter = ' '
}
// On backspace, move cursor to the left, clean character,
// and move the cursor again to the left. Delete last input element
if key == keyboard.KeyBackspace || key == keyboard.KeyBackspace2 {
        // If cursor is already at the home position, don't move
        if curPosition < 1 {
                continue
        }
        fmt.Fprintf(os.Stdout, "\b \b")
        input = input[:curPosition-1]
        curPosition--
        continue
}
// On arrow up press, clean out the terminal and replace the user input
// with whatever previous good command we can find. Works on multiple
// arrow up key presses too
if key == keyboard.KeyArrowUp {
        if len(goodHistory) < 1 {
```

```
                continue
        }
        // Clear the input first
        resetTermInput(len(input))
        cmdNum = prevCmdNum(cmdNum)
        input = printOldGoodCommand(cmdNum)
        curPosition = len(input)
        continue
}
// On arrow down press, clean out the terminal and replace with whatever
// command came after. Only makes sense if run after one or mory presses
// of the arrow up key. On the bottom it will set user input to just clean
if key == keyboard.KeyArrowDown {
        if len(goodHistory) < 1 {
                continue
        }
        resetTermInput(len(input))
        // If at the end of history, just clear the input
        if cmdNum >= len(goodHistory)-1 {
                input = ""
                cmdNum = len(goodHistory)
                curPosition = 0
                continue
        }
        // Get the later good command
        cmdNum = nextCmdNum(cmdNum)
        input = printOldGoodCommand(cmdNum)
        curPosition = len(input)
        continue
}
// Ignore left and right arrow keys
if key == keyboard.KeyArrowLeft || key == keyboard.KeyArrowRight {
        continue
}
// Send kill signals if ctrl is encountered or clear the input
if key == keyboard.KeyCtrlC {
        // Don't do anything if we have an empty command
        if curPosition == 0 && len(input) == 0 {
                sigintChan <- syscall.SIGINT
                continue
        }
        fmt.Fprintf(os.Stdout, "\033[41m^C\033[0m\n")
        input = ""
```

```
                        curPosition = 0
                        greet()
                        continue
                }
                // Ctrl-L should clear the screen
                if key == keyboard.KeyCtrlL {
                        executeInput("clear")
                        greet()
                        // Reprint whatever we had before
                        fmt.Fprintf(os.Stdout, "%s", input)
                        continue
                }
                // If the character is NOT printable, skip saving it
                if !unicode.IsPrint(readCharacter) {
                        continue
                }
                // Print the character that we swallowed up and append to input
                fmt.Fprint(os.Stdout, string(readCharacter))
                input += string(readCharacter)
                curPosition = len(input)
        }
}l
```