SMALL SHELL IN C: FUNCTIONAL DECOMPOSITION

RUNTIME EXAMPLE

```
os1 ~/cs344/cs344_fall2017/block3 807$ smallsh
: pwd
/nfs/stak/users/maass/cs344/cs344_fall2017/block3
: cd
: pwd
/nfs/stak/users/maass
: cd cs344
: date
Fri Dec  1 22:56:17 PST 2017
: sleep 20
^C
terminated by signal 2
: status
terminated by signal 2
: date
Fri Dec  1 22:56:28 PST 2017
: status
exit value 0
: sleep 5 &
background pid is 36034
: ls > junk
: cat junk
cs344_fall2017
junk
background pid 36034 is done: exit value 0
: wc < junk
 2  2 20
: wc < junk > junk2
: cat junk2
 2  2 20
: mkdir testdir$$
: ls
cs344_fall2017  junk  junk2  testdir24913
: cd testdir$$
: pwd
/nfs/stak/users/maass/cs344/testdir24913
: ^Z
Entering foreground-only mode (& is now ignored)
date
Fri Dec  1 22:57:47 PST 2017
: sleep 5 &
date
: Fri Dec  1 22:57:56 PST 2017
: exit
```

SIGNAL HANDLER

```
/*************************************************************************
 * catchSIGTSTP: This function is a signal handler for SIGTSTP. It toggles
 * foreground process only mode on and off. All background process requests
 * are run in the foreground when this is on.
 ************************************************************************/
void catchSIGTSTP(int signo) {
    // toggle flag
    ForegroundOnlyFlag = (ForegroundOnlyFlag == TRUE) ? FALSE: TRUE;
    // print appropriate message
    if (ForegroundOnlyFlag == TRUE) {
        char* message = "\nEntering foreground-only mode (& is now ignored)\n";
        write(STDOUT_FILENO, message, 50);
    } else {
        char* message  = "\nExiting foreground-only mode\n";
        write(STDOUT_FILENO, message, 30);
    }
    fflush(stdout);
}
```

STATUS COMMAND

```
/*************************************************************************
 * status: This function will print either the exit status or terminating
 * signal of the last foreground process.
 ************************************************************************/
void status() {
    if (WIFEXITED(pidExitStatus)) {
        printf("exit value %d\n", WEXITSTATUS(pidExitStatus));
    } else if (WIFSIGNALED(pidExitStatus)) {
        printf("terminated by signal %d\n", WTERMSIG(pidExitStatus));
    }
    fflush(stdout);
}
```

Oregon State University

smallsh: I broke my code down into small, reusable functions that are clear and easy to follow. This made development and testing go very quickly.

KEY CONCEPTS

- Signals – custom signal handlers with sigaction
- File I/O and redirection
- Forking processes to emulate foreground and background, including reaping zombie processes
- Checking exit status of processes
- “Built-in” commands – exit, cd, status
- C string manipulation
- Memory management

SOURCE CODE

https://github.com/sarahmaas/small-shell/

STRING REPLACEMENT

```
/*************************************************************************
 * replaceString: This function will replace a given substring with another
 * given substring and returns the new string.
 ************************************************************************/
char* replaceString(char* string, char* search, char* replace) {
    int bufferSize = strlen(string) + strlen(replace) + 1;
    char* buffer = (char*) calloc(bufferSize, sizeof(char));
    // set up iteration pointers
    char* strPtr = string;
    char* bufPtr = buffer;
    char* lastStrPtr = string;
    // while search term is found
    while(strPtr = strstr(strPtr, search)) {
        // check length for overflow, resize buffer
        // current length + new str piece + increase + null term.
        int reqLength = strlen(buffer) + (int)(strPtr - lastStrPtr)
            + strlen(replace) - strlen(search) + 1;
        if (reqLength > bufferSize) {
            bufferSize += reqLength;
            char* newBuffer = calloc(bufferSize, sizeof(char));
            memcpy(newBuffer, buffer, strlen(buffer));
            // update iteration pointer to new location
            bufPtr = newBuffer + strlen(newBuffer);
            free(buffer);
            buffer = newBuffer;
        }
        // concatenate string up through replacement
        sprintf(bufPtr, "%.*s%s", (int)(strPtr - lastStrPtr), lastStrPtr,
            replace);
        // update to point at end of string
        bufPtr = strlen(buffer) + buffer;
        // update to point at the remaining string after $$
        strPtr += strlen(search);
        lastStrPtr = strPtr;
    }
    // concatenate the end
    strcat(bufPtr, lastStrPtr);
    return buffer;
}
```

FUNCTION LIST

```
// function prototypes
void initGlobalVars();
void cleanGlobalVars();
void runShell();
char* getCommand();
char** parseCommand(char** args, char* userInput);
char* replaceString(char* string, char* search, char* replace);
void processCommand(char** args);
void exitShell();
void cd(char** args);
void status();
void executeExternalCommand(char** args);
boolean parseIO(char** args, char** inputFileName, char** outputFileName);
boolean redirectInput(char* inputFileName);
boolean redirectOutput(char* outputFileName);
void printFinishedChildren();
void setSignalHandlers();
void catchSIGINT(int signo);
void catchSIGTSTP(int signo);
ProcessList* initProcessList(int capacity);
void addToProcessList(ProcessList* processList, pid_t processID);
boolean removeFromProcessList(ProcessList* processList, pid_t processID);
void deleteProcessList(ProcessList* processList);
void printProcessList(ProcessList* processList);
```

FORKING PROCESSES

```
/*************************************************************************
 * executeExternalCommand: This function executes an external command in its
 * own child process, including background processes as requested.
 ************************************************************************/
void executeExternalCommand(char** args) {
    char* inputFileName = NULL;
    char* outputFileName = NULL;
    // i/o redirection detection, skip command if bad input
    boolean valid = parseIO(args, &inputFileName, &outputFileName);
    if (!valid) {
        pidExitStatus = 1;
        return;
    }
    // fork
    pid_t spawnpid = -5;
    pidExitStatus = -5;
    spawnpid = fork();
    if (spawnpid == 0) {
        // I am the child! Execute things.
        // ignore SIGTSTP, changes made during process only affect next command
        struct sigaction ignore_action = {0};
        ignore_action.sa_handler = SIG_IGN;
        sigaction(SIGTSTP, &ignore_action, NULL);
        // input, output, files, oh my! set background
        if (ForegroundOnlyFlag == FALSE && BackgroundProcessFlag == TRUE) {
            // don't kill background with SIGINT
            sigaction(SIGINT, &ignore_action, NULL);
            // background process, redirect to /dev/null if not to/from file
            if (inputFileName == NULL) {
                asprintf(&inputFileName, "%s", "/dev/null");
            } else if (outputFileName == NULL) {
                asprintf(&inputFileName, "%s", "/dev/null");
            }
        }
        // open files to redirect I/O
        redirectInput(inputFileName);
        redirectOutput(outputFileName);
        // execute command
        execvp(args[0], args);
        printf("%s: command not found\n", args[0]);
        fflush(stdout);
        exit(1);
    }
    // waits for nonbackground and prints if terminated
    pid_t caught;
    if (ForegroundOnlyFlag == TRUE || BackgroundProcessFlag == FALSE) {
        caught = waitpid(spawnpid, &pidExitStatus, 0);
        if (WIFSIGNALED(pidExitStatus)) {
            printf("terminated by signal %d\n", WTERMSIG(pidExitStatus));
            fflush(stdout);
        }
    } else {
        printf("background pid is %d\n", (int)spawnpid);
        fflush(stdout);
        addToProcessList(BackgroundProcessList, spawnpid);
    }
    free(inputFileName);
    free(outputFileName);
}
```

PARSE/UPDATE CHAR**

```
/*************************************************************************
 * parseIO: This function parses through the given command to set input and
 * sources as found and remove them from the command to for processing.
 ************************************************************************/
boolean parseIO(char** args, char** inputFileName, char** outputFileName) {
    boolean InputFlag = FALSE;
    boolean OutputFlag = FALSE;
    int i = 0;
    // iterate through all arguments to parse out I/O symbols
    while ((i + 1 < MAX_ARGS) && args[i] != NULL) {
        // check for I/O and remove as needed
        boolean argIsInput = (strcmp(args[i], "<") == 0);
        boolean argIsOutput = (strcmp(args[i], ">") == 0);
        if (argIsInput == TRUE || argIsOutput == TRUE) {
            // this is I/O, save the two operators
            char* redirectOperator = args[i];
            char* fileName = args[i + 1];
            if(fileName == NULL){
                printf("Error: No %s file provided.\n",
                    (argIsInput == TRUE) ? "input": "output" );
                fflush(stdout);
                return FALSE;
            } else {
                // a file name has been given!! Hooray.
                // set flags and stash filename
                if(argIsInput == TRUE) {
                    // store input filename
                    InputFlag = TRUE;
                    asprintf(inputFileName, "%s", fileName);
                } else if (argIsOutput == TRUE) {
                    // store output filename
                    OutputFlag = TRUE;
                    asprintf(outputFileName, "%s", fileName);
                }
                // checks if NEXT is NULL, so it resets last one in the loop :)
                // free both the redirection and filename
                free(redirectOperator);
                free(fileName);
                int j = i + 2;
                //shift array after removal of the I/O command
                while((j < MAX_ARGS) && args[j] != NULL){
                    args[j - 2] = args[j];
                    j++;
                }
                // set last two positions to NULL
                args[j - 1] = NULL;
                args[j - 2] = NULL;
                // shift i back one to account for the shift
                i--;
            }
        }
        i++;
    }
    // passed, GOOD INPUT!
    return TRUE;
}
```

FILE I/O REDIRECTION

```
/*************************************************************************
 * redirectOutput: This function redirects output to the specified file name
 * including handling if the output is to /dev/null.
 ************************************************************************/
boolean redirectOutput(char* outputFileName) {
    int outputFile = STDOUT_FILENO;
    if(outputFileName != NULL) {
        // opens output file if exists, creates one if not
        outputFile = open(outputFileName, O_RDWR | O_CREAT, 0777);
        if(outputFile == -1) {
            // error opening the file
            perror("Error opening output file");
            pidExitStatus = 1;
        }
        if(strcmp(outputFileName, "/dev/null") != 0) {
            if(chmod(outputFileName, S_IRWXU | S_IRWXG | S_IXOTH) == -1) {
                perror("Error with chmod of output file");
                exit(1);
            }
        }
        //send output file to stdout
        dup2(outputFile, STDOUT_FILENO);
        close(outputFile);
    }
}
```

Sarah Maas

expected graduation June 2018