Rose Lin, Eileen Feng
Shell Design Document

```
/****************************** Data Structure ****************************/

// node for a single job
struct jobnode {
    int index; //keeps track the the index of the current job
    pid_t pid;
    pid_t gpid;
    char* origin_input;
    int status; //initially set to -1, later set accoridng to enum status
    jobnode* next;
    jobnode* previous;
}jobnode;

//doubly-linked list of jobs
struct dlist {
    int size;
    jobnode* head;
    jobnode* tail;
}dlist;


/******************************** globals ***********************************/

enum status {background, foreground, suspended};
enum flags {fg_to_sus, sus_to_bg, bg_to_fg, fg_to_kill, exit_shell, start_bg};
// fg_to_sus: control-Z, sus_to_bg: 'bg', bg_to_fg: remove from background,
fg_to_kill: control-C, exit_shell: control-D; start_bg: launched as background

#define FALSE 0
#define TRUE 1

// semaphores;
sem_t* all;
sem_t* bglis;
sem_t* suslis;

// global variables
int stat_flag_c = -1; // set to 'fg_to_kill' when keyboard input 'Control-C'
int stat_flag_z = -1; //set to 'fg_to_sus' when keyboard input 'Control-Z'
int multi_jobs = FALSE; //used to check whether user input contains ';'
int launch_bg = FALSE; // used to check whether a new process is launched as
background
pid_t curr_fg_job; // keep track of the group pid of the current foreground job
int job_num = 0;

// job lists
dlsit* all_jobs;
dlist* bg_jobs;
dlist* sus_jobs;


/************************* Function Prototypes **************************/
```

```
// signal handlers

void* sig_z();
/* signal handler for  CTRL-Z,
*
 * set stat_flag_z = suspended
*
 * obtain and store the group pid of the current foreground job into ‘curr_fg_job’
*
 * also send the current foreground running process with a SIGTSTP signal
*/

void* sig_c();
/* signal handler for  CTRL-C
*
 * set stat_flag_c =   fg_to_kill
*
 * obtain and store the group pid of the current foreground job into ‘curr_fg_job’
*
 * also send a SIGINT signal to the current foreground process
*/


void init_sems ();
/* initializing all the semaphores */

void signal_handlers();
/* register signal handler for all keyboard inputs: CTRL-C, CTRL-D , CTRL-Z;
*
 * contains functions sig_c(), sig_z()
*/

void print_jobs();
/* prints out suspended and background jobs when user input ‘jobs’ command */

char* read_input();
/* return the input line */

int check_special_symbols(char* input);
/* check for special symbols    *
 * returns the number of jobs   */

char** parse_input (char* input);
/* parse the string 'input'                                              *
 * return an array of tokens parsed                                      *
 * should be modified to use the get_token structure talked in class    */

int execute_input(char* tokens, jobnode* job);
/* return 1 when shell should keep running, 0 otherwise


void update_list(pid_t gid, int flag);
/* replace the job corresponding to the group pid 'gid' according to the stat_flag
*
 * if 'flag == sus_to_bg': move the job with group pid 'gid' from 'sus_jobs' to
```

```
'bg_jobs'   *
 * if 'flag == bg_to_fg': remove this job from 'bg_jobs' and 'all_jobs'
*
 * use semaphores for each joblist to prevent race condition
*/


/******************** Functions for doubly linked list 'dlist' ******************/

dlist* new_dlist();
/* Initialize a new dlist with size = 0, head = NULL, tail = NULL; return a pointer
to the dlist */

jobnode* set_head(jobnode* new_head, dlist* dl);
/* Set the new head of a dlist */

jobnode* set_tail(jobnode* new_tail, dlist* dl);
/* set the current tail of a dlist */

void set_size(dlist* dl);
/* set the size of a dlist */

jobnode* get_job(dlist* dl, int index);
/* get the index^th child from dl, counting backwards */

void freedlist(dlist *dl);
/* free dlist */


/******************* Functions for 'jobnode' data structure ********************/

jobnode* new_job();
/* Return  the pointer to the node that is successfully added
*
 * Initialize the new job with origin_input = "", pid = -1; status = -1, next =
NULL; previous = NULL;  */

jobnode*  add_job(dlist* dl, jobnode* new_job);
/* add the new_job jobnode to the end of the dl
*
 * update tail and new_jobs's previous job
*
 * update size of the doubly linked list
*/

void set_job_input(jobnode* jobNode, char* input);
/* set the 'origin_input" field of 'jobNode' */

void set_job_status(jobnode* jobNode, int status);
/* set the 'status' field of jobNode */

void set_job_pid(jobnode* jobNode, pid_t pid);
/* set the 'pid' fied of jobNodes */

void set_job_next(jobnode* jobNode, jobnode* next);
/* set the 'next' field of jobNode */
```

```
void set_job_next(jobnode* jobNode, jobnode* previous);
/* set the ‘previous’ field of jobNode */

void remove_job(dlist* dl, jobnode* node);
/* remove the  node from the dl                 *
 * update tail                                  *
 * update size                                  */

jobnode* find_job(pid_t pid, dlist* dl);
/* find and returns a jobnode pointer to the job with id ‘pid’ */

void remove_job_pid(dlist* dl, jobnode* node);
/* remove the node from the dl according to its pid */

void remove_job_index(dlist* dl, int index);
/* remove the job from the dl according to its index */

void free_node(jobnode* head);
/* free jobnodes */


/******************** check_special_symbols function *******************/

int check_special_symbols(char* input) {
    int count_bg = 0; // used to keep count of the number '&'
    int count = 0; //used to keep count of the number of ';'
    // go through the whole string 'input'
    if (input contains ‘&’) {
        launch_bg = TRUE;
        count_bg ++;
        replace '&' with '\n'
    } else if (input contains ';') {
        count ++;
        replace ';' with '\n'
    }
    if(count_bg + count > 0) { // if do have multi-jobs
        multi_jobs = TRUE;
        return (count_bg + count + 1); // return the number of jobs
    } else {
        multi_jobs = FALSE;
        return 1;
    }
}


/********************** execute_input function ************************/

int execute_input(char** tokens, jobnode* job) {
    if(tokens contains nothing) {
        ctrl-D, print out a no input message
    } else if(tokens[0] == "jobs") {
        printf_jobs();
    } else if (tokens[0] == "fg") {
        jobnode* tofg;
        if(tokens[1] == NULL) {
            tofg = get_job(bglis, 1);

        } else {
            num = tokens[2] convert to integer;
            tofg = get_job(bglis, num);
        }
        jobgid = tofg.gpid; // get the job's group pid
        shell_pid = getpid(); // get the pid for the shell
        store the terminal attributes: tcgetattr();
        update_list(jobgid, bg_to_fg); // remove this job from background
        send 'SIGCONT' signal to 'tofg' process group with group pid 'jobgid';
        switch the 'tofg' job to foreground: tcgetpgrp();
        wait for the job 'tofg': waitpid()
        switch myshell back to foreground: tcsetpgrp() with shell_pid
        restore myshell's attributes: tcsetattr()
    } else if (tokens[0] == "bg") {
        jobnode* tobg;
        if(tokens[1] == NULL) {
            tobg = get_job(sus_jobs, 1);
        } else {
            num tokens[2] converts to integer;
            tobg = get_job(bg_jobs, num);
        }
        jobgid = tobg.gpid;
        send 'SIGCONT' signal to the 'tobg' process group (group pid jobgid);
        update_list(jobpid, sus_to_bg);
        wait for the job 'tofg' with status checking: waitpid() with flag 'WNOHANG'
and 'WUNTRACED'
    } else if (tokens[0] == "kill") {
        if(tokens[1] == "-9") {
            target_pid = tokens[2] convert to num
            target_job = find(target_pid, bg_jobs);
            if(target_job == NULL) { // if not a background job
                target_job = get_job(target_pid, sus_jobs);
            }
            target_gid = target_job.gpid;
            send 'SIGKILL' signal to target_gid
        } else {
            target_pid = tokens[2] convert to num
            target_job = find(target_pid, bg_jobs);
            if(target_job == NULL) { // if not a background job
                target_job = get_job(target_pid, sus_jobs);
                remove 'target_job' from 'sus_jobs'
            } else {
                remove 'target_job' from 'sus_jobs'
            }
            target_gid = target_job.gpid;
            send 'SIGTERM' signal to target_gid
        }
    } else if(launch_bg) {
        jobpid = fork() // fork a new process
        child:
            reset the group pid of the new process: jgid = setgpid()
            set the pid and gpid of the new job
            update_list(jgid, start_bg);
            execvp() // execute the job
        parent:
            wait for this job: waitpid() with flag 'WNOHANG' and 'WUNTRACED'
            when child terminates: update_list() remove child of 'bg_jobs'
    } else { // normal launching jobs
        jobpid = fork() // fork a new process
        child:
            reset the group pid for this new process: jpid = setgpid()
            execvp() // execute the job
        parent:
            wait for this job to terminates
    }

}



/*************************** Main Program *****************************/

int main (int argc, char** argv) {
    init_sems();
    signal_handlers();

    /* initialized all job lists */
    all_jobs = new_dlist();
    bg_jobs = new_dlist();
    sus_jobs = new_dlist();

    do {
    int run = 0;
    jobnode* njob = new_job();
    add_job(all_jobs, njob); // add the current job to the all_job list
    char* input = read_input();
    set_job_input(njob, input); // store the current input into the current jobnode
struct
    job_num = check_special_symbols(input);
    int loop = 0;
    while(loop < job_num) {
        // first loop to find the current index ‘ind’ of the first ‘\n’ index
        char** tokens = parse_input(input);
        set ‘input’ to the string starting from ‘ind’
        run = execute_input(tokens, njob);
        create a new job and let 'njob' points to the newly created job
        add_job(all_jobs, njob);
        set_job_input(njob, input);
    }
    free(input);
    free(tokens);
    } while(run);
    sem_unlink();
    sem_close();
    free_node(dlist->head);
    freedlist()
}
```