SPECIAL SHELL SCRIPT

Program works with while cycle. Previously, setup() method works for holding on the inputBuffer and seperating command line arguments.

```
while (1){

        background = 0;
        printf(" 333sh: ");
            fflush(stdout);
            pid_t childpid;
        /*setup() calls exit() when Control-D is entered */
        setup(inputBuffer, args, &background);
        .
        .
}
```

setup() method

The inputs entered by user are read by setup() method. This method have three parameters. These parameters:

1. inputBuffer (char array) → Inputs are written on it.
2. args (char pointer array) → Inputs are seperated and each args pointer shows seperated inputs.
3. background (int pointer) → If input has '&', background is 1.

Now, we look how to read and seperate the entered input.

What the user enters is recorded into inputBuffer array by the following line.

```
length = read(STDIN_FILENO,inputBuffer,MAX_LINE);
```

length is kept the entered input size.

Input is seperated according to space, tabs and enter characters. If each of them is met on inputBuffer, this character is null and the part that is from start character to this character is pointed to next args. Then, start character is the next character. If the compared character is '&' , the previous character is checked whether it is '>'. This character in null if the previous is not '>'. And also background is 1.

```
for (i=0;i<length;i++){ /* examine every character in the inputBuffer */


    switch (inputBuffer[i]){
        case ' ':
        case '\t' :              /* argument separators */
            if(start != -1){
           args[ct] = &inputBuffer[start];    /* set up pointer */
                ct++;
            }
        inputBuffer[i] = '\0'; /* add a null char; make a C string */
            //printf("%s\n",args[ct-1]);
            start = -1;
            break;


    case '\n':                 /* should be the final char examined */
            if (start != -1){
           args[ct] = &inputBuffer[start];
                ct++;
```

```
                }
            inputBuffer[i] = '\0';
                //printf("%s\n",args[ct-1]);
            args[ct] = NULL; /* no more arguments to this command */
                //printf("%s\n",args[ct]);
                break;


            default :            /* some other character */
                if (start == -1)
                   start = i;
            if (inputBuffer[i] == '&'){
                   *background  = 1;
                   //for ">&" redirection
                   if(inputBuffer[i-1]!='>'&&inputBuffer[i-2]!='\0')
                 inputBuffer[i] = '\0';
                }
        } /* end of switch */
    }    /* end of for */
    args[ct] = NULL; /* just in case the input line was > 80 */
```

end of setup() method

After that, seperate() method is continued with.

```
while (1){


                .
```

```
        .
        setup(inputBuffer, args, &background);
        char *cmd[10];
        int i=0;
        char* sep=seperate(args,cmd,&i,&background);
        .
        .
}
```

seperate() method

seperate() method has four parameters:

1. args (char pointer array) → command line arguments
2. cmd (char pointer array) → new pointer array for holding command line arguments without '&'
3. i (int pointer) → index for holding the placement of args pointer array.
4. background (int pointer) → If '&' is not found in args, background is 0.

seperate() method returns a char pointer. This char pointer is used while recording the background process for 'ps_all' command. This char pointer gets a command completely.

The command arguments are moved on cmd pointer array until args pointer arrives null and '&'. The aim of using the cmd pointer array for this method is the checking of whether there is any argument after '&' argument in the args pointer array. Also all commands are added to sep char pointer. These operations is made through while cycle.

```
char* seperate(char *args[],char *cmd[],int *i,int *background){
    int k=0;
    char *sep=malloc(sizeof(args[*i]));
    while(args[*i]!=NULL){
```

```
            if(strcmp(args[*i],"&")==0){
                cmd[k]=NULL;
                *i=*i+1;
                return sep;
            }
            cmd[k]=args[*i];
            strcat(sep,args[*i]);
            strcat(sep," ");
            k++;
            *i=*i+1;
        }
        *background=0;
        cmd[k]=NULL;
        return sep;
}
```

end of seperate() method

After seperate() method, firstly, last args pointer is checked NULL or not. The reason of that control is that seperate() method returns when '&' is seen by it so, the next argument is controlled in main() method. If the argument is not null, an error is returned the user as “wrong request”.

Secondly, the first argument of cmd is checked whether it is null or not. While comparing an argument with strcmp() method for other checks, if the argument is null, an error message returns and program exits. Therefore, this check should be.

After that, the checking for built-in commands starts. If we look these controls for these commands, the following conditions should be provided:

- exit, ps_all → next argument null
- kill, fg→ next argument is not null, but after that is null.

```
char* sep=seperate(args,cmd,&i,&background);
if(args[i]!=NULL){
        background=0;
        perror("wrong request");
}
else if(cmd[0]==NULL){
        continue;
}
else if(strcmp(cmd[0],"exit")==0&&cmd[1]==NULL){
        exitShell();
}
else if(strcmp(cmd[0],"kill")==0&&cmd[1]!=NULL&&cmd[2]==NULL){
        killProcess(cmd[1]);
}
else if(strcmp(cmd[0],"fg")==0&&cmd[1]!=NULL&&cmd[2]==NULL){
        foregroundFromBackground(cmd[1]);
}
else if(strcmp(cmd[0],"ps_all")==0&&cmd[1]==NULL){
        showProcess();
}
else if((childpid=fork())==0)
.
.
```

```
}
```

Before the executed methods for built-in commands, we shall examine the PROCESS structure.

This structure is composed to hold the background processes. It includes 4 variables:

- index → holds the job number in struct.
- value → holds the all command.
- pid → holds the process id.
- next → shows the process.

Two processes including first and last are initially defined for PROCESS structure. PROCESS first holds the initial process in structure and also last is the last one.

```
struct process{
        int index;
        char *value;
        pid_t pid;
        struct process *next;
};
typedef struct process PROCESS;
PROCESS *first=NULL;
PROCESS *last=NULL;
```

Now, we look the executed methods for built-in commands.

exitShell() method

exitShell() method is executed by exit command. This method does not get any argument.

exitShell() method checks whether there is any background process. These background processes is hold in a PROCESS structure. This check is made with waitpid() method in while cycle. pid of the controlled process is given to waitpid() method as parameter. If waitpid() method returns 0, program gives the warning “There are running processes.You need to terminate the running processes” and leaves from the method. Otherwise, continuing with the next process. When any background process remains, exit() method is run and the shell program is closed.

```
void exitShell(){
        PROCESS *temp;
        temp=first;
        while(temp!=NULL){
                if(waitpid(temp->pid,NULL,WNOHANG)==0){
                        printf("There are running processes.You need to terminate the running processes.\n");
                        return;
                }
                temp=temp->next;
        }
        exit(0);
}
```

end of exitShell() method

killProcess() method

killProcess() is executed by kill command. This method gets one parameter.

- cmd → char pointer

Second argument of the command is sent to this method. This argument should be '%number' or only number. Otherwise, the warning message as “wrong request” is returned.

The method firstly checks first character of the argument. If first character is '%', it is passed and other characters are converted an integer value by helping atoi() method. atoi() method returns 0 if these characters have any characters in external of the numbers. The returned value is checked whether it equals to 0 or not. If it is 0, the warning message is returned. Otherwise, this job number is researched in background processes by while cycle. When it is found in background processes, this process is terminated by sending process id of the process to kill() method. While running kill() method, if there is any error, the warning message "Failed to send the SIGUSR1 signal to child" is sent. If any such job number is not found in background processes, program continues with the last line in the method and “such process is not found” message is printed on the screen.

If first character is not '%', this argument is process id so, comparison with background processes is made by comparing this number with pid numbers of the processes. The same way in above is applied for others.

```
void killProcess(char *cmd){
        if(*cmd=='%'){
                cmd++;
                int ind=atoi(cmd);
                if(ind==0){
                        printf("wrong request\n");
                        return;
                }
                PROCESS *temp;
                temp=first;
                while(temp!=NULL){
                        if(temp->index==ind){
```

```
                        if(kill(temp->pid,SIGUSR1)==-1){
                                perror("Failed to send the SIGUSR1 signal to child");
                                return;
                        }
                        else{
                                return;
                        }
                }
                temp=temp->next;
        }
}
else{
        pid_t childpid=atoi(cmd);
        if(childpid==0){
                printf("wrong request\n");
                return;
        }
        PROCESS *temp;
        temp=first;
        while(temp!=NULL){
                if(temp->pid==childpid){
                        if(kill(temp->pid,SIGUSR1)==-1){
                                perror("Failed to send the SIGUSR1 signal to child");
                                return;
                        }
                        else{
                                return;
```

```
                        }
                    }
                    temp=temp->next;
                }
            }
            printf("such process is not found\n");
}
```

end of killProcess() method

foregroundFromBackground() method

This method gets one parameter. It is used by fg command.

- cmd → char pointer

This argument should start with '%' character. If it is not, the warning message is sent and exited from the method. Researching in the background processes is the same with first condition in the above method. But there are two PROCESS parameters. temp is kept the current comparison process and prev is previous process. This helps us while clearing the process.

If process is found in the background processes, main process is waited with waitpid() method. During this time waitpid() method returns 0 and is cycled in while. If process is terminated, it returns -1 and program leaves from while cycle. After that, it passes the process of clearing from background processes. If prev and temp equals to each others, first PROCESS equals to the next of prev process and temp process is freed. Otherwise, the next of temp equals to the next of prev and temp is freed. Then, exited from the method.

```
void foregroundFromBackground(char *cmd){
        if(*cmd=='%'){
```

```
cmd++;
int ind=atoi(cmd);
if(ind==0){
        printf("wrong request\n");
        return;
}
PROCESS *temp,*prev;
temp=first;
prev=temp;
while(temp!=NULL){
        if(temp->index==ind){
                pid_t childpid;
                childpid = waitpid(temp->pid, NULL, WNOHANG);
                while(childpid!=-1){
                        childpid = waitpid(temp->pid, NULL, WNOHANG);
                }
                if(prev==temp){
                        first=prev->next;
                        if(first==NULL)
                                last=NULL;
                        free(temp);
                }
                else{
                        prev->next=temp->next;
                        if(prev->next==NULL)
                                last=prev;
                        free(temp);
```

```
                        }
                        return;
                    }
                    prev=temp;
                    temp=temp->next;
                }

        }
        printf("wrong request\n");
        return;
}
```

end of foregroundFromBackground() method

showProcess() method

This method is used by ps_all. ps_all command shows the background processes in two parts including “Running” and “Terminated”. The method does not get any parameter.

Firstly running part is shown on the screen. The running processes is checked waitpid() method in while cycle. If waitpid() method returns 0, this process is printed with printf() on screen.

Secondly terminated part is printed on the screen. If waitpid() method returns -1, the process is printed and the process of the clearing it from background processes is passed. The clearing process is the same with the above method but, in here prev and temp equals to each others and continue for next background process after the process is freed.

```
void showProcess(){
        PROCESS *temp;
```

```
temp=first;
printf("\tRunning:\n\n");
while(temp!=NULL){
        if(waitpid(temp->pid,NULL,WNOHANG)==0)
                printf("\t\t[%d] %s(%ld)\n",temp->index,temp->value,(long)temp->pid);
        temp=temp->next;
}
printf("\tTerminated:\n\n");
temp=first;
PROCESS *prev;
prev=first;
while(temp!=NULL){
        if(waitpid(temp->pid,NULL,WNOHANG)<0){
                printf("\t\t[%d] %s(%ld)\n",temp->index,temp->value,(long)temp->pid);
                if(prev==temp){
                        first=prev->next;
                        if(first==NULL)
                                last=NULL;
                        free(temp);
                        temp=first;
                        prev=temp;
                        continue;
                }
                else{
                        prev->next=temp->next;
                        if(prev->next==NULL)
                                last=prev;
```

```
                    free(temp);
                    temp=prev;
                }
            }
            prev=temp;
            temp=temp->next;

        }
}
```

end of showProcess() method

If command does not match with built-in commands, a child process is created with fork() method for system commands. Child process continues within if condition. It firstly runs seperateForRedirection() method. This method finds the file names and gets redirection command if there is redirection command in command. std variable represents redirection command.

If std command is not null, redirection() method is run. This method creates redirections and files -if it needs – according to redirection commands.

After that, addForPipe() method runs the command as standard or by using pipe if command has pipe argument.

```
while(1){
                .
                .
                else if((childpid=fork())==0){
                    char* fileName[2];
                    char* std=seperateForRedirection(cmd,fileName);
```

```
                    if(std!=NULL){
                        if(cmd[0]==NULL){
                            perror("wrong request");
                            exit(-1);
                        }
                        redirection(std,fileName);

                    }
                    addForPipe(cmd);


                }
                .
                .
}
```

seperateForRedirection()method

This method gets two parameters. Command arguments are separated file names for redirection and redirection command is returned to use by this method.

- cmd → (char pointer array) it holds all commands.
- fileName → (char pointer array) file names are pointed to this variable by the method

This method investigates for redirection commands by while cycle. If any of “>”, “>>” and “>&” redirection is found in command, next argument and the second next argument is looked. If first next is not null and second next is null, first next argument is pointed to fileName as file name and also redirection command is returned. Otherwise, if this condition is not provided, redirection command is null and fileName equals to null.

If redirection command is “<”, two conditions are valid. These are “command < filename” and “command < filename > filename2”. Therefore, if the next argument after '<' redirection command is not null, it is pointed as a file name. Then, second next is checked whether null or not. If it is not null and “>” command, the next argument after from it is pointed as second file name. Otherwise, second file name is null.

```
char *seperateForRedirection(char *cmd[],char* fileName[]){
        char *sep;
        int i=0;
        while(cmd[i]!=NULL){
                //one condition for >, >> and >&
                if(strcmp(cmd[i],">")==0||strcmp(cmd[i],">>")==0||strcmp(cmd[i],">&")==0){
                        if(cmd[i+1]!=NULL&&cmd[i+2]==NULL){
                                fileName[0]=cmd[i+1];
                                fileName[1]=NULL;
                                sep=malloc(sizeof cmd[i]);
                                strcpy(sep,cmd[i]);
                                cmd[i]=NULL;
                                return sep;
                        }
                        fileName[0]=NULL;
                        fileName[1]=NULL;
                        return NULL;
                }
                //two condition for <, either of "< filename" or "< filename > filename2"
                else if(strcmp(cmd[i],"<")==0&&cmd[i+1]!=NULL){
                        fileName[0]=cmd[i+1];
                        sep=malloc(sizeof cmd[i]);
```

```
                    strcpy(sep,cmd[i]);
                    cmd[i]=NULL;
                    if(cmd[i+2]==NULL){
                            fileName[1]=NULL;
                            return sep;
                    }
                    else if(strcmp(cmd[i+2],">")==0&&cmd[i+3]!=NULL){
                            fileName[1]=cmd[i+3];
                            if(cmd[i+4]==NULL){
                                    return sep;
                            }
                    }
                    fileName[1]=NULL;
                    return NULL;
            }
            i++;
        }
        fileName[0]=NULL;
        fileName[1]=NULL;
        return NULL;
}
```

end of seperateForRedirection()method

redirection() method

This method gets two arguments.

- std → (char pointer) it holds redirection command.
- fileName → (char pointer array) it holds file names.

“>” and “>>” redirection commands has the same redirection. Both of them prints the output to file. Therefore, they are in same category. It is only one difference that “>” command truncates and other appends while printing to the file. The comparison is made for that difference in the category.

```
void redirection(char *std,char *fileName[]){
    if(strcmp(std,">")==0||strcmp(std,">>")==0){
        int fd;
        if(strcmp(std,">")==0)
            fd = open(fileName[0], CREATE_FLAGS, CREATE_MODE);
        else
            fd = open(fileName[0], CREATE_FLAGS2, CREATE_MODE);
        if (fd == -1) {
            perror("Failed to open");
            exit(0);
        }
        if (dup2(fd, STDOUT_FILENO) == -1) {
            perror("Failed to redirect standard output");
            exit(0);
        }
        if (close(fd) == -1) {
            perror("Failed to close the file");
            exit(0);
        }
    }
```

```
else if(strcmp(std,">&")==0)
.
.
```

">&" command prints the errors to the file so, error output is redirected to the file.

```
.
.
else if(strcmp(std,">&")==0){
        int fd;
        fd = open(fileName[0], CREATE_FLAGS3, CREATE_MODE);
        if (fd == -1) {
                perror("Failed to open");
                exit(0);
        }
        if (dup2(fd, STDERR_FILENO) == -1) {
                perror("Failed to redirect standard error");
                exit(0);
        }
        if (close(fd) == -1) {
                perror("Failed to close the file");
                exit(0);
        }
}
else if(strcmp(std,"<")==0)
.
.
```

If redirection command is “<”, there are two probabilities so, firstly a file is opened and input direction is redirected from the file and then the method checks whether there is second file name. If second file name is not null, output is redirected to the same file or a different file for “>” redirection command.

If there is any error while making these redirections, a warning messsage is printed on the screen and process is stopped.

```
.
.
else if(strcmp(std,"<")==0){
        int fd;
        int fd2;
        fd = open(fileName[0], CREATE_FLAGS4, CREATE_MODE);
        if (fd == -1) {
                perror("Failed to open");
                exit(0);
        }
        if (dup2(fd, STDIN_FILENO) == -1) {
                perror("Failed to redirect standard error");
                exit(0);
        }
        if (close(fd) == -1) {
                perror("Failed to close the file");
                exit(0);
        }
        if(fileName[1]!=NULL){
                fd2 = open(fileName[1], CREATE_FLAGS3, CREATE_MODE);
```

```
                    if (fd2 == -1) {
                        perror("Failed to open");
                        exit(0);
                    }
                    if (dup2(fd2, STDOUT_FILENO) == -1) {
                        perror("Failed to redirect standard error");
                        exit(0);
                    }
                    if (close(fd2) == -1) {
                        perror("Failed to close the file");
                        exit(0);
                    }
                }
            }
        }
```

end of redirection() method

addForRedirection() method

This method gets one parameter.

- cmd → (char pointer array) it holds command arguments.

The method firstly looks pipe arguments in command and this argument is null. Also the next argument index after from pipe argument is hold an integer array(ptr) for using later. If command starts with pipe argument or more than one pipe argument is consecutive, the warning message - “unexpected token” **-** is returned and exited from process. First and last elements of the ptr is -1. These show ending of the pipe argument for the next operation.

```
void addForPipe(char *cmd[]){

        char *value[10];
        int i=0;
        int k=0;
        int *ptr=malloc(15*sizeof (int*));
        ptr[0]=-1;
        ptr[1]=0;
        int a=1;
        while(cmd[i]!=NULL){
                if(strcmp(cmd[i],"|")==0){
                        cmd[i]=NULL;
                        k=0;
                        if(i==0||i==ptr[a]){
                                perror("unexpected token");
                                exit(-1);
                        }
                        a++;
                        i++;
                        ptr[a]=i;
                        continue;
                }
                value[k]=cmd[i];

                i++;
                k++;
```

```
}
ptr[a+1]=-1;
.
.
```

It starts with the last command for ending with the last command in pipe operations. Argument index takes from ptr array. If command line has pipe argument for previous command, an extra memory is opened with pipe() for communicating between commands. And also created a child process. The current process continues as parent process and other command is child process. Child process redirects the output to the pipe memory and parent process redirects input from the pipe memory. Parent process runs the current command with execv() method after the redirection. Child looks whether there is any command before it. If it is, pipe() and fork() operations repeats. Otherwise, the child command is run.

```
.
.
const char *path;
while(ptr[a-1]!=-1){
        pid_t childpid;
        int fd[2];
        if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
                perror("Failed to setup pipeline");
                exit(-1);
        }
        if (childpid == 0) {
                if (dup2(fd[1], STDOUT_FILENO) == -1)
                        perror("Failed to redirect stdout");
                else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
                        perror("Failed to close extra pipe descriptors");
                else {
```

```
                a--;
        }
}
else{
        if (dup2(fd[0], STDIN_FILENO) == -1)
                perror("Failed to redirect stdin");
        else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
                perror("Failed to close extra pipe file descriptors");
        else {
                char *tmp[10];
                int k=ptr[a];
                int i=0;
                if(cmd[k]==NULL){
                        perror("unexpected");
                        exit(-1);
                }
                while(cmd[k]!=NULL){
                        tmp[i]=cmd[k];
                        k++;
                        i++;
                }
                tmp[i]=NULL;
                if((path=execute(tmp[0]))!=NULL){
                        //printf("girdi0\n");
                        execv(path,tmp);
                        exit(-1);
                }
```

```
                                else{
                                        perror("not found");
                                        exit(0);
                                }
                        }
                }
        }
        if((path=execute(cmd[0]))!=NULL){
                //printf("girdi0\n");
                execv(path,cmd);
                exit(-1);
        }
        else{
                perror("not found");
                exit(0);
        }
}
```

First argument of these commands is investigated by execute() method before running with execv() method. If execute() returns null, the warning message - “not found” - is returned.

execute() method

This method gets one argument and returns the path of the command.

- args → (char pointer) first argument of command.

Firstly, paths includes system commands are taken getenv() method. These paths are separated by strtok() method. The command hold by args variable is inserted to the end of the paths and each added path is checked whether it can be run. If runnable path is found, this path is returned. Otherwise, if any runnable is not found, the method returns null.

```
const char *execute(char *args){
        //must be restricted with null
        char *path=getenv("PATH")+'\0';
        strtok(path,":");
        while(path){
                char *path2=(char *)malloc(sizeof path);
                strcpy(path2,path);
                strcat(path2,"/");
                strcat(path2,args);
                const char *path3=path2;
                if(access(path3,F_OK)==0)
                        return path2;
                path=strtok(NULL,":");

        }
return NULL;
}
```

end execute() method

After that, this operations is generated by parent process. If child is not created because of any error, the warning message - “signal is failed”- is returned. Child process is waited by wait() method until it is terminated if the process is foreground. Otherwise, the process is recorded as background process in PROCESS structure by addProcess() method.

```
                .
                .
                else if(childpid==-1){
                        perror("signal is failed\n");
                }
                else if(background==0){
                        while(wait(NULL)!=childpid);
                }
                else{
                        addProcess(sep,childpid,&ind);
                }
        }
}
```

addProcess() method

This process gets three arguments.

- cmd → (char pointer array) it holds the all command.
- childpid → (pid_t) it holds process id.
- ind → (int pointer) it holds the next job number.

A new PROCESS variable(newProc) is defined and a memory space is allocated for it. It is filled with sent parameter. If PROCESS first is null, first and last equals to newProc and also job number which is hold in pointer ind is reseted. Otherwise, after the next of PROCESS last is newProc, last equals to the next of it. At the end of the method job number is increased by 1 for the next process.

```
void addProcess(char *cmd,pid_t childpid,int *ind){
        PROCESS *newProc=malloc(sizeof(PROCESS));
        newProc->index=*ind;
        newProc->value=malloc(sizeof(cmd));
        strcpy(newProc->value,cmd);
        newProc->pid=childpid;
        newProc->next=NULL;
        if (first==NULL){
                *ind=1;
                newProc->index=*ind;
                first=newProc;
                last=newProc;
        }
        else{
                last->next=newProc;
                last=last->next;
        }
        *ind=*ind+1;
}
```

end of addProcess() method

END