1 README

1.1 Usage

```
./mysh
./mysh inputfile
```

1.2 Data Structure

- JOB JOB stores every command line and its relating information.

```
struct JOB
{
char *command;
//command Line
char **argv;
//parameters
char *inputFile;
char *outputFile;
int argNum;
//number of parameters
pid\_t jobid;
//process id
};
```

- COMMAND_EXE

```
struct COMMAND\_EXE
{
int mode;
//BACKGROUND or FOREGROUND or PIPELINE
struct JOB *task[MAX\_PIPELINE];
//maybe pipelined, separate it to several parts
int taskNum ;
//numofpipeline + 1
pid\_t cmdid;
//process id
};
```

- SHELLINFO

```
struct SHELLINFO
{
        char wd[PATH\_SIZE];
        //current working directory
```

```
        int   jobs;
        //exclude command "wait"
        struct COMMAND\_EXE* tsk[MAX\_COMMAND];
        //all the JOBS
}shell;
```

1.3 Function

- char *ReadLine
  ReadLine read a line or to the end of file, as an extend of fgets(). For the memory allocation reason(we don't know how big the memory block should malloc beforehand), I malloc and fgets, and realloc and fgets again and again until the space satisfy the request. Also, in some annoying test script which lacks of a endline in the end of file, the function can add a '\n' to it.

- struct COMMAND_EXE* Split_Line_To_Segment
  separate the command line by '\n'. What's more, deal with '\",'\"'(Extract the contents from every couple of quotations)

- int Split_Line
  And I begin to store every option, every argv and command name into argv[][]. For the redirection requirement, the function also points out what input file is and output file is.(if no specification, stdin and stdout as default)

- void Csh_Execute
  Main execution function. If the command is built in, execute it by ourselves. If not, call execvp(). There are some differences in FOREGROUND and BACKGROUND mode.

  - FOREGROUND call block waitpid
  - BACKGROUND call non-block waitpid

- void Csh_Exe_Command_Recursive(PipeLine Accomplish)
  I haven't figured out a way to combine pipeline mode with normal mode(some principles unclear, like the redirection and pipeline's priority), so I write a simple recursive function to realize the pipeline. Here's general method: except the leftmost and rightmost of pipeline, every sub-command's input is from pipe(derived from its parent process)'s read-end, and output is passing through a new pipe. Just paste the core code.

```
void Csh\_Exe\_Command\_Recursive(int i,
struct COMMAND\_EXE* cmd,int outfd[2])
{
        int infd[2] ;
        int io[2];
```

```
        io[0] = STDIN\_FILENO;
        io[1] = STDOUT\_FILENO;
        if(i!=0)
                pipe(infd);
        if(i!=0)
        {
                pid\_t child\_id = fork();
                if(child\_id == 0)
                {
                Csh\_Exe\_Command\_Recursive(i-1,cmd,infd);
                }
                else
                {
                wait(0);
                }
                }
                if(i==0)
                {
                close(outfd[0]);
                dup2(outfd[1],STDOUT\_FILENO);
                io[1] = outfd[1];
                }
                else if(i==(cmd->taskNum-1))
                {
                close(infd[1]);
                dup2(infd[0],STDIN\_FILENO);
                io[0] = infd[0];
                }
                else
                {
                close(infd[1]);
                close(outfd[0]);
                dup2(outfd[1],STDOUT\_FILENO);
                dup2(infd[0],STDIN\_FILENO);
                io[0] = infd[0];
                io[1] = outfd[1];
                }
}
```

- Release_Zombie Initially, I thought it should be a skillful task, and I was track off into a relatively more complicated way to solve it. First, I used the share memory to achieve processes' communication. Then I used signal(SIGCHILD,ZombieClr) to awake the ZombieClr()(which function used to release Zombies). What's more, to avoid conflicts between different processes, I used mutex to realize mutual exclusive. But then I found that I

zigzagged...After that, I just use waitpid whose option is set to WNOANG to all the processes every period of time. And it does work better than the former version...