Prologue

I'll have a quick look at XV6 shell.c to understand the implementation of its features, such as running command, redirection, pipe, list...

You can find the source code in xv6's repository. (https://github.com/mit-pdos/xv6-public/blob/master/sh.c)

Main Entance

- main
  - chdir (command `cd`)
  - Parse & Run Command

Structions

We have 6 structions and 5 kinds of command types, including `execcmd`, `redircmd`, `pipcmd`, `listcmd`, and `backcmd`, and they have a common base "class/struct" -- `cmd`.

```
struct cmd {
  int type;
};

struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};

struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};

struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
```

```
};

struct backcmd {
  int type;
  struct cmd *cmd;
};
```

It's meaning less to analysis them now, you can go back to check the struct when analysising the parsing part.

Primitive Functions

peek

Source:

```
int
peek(char **ps, char *es, char *toks)
{
  char *s;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return *s && strchr(toks, *s);
}
```

This simple function has 9 references. `ps` is a pointer which points to a input start pointer: `ps->string_start_pointer->address_of_string_start`. And `es` points to the end of the input string while `toks` is a sting.

The first `while` will jump over the the whitespaces at the start of the input string. And store the new input string start in the pointer `ps`.

Then, it would return if the first character of pruned input string in the `toks` or not.

In a word, this function prunes the input string and check if the first character in `toks`.

In shell.c, this function is used to prune the comamnd and check if the/(a part of) command starts with "(&;| <>)".

gettoken

Source:

```
gettoken(char **ps, char *es, char **q, char **eq)
{
  char *s;
```

```
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  if(q)
    *q = s;
  ret = *s;
  switch(*s){
    case 0:
      break;
    case '|':
    case '(':
    case ')':
    case ';':
    case '&':
    case '<':
      s++;
      break;
    case '>':
      s++;
      if(*s == '>'){
        ret = '+';
        s++;
      }
      break;
    default:
      ret = 'a';
      while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
        s++;
      break;
  }
  if(eq)
    *eq = s;

  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return ret;
}
```

This function is more complex than the previous one.

- `ps` is a pointer which points to a input string start pointer
- `es` points to the end of input string
- `q` is a pointer, it would store the start of splited command.
- `eq` is a pointer, it would store the end of splited command.

This function would:

First, jump over the whitespaces at the start of input

Sencond, if the input string's first character is a symbol ("|();&<>"), it would return the symbol after prune the rest part of input. If not, it would set return value as `a` and jump over the no-symbol characters.
Btw, it could also recognize ">>" and set the return value to '+'.

In a word, this function would parse the command

return the type of the first part of the command. Besides, the parsed part would be stored in `q` and `eq`.

The shell.c always combines `peek` function and `gettoken` function. we gonna analysis them later.

parsecmd

```
struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    printf(2, "leftovers: %s\n", s);
    panic("syntax");
  }
  nulterminate(cmd);
  return cmd;
}
```

It's just a wrapper and it would call `parseline` to do the parsing work.

parseline

```
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

This is the entry of parses, you can just leave it and go back to this function after analysing all the specific parsers.

Misc Functions

nulterminate

This function is not important in my view. It would set the end ponters to 0.

getcmd

read the input from the users.

panic

leave a message and exit

fork1

panic fork

Parsing & Execution

Before analyzing specific parser, we'd better move to some special cases, such as running a simple command, a command with redirection, a command with pipe... That would help you to have a high level understanding of the the shell.

Execute Simple Commands

Let's start from some simple commands, such as "/bin/ls /tmp".

For these simple commands without special symbols, the shell uses parseexec to parse it and use runcmd to execute it. We'll start with the parser.

```
struct cmd*
parseexec(char **ps, char *es)
{
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  if(peek(ps, es, "("))
    return parseblock(ps, es);

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
```

```
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != 'a')
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}
```

parseexec is the most common parser. It take input between `*ps` and `es`.

It would check if we have a block in the command, if yes, it would return block's `cmd` and it would also check the redirection. We would talk about thses more complex cases later.

For simple commands, the shell would go into the while loop. The shell uses `gettoken` to get the command pieces. Because " " is one of whitespaces so "/bin/ls /tmp" would be splited to "/bin/sh" and "/tmp". Meanwhile, these pieces's starts would be stored in `cmd->argv` and the ends would be stored in `cmd->eargve` so that we could use "execvp" in `runcmd` to execute the command. Btw, we would also check the redirection before return. The redirection symbol would show after the command because of the redirection syntax.

```
runcmd(struct cmd *cmd)
{
...
  switch(cmd->type){
...
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit();
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
```

We just go through the procedure of parsing/running a simple command!

Execute Commands with Redirection

This case is just a little more complex than the simple commands, the only difference is we gonna use a specific file as our input/output rather than the stdin/stdout.

We can just replace the original input/output with the redirection file. In `Execute Simple Commands`, we know that the redirection would be check after parsing mornal command pieces. We don't need to parse the command again. We can just replace the stdin/stdou.

`parseredirs` is the parser of commands with redirection.

```
struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;

  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);
    if(gettoken(ps, es, &q, &eq) != 'a')
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    case '+':  // >>
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    }
  }
  return cmd;
}
```

If the function finds redirection symbols "<>" at the start of the command, the parameter `cmd` would be converted to a `redircmd`. And the parser would build the `redircmd` agaist the redirection symbol. We have three types of redirection, including ">", "<", ">>". According to the redirection symbols, we open the redirection file with corresponding permissions.

```
struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};
struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
```

```
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = REDIR;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->efile = efile;
  cmd->mode = mode;
  cmd->fd = fd;
  return (struct cmd*)cmd;
}
```

The `redircmd` strcuture would store the `cmd` as an element. The type would alway be REDIR. Besides, the redirection file's information would be stored in `file` and `efile`. The `fd` element means the file descriptor would be replaced and the `mode` element store the open-permission infromation.

Now, we get all the necessnrary information and we gonna run the command in `runcmd`.

```
runcmd(struct cmd *cmd)
{
    ...
    case REDIR:
        rcmd = (struct redircmd*)cmd;
        close(rcmd->fd);
        if(open(rcmd->file, rcmd->mode) < 0){
        printf(2, "open %s failed\n", rcmd->file);
        exit();
        }
        runcmd(rcmd->cmd);
        break;
    ...
}
```

In this function, we gonna convert the type of `cmd` so that we can use the information we stored earlier. Then, we close the old input/output fd and open the redirection file to replace the file describtor. For example, if we close "stdin", the `fd` 0 would be released and the new file would use 0 as its `fd`. At the end, we call `runcmd` to run our sub command. The command would run as a redirection command because we have replaced the input/output.

Execute Commands with PIPE

Pipe is a little more complex than the redirection. The command is splited to two parts by the pipe symbol and we use a pipe to connect them. I think there are three steps.

1. Creat the pipe
2. Fork two process to run two parts of command
3. Replace the stdout of left part with the pipe-in.
4. Replace the stdin of right part with the pipe-out
5. run the commands

For the parser, we need to split the command with pipe symbol. The parser is elegant.

```
struct cmd*
parsepipe(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}
```

It use `parseexec` to get the left part and jump over the pipe symbol. The it call parsepipe to parse it's right part. I have a question at the first time I saw this line. I was confused why don't it use parseexec to parse the right part. The answer is it's different from the redirection. We only have one redirection symbol in one command. But we don't know how many pipes symbols would be in the command. Using parsepipe is very elegant!

The function `pipecmd` would use the information got from parser to fills the element in `strcut pipecmd`.

```
struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};
struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

It fills the left and right of the pipe-cmd with the parsing results. Assuming we have a command "ls | grep c* | wc", the shell would parse it as following procedure:

1. Get execcmd("ls") from parseexec
2. Get execcmd("grep c*") from parseexec
3. Get execcmd("wc") from parseexec
4. 2 and 3 would be used to built a pipcmd
5. 1 and 4 would also be used to build a pipcmd
6. return 5

Awesome! Now we can run them one by one in `runcmd`.

```
runcmd(struct cmd *cmd)
{
    ...
case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait();
    wait();
    break;
    ...
}
```

It would convert the type of `cmd` and create a pair `pipe`. Just as we said, it would create two kids to run left part and right part. The parent process would do nothing but exit after waiting two kids's termination while the child processes, use dup to replace its original stdin/stdout. The part is also amazing it build a pipe between two processes. The "close+dup" pair is amazing, too! The dup would take the smallest fd and close would release the file describtor.

Let's review the previous example "ls | grep c* | wc". We have (1,4) as at the end of parsing. Now, in `runcmd` it would run 1 first because 4 is waiting for the input of 1. After(not exactly but that doesn't matter) getting the output of 1, runcmd would split 4 to (2,3) so that 1's out put would be the input of 2! Now we can see the command run one by one!

Execute a List of Commands

Actually list of commands is easier than pipe. And we just need to split the commands to several sub-commands and we don't need to deal with the input/output.

The entry of hander is in `parseline`

```
struct cmd*
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

After parsing the first part of command, this function would check if we have a ";". If yes, we gonna creat a lists. As we know, the order is important, the second command would only be executed after the first one. In this part, we reuse the "parseline". It is like building a linked list. Every node would include a command and a `next` pointer. We can see it clearly by combining constrcuting part and the parsing part.

```
struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
  struct listcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = LIST;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

So in `runcmd` we can iterate the linked list and run them in order!

```
void
runcmd(struct cmd *cmd)
{
    ...
  case LIST:
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait();
    runcmd(lcmd->right);
    break;
```

```
    ...
}
```

The `panic fork` and the `wait` are very important. `panic fork` could make sure the child process would terminate if current command would have some error while `wait` could make sure the commands would be executed in correct order! And because we use fork, our parent process would continue to run the second command even if the child was terminated.

Execute the Commands with &

`backcmd` is similar to the previous one, a list of commands. The differences are

1. `backcmd` should be seen as one command
2. The second command would be executed only when the first one return normally

The entry of dealer is also in `parseline` and because it's only a symbol connecting different parts of one command, the order is important. We should run `backcmd` earlier than `listcmd`.

```
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

As the code shown above, we gonna deal with `backcmd` earlier than `listcmd`. Also we can parse the first command normally, and check if we need to parse the `backcmd`. For example, we have a command "echo 1 & ls".

```
struct cmd*
backcmd(struct cmd *subcmd)
{
  struct backcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = BACK;
  cmd->cmd = subcmd;
```

```
  return (struct cmd*)cmd;
}
```

The parser is simple. It uses the linked list structure like the `listcmd`.

```
void
runcmd(struct cmd *cmd)
{
...
  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
...
}
```

The runner part is also elegant. The only different is it would use one ...

Oh, waite a sec, would it work? I don't think the second part would be executed!

I test on xv6 terminal and find it doesn't work... Okay, I think I know why its name is back rather than and. It means backend. So the key is keep it back end. We can just run it in the child process without `wait`.

It's not eleagnt, it's trite.

Epilogue

This shell support `cd` and 5 types of commands, including normal, list, backend, pipe, redirection.

For `cd`, we gonna use chdir to complete it simply. But unluckily, we can't combine `cd` with other symbols, such as "cd .. ; ls".

For other types of commands, we have corresponding parser and runner. We analysised them in previous parts. And they are related, if you dive deeper about the constrcution of parsing, you will find this elegant shell is amazing.

All in one, we went through the shell.c. And we find it's not as powerful as our bash/zsh. However, we know the constrcution of a basic shell and know the way to parse and run different types of commands. That's amazing to read these awesome codes, I feel like I talked to a great programmer, that's really precious for me.