Changes in shell code:

1. Add two new structures (sequencecmd and parallalcmd) to present ";" operation command and "&" operation command. As the sequence operation should run both sides of commands, it has two cmd attributes: *left and *right.

2. Complete the cases in runcmd function. For case ' ', just execute the command in argv[]. For case '>' or '<', just will open the file in highest right by corresponding mode. For case '|', create a pipe and use the pipe input and output to transmit data between parent process and child process, the parent process should wait the child process at last to guarantee the transmission is finished before next command.

3. Add required operator ";" and "&" in runcmd function. For ";" case, create a child process by fork function and run left command in child process. The parent process will wait child process (by wait function), so the left command will finished before the right command executes. For "&" case, the command will run in child process and the parent process will not wait child, so the parent can go on to receive other input command.

4. Add the functions to set up sequencecmd and parallelcmd as following. Their aims are just allocating initialized space and setting corresponding type and commands.
struct cmd sequencecmd(struct cmd *left, struct cmd *right)*
struct cmd parallelcmd(struct cmd *subcmd)*

5. Add ";" "&" symbols in array symbols[], so the gettoken function will be aware of these operators. Also add the ';' and '&' cases in gettoken function to make sure the pointer s will pass these operator letters.

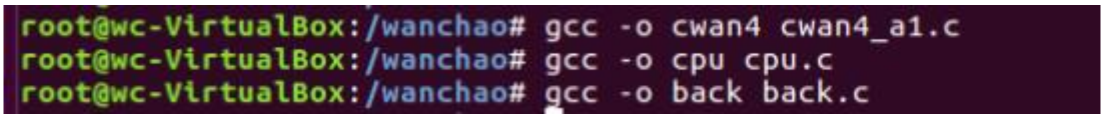
6. Add code in parseline function to distinguish "&" and ";" operators. First check whether it's a parallel command and then whether it's sequence command because the user will also type operator in this order ("&" should in front of ";" if they both be typed in, because the ";" should be the boundary of two commands). And other operator " ", "<>", "|" should in front of them by same reason. Also due to the function name "parseline", I should put "&" and ";" there but not in parsepipe function to present the right meaning. The code I add will simply check the character whether is "&" or ";". If it is, build the corresponding structure to save the command.

7. In parseexec function, I add "&;" in while loop's condition to make sure the loop will stop when discovers them and go back to correspond function.

Test data:

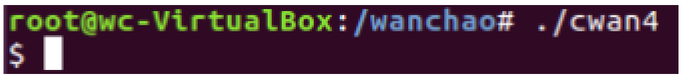
I use the ubuntu operating system to run the shell code. To show the parallel result clearly, I also copy and modify the cpu.c code from text book as auxiliary. It will display the string five times in five seconds. Also I create an infinite loop in back.c code.

Firstly, compile these code to be executive with the same name of c code



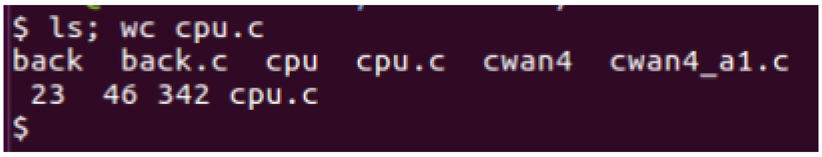
```
root@wc-VirtualBox:/wanchao# gcc -o cwan4 cwan4_a1.c
root@wc-VirtualBox:/wanchao# gcc -o cpu cpu.c
root@wc-VirtualBox:/wanchao# gcc -o back back.c
```

For example "gcc –o cwan4 cwan4_a1.c" and "gcc –o cpu cpu.c".
Then run "./cwan4" to execute the shell program.



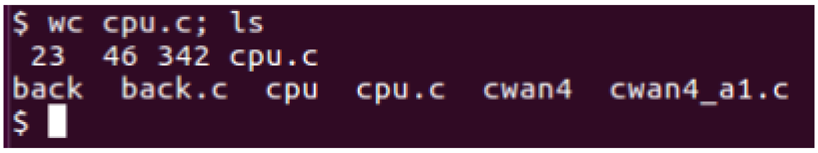
```
root@wc-VirtualBox:/wanchao# ./cwan4
$ 
```

Run "ls; wc cpu.c" in sequence, we will see the ls command run first and then wc command runs, which shows that the ";" realize the sequence function.



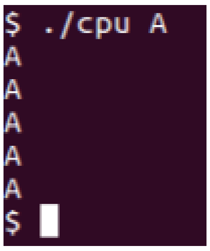
```
$ ls; wc cpu.c
back  back.c  cpu  cpu.c  cwan4  cwan4_a1.c
 23  46 342 cpu.c
$
```

Then try to reverse the order, run "wc cpu.c; ls". The result shows the execution order is also reversal.

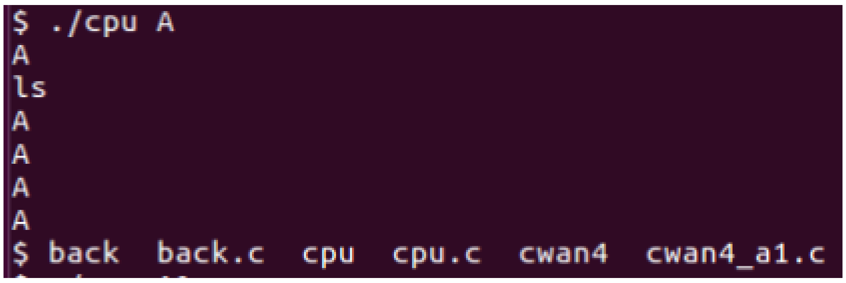


```
$ wc cpu.c; ls
 23  46 342 cpu.c
back  back.c  cpu  cpu.c  cwan4  cwan4_a1.c
$ 
```

Next let's try parallel operator, run "./cpu A". We will see the "$" notion will be display after the code finished, which reflects the shell will wait the "./cpu A" process. And if we execute other command during when "./cpu A" process is running. We cannot see the result instantly but after the "./cpu A" process finish.

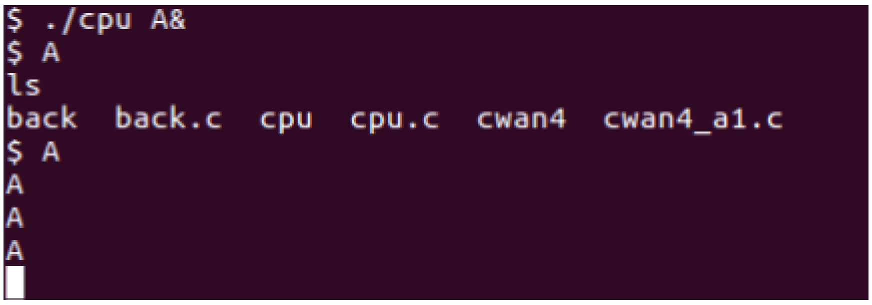


```
$ ./cpu A
A
A
A
A
A
$ 
```



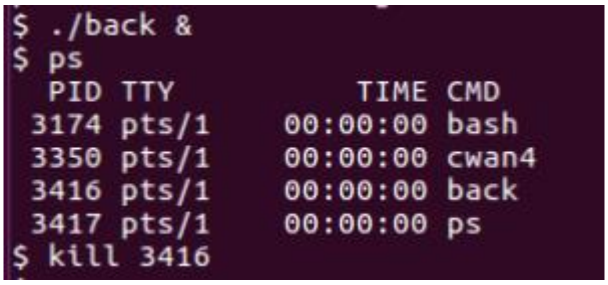
```
$ ./cpu A
A
ls
A
A
A
A
$ back  back.c  cpu  cpu.c  cwan4  cwan4_a1.c
```

Then let run with "&", i.e. "./cpu A&". And we can execute the "ls" again during the execution. This time we will see the ls runs instantly, which show that "./cpu A" process runs in parallel. And another thing we can discover is the "$" show instantly just after we run "./cpu A&", which is also show that we can run other command at that time.



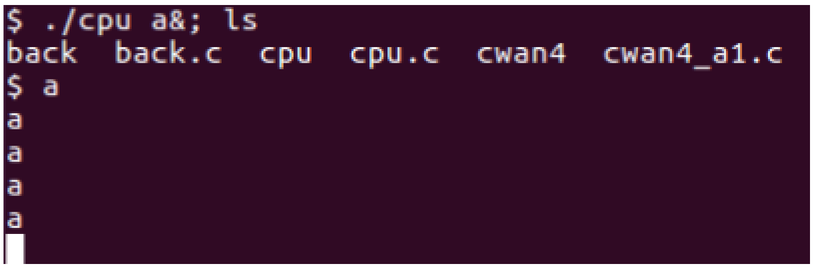
```
$ ./cpu A&
$ A
ls
back  back.c  cpu  cpu.c  cwan4  cwan4_a1.c
$ A
A
A
A
```

Then let try the infinite loop program. Run "./back &", we will see the "$" shows instantly. And we can see the process by run "ps". It will show that there is process named back. Then we can kill it with its PID. This can also prove the "&" realize parallel function.



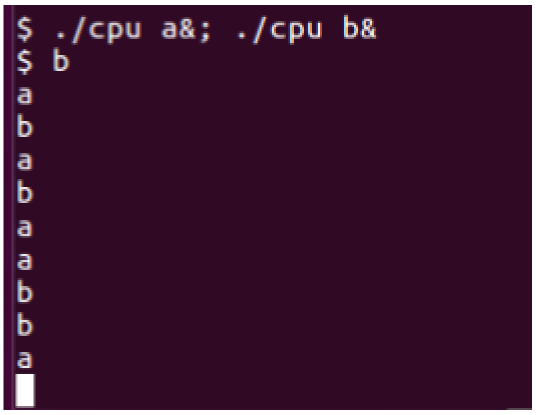
```
$ ./back &
$ ps
  PID TTY          TIME CMD
 3174 pts/1    00:00:00 bash
 3350 pts/1    00:00:00 cwan4
 3416 pts/1    00:00:00 back
 3417 pts/1    00:00:00 ps
$ kill 3416
```

At last, let try ";" and "&" together. Run "./cpu a&; ls". We will see the similar result that the "./cpu" runs in parallel with "ls" and "ls" can execute instantly.

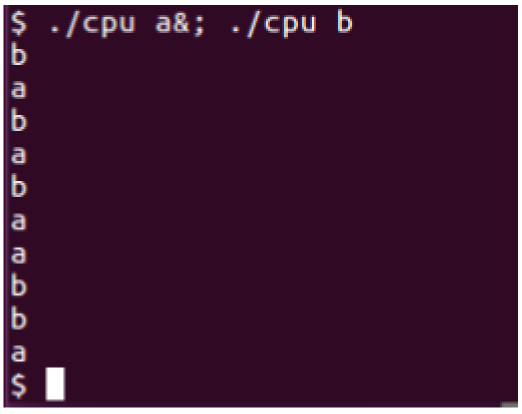


```
$ ./cpu a&; ls
back  back.c  cpu  cpu.c  cwan4  cwan4_a1.c
$ a
a
a
a
a
```

Let try "./cpu a&; ./cpu b&" or "./cpu a&; cpu b", we can see the similar result that shows they run in parallel. The a and b will display alternately (not exactly alternately). So the shell realize the parallel and sequence command in same time.



```
$ ./cpu a&; ./cpu b&
$ b
a
b
a
b
a
a
b
b
a
```



```
$ ./cpu a&; ./cpu b
b
a
b
a
b
a
a
b
b
a
$
```

To summarize, the test commands I used are as the following:

ls; wc cpu.c

wc cpu.c; ls

./cpu A (with ls after its execution)

./cpu A&

./back & (see the process by "ps")

./cpu a&; ls

./cpu a&; ./cpu b&

./cpu a&; ./cpu b